Intel® Fortran Compiler 19.1 Developer Guide and Reference

Disclaimer and Legal Information

Contents

Notices and Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Intel, the Intel logo, Intel Atom, Intel Core, Intel VTune, MMX, Pentium, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

This software and the related documents are Intel copyrighted materials, and your use of them is governed by the express license under which they were provided to you (**License**). Unless the License provides otherwise, you may not use, modify, copy, publish, distribute, disclose or transmit this software or the related documents without Intel's prior written permission.

This software and the related documents are provided as is, with no express or implied warranties, other than those that are expressly stated in the License.

Intel®Fortran Compiler 19.1 Developer Guide and Reference

The following are some important features of the compiler:

Compiler Setup

Compiler Setup explains how to invoke the compiler on the command line or from within an IDE.

OpenMP* Support

The compiler supports many OpenMP* features, including most of OpenMP* Version TR4: Version 5.0.

Compiler Options

Compiler Options provides information about options you can use to affect optimization, code generation, and more.

Language Reference

The Language Reference provides information on language syntax and semantics, on adherence to various Fortran standards, and on extensions to those standards.

For information about the Fortran standards, visit the Fortran standards technical committee website at http://j3-fortran.org/.

Fortran Language Extensions

Fortran Language Extensions provides information about how to use additional implementation features, such as creating a Component Object Model server and generating listing and map files, among others.

Mixed Language Programming

Mixed Language Programming provides information about Fortran and C interoperable procedures and data types, as well as various specifics of mixed-language programming.

List of Run-Time Error Messages

List of Run-Time Error Messages describes the errors processed by the Intel® Fortran run-time library (RTL).

Context Sensitive/F1 Help

To use the Context Sensitive/F1 Help feature, visit the Download Documentation: Intel® Compiler (Current and Previous) page and follow the instructions provided there.

Part I

Introducing the Intel®Fortran Compiler

Using the Intel®Fortran Compiler , you can compile and generate applications that can run on Intel® 64 architecture. You can also create programs for the IA-32 architecture on Windows* and Linux*.

Intel® 64 architecture applications can run on the following:

- Windows* operating systems for Intel® 64 architecture-based systems.
- Linux* operating systems for Intel® 64 architecture-based systems.
- macOS* operating systems for Intel® 64 architecture-based systems.

IA-32 architecture applications can run on the following:

- Supported Windows* operating systems
- Supported Linux* operating systems

NOTE
Starting with the 19.0 release of the Intel®Fortran Compiler, macOS* 32-bit applications are no longer supported. If you want to compile 32-bit applications, you must use an earlier version of the compiler and you must use Xcode* 9.4 or earlier.

Unless specified otherwise, assume the information in this document applies to all supported architectures and all operating systems.

You can use the compiler in the command-line or in a supported Integrated Development Environment (IDE):

- Microsoft Visual Studio* (Windows* only)
- Eclipse*/CDT (Linux* only)
- Xcode* (macOS* only)

See the Release Notes for complete information on supported architectures, operating systems, and IDEs for this release.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Feature Requirements

To use these tools and features, you need licensed versions of the tools and you must have an appropriately supported version of the product edition. For more information, check the product release notes.

NOTE Some features may require additional product installation.

The following table shows components (tools) and where to find additional information on them.

Component	More Information
Intel® Inspector Intel® Trace Analyzer and Collector	More information on tools and features can be found on the Intel IDZ: http://software.intel.com/

The following table lists dependent features and their corresponding required products. For certain compiler options, the compilation may fail if the option is specified but the required product is not installed. In this case, remove the option from the command line and recompile. For more information about requirements for a particular product, see http://www.intel.com/software/products/.

Feature Requirements

Feature	Requirement
Thread Checking	Intel® Inspector
Trace Analyzing and Collecting	Intel® Trace Analyzer and Collector Compiler options related to this feature may require a set-up script. For further information, see the product documentation.
Coarray programs built to run using distributed memory	Intel® Parallel Studio XE Cluster Edition for Windows* or Linux*

Other Tools

Feature	Requirement
Privatization of static data for the MultiProcessor Computing (MPC) unified parallel runtime	MPC framework elements The MPC unified parallel runtime must be installed. For more information, see http://mpc.hpcframework.com/

Refer to the Release Notes for detailed information about system requirements, late changes to the products, supported architectures, operating systems, and Integrated Development Environments (IDEs).

Getting Help and Support

Windows*

Documentation is available from within the version of Microsoft Visual Studio*. You must install the documentation on your local system. To use the feature, visit the Download Documentation: Intel® Compiler (Current and Previous) page and follow the instructions provided there. From the **Help** menu, choose **Intel Compilers and Libraries** to view the installed user and reference documentation.

Linux* and macOS*

On Linux and macOS*, the documentation has limited integration in the Eclipse*/CDT and Xcode*. In both cases, the integrated documentation only provides details about where to find the product documentation on your local system.

Intel Software Documentation

You can find product documentation for many released products at: http://software.intel.com/en-us/intel-software-technical-documentation/

Product Website and Support

To find product information, register your product, or contact Intel, visit: https://software.intel.com/en-us/support/

At this site, you will find comprehensive product information, including:

- Links to Get Started, Documentation, Individual Support, and Registration
- Links to information such as white papers, articles, and user forums
- Links to product information
- Links to news and events

Online Service Center

Each purchase of an Intel® Software Development Product includes a year of support services, which includes priority customer support at our Online Service Center. For more information about the Online Service Center visit: https://software.intel.com/en-us/support/online-service-center

> **NOTE** To access support, you must register your product at the Intel® Registration Center: https://registrationcenter.intel.com

Release Notes

For detailed information on system requirements, late changes to the products, supported architectures, operating systems, and Integrated Development Environments (IDE) see the Release Notes for the product.

Forums

You can find helpful information in the Intel Software user forums. You can also submit questions to the forums. To see the list of the available forums, go to https://software.intel.com/en-us/forum/

Related Information

Reference and Tutorial Information

The following commercially published documents provide reference or tutorial information about Fortran:

- *Introduction to Programming with Fortran with coverage of Fortran 90, 95, 2003, 2008 and 77*, by I.D. Chivers and J. Sleightholme; published by Springer, ISBN 9780857292322
- *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*, by Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T., published by Springer Verlag, ISBN 9781846283789
- *Fortran 95/2003 For Scientists and Engineers*, by Chapman S.J., published by McGraw- Hill, ISBN 0073191574
- *Modern Fortran Explained: Incorporating Fortran 2018*, by Metcalf M., Reid J. and Cohen M., 2018, published by Oxford University Press, ISBN-13: 978-0198811886

Intel does not endorse these books or recommend them over other books on the same subjects.

Additional Product Information

For additional technical product information including white papers, forums, and documentation, visit https://software.intel.com/en-us/intel-sdp-home/

Additional Language Information

- For information about the OpenMP* standards, visit the OpenMP website: http://www.openmp.org/
- For information about the Fortran standards, visit the Fortran standards technical committee website: http://j3-fortran.org/

Notational Conventions

Information in this documentation applies to all supported operating systems and architectures unless otherwise specified. This documentation uses the following conventions:

Notational Conventions

THIS TYPE	Indicates statements, data types, directives, and other language keywords. Examples of statement keywords are WRITE, INTEGER, DO, and OPEN.
this type	Indicates command-line or option arguments, new terms, or emphasized text. Most new terms are defined in the Glossary.
`This type`	Indicates a code example.
`This type`	Indicates what you type as input.
This type	Indicates menu names, menu items, button names, dialog window names, and other user-interface items.
File > Open	Menu names and menu items joined by a greater than (>) sign to indicate a sequence of actions. For example, **Click File > Open** indicates that in the **File** menu, you would click **Open** to perform this action.
{value | value}	Indicates a choice of items or values. You can usually only choose one of the values in the braces.
[*item*]	Indicates items that are optional. Brackets are also used in code examples to show arrays.
item [, *item*]...	Indicates that the item preceding the ellipsis (...) can be repeated. In some code examples, a horizontal ellipsis means that not all of the statements are shown.
Intel® Fortran	This term refers to the name of the common compiler language supported by the Intel® Fortran Compiler.
Windows* or Windows* operating system	These terms refer to all supported Microsoft Windows* operating systems.

Linux* or Linux* operating system	These terms refer to all supported Linux* operating systems.
macOS* or macOS* operating system	These terms refer to all supported macOS* operating systems.
Microsoft Visual Studio*	An asterisk at the end of a word or name indicates it is a third-party product trademark.
compiler option	This term refers to Linux*, macOS*, or Windows* options, which are used by the compiler to compile applications. The following conventions are used as shortcuts when referencing compiler option names in text: • Many options have names that are the same on Linux*, macOS*, and Windows*, except that the Windows* form starts with an initial / and the Linux* and macOS*form starts with an initial -. Within text, such option names are shown without the initial character. For example, `check`. • Many options have names that are the same on Linux*, macOS*, and Windows*, except that the Windows* form starts with an initial Q. Within text, such option names are shown as `[Q]option-name`. For example, if you see a reference to `[Q]ipo`, the Linux* and macOS* form of the option is `-ipo` and the Windows* form of the option is `/Qipo`. • Several compiler options have similar names except that the Linux* and macOS* forms start with an initial q and the Windows* form starts with an initial Q. Within text, such option names are shown as `[q or Q]option-name`. For example, if you see a reference to `[q or Q]opt-report`, the Linux* and macOS* form of the option is `-qopt-report` and the Windows* form of the option is `/Qopt-report`. Other dissimilar compiler option names are shown in full.

Conventions Used in Compiler Options

`/option` or `-option`	A slash before an option name indicates the option is available on Windows*. A dash before an option name indicates the option is available on Linux* and macOS* systems. For example: • Windows* option: `/help` • Linux* and macOS*option: `-help`

NOTE If an option is available on all supported operating systems, no slash or dash appears in the general description of the option. The slash and dash only appear where the option syntax is described.

`/option:argument` or `-option=argument`	Indicates that an option requires an argument (parameter). For example, you must specify an argument for the following options: • Windows* option: `/tune:`*processor* • Linux* and macOS* option: `-mtune=`*processor*
`/option:`*keyword* or `-option=`*keyword*	Indicates that an option requires one of the *keyword* values.
`/option`[`:`*keyword*] or `-option`[`=`*keyword*]	Indicates that the option can be used alone or with an optional keyword.
`option`[*n*] or `option`[`:`*n*] or `option`[`=`*n*]	Indicates that the option can be used alone or with an optional value. For example, in `/Qunroll`[`:`*n*] and `-unroll`[`=`*n*], the *n* can be omitted or a valid value can be specified for *n*.
`option`[`-`]	Indicates that a trailing hyphen disables the option. For example, `/Qglobal_hoist-` disables the Windows* option `/Qglobal_hoist`.
`[no]option` or `[no-]option`	Indicates that `no` or `no-` preceding an option disables the option. For example, in the Windows* option `/[no]traceback`, `/traceback` enables the option, while `/notraceback` disables it. In the Linux* and macOS* option `-[no-]global_hoist`, `-global_hoist` enables the option, while `-no-global_hoist` disables it. In some options, the `no` appears later in the option name. For example, `-fno-common` disables the `-fcommon` option.

Conventions Used in Language Reference

This color	Indicates Intel extensions (non-standard features) that may or may not be implemented by other compilers. Features defined by the Fortran Standards or the OpenMP* Standards are shown in black.
Fortran	This term refers to language information that is common to previously supported Fortran standards, Fortran 2008, and the Intel® Fortran Compiler.

Standard Fortran	This term refers to language information that is common to ANSI/ISO Fortran 95, ANSI/ISO Fortran 90, and Intel® Fortran.
Fortran 95	This term refers to language features specific to ANSI/ISO Fortran 95.
Fortran 2003	This term refers to language features specific to ANSI/ISO Fortran 2003.
Fortran 2008	This term refers to language features specific to ISO/IEC 1539-1:2010 (Fortran 2008).
Fortran 2018	This term refers to language features specific to the Draft International Fortran standard expected to be published late 2018 or early 2019.
integer	This term refers to the INTEGER(KIND=1), INTEGER(KIND=2), INTEGER (INTEGER(KIND=4)), and INTEGER(KIND=8) data types as a group.
INTEGER	This term refers to the default data type of objects declared to be INTEGER. INTEGER is equivalent to INTEGER(KIND=4), unless a compiler option specifies otherwise.
real	This term refers to the REAL (REAL(KIND=4)), DOUBLE PRECISION (REAL(KIND=8)), and REAL(KIND=16) data types as a group.
REAL	This term refers to the default data type of objects declared to be REAL. REAL is equivalent to REAL(KIND=4), unless a compiler option specifies otherwise.
complex	This term refers to the COMPLEX (COMPLEX(KIND=4)), DOUBLE COMPLEX (COMPLEX(KIND=8)), and COMPLEX(KIND=16) data types as a group.
COMPLEX	This term refers to the default data type of objects declared to be COMPLEX. COMPLEX is equivalent to COMPLEX(KIND=4), unless a compiler option specifies otherwise.
logical	This term refers to the LOGICAL(KIND=1), LOGICAL(KIND=2), LOGICAL (LOGICAL(KIND=4)), and LOGICAL(KIND=8) data types as a group.
LOGICAL	This term refers to the default data type of objects declared to be LOGICAL. LOGICAL is equivalent to LOGICAL(KIND=4), unless a compiler option specifies otherwise.
< Tab>	This symbol indicates a nonprinting tab character.
^	This symbol indicates a nonprinting blank character.

Platform Labels

A platform is a combination of an operating system (OS) and a central processing unit (CPU), which provides a distinct environment for product use (in this case, a computer language). An example of a platform is Microsoft Windows* on processors using Intel® 64 architecture.

In this documentation, the information applies to all supported platforms unless it is otherwise labeled for a specific platform (or platforms).

These labels may be used to identify specific platforms:

L*X	Applies to a Linux* operating system.
M*X	Applies to a macOS* operating system.
W*S	Applies to a Microsoft Windows* operating system.

Part II

Compiler Setup

You can use the Intel®Fortran Compiler from the command line, or from the IDEs listed below.

These IDEs are described in further detail in their corresponding sections.

Using the Command Line

Specifying the Location of Compiler Components with compilervars

Before you invoke the compiler, you may need to set certain environment variables that define the location of compiler-related components.

The Intel® Fortran Compiler includes `compilervars` scripts to set environment variables:

- On Linux* and macOS*, the file is a shell script called `compilervars.sh` or `compilervars.csh`.
- On Windows*, the file is a batch file called `compilervars.bat`.

The following information is operating system dependent.

> **NOTE** IA-32 architecture is no longer supported on macOS*.

Linux and macOS*:

Set the environment variables before using the compiler by sourcing the shell script `compilervars.sh` or `compilervars.csh`. Depending on the shell, you can use the `source` command or a `.` (dot) to source the shell script, according to the following rules:

- **`.csh` script**: Use the `source` command.
- **`.sh` script**:
  - Bash: Use either the `source` command or `.` (dot space).
  - Dash or other POSIX-compliant shell: Use `.` (dot space).

```
//# Bash shell:
source /<install-dir>/bin/compilervars.sh <arg>
. /<install-dir>/bin/compilervars.sh <arg>

//# examples: (assuming <install-dir> is /installed/compiler/)
prompt> source /installed/compiler/bin/compilervars.sh intel64
prompt> . /installed/compiler/bin/compilervars.sh intel64

// OR//# C shell:
source /<install-dir>/bin/compilervars.csh <arg>

//# example: (assuming <install-dir> is /installed/compiler/)
```

```
prompt> source /installed/compiler/bin/compilervars.csh intel64

// OR//# Dash or other POSIX-compliant shell:
. /<install-dir>/bin/compilervars.sh <arg>

//# example: (assuming <install-dir> is /installed/compiler/)
prompt> . /installed/compiler/bin/compilervars.sh intel64
```

The environment script file requires a target architecture argument `<arg>`:

- `ia32`: Compiler and libraries for IA-32 architecture-based targets only.
- `intel64`: Compiler and libraries for Intel® 64 architecture-based targets only.

If you want the script to run automatically, add the same command to the end of your startup file.

For example, a `.bash_profile` entry for `compilervars.sh` for IA-32 architecture targets only:

```
# set environment vars for Intel(R)  Fortran Compiler
source <install-dir>/bin/compilervars.sh ia32
```

> **NOTE** Symbolic links are created in the `/opt/intel` directory at install. The environment variables use symbolic links. When two versions of the Intel® Fortran Compiler are installed, the newest installed version is symbolically linked.

If the proper environment variables are not set, an error similar to the following appears when attempting to execute a compiled program:

```
./a.out: error while loading shared libraries:
libimf.so: cannot open shared object file: No such file or directory
```

Windows:

Under normal circumstances, you do not need to run the `compilervars.bat` batch file. The installed shortcut **Compiler 19.1 for IA-32 Visual Studio <Year> environment** sets these variables automatically.

For information on using the command line see Using the Command Line on Windows.

> **NOTE** You need to run the batch file if a command line is opened without using one of the provided menu items in the **Start** menu, or if you want to use the compiler from a script of your own.

The batch file inserts the directories used by the Intel® Fortran Compiler at the beginning of the existing paths. Because these directories appear first, they are searched before the directories in the path lists that are provided by Windows. This is especially important if the existing path includes directories with files that have the same names as those needed by the Intel® Fortran Compiler .

If needed, you can run `compilervars.bat` each time you begin a session on Windows systems by specifying it as the initialization file with the PIF Editor.

The batch file takes two arguments:

```
<install-dir>\bin\compilervars.bat <arg1> [<arg2>]
```

Where *<arg1>* is one of the following:

- `intel64`: Compiler and libraries for Intel® 64 architecture only (host and target).
- `ia32`: Compiler and libraries for IA-32 architecture only (host and target).

The *<arg2>* is optional. If specified, it is one of the following:

- `vs2019`: Microsoft Visual Studio* 2019
- `vs2017`: Microsoft Visual Studio 2017

> **NOTE** If <*arg2*> is not specified, the script uses the highest installed version of Microsoft Visual Studio detected during the installation procedure.

Invoking the Intel® Compiler

Requirements before Using the Command Line

You may need to set certain environment variables before using the command line. For more information, see Specifying the Location of Compiler Components with compilervars.

macOS*: Alternatively, you can invoke the compiler with the commands described below using symbolic links in `/usr/local/bin`.

Using the Intel® Fortran Compiler from the Command Line

You can invoke the Intel® Fortran Compiler on the command line using the `ifort` command.

> **NOTE** For Windows* and macOS* systems, you can use the compiler within the IDE.
> For more information on using the Microsoft Visual Studio* IDE, see Using Microsoft Visual Studio*.
> For information on using the Xcode* IDE, see Using Xcode*.

The syntax of the `ifort` command is:

```
ifort [options]input_file(s)
```

The `ifort` command can compile and link projects in one step, or can compile them and then link them as a separate step.

In most cases, a single `ifort` command invokes the compiler and linker. You can also use `ld` (Linux and macOS*) or `link` (Windows*) to build libraries of object modules. These commands provide syntax instructions at the command line if you request it with the `-help` (Linux and macOS*, or the `/help` or `/?` (Windows*) options.

The `ifort` command automatically references the appropriate Intel® Fortran Run-Time Libraries when it invokes the linker. To link one or more object files created by the Intel® Fortran compiler, you should use the `ifort` command instead of the `link` command.

The `ifort` command invokes a *driver program* that is the user interface to the compiler and linker. It accepts a list of command options and file names and directs processing for each file. The driver program does the following:

- Calls the Intel® Fortran Compiler to process Fortran files.
- Passes the linker options to the linker.
- Passes object files created by the compiler to the linker.
- Passes libraries to the linker.
- Calls the linker or librarian to create the executable or library file.

Because the driver calls other software components, they may return error messages. For instance, the linker may return a message if it cannot resolve a global reference. The `watch` option can help clarify which component is generating an error.

For a complete listing of compiler options, see the Compiler Options reference.

NOTE
The compiler recognizes language extensions for offloading in the source program by default and builds a heterogeneous binary that runs on the target and host when any are present. If your program includes these language extensions and you do not want to build a heterogeneous binary, specify the `-no-offload` compiler option. For more information, see the -qno-offload compiler option.

Offload is not supported on Windows systems.

NOTE
Windows systems support characters in Unicode* (multibyte) format. The compiler processes the file names containing Unicode characters.

Syntax Rules

The following rules apply when specifying `ifort` on the command line:

Argument	Description
`options`	An option is specified by one or more letters preceded by a hyphen (`-`) for Linux and macOS* or a slash (`/`) for Windows. (You can use a hyphen (`-`) instead of a slash (`/`) for Windows*, but this is not the preferred method.) Options cannot be combined with a single slash or hyphen, you must specify the slash or hyphen for each option specified. For example: `/1 /c` is correct, but `/1c` is not. Options can take arguments in the form of filenames, strings, letters, and numbers. If a string includes spaces, they must be enclosed in quotation marks. Some options take arguments in the form of filenames, strings, letters, or numbers. Except where otherwise noted, a space between the option and its argument(s) can be entered or combined. For a complete listing of compiler options, see the Compiler Options reference. Some compiler options are case-sensitive. For example, `c` and `C` are two different options. Option names can be abbreviated, enter as many characters as are needed to uniquely identify the option. Compiler options remain in effect for the whole compilation unless overridden by a compiler directive. Certain options accept one or more keyword arguments following the option name on Windows*. To specify multiple keywords, you typically specify the option multiple times. However, some options allow comma-separated keywords. For example: • Options that use a colon can use an equal sign (`=`) instead. • Standard output and standard error can be redirected to a file, avoiding displaying excess text, which slows down execution; scrolling text in a terminal window on a workstation can cause an I/O bottleneck (increased elapsed time) and use more CPU time. See the examples in the next section.

Argument	Description
	NOTE Options on the command line apply to all files. In the following example, the `-c` and `-nowarn` options apply to both files `x.f` and `y.f`: `ifort -c x.f -nowarn y.f`
`input file(s)`	Multipleinput_files can be specified, using a space as a delimiter. When a file is not in PATH or working directory, specify the directory path before the file name. The filename extension specifies the type of file. See Understanding File Extensions.
`Xlinker` (Linux and macOS*) or `/link` (Windows)	Unless specified with certain options, the command line compiles and links the files you specify. To compile without linking, specify the `c` option. All compiler options must precede the `-Xlinker` (Linux and macOS*) or `/link` (Windows*) options. Options that appear following `-Xlinker` or `/link` are passed directly to the linker.

Examples of the ifort Command

The following command compiles `x.for`, links, and creates an executable file. This command generates a temporary object file, which is deleted after linking:

```
ifort x.for
```

The following command compiles `x.for` and generates the object file `x.o` (Linux and macOS*) or `x.obj` (Windows*). The `c` option prevents linking (it does not link the object file into an executable file):

```
//  (Linux and macOS*)
ifort -c x.for
//  (Windows*)
ifort x.for /c
```

The following command links `x.o` or `x.obj` into an executable file. This command automatically links with the default Intel® Fortran libraries:

```
//  (Linux and macOS*)
ifort x.o
//  (Windows*)
ifort x.obj
```

The following command compiles `a.for`, `b.for`, and `c.for`, creating three temporary object files, then linking the object files into an executable file named `a.out` (Linux and macOS*) or `a.exe` (Windows*).

```
ifort a.for b.for c.for
```

Compile the source files that define modules **before** the files that reference the modules (in `USE` statements) when using modules and compile multiple files.

When you use a single `ifort` command, the order in which files are placed on the command line is significant. For example, if the free-form source file `moddef.f90` defines the modules referenced by the file `projmain.f90`, use the following syntax:

```
ifort moddef.f90 projmain.f90
```

To specify a particular name for the executable file, specify the option `-o` (Linux and macOS*) or `/exe` (Windows*):

```
//  (Linux and macOS*)
ifort x.for -o myprog.out
//  (Windows*)
ifort x.for /exe:myprog.exe
```

To redirect output to a file and then display the program output (Linux and macOS*):

```
//  (Linux and macOS*)
myprog > results.lis
more results.lis
```

To place standard output into file `one.out` and standard error into file `two.out`:

```
//  (Windows*)
ifort filenames /options 1>one.out 2>two.out
// OR
ifort filenames /options >one.out 2>two.out
```

To place standard output and standard error into a single file `both.out` (Windows*):

```
//  (Windows*)
ifort filenames /options 1>both.out 2>&1
// OR
ifort filenames /options >both.out 2>&1
```

Other Methods for Using the Command Line to Invoke the Compiler

- **Using makefiles from the Command Line:** Use makefiles to specify a number of files with various paths and to save this information for multiple compilations. For more information on using makefiles, see Using Makefiles to Compile Your Application.
- **Using the devenv Command from the Command Line (Windows Only):** Use `devenv` to set various options for the IDE, and to build, clean, and debug projects from the command line. For more information on the `devenv` command, see the `devenv` description in the Microsoft Visual Studio documentation.
- **Using a Batch File from the Command Line:** Create and use a `.bat` file to consistently execute the compiler with a desired set of options instead of retyping the command each time you need to recompile.

See Also

Using the compilervars File to Specify Location of Components
Understanding File Extensions
Using Microsoft Visual Studio
Using Xcode
Using Makefiles to Compile Your Application
watch
qoffload

Using the Command Line on Windows*

The compiler provides a shortcut to access the command line with the appropriate environment variables already set.

To invoke the compiler from the command line:

- **Windows* 10:**

  1. From the **Start** screen, swipe up from the bottom edge, or click the arrow to move the **Apps** screen.

2. In the **Apps** screen, scroll to the right until you see the program group for your Intel product.
3. Click **Intel 64 Visual Studio mode**, **IA-32 Visual Studio mode**, or **Intel® oneAPI command prompt** to open a command window on your desktop.

- **Older Windows Versions:**

  Click the appropriate mode from the **Start** menu item for your Intel product under **All Programs** > **the product** > **Compiler and Performance Libraries** > **Command Prompt with Intel Compiler**.

  NOTE Older Windows versions are not supported for oneAPI content.

The command line opens.

You can use any command recognized by the Windows command prompt, plus some additional commands.

Because the command line runs within the context of Windows, you can easily switch between the command line and other applications for Windows or have multiple instances of the command line open simultaneously.

When you are finished working in a command line, use the **exit** command to close and end the session.

Running Fortran Applications from the Command Line

For programs run from the command line, the operating system searches directories listed in the PATH environment variable to find the requested executable file.

The program can also be run by specifying the complete path of the executable file. On Windows* operating systems, any DLLs you are using must be in the same directory as the executable or in one specified in the path.

Multithreaded Programs

If the program is multithreaded, each thread starts on whichever processor is available at the time. On a computer with one processor, the threads all run in parallel, but not simultaneously; the single processor switches among them. On a computer with multiple processors, the threads can run simultaneously.

Using the -fpscomp `filesfromcmd` Option

If you specify the `fpscomp` option with keyword `filesfromcmd`, the command line that executes the program can include additional filenames to satisfy `OPEN` statements in the program for which the filename field (`FILE` specifier) has been left blank. The first filename on the command line is used for the first `OPEN` statement executed, the second filename for the second `OPEN` statement, and so on.

NOTE
In the Visual Studio* IDE, you can provide these filenames using **Project** > **Properties**. Choose the Debugging category and enter the filenames in the **Command Arguments** text box.

Each filename on the command line (or in an IDE dialog box) must be separated from the names around it by one or more spaces or tab characters. You can enclose each name in quotation marks ("<*filename*>"), but this is not required unless the argument contains spaces or tabs. A null argument consists of an empty set of quotation marks with no filename enclosed ("").

The following example runs the program `MYPROG.EXE` from the command line:

```
MYPROG "" OUTPUT.DAT
```

Because the first filename argument is null, the first OPEN statement with a blank filename field produces the following message:

```
File name missing or blank - please enter file name
UNIT number  ?
```

The *number* is the unit number specified in the OPEN statement. The filename OUTPUT.DAT is used for the second OPEN statement executed. If additional OPEN statements with blank filename fields are executed, you will be prompted for more filenames.

Instead of using the fpscomp option with keyword filesfromcmd, you can:

- Call the GETARG library routine to return the specified command-line argument. To execute the program in the Visual Studio* IDE, provide the command-line arguments to be passed to the program using **Project** > **Properties**. Choose the Debugging category and enter the arguments in the **Command Arguments** text box.
- On Windows* OS, call the GetOpenFileName Windows* API routine to request the file name using a dialog box.

See Also

fpscomp option

Understanding File Extensions

Input File Extensions

The Intel® Fortran Compiler interprets the type of each input file by the file name extension.

The file extension determines if a file gets passed to the compiler or to the linker. The following types of files are used with the compiler:

- Files passed to the compiler: .f90, .for, .f, .fpp, .i, .i90, .ftn

  Typical Fortran source files have a file extension of .f90, .for, and .f. When editing your source files, you need to choose the source form, either free-source form or fixed-source form (or a variant of fixed form called tab form). You can use a compiler option to specify the source form used by the source files (see the description for the free or fixed compiler option) or you can use specific file extensions when creating or renaming your files. For example, the compiler assumes that files with an extension of:

  - .f90 or .i90 are free-form source files.
  - .f, .for, .ftn, or .i are fixed-form (or tab-form) files.
- Files passed to the linker: .a, .lib, .obj, .o, .exe, .res, .rbj, .def, .dll

The most common file extensions and their interpretations are:

Filename	Interpretation	Action
file.a (Linux and macOS*)	Object library	Passed to the linker.
file.lib (Windows)		
file.f	Fortran fixed-form source	Compiled by the Intel® Fortran compiler.
file.for		
file.ftn		
file.i		

Filename	Interpretation	Action

`file.fpp`
On Linux, filenames with the following uppercase extensions:
`file.FPP`
`file.F`
`file.FOR`
`file.FTN`
	Fortran fixed-form source	Automatically preprocessed by the Intel® Fortran preprocessor `fpp`; then compiled by the Intel® Fortran compiler.

`file.f90` `file.i90`	Fortran free-form source	Compiled by the Intel® Fortran compiler.
`file.F90` (Linux and macOS*)	Fortran free-form source	Automatically preprocessed by the Intel® Fortran preprocessor `fpp`; then compiled by the Intel® Fortran compiler.
`file.s` (Linux and macOS*) `file.asm` (Windows)	Assembly file	Passed to the assembler.
`file.o` (Linux and macOS*) `file.obj` (Windows)	Compiled object file	Passed to the linker.

When you compile from the command line, you can use the compiler configuration file to specify default directories for input libraries. To specify additional directories for input files, temporary files, libraries, and for the files used by the assembler and the linker, use compiler options that specify output file and directory names.

Output File Extensions (Windows)

On Windows operating systems, many compiler options allow you to specify the name of the output file being created. These compiler options are summarized in the table below.

If you specify only a filename without an extension, a default extension is added for the file.

Compiler option	Default file extension
/Fa*file*	`.ASM`
/dll:*file*	`.DLL`
/exe:*file*	`.EXE`
/map:*file*	`.MAP`

See Also

Invoking the Intel® Fortran Compiler

Using Makefiles to Compile Your Application

This topic describes the use of makefiles to compile your application. You can use makefiles to specify a number of files with various paths, and to save this information for multiple compilations.

Using Makefiles to Store Information for Compilation on Linux* or macOS*

To run `make` from the command line using the Intel®Fortran Compiler, make sure that `/usr/bin` and `/usr/local/bin` are in your *PATH* environment variable.

If you use the C shell, you can edit your `.cshrc` file and add the following:

```
setenv PATH /usr/bin:/usr/local/bin:$PATH
```

Then you can compile using the following syntax:

```
make -f yourmakefile
```

Where `-f` is the `make` command option to specify a particular makefile name.

Using Makefiles to Store Information for Compilation on Windows*

To use a makefile to compile your source files, use the `nmake` command . For example, if your project is `your_project.mak`, you can use the following syntax:

```
nmake /f [makefile_name.mak] FPP=[compiler_name.exe] LINK32=[linker_name.exe]
```

For example:

```
prompt> nmake /f your_project.mak FPP=ifort.exe LINK32=xilink.exe
```

Argument	Description
`/f`	The `nmake` option to specify a makefile.
`your_project.mak`	The makefile used to generate object and executable files.
`FPP`	The preprocessor/compiler that generates object and executable files. (The name of this macro may be different for your makefile.)
`LINK32`	The linker that is used.

The `nmake` command creates object files (`.obj`) and executable files (`.exe`) from the information specified in the `your_project.mak` makefile.

Generating Build Dependencies for Use in a Makefile

Use the `gen-dep` compiler option to generate build dependencies for a compilation.

Build dependencies include a list of all files included with `INCLUDE` statements and `.mod` files accessed with `USE` statements. The resulting output can be used to create a makefile to with the appropriate dependencies resolved.

Consider a source file that contains the following:

```
 module b
 include 'gendep001b.inc'
 end module b

 program gendep001
 use b
 a_global = b_global
end
```

When you compile the source using the `gen-dep` option, the following output is produced:

```
b.mod : \
 gendep001.f90
gendep001.obj : \
 gendep001.f90 gendep001b.inc
```

This output indicates that the generated file, `b.mod`, depends on the source file, `gendep001.f90`. Similarly, the generated file, `gendep001.obj`, depends on the files, `gendpe001.f90` and `gendep001b.inc`.

Using Microsoft Visual Studio* (Windows*)

You can use the Intel® Fortran Compiler within the Microsoft Visual Studio* integrated development environment (IDE) to develop Fortran applications, including static library (`.LIB`), dynamic link library (`.DLL`), and main executable (`.EXE`) applications. This environment makes it easy to create, debug, and execute programs. You can build your source code into several types of programs and libraries, using the IDE or from the command line.

The IDE offers these major advantages:

- Makes application development quicker and easier by providing a visual development environment.
- Provides integration with the native Microsoft Visual Studio* debugger.
- Makes other IDE tools available.

See Also

Performing Common Tasks with Microsoft Visual Studio*
Using Microsoft Visual Studio* Solution Explorer
Using Breakpoints in the Debugger

Using Microsoft Visual Studio* Solution Explorer

After starting Microsoft Visual Studio*, a screen appears. This shows an open **Solution** named `Console1`, a **Project** named `Console1`, and the **source file**`Console1.f90` have been opened. The right pane shows the file `Console1.f90`, opened in the default language-sensitive integrated development environment text editor, which uses different colors to identify the following:

- Source comments (green)
- Fortran standard language elements (blue)
- Other language text (black)

Solution Explorer View

The left pane shows the **Solution Explorer** view, which lets you view different aspects of your solution, such as the source files in your solution. The tabs displayed in **Solution Explorer** vary depending upon the products installed and the files associated with the current solution. The sample screen shown above shows a **Solution** tab and a **Property Manager** tab (not used by Intel® Fortran). To display the **Solution Explorer** view, select **View** > **Solution Explorer**.

To edit a file listed in **Solution Explorer**, either double-click its file name or select **File** > **Open** and specify the file.

The **Output** window displays compilation and linker messages. To display the **Output** window, select **View** > **Output**. The **Output** window also links to the build log, if the **Generate Build logs** option is enabled in **Tools** > **Options** > **Intel Compilers and Tools** > **Visual Fortran**.

Creating a New Project

Creating a New Project

When you create a project, Microsoft Visual Studio* automatically creates a corresponding solution to contain it. To create a new Intel® Fortran project using Microsoft Visual Studio*:

The `hello32` project assumes focus in the **Solution Explorer** view. The default Microsoft Visual Studio* solution is also named `hello32`

1. Select **File** > **New** > **Project**.
2. In the left pane, click **Intel® Visual Fortran** to display the Fortran project types. For each project type, available templates are listed in the right pane.
3. Click the appropriate project type (see Understanding Project Types).
4. Accept or specify a project name.
5. Accept or specify the Location for the project directory. Project files will be stored here. If the directory specified does not exist, it will be created.
6. Click **OK** to complete the new project.

The project and its files appear in the **Solution Explorer** view. For a COM Server project, you will see a second page with additional user options.

Add an Existing File to the Project

1. If not already open, open the project (use the **File** menu).
2. Select **Project** > **Add Existing Item**.
3. In the **Add Existing Item** dialog box that appears, select the Fortran files to be added to the project.

Add a New File to the Project

1. If not already open, open the project (use the **File** menu).
2. Select **Project** > **Add New Item**.
3. In the **Add New Item** dialog box that appears, choose the type of file.
4. Specify the file name. Click **Open**. The file name appears in the **Solution Explorer** view.
5. Use the Microsoft Visual Studio* editor to type in source code. Be sure to save your work when you are finished.

Organizing Existing Source Code

If you have existing source code, you should organize it into directories before creating a project, although it is easy to move files and edit your project definitions if you should later decide to reorganize your files.

Working with Fortran Modules

If your program uses Fortran modules, you do not need to explicitly add them to your project; they appear as dependencies (`.MOD` files).

A module file is a precompiled, binary version of a module definition, stored as a `.mod` file. When you change the source definition of a module, you can update the `.mod` file before you rebuild your project. To do this, compile the corresponding source file separately by selecting the file in the **Solution Explorer** window and selecting **Build** > **Compile**. If the module source file is part of your project, you do not need to compile the file separately. When you build your project, the Intel® Fortran Compiler determines what files need to be compiled.

To control the placement of module files in directories, use **Project** > **Properties** > **Fortran** > **Output Files** > **Module Path** in the IDE or the compiler option `[no]module` on the command line. The location you specify is automatically searched for `.mod` files.

To control the search for module files in directories, select one of the following:

- **In the IDE:**
  - **Project** > **Properties** > **Fortran** > **Preprocessor** > **Default Include and Use Path**
  - **Project** > **Properties** > **Fortran** > **Preprocessor** > **Ignore Standard Include Path**
- **On the Command Line:**
  - `X` or `I` and `assume:source_include` compiler options.

For a newly created project (or any other project), the IDE scans the file list for sources that define modules and compiles them before compiling the program units that use them. The IDE automatically scans the added project files for modules specified in `USE` statements, as well as any `INCLUDE` statements. It scans the source files for all tools used in building the project.

See Also

Understanding Project Types
I
X
assume

Performing Common Tasks with Microsoft Visual Studio*

This topic outlines the basic steps for using the Intel® Fortran Compiler with Microsoft Visual Studio*.

Building and Running a Fortran Project

- To build the application, select **Build** > **Build Solution**. Any errors will be displayed in the **Output Window**. Double-click on a message to go to the line in error.
- To run without debugging, select **Debug** > **Start Without Debugging**. The console window will remain open after the program exits until you press Enter.
- To run under the debugger, first set a breakpoint at the first executable line of the program by clicking in the gray column to the left of the source line. Then select **Debug** > **Start**. If the program exits normally, the console window will be closed automatically.

Converting Compaq* Visual Fortran Projects

For information on converting projects from Compaq* Visual Fortran to Intel® Fortran, see Converting Projects.

See Also

Creating a New Project
Converting Projects

Selecting a Version of the Intel® Compiler

If you have more than one version of the Intel® Fortran Compiler installed, you can choose which version to be used when building applications. You can also select different versions for different target platforms and the version of the compiler you are using. The target platform you select determines the compiler versions that appear in the **Selected** drop-down box.

To select the compiler version:

1. Select **Tools** > **Options** > **Intel Compilers and Tools** > **Visual Fortran** > **Compilers**.
2. Select a compiler from **Selected compiler**. Click **OK**.

See Also

Specifying a Target Platform for Solutions and Projects

Specifying Fortran File Extensions

You can specify additional Fortran free format and fixed format file extensions to be recognized as valid file extensions within the IDE. The IDE treats these additional extensions as compilable Fortran source files. You can also remove or modify existing extensions.

When you add a new extension, the IDE checks the registry to determine whether the extension is already associated with a language, tool, or file format. If there is such an association, a message informs you of this and you will not be allowed to add the extension.

To specify Fortran file extensions:

1. Open **Tools** > **Options**.
2. In the left pane, go to **Intel Compilers and Tools** > **Visual Fortran** > **General**.
3. Specify one or more Fortran File Extensions, each beginning with a period and separated by semi-colons. You can specify extensions for both **Free Format** and **Fixed Format**. Click **OK**.

These new settings take effect the next time you start Visual Studio*.

Use the **Reset** button to restore Fortran file extensions to their default settings.

Understanding Solutions, Projects, and Configurations

The Microsoft* Visual Studio* integrated development environment (IDE) consists of one or more projects contained within a **solution**. A solution can contain multiple projects. For example, if you have several Fortran applications that do different calculations, but are related to the same research application you are working on, you can store all the individual projects in a single solution. Along with a solution file (`.sln`), the IDE creates a solution user options (`.suo`) file for storing IDE customization.

The following table summarizes the files created by Microsoft* Visual Studio* when a new project is created:

File	Extension	Description
Project Solution file	`.sln`	Stores solution information, including the projects and items in the solution and their locations on disk.
Project file	`.vfproj` `.vcxproj`	Contains information used to build a single project or sub-project.
Solution options file	`.suo`	Contains IDE customization for the solution, based on the selected options.

Caution
Directly modifying these files with a text editor is not supported.

Caution
Before opening Compaq* Visual Fortran 6.0 projects or Intel® Visual Fortran 7.x projects in Microsoft* Visual Studio*, review the guidelines listed in Converting Projects.

Each project can specify one or more configurations to build from its source files. A configuration specifies such information as the type of application to build, the platform it runs on, and the tool settings to use when building. Having multiple configurations extends the scope of a project, but maintains a consistent source code base to work with.

Microsoft* Visual Studio* automatically creates **Debug** and **Release** (also known as **Retail**) configurations when a new project is started. The default configuration is the **Debug** configuration. To specify the current configuration, select **Configuration Manager** from the **Build** menu.

Specify build options in the **Project** > **Properties** dialog box, for one of the following:

- For all configurations (project-wide)
- For certain configurations (per configuration)
- For certain files (per file)

For example, specify compiler optimizations for all general configurations, but turn them off for certain configurations or certain files.

Once the files in the project are specified and the configurations for your project build are set, including the tool settings, build the project with the commands on the **Build** menu.

> **NOTE**
> For a multiple-project solution, make sure that the executable project is designated as the startup project (shown in bold in the **Solution Explorer** view). To modify the startup project, right click on the project and select **Set as StartUp Project**.

See Also

Converting Projects

Navigating Programmatic Components in a Fortran File

You can quickly navigate the code of the file currently open in the source editor using the **Tree Navigation Window**. The **Tree Navigation Window** displays the following components of the file as nested, selectable nodes in a tree:

- Programs
- M odules
- Subroutines with signature
- Functions with signature
- Types
- Interfaces

Nodes at each nested level are sorted alphabetically.

Any changes you make to a file, such as adding or deleting a component or changing a signature, are immediately reflected in the tree.

To navigate a file:

1. Select **View** > **Other Windows** > **Tree Navigation Window**.
   The **Tree Navigation Window** tab appears near the **Solution Explorer** tab. When no Fortran project is opened, the **Tree Navigation Window** is empty. When you open a Fortran file in the source editor, all components of the file appear in the window.
2. Select a node in the tree to view the corresponding component in the source editor.
   The cursor appears at the correct location in the file.

Selecting a Configuration

A configuration contains settings that define the final binary output file that you create within a project. It specifies the type of application to build, the platform on which it is to run, and the tool settings to use when building.

Debug and Release Configurations

When you create a new project, Visual Studio* automatically creates the following configurations:

Configuration	Description
Debug configuration	By default, the debug configuration sets project options to include the debugging information in the debug configuration. It also turns off optimizations. Before you can debug an application, you must build a debug configuration for the project.
Release (Retail) configuration	The release configuration does *not* include the debugging information, and it uses any optimizations that you have chosen.

Use the Visual Studio* **Configuration Manager** to select:

- **Release** or **Debug** configuration for the active solution.
- **Release** or **Debug** configuration for any project within the active solution.
- Target platform for each project.

Multiple configurations allow extending the scope of a project while maintaining a consistent source code base from which to work. The default configuration is the **Debug** configuration. To add, change, or delete a configuration, select **Build** > **Configuration Manager** from the main menu or use the drop-down box on the main menu bar. Only one configuration can be active at one time. Configurations can be used for the whole solution or for specific projects in the solution.

When you build your project, the currently selected configuration is built, as follows:

- If you selected the debug configuration, a directory called `Debug` contains the output files created by the build for the debug version of your project.
- If you selected the release configuration, a directory called `Release` contains the output files created by the build for the release version of your project.

(You can change the output directory using the **General** category in **Project** > **Properties**.)

A configuration has the following characteristics:

- **Configuration type:** specifies the type of Fortran application to build.
- **Build options:** specifies the build options, which include the compiler and linker options.

By default, the **Debug** configuration sets project options to include debug symbol information and turns off optimizations. Before you can debug an application, you must build a debug configuration for the project. Although debug and release configurations usually use the same set of source files, the project setting information usually differs. For example, the default debug configuration supplies full debug information and no optimizations, whereas the default release configuration supplies minimal debug information and full optimizations.

To make configuration changes for your project:

1. Choose an active solution in the **Solution Explorer**.
2. Go to **Build** > **Configuration Manager**.
3. Select a configuration.

Platform Types

The platform type sets options required specifically for the selected platform, such as options that the compiler uses for the source files, the libraries that the linker uses for the platform, defined constants, and so on.

New Configurations

In addition to the default **Debug** and **Release** configurations, you can also define new configurations within your project. These configurations may use the existing source files in your project, the existing project settings, or other characteristics of existing configurations.

See Also

Setting Compiler Options in the Visual Studio* IDE Property Pages
Specifying a Target Platform for Solutions and Projects
Understanding Errors During the Build Process

Specifying a Target Platform

You can specify a target platform for a Microsoft Visual Studio* solution or an individual project within a solution.

1. Select a solution or project in the **Solution Explorer**.
2. Select **Build** > **Configuration Manager**.
   - Use the **Active solution platform** drop-down list to specify the target platform for the whole solution.
   - Use the **Platform** column to specify the target platform for an individual project within a solution.
3. Select **<New...>** on the **Active solution platform** drop-down menu to add a platform to the current list of active solution platforms.
4. Select or type a new platform in the **New Solution Platform** dialog box.
5. In **Copy settings from** choose a platform to use as a template or choose **<Empty>**.
6. If you want the platform to be set for the projects within the solution, select **Create new project platforms**. Click **OK**.
7. Close the **Configuration Manager**.

NOTE You can also change target platforms from the Microsoft Visual Studio* toolbar by selecting a platform from the **Solution Platforms** drop-down selection.

Removing Target Platforms

You can remove a target platform from the **Configuration Manager**:

1. Select **<Edit...>** from the **active solution platform** list of options.
2. In the **Edit Solutions Platforms** dialog that opens, select a platform and click **Remove**.
3. Click **Yes** in the confirmation box.

Specifying Path, Library, and Include Directories

You can specify directories that the Microsoft Visual Studio* project system should search for certain types of files.

To set path, library, and include directories for your Intel® Fortran project environment on a particular machine:

1. Select **Tools** > **Options**.
2. In the left pane, select **Intel Compilers and Tools** > **Visual Fortran** > **Compilers**.
3. In the right pane, specify directories where the Microsoft Visual Studio* project system should look for files:
   - **Executables:** The directories to be searched for executable files. (Works like the PATH environment variable.)
   - **Libraries:** The directories to be searched for libraries. (Works like the LIB environment variable.)
   - **Includes:** The directories to be searched for include files. (Works like the INCLUDE environment variable.) You can use macros like `$(VSInstallDir)` in directory names. For list of supported macros, see Supported Build Macros.
4. Click **OK**.

Use the **Reset** and **Reset All** buttons to restore original installation settings for Executables, Libraries, and Includes fields. **Reset** restores initial settings for the currently selected compiler; **Reset All** restores initial settings for all compilers listed in the **Selected** drop-down box.

> **NOTE** If you specify `devenv` or `useenv` on the command line to start the IDE, the IDE uses the PATH, INCLUDE, and LIB environment variables as defined for that command line when performing a build. It uses these values instead of the values defined in **Tools** > **Options**.
> For more information on the `devenv` command, see the `devenv` description in the Microsoft Visual Studio* documentation.

For more information on environment variables, see Supported Environment Variables.

See Also

Supported Environment Variables

Supported Build Macros

Setting Compiler Options in the Microsoft Visual Studio* IDE Property Pages

To set compilation and related options for the current project:

1. Select the project name in the **Solution Explorer** view.
2. In the **Project** menu, select **Properties**.

The Intel® Fortran Compiler also lets you specify compiler options for individual source files. To do this, select the file name and select **View** > **Property Pages**.

> **NOTE** For convenience, context-sensitive pop-up menus containing commonly used menu options are available by right-clicking on an item (for example, a file name or solution) in the IDE.

Displaying the Options

To display the Fortran compiler option categories, click the **Fortran** folder in the left pane to display the compiler option categories. The following sample screen shows the compiler options in the **General** category in the right pane. The selected option within the **General** category is **Suppress Startup Banner**, with a default value of Yes. The corresponding command line compiler option is `nologo`, as shown in the **Help** text at the bottom of the right pane.

> **NOTE** Option values that are different than the compiler defaults are displayed in bold.

Changing Option Settings

To change the setting for a compiler option:

1. Select a category and then click the desired option. Click the button at the right of an option line to display the available settings or display a dialog box. Available settings may include ***<inherit from project defaults>***, which resets the option value to the compiler default.
2. Select the desired setting and click **OK**.

To change the configuration (such as from **Debug** to **Release**), do one of the following:

- Select a different configuration in the **Configuration:** drop-down box in the upper-left of the window, or
- Click the **Configuration Manager** button in the upper-right of the window and reset the configuration in the dialog box that appears.

Fortran Option Categories

The Intel® Fortran compiler options available from the IDE are grouped in categories. Some options appear in multiple categories. Available options in each category may vary, depending the platform you have selected in the **Platform** box at the top of the dialog box. Options not listed in one of the categories can be typed into the **Command Line** category window.

Command Line Category

The **Command Line** category contains the **Additional Options** field where you can type in an option as you would from the command line. The IDE will process them as part of the **Property Pages** options for the particular project. For instance, you can use the **Command Line** category to type in miscellaneous Intel® Fortran compiler options that are not represented in any of the listed categories. If you type in a compiler option that is represented in one of the listed categories, the option you specify in the **Command Line** category will take precedence and override the equivalent setting in another category.

Supported Build Macros

The Intel® Fortran compiler supports certain build macros for use in the **Property Pages** dialog boxes associated with a project. Use these macros where character strings are accepted. The macro names are not case-sensitive.

The following table lists macros supported by Visual Studio* that are also supported by the Intel® Fortran Compiler.

Macro Name	Format
Configuration name	`$(ConfigurationName)`
Platform name	`$(PlatformName)`
Intermediate directory	`$(IntDir)`
Output directory	`$(OutDir)`
Input directory	`$(InputDir)`
Input path	`$(InputPath)`
Input name	`$(InputName)`
Input filename	`$(InputFileName)`
Input file extension	`$(InputExt)`
Inherit properties	`$(Inherit)`
Do not inherit properties	`$(NoInherit)`
Project directory	`$(ProjectDir)`
Project path	`$(ProjectPath)`
Project name	`$(ProjectName)`
Project filename	`$(ProjectFileName)`
Project file extension	`$(ProjectExt)`

Macro Name	Format
Solution directory	`$(SolutionDir)`
Solution path	`$(SolutionPath)`
Solution name	`$(SolutionName)`
Solution filename	`$(SolutionFileName)`
Solution file extension	`$(SolutionExt)`
Target directory	`$(TargetDir)`
Target path	`$(TargetPath)`
Target name	`$(TargetName)`
Target filename	`$(TargetFileName)`
Target file extension	`$(TargetExt)`
Visual Studio* installation directory	`$(VSInstallDir)`
Visual C++* installation directory	`$(VCInstallDir)`
.NET Framework directory	`$(FrameworkDir)`
.NET Framework version	`$(FrameworkVersion)`
.NET Framework SDK Directory	`$(FrameworkSDKDir)`

The Intel® Fortran compiler also supports the following macros (not supported by Visual Studio*).

Macro Name	Format
Intel® Fortran IDE installation directory	`$(IFIDEInstallDir)`

For additional information on using build macros, see the Microsoft MSDN* online documentation.

Using Manifests

The Intel® Fortran Compiler supports manifests, a Visual Studio* feature. Manifests describe run-time dependencies of a built application. A manifest file can be embedded in the assembly, which is the default behavior, or can be a separate standalone file. You can use the **Manifest Tool** property pages, which are accessed through **Project** > **Properties**, to change project settings that affect the manifest.

NOTE
In earlier releases, manifest files were embedded in the assembly, and were not able to be accessed or changed.

Using Intel® Performance Libraries with Microsoft Visual Studio*

You can use the Intel®Fortran Compiler with the following Intel® Performance Libraries that may be included as a part of the product:

- Intel® Integrated Performance Primitives (Intel® IPP)
- Intel® Threading Building Blocks (Intel® TBB)
- Intel® Math Kernel Library (Intel® MKL)

Use the property pages to specify Intel® Performance Libraries to use with the selected project configuration.

To specify Intel® Performance Libraries, select **Project** > **Properties**. In **Configuration Properties**, select **Intel Performance Libraries**, then do the following:

1. To use **Intel® Integrated Performance Primitives**, change the **Use Intel IPP** settings as follows:
   - **No:** Disable use of Intel® IPP libraries.
   - **Default Linking Method:** Use Intel® IPP libraries that depend on the settings for the **Code Generation** > **Runtime Library** property.
   - **Multi-threaded Static Library:** Use parallel static Intel® IPP libraries.
   - **Single-threaded Static Library:** Use parallel sequential static Intel® IPP libraries.
   - **Multi-threaded DLL:** Use parallel dynamic Intel® IPP libraries.
   - **Single-threaded DLL:** Use parallel sequential static Intel® IPP libraries.
2. To use **Intel® Threading Building Blocks** in your project, change the **Use Intel TBB** settings as follows:
   - **No:** Disable use of Intel® TBB libraries.
   - **Use TBB:** Set to **Yes** to use Intel® Threading Building Blocks.
   - **Instrument for use with Intel® Threading Analysis Tools:** Set to **Yes** to add the preprocessor definition TBB_USE_THREADING_TOOLS to the project property **Preprocessor** > **Preprocessor Definitions**.

   NOTE The TBB_USE_THREADING_TOOLS definition will be added only if the project property **Code Generation** > **Runtime Library** is set to **Multi-threaded DLL (option `MD`)**.

3. To use **Intel® Math Kernel Library** in your project, change the **Use Intel MKL** property settings as follows:
   - **No:** Disable use of Intel® MKL libraries.
   - **Parallel:** Use parallel Intel® MKL libraries.
   - **Sequential:** Use sequential Intel® MKL libraries.
   - **Cluster:** Use cluster Intel® MKL libraries.

   If your target platform is set to **x64**, a final selection appears: **Use ILP64 interfaces**. If selected, the corresponding ilp MKL libraries is added to the linker command line. Additionally, the MKL_ILP64 preprocessor definition is added to the compiler command line. If you do not make this selection, lp MKL libraries are used.

   NOTE Sets of libraries for each option depend on selected target platform and project properties defined by **Code Generation** > **Runtime Library** and **Advanced** > **Calling Convention**.

For more information, see the Intel® Integrated Performance Primitives, Intel® Threading Building Blocks, and Intel® Math Kernel Library documentation.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Using Guided Auto Parallelism in Microsoft Visual Studio*

The Guided Auto Parallelism (GAP) feature helps you locate portions of your serial code that can be parallelized. When you enable analysis using GAP, the compiler guides you to places in your code where you can increase efficiency through automatic parallelization and vectorization.

Running Analysis on a Project

You can start analysis from the Microsoft Visual Studio* IDE in several ways:

- From the **Tools** menu: Select **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis...**

  Starting analysis in this way results in a one-time run for the current project. The default values are taken from **Tools** > **Options** unless you have chosen to override them in the dialog box.
- From the Diagnostics property page: Use the **Guided Auto Parallelism Analysis** property.

  Specifically, choose **Project** > **Properties** > **Fortran** > **Diagnostics** and enable analysis using the **Guided Auto Parallelism Analysis** property. Enabling analysis in the property page allows you to run an analysis as part of a normal project **Build** request in Microsoft Visual Studio*. In this mode, GAP-related settings in **Tools** > **Options** are ignored, in favor of other GAP-related settings available in the property page.
- From the context menu: Right-click and select **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis...**.

  This is equivalent to using the **Guided Auto Parallelism** > **Run Analysis** option on the **Tools** menu.

To receive advice for auto parallelization, be sure that certain property page settings are correct. Select **Project** > **Properties** > **Fortran** > **Optimization** and set **Parallelization** to **Yes** to enable auto-parallelization optimization. You may also need to set the **Optimization** level at option `O2` or higher.

GAP Scenarios

To illustrate how the various GAP settings work together, consider the following scenarios:

Scenario	Result
The GAP analysis setting in the property pages is set to Enabled.	Analysis always occurs for the project, whenever a regular project build occurs. Other analysis settings specified in the property pages are used. Analysis setting in **Tools** > **Options** are ignored.
The Gap analysis setting in the property pages is set to Disabled, and GAP is run from the **Tools** menu.	Analysis occurs for this one run. The default values for this analysis are taken from **Tools** > **Options** and can be overridden in the dialog box. Options specified in the property pages are also used, but will be overridden by any specified analysis option.

Scenario	Result
The GAP analysis setting in the property pages is set to Disabled, and GAP options are set in **Tools** > **Options**.	No analysis occurs, unless analysis is explicitly run from the **Tools** menu.

Running Analysis on a File or within a File

Right-click on **Guided Auto Parallelism** context menu item to run analysis on the following:

- **Single file:** Select a file and right-click.
- **Function (routine):**Right-click within the function scope.
- **Range of lines:** Select one or more lines for analysis and right-click.

See Also

Options: Guided Auto Parallelism dialog box

Guided Auto Parallelism

Using Guided Auto Parallelism

Using Code Coverage in Microsoft Visual Studio*

The code coverage tool provides the ability to determine how much application code is executed when a specific workload is applied to the application. The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate a report in HTML-format and export data in both text- and XML-formatted files. The reports can be further customized to show color-coded, annotated source-code listings that distinguish between used and unused code.

To start code coverage:

1. Select **Tools** > **Intel Compiler** > **Code Coverage...**
2. Specify settings for the various phases.
3. Click **Run**.

The **Output** window shows the results of the coverage and a general summary of information from the code coverage.

See Also

Code Coverage dialog box

Code Coverage Settings dialog box

Code Coverage Tool

Using Profile Guided Optimization in Microsoft Visual Studio*

Profile Guided Optimization (PGO) improves application performance by reorganizing code layouts to reduce instruction-cache problems, shrinking code size, and reducing branch misprediction. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

To start PGO:

1. Choose **Tools** > **Intel Compiler** > **Profile Guided Optimization...**
2. Specify settings for the various phases.
3. Click **Run**.

The **Output** windows show the results of the optimization with a link to the composite log.

See Also

Profile Guided Optimization dialog box

Options: Profile Guided Optimization dialog box

Profile Guided Optimization

Using Source Editor Enhancements in Microsoft Visual Studio*

A number of Fortran source editor enhancements are available in Microsoft Visual Studio*.

Modules and Procedures Navigation Bar

A two-part navigation bar, located above the source editor pane, lets you navigate to a specific module (left part) and procedure (right part). To enable the navigation bar, choose **Navigation Bar** in **Tools** > **Options** > **Text Editor** > **Fortran** > **General**.

Source Editor Pane

Smart indenting: Smart indenting automatically indents block constructs (such as `IF` and `DO`) and left justifies the corresponding end statement. To enable smart indenting, select it in **Tools** > **Options** > **Text Editor** > **Fortran** > **Tabs**.

Code snippet insertion: Code snippet insertion lets you insert a prototype construct (such as `DO`, `WHILE`, or `MODULE`) from a list. Use the right-click context menu **Insert Snippet... option** to display the list and insert a snippet.

Delimiter matching: Delimiter (brace) matching lets you jump to a matching statement in a block construct (`IF...THEN...END IF`, `DO...END DO`). Use Ctrl] to jump. To enable delimiter matching, use **Automatic delimiter highlighting** in **Tools** > **Options** > **Text Editor** > **General**.

Call/Callers graph: Call/Caller information can be collected and shown visually in a graph that indicates the call stacks that lead to a unit of code. To enable Call/Callers graph information, change **Collect Call/Callers graph information** to **True** in **Tools** > **Options** > **Text Editor** > **Fortran** > **Advanced** > **Browsing/Navigation**.

Fortran Editor Options Page

Use **Tools** > **Options** > **Text Editor** > **Fortran** > **Advanced** to view the Fortran Advanced Options page.

Browsing/Navigation Section

Collect Object Browser information: Choose this option to enable the display of procedures in your project in a hierarchical tree. Once enabled, you can use **View** > **Object Browser** to display your procedures.

NOTE If the procedures in your project do not show when browsing in **My Solution**, you can right-click the message (No information. Try browsing a different component set.) and select **View Containers** as a possible solution.

Disable Database: Choose this option to disable creation of the code browsing database. This may help increase performance on slow machines. If you disable the database, all features that rely on code browsing information will not work.

Enable Find all References: Choose this option to enable display of the location(s) in your code where a symbol is referenced. When this option is enabled, you can use the right-click context menu **Find All References** option to display a list of references to the selected symbol. Double-click on a reference to find that reference.

Enable Go To Definition: Choose this option to enable quick navigation to an object definition. When this option is enabled, you can use the right-click context menu **Go to Definition** option to locate where the selected object was declared, opening the associated source file if required. (If you have also enabled **Scan system includes**, any objects declared in system modules such as IFWINTY cause the associated source for that module to be opened.)

Scan system includes: Choose this option to scan system include files. This option is used with one or more of the following options: **Collect Object Browser Information**, **Enable Find All References**, **Enable Go To Definition**.

Intrinsics Section

Enable Intrinsic Parameter Info: Choose this option to enable the display of intrinsic function and subroutine parameter information. When this option is enabled, you can type a name of an intrinsic procedure, followed by an open parenthesis, and information about the procedure and its arguments appears.

Enable Intrinsic Quick Info: Choose this option to enable the display of additional information when the mouse pointer is moved over an intrinsic function or subroutine name.

Miscellaneous Section

Enumerate Comment Tasks: Choose this option to enable the display of a list of tasks consisting of source files containing comments. Comments take the form of the ! character, followed by a token such as TO DO. Valid tokens are those listed in **Tools** > **Options** > **Environment** > **Task List**. When this option is enabled, you can select **Comments** from the task list using **View** > **Other Windows Task List**. Double-click on a comment in the list to jump to its location.

Highlight Matching Tokens: Choose this option to allow identifier highlighting and block delimiter matching. When enabled, this option highlights all references to the identifier under the cursor.

Outlining Section

Enable Outlining: Choose this option to allow the collapsing of whole program units. When this option is enabled, you can click the minus (-) or plus (+) symbols for `PROGRAM`, `SUBROUTINE`, `FUNCTION`, `MODULE`, and `BLOCK DATA` statements.

Outline Statement Blocks: Choose this option to allow collapsing of block constructs such as `IF` and `DO`. You must also choose **Enable Outlining**.

Creating the Executable Program

When you are ready to create an executable image of your application, use the options on the **Build** menu. You can:

- Compile a file without linking.
- Build a project or solution.
- Rebuild a project or solution.
- Batch build several configurations of a project.

- Clean a project or solution (which deletes all files created during the Build).
- Select the active solution and configuration.
- Edit the project configuration.

When you have completed your project definition, you can build the executable program.

When you select **Build <*projectname*>** from the **Build** menu (or **Build** toolbar), the integrated development environment (IDE) automatically updates dependencies, compiles and links all files in your project. When you build a project, the IDE processes only the files in the project that have changed since the last build and those files dependent on the changed files. The following example illustrates this.

> **NOTE** To define the build order of projects, right-click on the solution and choose **Properties** > **Project Dependencies**.

Example: Assume you have multiple projects (A, B, and C) in a solution with the following defined dependencies:

- A depends on B
- B depends on C

If you build A, the build process verifies that B is up-to-date. During verification of B, C is also verified that it is likewise up-to-date. When either, or both, are determined to be out of date, the appropriate build operations will occur to update them. When C and B produce `.lib` or `.dll` output, the output of C is linked into B and the output of B is linked into A.

The **Rebuild <*project name*>** option forces a new compilation of all source files listed for the project.

You either can choose to build a single project, the current project, or multiple project configurations (using the **Batch Build...** option) in one operation. You can also choose to build the entire solution.

You can execute your program from the IDE using **Debug** > **Start Without Debugging** (Ctrl and F5) or **Debug** > **Start** (F5). You can also execute your program from the command line prompt.

Compiling Files in a Project

You can select and compile individual files in any project in your solution. To do this, select the file in the **Solution Explorer** view. Then, do one of the following:

- Select **Compile** from the **Build** menu (or **Build** toolbar).
- Right-click to display the pop-up menu and select **Compile**.

You can also use **Compile** from the **Build** menu (or **Build** toolbar) options when the source window is active and has input focus.

Encountering Errors during Compilation

When the compiler encounters an error in a file, compilation stops and the error is reported. You can change this default behavior and allow compilation to continue despite an error in the current file. When you do this, an error in the current file will cause the compiler to begin compiling the next file.

To enable **Continue on errors** behavior:

1. In **Tools** > **Options** > **Intel Compilers and Tools** > **Visual Fortran**, select the **General** category.
2. Check **Continue on errors** and click **OK**.

To set the maximum number of errors to encounter before compilation stops, choose **Configuration Properties** > **Fortran** > **Diagnostics** > **Error Limit**.

Performing Parallel Project Builds

Visual Studio* provides a parallel project build feature, allowing you to build multiple projects within a solution simultaneously, using separate threads. The Visual Studio* IDE initially sets the number of parallel project builds to equal the number of CPUs. To change this setting, do the following:

1. Choose **Tools** > **Options** > **Projects and Solutions**.
2. In the **Build and Run** property page, change the number in maximum number of parallel project builds and click **OK**.

For more information on using this feature, see the Microsoft MSDN* documentation.

Converting and Copying Projects

Converting Projects

In general, you can open projects created by older versions of Intel® Visual Fortran and use them directly. If the projects were created in older versions of Microsoft Visual Studio*, the solution file is converted first and then any non-Fortran projects it contains. Projects created in newer versions of Intel® Visual Fortran might not be usable in older versions.

Projects created in Compaq* Visual Fortran 6.0 or later can usually be converted to Intel® Visual Fortran as follows:

1. Open the Microsoft Visual Studio* 6 workspace file (.dsw) in a newer version of Microsoft Visual Studio*. The project is converted to the new format.
2. Right click on the solution and select `Extract Compaq Visual Fortran Project Items`. This option is available only if your installation of Microsoft Visual Studio* includes Microsoft Visual C++.

Some general conversion principles apply:

- It is good practice to make a backup copy of the project before starting conversions.
- Intel® Fortran projects are created and built in a particular version of Microsoft Visual Studio*. If you open the project in a later version, you will be prompted to convert the solution. Once converted, a solution cannot be used in its previous environment.
- Compaq* Visual Fortran 6.x projects can be converted to Intel® Fortran projects in Microsoft Visual Studio* 2013, 2015, or 2017 environments. Fortran-only projects are simpler to convert.
- Project conversion support is provided for Compaq* Visual Fortran Version 6.x only. Compaq* Visual Fortran projects created with earlier versions may not convert correctly.
- Fortran source files, resource files, and `MIDL` files lose any custom build step information when converted from Compaq* Visual Fortran to Intel® Fortran. For other file types, custom build steps are propagated during the project's conversion.
- Conversion of Fortran and C/C++ mixed language projects results in the creation of two separate projects (a Fortran project and a C/C++ project) in a single solution.
- Intel® Fortran projects that are created with a point release (for instance, 18.x) are typically backward compatible to the first release of that number (in this case, 18.0). Projects are not backward-compatible between major release numbers.

See Also

Copying Projects

Copying Projects

You need to follow certain procedures to move a project's location if you copy a project to:

- Another disk or directory location on the same system.
- Another system where the Intel® Fortran Compiler is installed.

If you upgrade your operating system version on your current system, you should delete the `*.SUO` and `*.NCB` files in each main project directory before you open solutions with the new operating system.

It is good practice to clean a solution before moving and copying project files. To do this, select **Clean** in the **Build** menu.

Copy an Existing Intel® Fortran Project to Another Disk or System

1. Copy all project files to the new location. You do not need to copy the subdirectories created for each configuration. Keep the directory hierarchy intact by copying the entire project tree to the new computer. For example, if a project resides in the folder `\MyProjects\Projapp` on one computer, you can copy the contents of that directory, and all subdirectories, to the `\MyProjects\Projapp` directory on another computer.
2. Delete the following files from the main directory at the new location. These files are disk- and computer-specific and should not be retained:
   - `*.SUO` files
   - `*.NCB` files (if present)
3. If you copied the subdirectories associated with each configuration (for example, **Debug** and **Release**), delete the contents of subdirectories at the new location. The files contained in these subdirectories are disk- and computer-specific files and should not be retained. For example, Intel® Fortran module (`.MOD`) files contained in these subdirectories should be recreated by the compiler, especially if a newer version of Intel® Fortran has been installed.

   NOTE The internal structure of module files can change between Intel® Fortran releases.

   If you copied the project files to the same system or a system running the same platform and major Intel® Fortran version, do the following steps to remove most or all of the files in the configuration subdirectory:

   1. Open the appropriate solution. In the **File** menu, either select **Open Solution** or select **Recent Solutions**. If you use **Open Solution**, select the appropriate `.SLN` file.
   2. Select **Set Active Configuration** in the **Build** menu and select a configuration.
   3. Select **Clean** in the **Build** menu.
   4. Repeat the previous two steps for other configurations whose subdirectories have been copied.
4. If possible, after copying a project, verify that you can open the project at its new location using the same Fortran version that it was created in. This ensures that the project has been moved successfully and minimizes the chance of conversion problems. If you open the project with a later version of Fortran, the project will be converted and you will not be able to convert the project back. For this reason, making an archive copy of the project files before you start is recommended.
5. View the existing configurations. To view the existing configurations associated with the project, open the solution and view available configurations using the drop-down box at the top of the screen.
6. Check and reset project options.

   Because not all settings are transportable across different disks and systems, you should verify your project settings on the new platform. To verify your project settings:

   a. From the **Project** menu, choose **Properties**. The **Project Property Pages** dialog box appears.
   b. Configure settings as desired. Pay special attention to the following items:
      - **General:** Review the directories for intermediate and output files. If you moved the project to a different system, be aware that any absolute directory paths (such as `C:\TEMP` or `\Myproj\TEMP`) will most likely need to be changed. Instead, use relative path directory names (without a leading back slash), such as `Debug`
      - **Custom Build Step:** Review for any custom commands that might change between platforms.
      - **Pre-build**, **Pre-link**, and **Post-build Steps** in **Build Events:** Review for any custom commands that may have changed.
7. Check your source code for directory paths referenced in `INCLUDE` or similar statements. Microsoft Visual Studio* provides a multi-file search capability called **Find in Files**, available from the **Edit menu**.

About Fortran Project Types

Understanding Project Types

When you create a project in Visual Studio*, you must choose a *project type*. You need to create a project for each binary executable file to be created. For example, the main Fortran program and a Fortran dynamic-link library (DLL) would each reside in the same solution as separate projects.

The project type specifies what to generate and determines some of the options that the visual development environment sets by default for the project. It determines, for instance, the options that the compiler uses to compile the source files, the static libraries that the linker uses to build the project, and the default locations for output files.

When you select a project type, an Application wizard (AppWizard) is launched, which guides you through project set-up. The AppWizard supplies default settings for both the **Release** and **Debug** Configurations of the project. For more information about configurations, see Understanding Solutions Projects and Configurations and Selecting a Configuration.

The following table lists the available Intel® Fortran Compiler project types. The first four projects listed are main project types, requiring main programs. The last two are library projects, without main programs.

Project Type	Key Features
Using Fortran Console Application Projects (`.EXE`)	Single window main projects without graphics (resembles character-cell applications). Requires no special programming expertise.
Using Fortran Standard Graphics Application Projects (`.EXE`)	Single window main projects with graphics. The programming complexity is simple to moderate, depending on the graphics and user interaction used.
Using Fortran QuickWin Application Projects (`.EXE`)	Multiple window main projects with graphics. The programming complexity is simple to moderate, depending on the graphics and user interaction used.
Using Fortran Windowing Application Projects (`.EXE`)	Multiple window main projects with full graphical interface and access to all Windows* API routines. Requires advanced programming expertise and knowledge of the Calling Windows API Routines.
Using Fortran Static Library Projects (`.LIB`)	Library routines to link into `.EXE` files.
Using Fortran Dynamic-Link Library Projects (`.DLL`)	Library routines to associate during execution.
Using the Fortran COM Server (`.DLL`)	Fortran in-process COM server

If you need to use the command line to build your project, you can:

- Use the command line compiler options to specify the project type (see Specifying Project Types with ifort Command Options).
- Create the application from the command line (see Invoking the Intel® Fortran Compiler).

See Also

Understanding Solutions Projects and Configurations

Selecting a Configuration

Additional Documentation: Creating Fortran Applications that Use Windows* OS Features

Specifying Project Types with ifort Command Options

This section provides the `ifort` command-line options that correspond to Visual Studio* project types.

Creating Main Project Types

The first four projects described below are main project types, requiring main programs. You can create any of the following project types with the `ifort` command:

- To create Console Application Projects, you do not need to specify any options. (If you link separately, specify the `link` option `/subsystem:console`.) This is the default project type created.
- To create Standard Graphics Application Projects, specify the `libs` option with keyword `qwins` (also sets certain linker options).
- To create QuickWin Application Projects, specify the `libs` option with keyword `qwin` (also sets certain linker options).
- To create Windowing Application Projects, specify the `winapp` option (also sets certain linker options).

Creating Library Project Types

The following project types are library projects, without main programs. You can create them with the `ifort` command:

- To create Dynamic-Link Library Projects, specify the `dll` option (which sets the `libs` option with keyword `dll`).
- To create Static Library Projects:
  - If your application will not call any QuickWin or Standard Graphics routines, specify the `libs` option with keyword `static` and `c` options to create the object files.
  - If your application will call QuickWin routines, specify the `libs` option with keyword `qwin` and `c` options to create the object files.
  - If your application will call Standard Graphics routines, specify the `libs` option with keyword `qwins` and `c` options to create the object files.
  - Use the `LIB` command to create the library.

See Also

Understanding Project Types
Using Fortran Console Application Projects
Using Fortran Standard Graphics Application Projects
Using Fortran QuickWin Application Projects
Using Fortran Windowing Application Projects
Using Fortran Dynamic-Link Library Projects
Using Fortran Static Library Projects

Using Fortran Console Application Projects

A Fortran Console application (`.EXE`) is a character-based Intel® Fortran program that does not require screen graphics output.

Fortran Console projects operate in a single window and let you interact with your program through normal read and write commands. Console applications are better suited to problems that require pure numerical processing rather than graphical output or a graphical user interface. This type of application is also more transportable to other platforms than the other types of application.

Fortran Console applications can be faster than Fortran Standard Graphics or Fortran QuickWin graphics applications, because of the resources required to display graphical output (see Using the Console).

Any graphics routine that your program calls will produce no output, but will return error codes. A program will not automatically exit if such an error occurs, so your code should be written to handle this condition.

With a Fortran Console project, you cannot use the QuickWin functions. However, you can use single- or multi-threaded static libraries, DLLs, and dialog boxes.

As with all Windows* command consoles, you can toggle between console viewing modes by pressing ALT and ENTER.

To create a console application from the IDE:

1. Select the **Console Application** project type.
2. Select from two templates: **Empty project** or **Main program code**, which includes sample code.

See Also

Using the Console

Using Fortran Standard Graphics Application Projects

A Fortran standard graphics application (`.EXE`) is an Intel® Fortran QuickWin program with graphics that runs in a single QuickWin window. A standard graphics (QuickWin single window, sometimes called single document) application looks similar to an MS-DOS* program when manipulating the graphics hardware directly, without Windows*.

A Fortran standard graphics application allows graphics output (such as drawing lines and basic shapes) and other screen functions, such as clearing the screen. Standard Graphics is a subset of QuickWin. You can use all of the QuickWin graphics functions in these projects. You can use dialog boxes with these and all other project types.

You can select displayed text either as a bitmap or as text. Windows* provides APIs for loading and unloading bitmap files. Standard graphics applications should be written as multithreaded applications.

Fortran standard graphics (QuickWin single window) applications are normally presented in full-screen mode. The single window can be either full-screen or have window borders and controls available. You can change between these two modes by using ALT and ENTER.

If the resolution selected matches the screen size, the application covers the entire screen; otherwise, scroll bars are present to resize the window. You cannot open additional windows in a standard graphics application. Standard graphics applications have neither a menu bar at the top of the window, nor a status bar at the bottom.

Fortran standard graphics applications are appropriate for problems that:

- Require numerical processing and some graphics.
- Do not require a sophisticated user interface.

To create a standard graphics application using the Visual Studio* IDE:

1. Select the **QuickWin Application** project type.
2. Select the **Standard Graphics Application** template.

When you select the Fortran standard graphics project type, the integrated development environment includes the QuickWin library automatically, which lets you use the graphics functions. When building from the command line, you must specify the `libs` option with keyword `qwins`. You cannot use the run-time functions meant for multiple-window projects if you are building a standard graphics project. You cannot make a Standard Graphics application a DLL.

See Also

Additional Documentation: Creating Fortran Applications that Use Windows* Features

Using Fortran QuickWin Application Projects

Fortran QuickWin graphics applications (`.EXE`) are multi-threaded and are more versatile than standard graphics (QuickWin single window) applications because you can open multiple windows while your project is executing. This multiple window capability is also referred to as multiple-document interface or MDI. Multiple windows can be used in a variety of ways. For example, you might want to generate several graphic plots and be able to switch between them while also having a window for controlling the execution of your program. These windows can be full screen or reduced in size and can be placed in various parts of the screen.

QuickWin library routines let you build applications with a simplified version of the Windows* interface using Intel® Fortran. The QuickWin library provides a rich set of Windows* features, but it does not include the complete Windows* Applications Programming Interface (API). If you need additional capabilities, you must set up a Windows* application to call the Windows* API directly rather than using QuickWin to build your program.

Applications that use a multiple-document interface (MDI) have a menu bar at the top of the window and a status bar at the bottom. The QuickWin library provides a default set of menus and menu items that you can customize with the QuickWin APIs. An application that uses MDI creates many "child" windows within an outer application window. The user area in an MDI application is a child window that appears in the space between the menu bar and status bar of the application window. Your application can have more than one child window open at a time.

Fortran QuickWin applications can use the `IFLOGM` module to access functions to control dialog boxes. These functions allow you to display, initialize, and communicate with special dialog boxes in your application. They are a subset of Windows* API functions, which Windows* applications can call directly.

To create a QuickWin application in Visual Studio*:

1. Select the **QuickWin Application** project type.
2. Select the **QuickWin Application** template in the right pane.

When you select the Fortran QuickWin project type, the IDE includes the QuickWin library automatically, which lets you use the graphics functions.

When building from the command line, you must specify the `libs` compiler option with keyword `qwin` to indicate a QuickWin application.

A QuickWin application covers the entire screen if the resolution selected matches the screen size; otherwise, the window will contain scroll bars.

You cannot make a Fortran QuickWin application a DLL.

See Also

Additional Documentation: Creating Fortran Applications that Use Windows* Features

Using Fortran Windowing Application Projects

Fortran Windowing applications (`.EXE`) are main programs that you create when you choose the Fortran Windowing Application project type. This type of project lets you call the Windows* APIs directly from Intel® Fortran. This provides full access to the Windows* APIs, giving you a larger (and different) set of functions to work with than QuickWin.

Although you can call some of the Windows* APIs from the other project types, Fortran Windowing applications allow you to use the full set of API routines and use certain system features not available for the other project types.

The `IFWIN` module contains interfaces to the most common Windows APIs. If you include the `USE IFWIN` statement in your program, the most common Windows* API Routines are available to you. The `IFWIN` module gives you access to a full range of routines including window management, graphic device interface, system services, multimedia, and remote procedure calls.

Window management routines give your application the means to create and manage a user interface. You can create windows to display output or prompt for input. Graphics Device Interface (GDI) functions provide ways for you to generate graphical output for displays, printers, and other devices. Windows* system functions allow you to manage and monitor resources such as memory, access to files, directories, and I/O devices. System service functions provide features that your application can use to handle special conditions such as errors, event logging, and exception handling.

Using multimedia functions, your application can create documents and presentations that incorporate music, sound effects, and video clips as well as text and graphics. Multimedia functions provide services for audio, video, file I/O, media control, joystick, and timers.

Remote Procedure Calls (RPC) gives you the means to carry out distributed computing, letting applications tap the resources of computers on a network. A distributed application runs as a process in one address space and makes procedure calls that execute in an address space on another computer. You can create distributed applications using RPC, each consisting of a client that presents information to the user and a server that stores, retrieves, and manipulates data as well as handling computing tasks. Shared databases and remote file servers are examples of distributed applications.

See Also

Calling Windows* API Routines

Additional Documentation: Creating Fortran Applications that Use Windows* OS Features

Using Fortran Static Library Projects

Fortran static libraries (`.LIB`) are blocks of code compiled and kept separate from the main part of your program. The Fortran static library is one of the Fortran project types.

To create a static library from the integrated development environment (IDE), select the **Static Library** project type. To create a static library from the command line, use the `c` option to suppress linking and use the `LIB` command.

> **NOTE**
> When compiling a static library from the command line, include the `c` option to suppress linking. Without this option, the compiler generates an error because the library does not contain a main program.

When you create a static library, you are asked to specify whether you want to prevent the insertion of link directives for default libraries. By default, this checkbox is selected, which means insertion of link directives is prevented. Select this option if you plan to use this static library with other Fortran projects. The option prevents the static library from specifying a version of the Fortran run-time library. When the static library is linked with another Fortran project, the Fortran run-time library choice in the other Fortran project is used for the static library as well.

You may decide against selecting this option if you plan to use this static library with C/C++ projects. If you do select it, you need to explicitly name the Fortran run-time library to use in the Linker **Additional Dependencies** property. You can change your selection after creating the project using the **Fortran Disable Default Library Search Rules** property.

A static library is a collection of source and object code defined in the **Solution Explorer** window. The source code is compiled when you build the project. The object code is assembled into a `.LIB` file without going through a linking process. The name of the project is used as the name of the library file by default. Static libraries offer important advantages in organizing large programs and in sharing routines between several programs. These libraries contain only subprograms, not main programs. A static library file has a `.LIB` extension and contains object code.

When you associate a static library with a program, any necessary routines are linked from the library into your executable program when it is built. Static libraries are usually kept in their own directories. If you use a static library, only those routines actually needed by the program are incorporated into the executable image (`.EXE`). This means that your executable image will be smaller than if you included all the routines in the library in your executable image. The Linker determines which routines to include.

Because applications built with a static library all contain the same version of the routines in the library, you can use static libraries to help keep applications current. When you revise the routines in a static library, you can easily update all the applications that use it by relinking the applications.

If you have a library of substantial size, you should maintain it in a dedicated directory. Projects using the library access it during linking.

When you link a project that uses the library, selected object code from the library is linked into that project's executable code to satisfy calls to external procedures. Unnecessary object files are not included.

To debug a static library, you must use a main program that calls the library routines. Both the main program and the static library should have been compiled using the debug option. After compiling and linking is completed, open the **Debug** menu and choose **Go** to reach breakpoints, or use the step controls on the **Debug** toolbar.

Using Static Libraries

To add static libraries to a main project in the IDE, use the **Add Existing Item...** option in the **Project** menu. You can enter the path and library name with a `.LIB` extension in the dialog box that appears. If you are using a makefile, you must add the library by editing the makefile for the main project. If you are building your project from the command line, add the library name with a `.LIB` extension and include the path specification if necessary.

Using Fortran Dynamic-Link Library Projects

A dynamic-link library (`.DLL`) is a source-code library that is compiled and linked to a unit independently of the applications that use it. A DLL shares its code and data address space with a calling application. A DLL contains only subprograms, not main programs.

A DLL offers the organizational advantages of a static library, but with the advantage of a smaller executable file at the expense of a slightly more complex interface. Object code from a DLL is not included in your program's executable file, but is associated as needed in a dynamic manner while the program is executing. More than one program can access a DLL at a time.

When routines in a DLL are called, the routines are loaded into memory at run-time, as they are needed. This is most useful when several applications use a common group of routines. By storing these common routines in a DLL, you reduce the size of each application that calls the DLL. In addition, you can update the routines in the DLL without having to rebuild any of the applications that call the DLL.

With Intel® Fortran, you can use DLLs in two ways:

1. You can build a DLL with your own routines. In Visual Studio*, select **Dynamic-link Library** as your project type. From the command line, use the `DLL` option with the `ifort` command.
2. You can build applications with the run-time library stored in a separate DLL instead of in the main application file. In the integrated development environment, open a solution and do the following:

   - From the **Project** menu, select **Properties** to display the project properties dialog box.
   - Click the **Fortran** folder.
   - Select the **Libraries** category.
   - In the **Runtime Library** option, select an option ending with "DLL."

   From the command line, use the `libs` compiler option with keyword `dll` to build applications with the run-time library stored in a separate DLL.

See Also

Additional Documentation: Creating Fortran Applications that Use Windows* OS Features

Using the Console

On Windows* OS, a console window allows input and output of characters (not graphics).

For example, data written (explicitly or implicitly) by Fortran `WRITE` (or other) statements to Fortran logical unit 6 display characters on a console window. Similarly, data read by Fortran `READ` (or other) statements to unit 5 accept keyboard character input.

The console consists of two components:

- The actual console window that shows the characters on the screen.
- The console buffer that contains the characters to be displayed.

If the console screen buffer is larger than the console window, scroll bars are automatically provided. The size of the console screen buffer must be larger (or equal to) the size of the console window. If the buffer is smaller than the size of the console window, an error occurs. For applications that need to display more than a few hundred lines of text, the ability to scroll quickly through the text is important.

Fortran Console applications automatically provide a console. Fortran QuickWin (and Fortran Standard Graphics) applications do not provide a console, but display output and accept input from Fortran statements by using the program window.

The following Fortran Project Types provide an application console window:

Project Type	Description of Console Provided
Fortran Console	Provides a console window intended to be used for character-cell applications that use text only. When running a Fortran Console application from the command prompt, the existing console environment is used. When you run the application from Windows* or Developer Studio* (by selecting **Start Without Debugging** in the **Debug** menu), a new console environment is created. Basic console use is described in Code Samples of Console Use.
Fortran QuickWin or Fortran Standard Graphics	Does not provide a console, but output to unit 6 and input to unit 5 are directed to the application program window, which can handle both text and graphics. Because the program window must handle both text and graphics, it is not as efficient as the console for just text-only use. A Fortran QuickWin or Fortran Standard Graphics program window (or child window) provides a console-like window. See Console Use for Fortran QuickWin and Fortran Standard Graphics Applications.
Fortran Windows	Does not provide a console window, but the user can create a console by using Windows* API routines. See Console Use for Fortran Windows* Applications and Fortran DLL Applications.
Fortran DLL	Does not provide a console window, but the user can create a console by using Win32 routines. See Console Use for Fortran Windows* Applications and Fortran DLL Applications.
Fortran Static Library	Depends upon the project type of the main application that references the object code in the library (see above project types).

In addition to the Windows* API routines mentioned below, there are other routines related to console use described in the Microsoft Platform SDK* documentation.

Console Use for Fortran QuickWin and Fortran Standard Graphics Applications

For a Fortran QuickWin or Fortran Standard Graphics application, because the default program window handles both graphics and text, the use of a QuickWin window may not be very efficient:

- QuickWin windows use lots of memory and therefore have size limitations.
- They can be slow to scroll.

Although you can access the console window using `WRITE` and `READ` (or other) statements, applications that require display of substantial lines of text, consider creating a DLL that creates a separate console window for efficiency. The DLL application needs to call Windows* API routines to allocate the console, display text, accept keyed input, and free the console resources.

Basic use of a console is described in Code Samples of Console Use.

Console Use for Fortran Windows Applications and Fortran DLL Applications

With a Fortran Windows* or Fortran DLL application, attempting to write to the console using `WRITE` and `READ` (or other) statements before a console is created results in a run-time error (such as error performing `WRITE`).

A console created by a Fortran DLL is distinct from any application console window associated with the main application. A Fortran DLL application has neither a console nor an application window created for it, so it must create (allocate) its own console using Windows* API routines. When used with a Fortran QuickWin or Fortran Standard Graphics application main program, the Fortran DLL can provide its main application with a very efficient console window for text-only use.

Like a Fortran DLL application, a Fortran Windows application has neither a console nor an application window created for it, so it must create its own console using Windows* API routines. After allocating a console in a Fortran DLL, the handle identifier returned by the `GetStdHandle` Windows* API routine refers to the actual console the DLL creates.

When the Fortran Windows application does create a console window, it is very efficient for text-only use. The handle identifier returned by the `GetStdHandle` Calling Windows* API Routines refers to the actual console the Fortran Windows application creates.

For information about creating a console, see Allocating and Deallocating a Console below.

Code Samples for Console Use

The following sections shows sample code for using a console:

- Allocating and Deallocating a Console for Fortran Windows* and DLL Applications.
- Extending Size of the Console Window and Console Buffer for console use in any project type.
- Writing and Reading Characters at a Cursor Position for console use in any project type.

Allocating and Deallocating a Console

To create a console, you use the `AllocConsole` routine. When you are done with the console, free its resources with a `FreeConsole` routine. For example, the following code allocates the console, enlarges the buffer size, writes to the screen, waits for any key to be pressed, and deallocates the console:

```
! The following USE statement provides Fortran interfaces to Windows routines
    USE IFWIN
! Begin data declarations
    integer lines,length
    logical status
    integer fhandle
    Type(T_COORD) wpos
! Set buffer size variables
    length = 80
     lines  = 90
! Begin executable code
!   Allocate a console
    status = AllocConsole() ! get a console window of the currently set size
    handle = GetStdHandle(STD_OUTPUT_HANDLE)
    wpos.x = length   ! must be >= currently set console window line length
    wpos.y = lines    ! must be >= currently set console window number of lines
    ! Set a console buffer bigger than the console window. This provides
    ! scroll bars on the console window to scroll through the console buffer
    status = SetConsoleScreenBufferSize(fhandle, wpos)
     ! Write to the screen as needed. Add a READ to pause before deallocation
     write (*,*) "This is the console output! It might display instructions or data "
```

```
     write (*,*) " "
     write (*,*) "Press any key when done viewing "
     read (*,*)
!    Deallocate the console to free its resources.
     status = FreeConsole()
```

Calling Windows* API routines is described in Calling Windows* API Routines.

If you are using a DLL, your DLL code will need to create subprograms and export their symbols to the main program.

Basic use of a console is described in Extending Size of the Console Window and Console Buffer and Writing and Reading Characters at a Cursor Position.

Extending the Size of the Console Window and Console Buffer

When you execute a Fortran Console application, the console is already allocated. You can specify the size of the console window, size of the console buffer, and the location of the cursor. If needed, you can extend the size of the console buffer and console window by using the following Windows* API routines:

1. You first need to obtain the handle of the console window using the `GetStdHandle` routine. For example:

```
! USE statements to include routine interfaces
use ifqwin
use ifport
use ifcore
use ifwin
! Data declarations
integer fhandle
logical lstat
! Executable code
fhandle = GetStdHandle(STD_OUTPUT_HANDLE)
! ...
```

2. If needed, you can obtain the size of the:
   - Console window by using the `GetConsoleWindowInfo` routine.
   - Console buffer by using the `GetConsoleScreenBufferInfo` routine.

   For example:

```
! USE statements to include routine interfaces
use ifqwin
use ifport
use ifcore
use ifwin
! Data declarations
integer fhandle
logical lstat
Type(T_CONSOLE_SCREEN_BUFFER_INFO) conbuf
type (T_COORD)         dwSize
type (T_SMALL_RECT)    srWindow
fhandle = GetStdHandle(STD_OUTPUT_HANDLE)
! Executable code to get console buffer size
lstat = GetConsoleScreenBufferInfo(fhandle, conbuf)
write (*,*) " "
write (*,*) "Window coordinates= ", conbuf.srWindow
write (*,*) "Buffer size= ", conbuf.dwSize
! ...
```

3. To set the size of the console window and buffer, use the `SetConsoleWindowInfo` and `SetConsoleScreenBufferSize` routines with the **fhandle** value returned previously:

```
!  USE statements to include routine interfaces
use ifqwin
use ifport
use ifcore
use ifwin
! Data declarations
integer nlines, ncols
logical lstat
Type(T_COORD) wpos
Type(T_SMALL_RECT) sr
Type(T_CONSOLE_SCREEN_BUFFER_INFO) cinfo
! Executable code to set console window size
sr.top    =   0
sr.left   =   0
sr.bottom =   40 ! <= console buffer height
 -1
sr.right  =   60 ! <=
 console buffer width  -1
lstat = SetConsoleWindowInfo(fhandle, .TRUE., sr)
! Executable code to set console buffer size
nlines = 100
ncols  =  80
wpos.x = ncols  ! columns >= console window width
wpos.y = nlines ! lines   >= console window height
lstat = SetConsoleScreenBufferSize(fhandle, wpos)
! ...
```

Writing and Reading Characters at a Cursor Position

You can position the cursor as needed using the `SetConsoleCursorPosition` routine before you write characters to the screen:

```
! Use previous data declarations
! Position and write two lines
  wpos.x = 5 ! 6 characters from left
  wpos.y = 5 ! 6 lines down
  lstat = SetConsoleCursorPosition(fhandle, wpos)
  write(*,*) 'Six across Six down'
! ...
```

You read from the screen at an appropriate place, but usually you should set the cursor relative to the starting screen location:

```
 ! Use previous and the following data declaration
  CHARACTER(Len=50) charin
! Go back to beginning position of screen
  wpos.x = 0 ! 0 characters from left
  wpos.y = 0 ! 0 lines down
  lstat = SetConsoleCursorPosition(fhandle, wpos)
! Position character input at start of line 11
  wpos.x = 0 ! first character from left
  wpos.y = 10 ! 11 lines down
  lstat = SetConsoleCursorPosition(fhandle, wpos)
  read(*,*) charin
! ...
```

For console I/O, you can use Windows* OS routines `WriteConsoleLine` and `ReadConsoleLine` instead of Fortran `WRITE` and `READ` statements.

See Also

Understanding Project Types
Calling Windows* API Routines
Using the Console
Code Samples of Console Use
Console Use for Fortran QuickWin and Fortran Standard Graphics Applications
Console Use for Fortran Windows* Applications and Fortran DLL Applications
Allocating and Deallocating a Console
Extending Size of the Console Window and Console Buffer
Writing and Reading Characters at a Cursor Position

Additional Documentation: Creating Fortran Applications that Use Windows* Features

A separate document is available that details creating Intel® Fortran applications that use Windows* features.

You can find *Using Intel® Visual Fortran to Create and Build Windows*-Based Applications* on the Intel® Software Documentation Library website at http://software.intel.com/en-us/intel-software-technical-documentation. To get to this document, filter the page for the product suite or the single product.

The document covers the following:

- **Creating Fortran Windowing Applications:** Windows-based applications use the familiar Windows* interface, complete with tool bars, pull-down menus, dialog boxes, and other features. You can include data entry and mouse control in your application and allow for interaction with programs written in other languages or commercial programs such as Microsoft Excel*.
- **Creating and Using Fortran DLLs:** A dynamic-link library (DLL) contains one or more subprogram procedures (functions or subroutines) that are compiled, linked, and stored separately from the applications using them. Because the functions or subroutines are separate from the applications using them, they can be shared or replaced easily.
- **Using QuickWin:** The Intel® Fortran QuickWin run-time library helps you turn graphics programs into simple Windows* applications. Though the full capability of Windows* is not available through QuickWin, QuickWin is simpler to learn and to use. QuickWin applications support pixel-based graphics, real-coordinate graphics, text windows, character fonts, user-defined menus, mouse events, and editing (select/copy/paste) of text, graphics, or both.
- **Using Dialog Boxes for Application Controls:** Dialog boxes are a user-friendly way to solicit application control. As your application executes, you can make a dialog box appear on the screen and the user can click on a dialog box control to enter data or choose what happens next. Using the dialog routines provided with Intel® Fortran, you can add dialog boxes to your application. These routines define dialog boxes and their controls (scroll bars, buttons, and so on), and call your subroutines to respond to user selections.

See Also

Intel® Software Documentation Library

Optimization Reports: Enabling in Microsoft Visual Studio*

Optimization reports can help you address vectorization and optimization issues.

When you build a solution or project, the compiler generates optimization diagnostics. You can view the optimization reports in the following windows:

- The **Compiler Optimization Report** window, either grouped by loops or in a flat format.
- The **Compiler Inline Report** window.

- The optimization annotations, which are integrated within the source editor.

To enable viewing for the optimization reports:

1. In your project's property pages, select **Configuration Properties** > **Fortran** > **Diagnostics**.
2. Set a non-default value for any of the following options:
   - **Optimization Diagnostics Level**
   - **Optimization Diagnostics Phase**
   - **Optimization Diagnostics Routine**
3. The Interprocedural Optimization (IPO) is turned off by default. To view the IPO diagnostics, set the property **Fortran** > **Optimization** > **Interprocedural Optimization** to **Single-file** or **Multi-file**.
4. Build your project to generate an optimization report.

When the compiler generates optimization diagnostics, the **Compiler Optimization Report** and the **Compiler Inline Report** windows open, and the optimization report annotations appear in the source editor.

> **NOTE** You can specify how you want the optimization reporting to appear with the **Optimization Reports** dialog box. Access this dialog box by selecting **Tools** > **Options** > **Intel Compilers and Tools** > **Optimization Reports**.

See Also

Options: Optimization Reports dialog box

Optimization Reports: Viewing

When the compiler generates optimization diagnostics, the **Compiler Optimization Report** and the **Compiler Inline Report** windows open, and optimization report annotations appear in the editor.

Compiler Optimization Report window

The **Compiler Optimization Report** window displays diagnostics for the following phases of the optimization report:

- PGO
- LOOP
- PAR
- VEC
- Offload (Linux* only)
- OpenMP
- CG

Information appears in this window grouped by loops, or in a flat format. To switch the presentation format, click the gear button on the toolbar of the window, and uncheck **Group by loops**.

In addition to sorting information by clicking column headers and resizing columns, you can use the windows described in the following table:

Do This	To Do This
Double-click a diagnostic.	Jump to the corresponding position in the editor.
Click a link in the **Inlined into** column.	Jump to the call site of the function where the loop is inlined.
Expand or collapse a diagnostic in **Group by loops** view.	View detailed information for the diagnostic.
Click on a column header.	Sort the information according to that column.

Do This	To Do This
Click the filter button.	Select a scope by which to filter the diagnostics that appear in the window. The title bar of the **Compiler Optimization Report** window shows the applied filter. Labels on optimization phase filter buttons show how many diagnostics of each phase are in the current scope.
Click a **Compiler Optimization Report** window toolbar button corresponding to an optimization report phase.	Turn filtering diagnostics on or off for an optimization phase. Labels on optimization phase filter buttons show the total number of diagnostics for each phase. By default all phases turned on.
Enter text in the search box in the **Compiler Optimization Report** window toolbar.	Filter diagnostics using the text pattern. Diagnostics are filtered when you stop typing. Pressing Enter saves this pattern in the search history. To disable filtering, clear the search box. To use a pattern from the search history, click on the down arrow next to the search box.

Compiler Inline Report window

The **Compiler Inline Report** window displays diagnostics for the IPO phase of the optimization report.

Information appears in this window in a tree. Each entry in the tree has corresponding information in the right-hand pane under the **Properties** tab and the **Inlining options** tab.

You can use the window as described in the following table:

Do This	To Do This
Double-click a diagnostic in the tree, or click on the source position link under the **Properties** tab.	Jump to the corresponding position in the editor.
Click **Just My Code**.	Only display functions from your code, filter all records from files that don't belong to the current solution file tree.
Right-click on a function body in the editor and select **Intel Compiler** > **Show Inline report for *function name***.	View detailed information for the specified function.
Right-click on a function body in the editor and select **Intel Compiler** > **Show where *function name* in inlined**.	Show where the specified function is inlined.
Enter text in the search box in the **Compiler Inline Report** window toolbar.	Filter diagnostics using the text pattern. Diagnostics are filtered when you stop typing. Pressing Enter saves this pattern in the search history. To disable filtering, clear the search box.

Do This	To Do This
	To use a pattern from the search history, click on the down arrow next to the search box.

Viewing Optimization Notes in the Editor

Viewing optimization notes in the editor provides context for the diagnostics that the compiler generates.

1. In Caller Site
2. In Callee Site
3. In Caller and Callee Site

You can use optimization notes as described in the following table:

Do This	To Do This

Right-click an optimization note	• Expand or collapse the current optimization note, or all of them.
	• Open the **Optimization Reports** dialog box to adjust settings for optimization report viewing. You can view optimization notes in one of the following locations:
	  • **Caller Site**
	  • **Callee Site**
	  • **Caller Site and Callee Site**

Double-click an optimization note heading.	Expand or collapse the current optimization note.
Double-click a diagnostic detail.	Jump to the corresponding position in the editor.
Click a hyperlink in the optimization note.	Show where the specified function is inlined.
Click the help **(?)** icon.	Get detailed help for the selected diagnostic. The default browser opens and, if you are connected to the internet, displays the help topic for this diagnostic.
Hover the mouse over a collapsed optimization note.	View a detailed tooltip about that optimization note.

See Also

Optimization Reports: Enabling in Visual Studio*
qopt-report-phase, Qopt-report-phase

Dialog Box Help

Options: General dialog box

To access the **General** page, click **Tools** > **Options** and then select **Intel Compilers and Tools** > **Visual Fortran** > **General**. Use this page to specify Fortran File Extensions and Build Options.

Build Options

Continue on Errors: Check this box to allow compilation to continue regardless of an error in the current file. The compiler will begin compiling the next file. (To set the maximum number of errors to encounter before compilation stops, choose **Configuration Properties** > **Fortran** > **Diagnostics** > **Error Limit**).

Generate Build Logs: Check this box to generate build logs.

Show Environment in Log: Check this box to show environment variable settings in the log file.

Fortran File Extensions

You can specify additional Fortran free format and fixed format file extensions to be recognized as valid file extensions within the IDE. The IDE treats these additional extensions as compilable Fortran source files. You can also remove or modify existing extensions that appear in the list.

When you add a new extension, the IDE checks the registry to determine whether the extension is already associated with a language, tool, or file format. If there is such an association, a message informs you of this and you will not be allowed to add the extension.

Headers: Specify one or more file extensions for header files, each beginning with a period and separated by semi-colons.

Sources: Specify one or more file extensions for source files, each beginning with a period and separated by semi-colons.

Click **OK** to save your changes.

Options: Compilers dialog box

To access the **Compilers** page:

1. Open **Tools** > **Options**
2. In the left pane, select **Intel Compilers and Tools** > **Visual Fortran** > **Compilers**.

See Also

Selecting the Compiler Version
Specifying Path, Library, and Include Directories

Options: Advanced dialog box

To access the **Advanced** page, expand the **Tools** > **Options** > **Text Editor** > **Fortran** nodes and select **Advanced**. Here you can specify advanced options for text editing.

Browsing/Navigation section

Collect Object Browser information: Choose this option to enable the display of procedures in your project in a hierarchical tree. Once enabled, you can use **View** > **Object Browser** to display your procedures.

Disable Database: Choose this option to disable creation of the code browsing database. This may help increase performance on slow machines. When you disable the database, all features that rely on code browsing information do not work.

Enable Database Saving/Loading: Choose this option to save collected data to a file on disk so that all source browsing features are available immediately when you open the project. When this option is disabled, the code browsing database is generated via background source parsing, so many features that rely on code browsing information do not work until this process completes. Saving and loading the database requires some additional time when saving and loading the project.

Enable Find all References: Choose this option to enable display of the location(s) in your code where a symbol is referenced. When this option is enabled, you can use the right-click context menu **Find All References** option to display a list of references to the selected symbol. Double-click on a reference to find that reference.

Enable Go To Definition: Choose this option to enable quick navigation to an object definition. When this option is enabled, you can use the right-click context menu **Go to Definition** option to locate where the selected object was declared, opening the associated source file if required. (If you have also enabled **Scan system includes**, any objects declared in system modules such as IFWINTY cause the associated source for that module to be opened.)

Scan system includes: Choose this option to scan system include files. This option is used with one or more of the following options: **Collect Object Browser Information**, **Enable Find All References**, **Enable Go To Definition**.

Intrinsics section

Enable Intrinsic Parameter Info: Choose this option to enable the display of intrinsic function and subroutine parameter information. When this option is enabled, you can type a name of an intrinsic procedure, followed by an open parenthesis, and information about the procedure and its arguments appears.

Enable Intrinsic Quick Info: Choose this option to enable the display of additional information when the mouse pointer is moved over an intrinsic function or subroutine name.

Miscellaneous section

Enumerate Comment Tasks: Choose this option to enable the display of a list of tasks consisting of source files containing comments. Comments take the form of the ! character, followed by a token such as TO DO. Valid tokens are those listed in **Tools** > **Options** > **Environment** > **Task List**. When this option is enabled, you can select **Comments** from the task list using **View** > **Other Windows Task List**. Double-click on a comment in the list to jump to its location.

Highlight Matching Tokens: Choose this option to allow identifier highlighting and block delimiter matching. When enabled, this option highlights all references to the identifier under the cursor.

Outlining Section

Enable Outlining: Choose this option to allow the collapsing of whole program units. When this option is enabled, you can click the minus (**-**) or plus (**+**) symbols for `PROGRAM`, `SUBROUTINE`, `FUNCTION`, `MODULE`, and `BLOCK DATA` statements.

Outline Statement Blocks: Choose this option to allow collapsing of block constructs such as `IF` and `DO`. You must also choose **Enable Outlining**.

Configure Analysis dialog box

Use the **Configure Analysis** dialog box to specify settings for Guided Auto Parallelism (GAP) analysis and run the analysis.

To access this dialog box, click **Tools** > **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis...**

Configure Analysis Options

Scope: Specify the desired scope.

Level of Analysis: Specify the desired level of analysis. Choose **Simple**, **Moderate**, **Maximum**, or **Extreme**.

Suppress Compiler Warnings: Check this box to suppress compiler warnings. This adds the option `W0` to the compiler command line.

Suppress remark IDs: Specify one or more remark IDs to suppress. Use a comma to separate IDs.

Send remarks to a file: Check this box to send GAP remarks to a specified text file.

Remarks file: Specify the filename where GAP remarks will be sent.

Save these settings as the default (in Tools -> Options for Guided Auto Parallelism): Check this box to save the specified settings as the default settings.

Show all GAP configuration and informational dialogs: Check this box to display this dialog box next time.

When you are done specifying settings, click **Run Analysis**.

See Also

Options: Guided Auto Parallelism dialog box

Options: Guided Auto Parallelism dialog box

Use the **Guided Auto Parallelism** page to specify settings for Guided Auto Parallelization (GAP) analysis.

To access the **Guided Auto Parallelism** page click **Tools** > **Options** and then select: **Intel Compilers and Tools** > **Visual Fortran** > **Guided Auto Parallelism**.

> **NOTE** These settings are used when running analysis using **Tools** > **Intel Compiler** > **Guided Auto Parallelism** > **Run Analysis on project...**

Guided Auto Parallelism Options

Level of Analysis: Specify the desired level of analysis. Choose **Simple**, **Moderate**, **Maximum**, or **Extreme**.

Suppress compiler warnings: Check this box to suppress compiler warnings. Selection adds option `W0` to the compiler command line.

Suppress Remark IDs: Specify one or more remark IDs to suppress. Use a comma to separate IDs.

Send remarks to a file: Check this box to send GAP remarks to a specified text file.

Remarks file: Specify the filename to send GAP remarks to.

Show all GAP configuration and informational dialogs: Check this box to display additional dialog boxes when you run an analysis.

Reset: Click this button to restore the previously selected settings.

See Also

Using Guided Auto Parallelism in Microsoft Visual Studio*

Guided Auto Parallelism

Using Guided Auto Parallelism

Profile Guided Optimization dialog box

This topic has information on the following dialog boxes:

- **Profile Guided Optimization (PGO)** dialog box
- **Application Invocations** dialog box
- **Edit Command** dialog box
- **Command** dialog box

Profile Guided Optimization dialog box

To access the **Profile Guided Optimization** dialog box, choose **Tools** > **Intel Compiler** > **Profile Guided Optimization**.

Use the **Profile Guided Optimization** dialog box to set the options for profile guided optimization.

Phase 1 - Instrument: This phase produces an instrumented object file for the profile guided optimization. The command line compiler option for each optimization instrument you choose appears in **Compiler Options**.

- **Enable Function Ordering in the optimized application:** Select this checkbox to enable ordering of static and extern routines using profile information. This optimization specifies the order in which the linker should link the functions of your application. This optimization can improve your application performance by improving code locality and by reducing paging.
- **Enable Static Data Layout in the optimized application:** Select this checkbox to enable ordering of static global data items based on profiling information. This optimization specifies the order in which the linker should link global data of your program. This optimization can improve application performance by improving the locality of static global data, reduce paging of large data sets, and improve data cache use.
- **Instrument with guards for threaded application:** Select this checkbox to produce an instrumented object file that includes the collection of PGO data on applications that use a high level of parallelism.

Selecting an option produces a static profile information file (`.spi`), but also increases the time needed to do a parallel build.

Deselect the checkbox to skip this phase to save time running profile guided optimization. When you skip this phase, you use the existing profile information when running profile guided optimization. For example, you may want to skip this phase when you change the code to fix a bug and the fix doesn't affect the architecture of the project.

Phase 2 - Run Instrumented Application(s): This phase runs the instrumented application produced in the previous phase as well as other applications in the **Applications Invocations** dialog box. To add a new application or edit an existing application in the list, click **Applications Invocations**.

Deselect the checkbox to skip this phase to save time running profile guided optimization. When you skip this phase, you do not run the applications in the list when running profile guided optimization. For example, you might want to skip this phase when you change the code to fix a bug and the fix doesn't affect the architecture of the project.

Phase 3 - Optimize with Profile Data: This phase performs the profile guided optimization.

Deselect the checkbox to skip this phase.

Profile Directory: The directory that contains the profile. Click **Edit** to edit the profile directory or the **Browse** button to browse for the profile directory.

Show this dialog next time: Deselect this checkbox to run profile guided optimization without displaying this dialog box. The profile guided optimization will use these settings.

Save Settings: Click to save your settings.

Run: Click to start the profile guided optimization.

Cancel: Click to close this dialog box without starting the profile guided optimization.

Application Invocations dialog box

To access the **Application Invocations** dialog box, click **Application Invocations...** in the **Profile Guided Optimization** dialog box. Use the **Profile Guided Optimization** dialog box to configure the application options for your application as well add additional applications when you run profile guided optimization.

The list of applications comes from the debug settings of the **Startup Project**.

Merge Environment: Select this checkbox to merge the application environment with the environment defined by the operating system.

To add, edit, or remove an application, click one of the buttons.

Add: Click to add a new application in the **Edit Command** dialog box.

Duplicate: Click after selecting an application to copy its settings so that you can use a different setting.

Edit: Click after selecting an application to change its settings in the **Edit Command** dialog box.

Delete: Click to remove the selected application from the list.

OK: Click to save the settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

Edit Command dialog box

To access the **Edit Command** dialog box, click **Add** or **Edit** in the **Application Invocations** dialog box. Use the **Edit Command** dialog box to add a new or edit an existing application in the **Application Invocations** dialog box.

Command: Add a new or edit an existing application. Click **Edit** to open the **Command** dialog box with a list of macros. Click **Browse** to navigate to another directory that contains the application.

Command Arguments: Enter the arguments required by the application.

Working Directory: Enter a new or edit the working directory for the application. Click **Edit** to open the **Command** dialog box with a list of macros. Click **Browse** to navigate to working directory of the application.

Environment: Enter the environment variable required by this application.

Merge Environment: Select this checkbox to merge the application environment with the environment defined by the operating system.

Load from Debugging Settings: Click to load the debug settings for this application.

OK: Click to save the settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

Command dialog box

To access the **Command** dialog box, click **Edit** in the **Edit Command** dialog box. Use the **Command** dialog box to specify or change the macro used in the application to run as part of the profile guided optimization.

Select a macro from the list and then click one of the buttons.

Macro: Click to show or close the list of available macros.

Insert: Click to use the selected macro.

OK: Click to save the settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

See Also

Profile-Guided Optimization

Using Profile Guided Optimization in Microsoft Visual Studio*

Options: Profile Guided Optimization

Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations

Options: Profile Guided Optimization (PGO) dialog box

Use the **Profile Guided Optimization** page to specify settings for PGO. To access the **Profile Guided Optimization** page, click **Tools** > **Options** and then select **Intel Compilers and Tools** > **Profile Guided Optimization**.

Profile Guided Optimization (PGO) Options

Show PGO Dialog: Specify whether to display the Profile Guided Optimization dialog box when you begin PGO.

See Also

Using Profile Guided Optimization in Microsoft* Visual Studio*

Profile Guided Optimization dialog box

Profile-Guided Optimizations

Code Coverage dialog box

To access the **Code Coverage** dialog box, select **Tools** > **Intel Compiler** > **Code Coverage...**.

Use the **Code Coverage** dialog box to set the code coverage feature.

Phase 1 - Instrument: Select this checkbox to compile your code into an instrumented application.

Select the **Instrument with guards for threaded application** checkbox to produce an instrumented object file that includes the collection of PGO data on applications that use a high level of parallelism.

The compiler option used is shown in **Compiler Options**.

Deselect the **Phase 1 - Instrument** checkbox to skip this phase.

Phase 2 - Run Instrumented Application(s): Select this checkbox to run your instrumented application as well as other applications.

You can specify the options to run with the applications by choosing the **Application Invocations...** button to access the **Applications Invocations** dialog box.

Deselect the **Phase 2 - Run Instrumented Application(s)** checkbox to skip this phase.

Phase 3 - Generate Report: Select this checkbox to generate a report with the results of running the instrumented application.

Choose the **Settings...** button to access the Code Coverage Settings dialog box to configure the settings.

Profile Directory: Where the profile is stored.

Browse: Button to browse for the profile directory.

Show this dialog next time: Choose this button to access the dialog box when you run profile guided optimization.

Save Settings: Choose this button to save your settings.

Run: Choose this button to start the profile guided optimization.

Cancel: Choose this button to close this dialog box without starting the profile guided optimization.

See Also

Using Code Coverage in Microsoft Visual Studio*
Code Coverage Settings dialog box
code coverage Tool

Options: Code Coverage dialog box

To access the **Code Coverage** page, click **Tools** > **Options** and then select **Intel Compilers and Tools** > **Code Coverage**.

Use this page to specify settings for code coverage. These settings are used when you run an analysis.

Code Coverage Options

Use the available options to:

- Select colors to be used to show covered and uncovered code.
- Enable or disable the progress meter.
- Set the email address and name of the web page owner.

General

Show Code Coverage Dialog: Specify whether to display the Code Coverage dialog box when you begin code coverage.

Profmerge Options

Suppress Startup Banner: Specify whether version information is displayed.

Verbose: Specify whether additional informational and warning messages should be displayed.

See Also

Using Code Coverage in Microsoft Visual Studio*

Code Coverage dialog box

Code Coverage Tool

Code Coverage Settings dialog box

To access the **Code Coverage Settings** dialog box, choose the **Settings** button in the **Code Coverage** dialog box. Use the **Code Coverage Settings** dialog box to specify settings for the generated report.

Code Coverage options

Ignore Object Unwind Handlers: Set to **True** to ignore the object unwind handlers.

Show Execution Counts: Set to **True** to show the dynamic execution counts in the report.

Treat Partially-covered Code As Fully Covered Code: Set to **True** to treat partially covered code as fully covered code.

Profmerge options

Dump Profile Information: Set to **True** to include profile information in the report.

Exclude Functions: Enter the functions that will be excluded from the profile. The functions must be separated by a comma (","). A period (".") can be used as a wildcard character in function names.

OK: Click to save your settings and close this dialog box.

Cancel: Click to discard the settings and close this dialog box.

See Also

Code Coverage dialog box
Using Code Coverage in the Microsoft Visual Studio* IDE
code coverage Tool

Options: Optimization Reports dialog box

To access the **Optimization Reports** page, click **Tools** > **Options** and then select**Intel Compilers and Tools** > **Optimization Reports**. Use this page to specify how you want optimization reporting to appear.

This page, in conjunction with the **Diagnostics** property page for your project or solution, defines settings for optimization report viewing in Visual Studio*.

General

Always Show Compiler Inline Report: Specify if the Compiler Inline Report appears after building or rebuilding your solution or project when inline diagnostics are present.

Always Show Compiler Optimization Report: Specify if Compiler Optimization Report appears after building or rebuilding your solution or project when optimization diagnostics are present. This option has higher priority than **Always Show Compiler Inline Report**. If both options are set to True, then this window has focus by default.

Show Optimization Notes in Text Editor Margin: Specify if optimization notes appear in the editor as source code annotations.

Optimization Notes in Text Editor

Collapse by Default: Specify if optimization notes appear expanded or collapsed by default.

Show Optimization Notes: Specify if source code annotations appear in the editor.

Site: Specify where optimization notes appear in the editor. Select from one of the following options:

- **Caller Site**
- **Callee Site**
- **Caller Site and Callee Site**

See Also

Optimization Reports: Enabling in Visual Studio*

Using Xcode* (macOS*)

Creating an Xcode* Project

The following topic applies to Xcode*.

To create a new Xcode* project:

1. Launch the Xcode* application.
2. Select **File** > **New** > **Project...**

   The **Choose a template for your new project** window opens.

3. In the left pane, select **macOS*** > **Application**.
4. Select a template, for example: **Command Line Tool**, and click **Next**.
5. Name your project, for example: Hello World, then enter a string for the **Organization Name** and **Organization Identifier** and select the C or C++ language (you can change it to Fortran later). Click **Next**.
6. Specify a directory for your project, and optionally select **Create local git repository for this project** to place your project under version control.
7. Click **Create**.

Xcode* creates the named project directory, with an `.xcodeproj` extension. Your new project directory contains a `main.cpp` source file and other project files. You can delete `main.cpp`.

Each Xcode* project has its own **Project Editor** window that displays project source files, targets, and executables.

See Also

Adding a New File

Selecting the Intel® Compiler

The following topic applies to Xcode*.

To select the Intel®Fortran Compiler:

1. Select the target you want to change and click **Build Rules**.
2. Add a new rule by clicking **Editor** > **Add Build Rule** or pressing the **+** button.
3. Under **Process**, choose **Fortran source files** .
4. Under **Using**, select one of the options for the **Intel® Fortran Compiler**:
   - ***Major_Version***, such as **19.1**: The most recently installed compiler, even if it is not the latest release.
   - **Latest Release**: The latest released compiler available on your system. This is useful when you have multiple installations of the Intel compiler and want to use the version most recently released by Intel.
   - ***Major_Version.n.nnn***: A specific package, such as 19.1.0.000. This is useful when you have multiple packages of one major version installed.

NOTE If the Intel® Compiler does not show up in the drop-down list, it may mean that the compiler does not support your version of Xcode*. To enable the Intel® Compiler in Xcode* specify the Xcode path during installation and restart the program. The installer checks for the supported Xcode* version and warns you in the case of an unsupported version.

NOTE

- If you select a tool that does not support the source file type, that source file type is processed by a later rule that specifies that type.
- Once you have set a new build rule for the Xcode* to use the Intel® Fortran Compiler for Fortran source files, you can view and adjust the option settings.

See Also

Setting Compiler Options

Adding a New File

The following topic applies to Fortran for Xcode*.

To add a new Fortran file:

1. In the Xcode* Project Editor, select **File** > **New** > **File...**
2. Select **Fortran** in the left pane, and then select **Fortran Fixed Format File** or **Fortran Free Format File** in the upper right pane. Click **Next**.
3. Enter a name for the file, including a Fortran source file extension, such as `.f90`, and select a location for the new file. Click **Create**.

Building the Target

The following topic applies to Xcode*.

A single project can contain multiple targets. The active target determines how your project is built. This topic describes how to build the target using the Xcode* IDE and documents the build steps using the `xcodebuild` command line utility.

> **NOTE** Starting with the 19.0 release of the Intel®Fortran Compiler, macOS* 32-bit applications are no longer supported. If you want to compile 32-bit applications, you must use an earlier version of the compiler and you must use Xcode* 9.4 or earlier.

Building Using the Xcode* IDE

1. Select the target in the project editor under **Targets**.
2. Select **Product** > **Build**.
3. To view the results of your build, click the **Log Navigator** button.

You can change the compilation order of the files in an Xcode* target. This may be necessary to compile Fortran source files that define modules (producing `.MOD` files). These can be used by Fortran sources that use the modules. To re-order the files listed under a target's **Compile Sources**, click a source file and drag it before, or after other compilations.

Building From the Command Line Using the xcodebuild Utility

You can use the `xcodebuild` utility to build a target. This utility uses the Xcode* project settings to build target projects from the command line. If you have previously configured your Xcode* project to build with the Intel compiler, `xcodebuild` invokes it from the command line.

To build from the command line:

1. Check that the Xcode* project is configured to use the Intel® Fortran Compiler.
2. Launch a terminal window from the finder by selecting **Applications** > **Utilities** > **Terminal** .
3. Change directories to the directory containing the Xcode* project file (`.xcodeproj`).
4. If you have multiple versions of Xcode*, use the **xcode-select** utility to verify the current Xcode* version.
5. Issue an `xcodebuild` command. For example:

```
xcodebuild -project HelloWorld.xcodeproj -target HelloWorld -configuration Debug
```

6. Run the program built in the example from the previous step by entering the following:

```
./build/Debug/HelloWorld
```

For more information, refer to the `xcodebuild` man page.

Setting the Executable's Architecture

Before building a 64-bit executable from within Xcode*, you may need to edit the executable's target architecture. To change the **Architectures** setting:

1. Click the target you want to change in the project editor under **Targets** and select the **Build Settings** tab.

2. Under **Architectures**, select the desired architecture.

NOTE The Intel® Fortran Compiler generates code solely for Intel® architectures.

Setting Compiler Options

The following topic applies to Xcode*.

To use the Xcode* environment to set compiler options, including options specific to Intel® architecture:

1. Select a target.
2. Under the **Build Settings** tab, click **All**.
3. Scroll down to the list of Intel® Fortran Compiler categories.
4. To set an Intel®Fortran Compiler option in the Floating Point category, scroll down to display **Floating Point**.
5. Select a **Setting**, such as **Extend Precision of Single-Precision Constants**, and set the option's state. If the **Quick Help** inspector is visible, information about the selected option appears under **Quick Help**.

The next time you build your project, the selected options are used in the compilation.

NOTE To view the settings that have changed from the established defaults, select the **Levels** button under **Build Settings**.

Running the Executable

The following topic applies to Xcode*.

Once you have built your Xcode* project, click the **Run** button. The output from the executable is displayed. This button runs the configuration currently associated with the button. Use the **Scheme Editor** to change the configuration associated with the button.

Tip To open the **Scheme Editor**, select **Product** > **Scheme...** > **Edit Scheme...**

Using Dynamic Libraries

Using the Dynamic Libraries does not assume that the Apple* System Integrity Protection feature purges environment variables, such as `DYLD_LIBRARY_PATH`, when launching the protected process. Refer to the https://developer.apple.com/library/archive/documentation/Security/Conceptual/System_Integrity_Protection_Guide/Introduction/Introduction.html for more information. Xcode must take this into account and set the proper environment variables in the Xcode environment.

You can build your Xcode project with the `-shared-intel` compiler option to link with the Intel dynamic libraries. Build your project with the `-qopenmp` or `-parallel` option to link in `libiomp5.dylib`. If you do this, you need to set Xcode build option `Runpath Search path` to an appropriate folder with the compiler and performance libraries, or specify the `DYLD_LIBRARY_PATH` environment variable in the Xcode environment.

To add the environment variable:

1. Open the **Scheme Editor** and select the **Run** action.
2. On the **Arguments** tab, under **Environment Variables**, click the **+** button.
3. Add `DYLD_LIBRARY_PATH`. Set the value to the full path to the Intel compiler's `/lib` directory.

NOTE If you build your project with the `-shared-intel`, `-qopenmp`, or `-parallel` compiler option without setting the `DYLD_LIBRARY_PATH` environment variable, a *library not found* error message results at runtime. Depending on your application, the error message may refer to a library other than the one noted in this example:

```
dyld: Library not loaded: libiomp5.dylib
Referenced
from: /Users/test/hello_world
Reason: image not found
```

Due to the Apple System Integrity Protection you may need to set the `DYLD_LIBRARY_PATH` explicitly in the launch string, or configure the `Runpath Search path` build option.

See Also

shared-intel option
openmp, Qopenmp option
parallel, Qparallel option

Using Intel® Performance Libraries with Xcode*

The following topic applies to Fortran for Xcode*.

The Intel® Fortran Compiler is bundled with the Intel® Math Kernel Library (Intel® MKL). You can access this library in Xcode* by using the **Use Intel® Math Kernel Library** property, located in the **Build Settings** > **Performance Library Build Components**.

The **Use Intel® Math Kernel Library** property provides the following options in a drop-down menu:

- **None:** Disables the use of the Intel® MKL.
- **Use threaded Intel® Math Kernel Library:** Links using the threaded version of the library.
- **Use non-threaded Intel® Math Kernel Library:** Links using the non-threaded version of the library.

For more information, see the Intel® MKL documentation.

Part II

Compiler Reference

Compiler Limits

The amount of data storage, the size of arrays, and the total size of executable programs are limited only by the amount of process virtual address space available, as determined by system parameters.

The table below shows the limits to the size and complexity of a single Intel® Fortran program unit and to individual statements contained within it:

Language Element	Limit
Actual number of arguments per CALL or function reference	Limited only by memory constraints
Arguments in a function reference in a specification expression	255
Array dimensions	31 (The Fortran 2008 standard supports a maximum array dimension of 15.)
Array elements per dimension	2**31-1 on systems using IA-32 architecture 2**63-1 on systems using Intel® 64 architecture Limited by current memory configuration.
Character lengths	2**31-1 on systems using IA-32 architecture 2**63-1 on systems using Intel® 64 architecture
Constants: character and Hollerith	7198
Constants: characters read in list-directed I/O	2048 characters
Continuation lines	No fixed limit; at least 255 lines of 132 or fewer characters. Longer lines may reduce the number of allowed continuations, subject to the limit on lexical tokens per statement.
Data and I/O implied DO nesting	31
DO, CASE, FORALL, WHERE, and block IF statement nesting (combined)	512
DO loop index variable	9,223,372,036,854,775,807= 2**63-1

Language Element	Limit
Format group nesting	8
Fortran source line length	fixed form: 72 (or 132 if `/extend_source` is in effect) characters free form: 7200 characters
INCLUDE file nesting	20 levels
Labels in computed or assigned GOTO list	Limited only by memory constraints
Lexical tokens per statement	40000
Named common blocks	Limited only by memory constraints
Nesting of array constructor implied DOs	31
Nesting of input/output implied DOs	31
Nesting of interface blocks	Limited only by memory constraints
Nesting of DO, IF, or CASE constructs	Limited only by memory constraints
Nesting of parenthesized formats	Limited only by memory constraints
Number of arguments to MIN and MAX	Limited only by memory constraints
Number of digits in a numeric constant	Limited by statement length
Parentheses nesting in expressions	Limited only by memory constraints
Structure nesting	30
Symbolic name length	63 characters
Width field for a numeric edit descriptor	2**15-1 on systems using IA-32 architecture 2**31-1 on systems using Intel® 64 architecture For limits of other edit descriptor fields, see Forms for Data Edit Descriptors.

For more information on memory limits for large data objects, see:

- The AUTOMATIC statement
- The /F compiler option
- The heap-array compiler option
- The product Release Notes

See Also

Forms for Data Edit Descriptors

Using Visual Studio* IDE Automation Objects (Windows*)

This topic briefly describes the Automation interfaces provided by Intel® Visual Fortran. Automation interfaces are programmable objects used to access underlying IDE components and projects to provide experienced developers with a means of automating common tasks and allow a finer degree of control over the IDE and the Fortran projects being used within it.

You can use the Visual Studio* Object Browser (**View** > **Object Browser**) to view an object and its associated properties. Open the following in the browser: **Browse** > **Edit Custom Component Set** > **.NET** > **Microsoft.VisualStudio.VFProject**.

NOTE
The objects listed here are provided as an advanced feature for developers who are already familiar with using Automation objects and the Visual Studio* object model.

Object	Description
IVFCollection	Contains the functionality that can be exercised on a collections object.
VFConfiguration	Programmatically accesses the properties in the **General** property page of a project's Property Pages dialog box. This object also allows access to the tools used to build this configuration.
VFCustomBuildTool	Programmatically accesses the properties in the **Custom Build Step** property page in a project's Property Pages dialog box.
VFDebugSettings	Contains properties that allow the user to programmatically manipulate the settings in the **Debug** property page.
VFFile	Describes the operations that can take place on a file in the active project.
VFFileConfiguration	Contains build information about a file (VFFile object), including such things as what tool is attached to the file for that configuration.
VFFilter	Exposes the functionality on a folder in Solution Explorer for an Intel® Visual Fortran project.
VFFortranCompilerTool	Exposes the functionality of the IFORT tool.
VFFortranCompilerVersion	Provides access to properties relating to the Intel® Visual Fortran compiler version.
VFLibrarianTool	Exposes the functionality of the LIB tool.
VFLinkerTool	Exposes the functionality of the LINK tool.
VFManifestTool	Programmatically accesses the properties in the **Manifest Tool** folder of a project's Property Pages dialog box.
VFMidlTool	Programmatically accesses the properties in the **MIDL folder** of a project's Property Pages dialog box.
VFPlatform	Provides access to properties relating to supported platforms.
VFPreBuildEventTool	Programmatically accesses the properties on the **Pre-Build Event** property page, in the Build Events folder in a project's Property Pages dialog box.

VFPreLinkEventTool	Programmatically accesses the properties on the **PreLink Event** property page, in the Build Events folder in a project's Property Pages dialog box.
VFPostBuildEventTool	Programmatically accesses the properties on the **Post-Build Event** property page, in the Build Events folder in a project's Property Pages dialog box.
VFProject	Exposes the properties on an Intel® Visual Fortran project
VFResourceCompilerTool	Programmatically accesses the properties in the Resources folder in a project's Property Pages dialog box.

The following example, written in Visual Basic*, demonstrates how to use automation objects to modify the list of available platforms and versions in the Visual Studio* IDE Configuration Manager:

```
Imports System
Imports EnvDTE
Imports EnvDTE80
Imports System.Diagnostics
Imports Microsoft.VisualStudio.VFProject
Imports System.Collections
Public Module MultiPV
    ' Create a Console application before executing this module
    ' Module demonstrates Multi Platform & Multi Version Automation Support
    ' Variable definition
    Dim Prj As Project          ' VS project
    Dim VFPrj As VFProject     ' Intel VF project
    Dim o As Object             ' Object
    Sub run()
        ' Get the Project
        Prj  = DTE.Solution.Projects.Item(1)
        '
        ' Get Intel VF project
        VFPrj  = Prj.Object
        '
        ' Get list of Supported platforms
        Dim pList As ArrayList = New ArrayList()    ' list of platforms
        Dim cList As ArrayList = New ArrayList()    ' lost of compilers
        Dim i As Integer
        pList  = getSupportedPlatforms()
        For i = 0 To pList.Count - 1
            cList  = getCompilers(pList.Item(i))
            printCompilers(pList.Item(i), cList)
        Next
        '
        ' Add configurations - x64
        For i = 0 To pList.Count - 1
            If pList.Item(i) <> "Win32" Then
                addConfiguration(pList.Item(i))
            End If
        Next
        Dim cfgsList As ArrayList = New ArrayList() ' list of configurations
        cfgsList = getAllConfigurations()
        '
        ' Set compiler
        For i = 0 To pList.Count - 1
            Dim pNm As String
            Dim cvList As ArrayList = New ArrayList()
            pNm = pList.Item(i)
            cList = getCompilers(pNm)
            cvList = getCompilerVersions(pNm)
```

```
        Dim j As Integer
        For j = 0 To cvList.Count - 1
            Dim cv As String = cvList.Item(j)
            If SetCmplrForPlatform(pNm, cv) Then
                setActiveCfg(pNm)
                SolutionRebuild()
                Dim sOut As String = GetOutput()
                Dim scv As String = CheckCompiler(sOut)
                MsgBox(pNm + " " + cv + " " + scv)
            End If
        Next
    Next
End Sub
' get context from Output window
Function GetOutput() As String
    Dim win As Window
    Dim w As OutputWindow
    Dim wp As OutputWindowPane
    Dim td As TextDocument
    win = DTE.Windows.Item(Constants.vsWindowKindOutput)
    w = win.Object
    Dim i As Integer
    For i = 1 To w.OutputWindowPanes.Count
        wp = w.OutputWindowPanes.Item(i)
        If wp.Name = "Build" Then
            td = wp.TextDocument
            td.Selection.SelectAll()
            Dim ts As TextSelection = td.Selection
            GetOutput = ts.Text
            Exit Function
        End If
    Next
End Function
Function CheckCompiler(ByVal log As String) As String
    Dim s As String
    Dim beg_ As Integer
    Dim end_ As Integer
    beg_ = log.IndexOf("Compiling with")
    beg_ = log.IndexOf("Intel", beg_)
    end_ = log.IndexOf("]", beg_)
    s = log.Substring(beg_, end_ - beg_ + 1)
    CheckCompiler = s
End Function
Function SetCmplrForPlatform(ByVal plNm As String, ByVal vers As String) As Boolean
    Dim pl As VFPlatform
    Dim cll As IVFCollection
    Dim cvs As IVFCollection
    Dim cv As VFFortranCompilerVersion
    Dim maj As String
    Dim min As String
    Dim ind As Integer
    Try
        ind = vers.IndexOf(".")
        maj = vers.Substring(0, ind)
        min = vers.Substring(ind + 1)
        cll = VFPrj.Platforms
        pl = cll.Item(plNm)
        If pl Is Nothing Then
```

```
            MsgBox("Platform " + plNm + " not exist")
            Exit Function
        End If
        cvs = pl.FortranCompilerVersions
        Dim j As Integer
        For j = 1 To cvs.Count
            cv = cvs.Item(j)
            If cv.MajorVersion.ToString() = maj And cv.MinorVersion.ToString() = min Then
                pl.SelectedFortranCompilerVersion = cv
                SetCmplrForPlatform = True
                Exit Function
            End If
        Next
        MsgBox("Compiler version " + maj + "." + min + " not exist for platform " + plNm)
        SetCmplrForPlatform = False
    Catch ex As Exception
        SetCmplrForPlatform = False
    End Try
End Function
Function getSupportedPlatforms() As ArrayList
    Dim list As ArrayList = New ArrayList()
    Dim pl As VFPlatform
    Dim pls As IVFCollection
    pls = VFPrj.Platforms
    Dim i As Integer
    For i = 1 To pls.Count
        pl = pls.Item(i)
        list.Add(pl.Name)
    Next
    getSupportedPlatforms = list
End Function
Function getCompilers(ByVal plNm As String) As ArrayList
    Dim list As ArrayList = New ArrayList()
    Dim pl As VFPlatform
    Dim pls As IVFCollection
    Dim cvs As IVFCollection
    Dim cv As VFFortranCompilerVersion
    Dim j As Integer
    pls = VFPrj.Platforms
    pl = pls.Item(plNm)
    cvs = pl.FortranCompilerVersions
    For j = 1 To cvs.Count
        cv = cvs.Item(j)
        list.Add(cv.DisplayName)
    Next
    getCompilers = list
End Function
Function getCompilerVersions(ByVal plNm As String) As ArrayList
    Dim list As ArrayList = New ArrayList()
    Dim pl As VFPlatform
    Dim pls As IVFCollection
    Dim cvs As IVFCollection
    Dim cv As VFFortranCompilerVersion
    pls = VFPrj.Platforms
    pl = pls.Item(plNm)
    cvs = pl.FortranCompilerVersions
    Dim j As Integer
    For j = 1 To cvs.Count
```

```
            cv = cvs.Item(j)
            Dim vers As String
            vers = cv.MajorVersion.ToString() + "." + cv.MinorVersion.ToString()
            list.Add(vers)
        Next
        getCompilerVersions = list
    End Function
    Sub printCompilers(ByVal plNm As String, ByVal list As ArrayList)
        Dim s As String
        s = "Platform " + plNm + Chr(13)
        Dim i As Integer
        For i = 0 To list.Count - 1
            s += "   " + list.Item(i) + Chr(13)
        Next
        MsgBox(s)
    End Sub
    Sub addConfiguration(ByVal cfgNm As String)
        Dim cM As ConfigurationManager
        cM = Prj.ConfigurationManager
        cM.AddPlatform(cfgNm, "Win32", True)
    End Sub
    Function getAllConfigurations() As ArrayList
        Dim list As ArrayList = New ArrayList()
        Dim cM As ConfigurationManager
        Dim i As Integer
        Dim c As Configuration
        cM = Prj.ConfigurationManager
        For i = 1 To cM.Count
            c = cM.Item(i)
            list.Add(c.ConfigurationName + "|" + c.PlatformName)
        Next
        getAllConfigurations = list
    End Function
    Sub setActiveCfg(ByVal pNm As String)
        Dim scs As SolutionConfigurations = DTE.Solution.SolutionBuild.SolutionConfigurations
        Dim i As Integer
        Dim j As Integer
        For i = 1 To scs.Count
            Dim sc As SolutionConfiguration
            Dim sctxs As SolutionContexts
            sc = scs.Item(i)
            sctxs = sc.SolutionContexts
            For j = 1 To sctxs.Count
                Dim sctx As SolutionContext = sctxs.Item(j)
                If sctx.ConfigurationName = "Debug" And sctx.PlatformName = pNm Then
                    sc.Activate()
                    Exit Sub
                End If
            Next
        Next
    End Sub
    Sub SolutionRebuild()
        DTE.Solution.SolutionBuild.Clean(True)
        DTE.Solution.SolutionBuild.Build(True)
    End Sub
End Module
```

Compiler Options

The Intel®Fortran compiler supports many compiler options you can use in your applications.

In this section, we provide the following:

- Lists of new options and lists of deprecated or removed options
- An alphabetical list of compiler options that includes their short descriptions
- General rules for compiler options and the conventions we use when referring to options
- Details about what appears in the compiler option descriptions
- A description of each compiler option. The descriptions appear under the option's functional category. Within each category, the options are listed in alphabetical order.

New Options

This topic lists the options or option settings that provide new functionality in this release.

If no label appears, the option is available on all supported systems.

If "only" appears in the label, the option is only available on the identified system.

For more details on the options, refer to the individual option descriptions.

For information on conventions used in this table, see Notational Conventions.

New compiler options or option settings are listed in tables below:

- The first table lists new options or option settings that are available on Windows* systems.
- The second table lists new options or option settings that are available on Linux* and macOS* systems. If an option is only available on one of these operating systems, it is labeled.

Windows* Options	Description	Default
`/assume`	New setting [no]old_inquire_recl has been added. It determines the value of the RECL= specifier on an INQUIRE statement for an unconnected unit or a unit connected for stream access. New setting [no]old_ldout_zero has been added. It determines the format of a floating-point zero produced by list-directed output. old_ldout_zero uses exponential format, noold_ldout_zero uses fractional format.	`/assume:old_inquire_recl` `/assume:old_ldout_zero`
`/check`	New setting [no]udio_iostat has been added. It determines whether conformance checking occurs when user defined derived type input/output routines are executed.	`/nocheck`
`/Qauto-arch`	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit.	OFF

Windows* Options	Description	Default
/Qconditional-branch=*keyword*	Lets you identify and fix code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction.	/Qconditional-branch:keep

Linux* and macOS* Options	Description	Default
-assume	New setting [no]old_inquire_recl has been added. It determines the value of the RECL= specifier on an INQUIRE statement for an unconnected unit or a unit connected for stream access. New setting [no]old_ldout_zero has been added. It determines the format of a floating-point zero produced by list-directed output. old_ldout_zero uses exponential format, noold_ldout_zero uses fractional format.	-assumeold_inquire_recl -assume old_ldout_zero
-check	New setting [no]udio_iostat has been added. It determines whether conformance checking occurs when user defined derived type input/output routines are executed.	-nocheck
-mauto-arch	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit.	OFF
-mconditional-branch=*keyword*	Lets you identify and fix code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction.	-mconditional-branch=keep

Alphabetical List of Compiler Options

The following table lists all the current compiler options in alphabetical order.

4Nportlib, 4Yportlib	Determines whether the compiler links to the library of portability routines.
align	Tells the compiler how to align certain data items.
allow	Determines whether the compiler allows certain behaviors.
altparam	Allows alternate syntax (without parentheses) for PARAMETER statements.
ansi-alias, Qansi-alias	Tells the compiler to assume certain rules of the Fortran standard regarding aliasing and array bounds.
arch	Tells the compiler which features it may target, including which instruction sets it may generate.
assume	Tells the compiler to make certain assumptions.
auto	Causes all local, non-SAVEd variables to be allocated to the run-time stack.
auto-scalar, Qauto-scalar	Causes scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute to be allocated to the run-time stack.
ax, Qax	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit.
Bdynamic	Enables dynamic linking of libraries at run time.
bigobj	Increases the number of sections that an object file can contain.
bintext	Places a text string into the object file (.obj) being generated by the compiler.
B	Specifies a directory that can be used to find include files, libraries, and executables.
Bstatic	Enables static linking of a user's library.
Bsymbolic	Binds references to all global symbols in a program to the definitions within a user's shared library.
Bsymbolic-functions	Binds references to all global function symbols in a program to the definitions within a user's shared library.
ccdefault	Specifies the type of carriage control used when a file is displayed at a terminal screen.
check	Checks for certain conditions at run time.
coarray, Qcoarray	Enables the coarray feature.
coarray-config-file, Qcoarray-config-file	Specifies the name of a Message Passing Interface (MPI) configuration file.
coarray-num-images, Qcoarray-num-images	Specifies the default number of images that can be used to run a coarray executable.
complex-limited-range, Qcomplex-limited-range	Determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.
convert	Specifies the format of unformatted files containing numeric data.
c	Prevents linking.

cxxlib	Determines whether the compiler links using the C++ run-time libraries provided by gcc.
dbglibs	Tells the linker to search for unresolved references in a debug run-time library.
D	Defines a symbol name that can be associated with an optional value.
debug (Linux* and macOS*)	Enables or disables generation of debugging information.
debug (Windows*)	Enables or disables generation of debugging information.
debug-parameters	Tells the compiler to generate debug information for PARAMETERs used in a program.
diag, Qdiag	Controls the display of diagnostic information during compilation.
diag-dump, Qdiag-dump	Tells the compiler to print all enabled diagnostic messages.
diag-error-limit, Qdiag-error-limit	Specifies the maximum number of errors allowed before compilation stops.
diag-file, Qdiag-file	Causes the results of diagnostic analysis to be output to a file.
diag-file-append, Qdiag-file-append	Causes the results of diagnostic analysis to be appended to a file.
diag-id-numbers, Qdiag-id-numbers	Determines whether the compiler displays diagnostic messages by using their ID number values.
diag-once, Qdiag-once	Tells the compiler to issue one or more diagnostic messages only once.
d-lines, Qd-lines	Compiles debug statements.
dll	Specifies that a program should be linked as a dynamic-link (DLL) library.
double-size	Specifies the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX declarations, constants, functions, and intrinsics.
dryrun	Specifies that driver tool commands should be shown but not executed.
dumpmachine	Displays the target machine and operating system configuration.
dynamiclib	Invokes the libtool command to generate dynamic libraries.
dynamic-linker	Specifies a dynamic linker other than the default.
dyncom, Qdyncom	Enables dynamic allocation of common blocks at run time.
E	Causes the preprocessor to send output to stdout.
EP	Causes the preprocessor to send output to stdout, omitting #line directives.
exe	Specifies the name for a built program or dynamic-link library.
extend-source	Specifies the length of the statement field in a fixed-form source file.
extfor	Specifies file extensions to be processed by the compiler as Fortran files.
extfpp	Specifies file extensions to be recognized as a file to be preprocessed by the Fortran preprocessor.
extlnk	Specifies file extensions to be passed directly to the linker.
F (macOS*)	Adds a framework directory to the head of an include file search path.

F (Windows*)	Specifies the stack reserve amount for the program.
f66	Tells the compiler to apply FORTRAN 66 semantics.
f77rtl	Tells the compiler to use the run-time behavior of FORTRAN 77.
falias, Oa	Specifies whether or not a procedure call may have hidden aliases of local variables not supplied as actual arguments.
falign-functions, Qfnalign	Tells the compiler to align procedures on an optimal byte boundary.
falign-loops, Qalign-loops	Aligns loops to a power-of-two byte boundary.
falign-stack	Tells the compiler the stack alignment to use on entry to routines.
Fa	Specifies that an assembly listing file should be generated.
FA	Specifies the contents of an assembly listing file.
fast	Maximizes speed across the entire program.
fast-transcendentals, Qfast-transcendentals	Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.
fasynchronous-unwind-tables	Determines whether unwind information is precise at an instruction boundary or at a call boundary.
fcode-asm	Produces an assembly listing with machine code annotations.
fcommon	Determines whether the compiler treats common symbols as global definitions.
Fd	Lets you specify a name for a program database (PDB) file created by the compiler.
feliminate-unused-debug-types, Qeliminate-unused-debug-types	Controls the debug information emitted for types declared in a compilation unit.
fexceptions	Enables exception handling table generation.
ffat-lto-objects	Determines whether a fat link-time optimization (LTO) object, containing both intermediate language and object code, is generated during an interprocedural optimization compilation (-c –ipo).
ffnalias, Ow	Determines whether aliasing is assumed within functions.
fimf-absolute-error, Qimf-absolute-error	Defines the maximum allowable absolute error for math library function results.
fimf-accuracy-bits, Qimf-accuracy-bits	Defines the relative error for math library function results, including division and square root.
fimf-arch-consistency, Qimf-arch-consistency	Ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture.
fimf-domain-exclusion, Qimf-domain-exclusion	Indicates the input arguments domain on which math functions must provide correct results.
fimf-force-dynamic-target, Qimf-force-dynamic-target	Instructs the compiler to use run-time dispatch in calls to math functions.
fimf-max-error, Qimf-max-error	Defines the maximum allowable relative error for math library function results, including division and square root.

fimf-precision, Qimf-precision	Lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.
fimf-use-svml, Qimf-use-svml	Instructs the compiler to use the Short Vector Math Library (SVML) rather than the Intel® Math Library (LIBM) to implement math library functions.
finline-functions	Enables function inlining for single file compilation.
finline-limit	Lets you specify the maximum size of a function to be inlined.
finline	Tells the compiler to inline functions declared with !DIR$ ATTRIBUTES FORCEINLINE .
finstrument-functions, Qinstrument-functions	Determines whether routine entry and exit points are instrumented.
fixed	Specifies source files are in fixed format.
fkeep-static-consts, Qkeep-static-consts	Tells the compiler to preserve allocation of variables that are not referenced in the source.
fltconsistency	Enables improved floating-point consistency.
fma, Qfma	Determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor.
fmath-errno	Tells the compiler that errno can be reliably tested after calls to standard math library functions.
fmerge-constants	Determines whether the compiler and linker attempt to merge identical constants (string constants and floating-point constants) across compilation units.
fmerge-debug-strings	Causes the compiler to pool strings used in debugging information.
fminshared	Specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.
fmpc-privatize	Enables or disables privatization of all static data for the MultiProcessor Computing environment (MPC) unified parallel runtime.
fnsplit, Qfnsplit	Enables function splitting.
fomit-frame-pointer, Oy	Determines whether EBP is used as a general-purpose register in optimizations.
foptimize-sibling-calls	Determines whether the compiler optimizes tail recursive calls.
fpconstant	Tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.
fpe-all	Allows some control over floating-point exception handling for each routine in a program at run-time.
fpe	Allows some control over floating-point exception handling for the main program at run-time.
fpic	Determines whether the compiler generates position-independent code.
fpie	Tells the compiler to generate position-independent code. The generated code can only be linked into executables.
fp-model, fp	Controls the semantics of floating-point calculations.
fpp-name	Lets you specify an alternate preprocessor to use with Fortran.

fp-port, Qfp-port	Rounds floating-point results after floating-point operations.
fpp	Runs the Fortran preprocessor on source files before compilation.
fprotect-parens, Qprotect-parens	Determines whether the optimizer honors parentheses when floating-point expressions are evaluated.
fpscomp	Controls whether certain aspects of the run-time system and semantic language features within the compiler are compatible with Intel® Fortran or Microsoft* Fortran PowerStation.
fp-speculation, Qfp-speculation	Tells the compiler the mode in which to speculate on floating-point operations.
fp-stack-check, Qfp-stack-check	Tells the compiler to generate extra code after every function call to ensure that the floating-point stack is in the expected state.
free	Specifies source files are in free format.
fsource-asm	Produces an assembly listing with source code annotations.
fstack-protector	Enables or disables stack overflow security checks for certain (or all) routines.
fstack-security-check	Determines whether the compiler generates code that detects some buffer overruns.
ftrapuv , Qtrapuv	Initializes stack local variables to an unusual value to aid error detection.
ftz, Qftz	Flushes subnormal results to zero.
fuse-ld	Tells the compiler to use a different linker instead of the default linker (ld).
fverbose-asm	Produces an assembly listing with compiler comments, including options and version information.
fvisibility	Specifies the default visibility for global symbols or the visibility for symbols in a file.
fzero-initialized-in-bss, Qzero-initialized-in-bss	Determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.
gcc-name	Lets you specify the name of the gcc compiler that should be used to set up the link-time environment, including the location of standard libraries.
gdwarf	Lets you specify a DWARF Version format when generating debug information.
Ge	Enables stack-checking for all functions. This is a deprecated option. The replacement option is /Gs0.
gen-depformat	Specifies the form for the output generated when option gen-dep is specified.
gen-depshow	Determines whether certain features are excluded from dependency analysis. Currently, it only applies to intrinsic modules.
gen-dep	Tells the compiler to generate build dependencies for the current compilation.
gen-interfaces	Tells the compiler to generate an interface block for each routine in a source file.
GF	Enables read-only string-pooling optimization.

global-hoist, Qglobal-hoist	Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.
grecord-gcc-switches	Causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.
GS	Determines whether the compiler generates code that detects some buffer overruns.
Gs	Lets you control the threshold at which the stack checking routine is called or not called.
gsplit-dwarf	Creates a separate object file containing DWARF debug information.
g	Tells the compiler to generate a level of debugging information in the object file.
guard	Enables the control flow protection mechanism.
guide, Qguide	Lets you set a level of guidance for auto-vectorization, auto parallelism, and data transformation.
guide-data-trans, Qguide-data-trans	Lets you set a level of guidance for data transformation.
guide-file, Qguide-file	Causes the results of guided auto parallelism to be output to a file.
guide-file-append, Qguide-file-append	Causes the results of guided auto parallelism to be appended to a file.
guide-opts, Qguide-opts	Tells the compiler to analyze certain code and generate recommendations that may improve optimizations.
guide-par, Qguide-par	Lets you set a level of guidance for auto parallelism.
guide-vec, Qguide-vec	Lets you set a level of guidance for auto-vectorization.
gxx-name	Lets you specify the name of the g++ compiler that should be used to set up the link-time environment, including the location of standard libraries.
heap-arrays	Puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.
help	Displays all available compiler options or a category of compiler options.
homeparams	Tells the compiler to store parameters passed in registers to the stack.
hotpatch	Tells the compiler to prepare a routine for hotpatching.
idirafter	Adds a directory to the second include file search path.
iface	Specifies the default calling convention and argument-passing convention for an application.
init, Qinit	Lets you initialize a class of variables to zero or to various numeric exceptional values.
inline-factor, Qinline-factor	Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.
inline-forceinline, Qinline-forceinline	Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.
inline-level, Ob	Specifies the level of inline function expansion.

inline-max-per-compile, Qinline-max-per-compile	Specifies the maximum number of times inlining may be applied to an entire compilation unit.
inline-max-per-routine, Qinline-max-per-routine	Specifies the maximum number of times the inliner may inline into a particular routine.
inline-max-size, Qinline-max-size	Specifies the lower limit for the size of what the inliner considers to be a large routine.
inline-max-total-size, Qinline-max-total-size	Specifies how much larger a routine can normally grow when inline expansion is performed.
inline-min-caller-growth, Qinline-min-caller-growth	Lets you specify a procedure size n for which procedures of size <= n do not contribute to the estimated growth of the caller when inlined.
inline-min-size, Qinline-min-size	Specifies the upper limit for the size of what the inliner considers to be a small routine.
inline	Specifies the level of inline function expansion.
intconstant	Tells the compiler to use FORTRAN 77 semantics to determine the kind parameter for integer constants.
integer-size	Specifies the default KIND for integer and logical variables.
intel-freestanding	Lets you compile in the absence of a gcc environment.
intel-freestanding-target-os	Lets you specify the target operating system for compilation.
ip, Qip	Determines whether additional interprocedural optimizations for single-file compilation are enabled.
ip-no-inlining, Qip-no-inlining	Disables full and partial inlining enabled by interprocedural optimization options.
ip-no-pinlining, Qip-no-pinlining	Disables partial inlining enabled by interprocedural optimization options.
ipo, Qipo	Enables interprocedural optimization between files.
ipo-c, Qipo-c	Tells the compiler to optimize across multiple files and generate a single object file.
ipo-jobs, Qipo-jobs	Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).
ipo-S, Qipo-S	Tells the compiler to optimize across multiple files and generate a single assembly file.
ipo-separate, Qipo-separate	Tells the compiler to generate one object file for every source file.
I	Specifies an additional directory for the include path.
isystem	Specifies a directory to add to the start of the system include path.
libdir	Controls whether linker options for search libraries are included in object files generated by the compiler.
libs	Tells the compiler which type of run-time library to link to.
link	Passes user-specified options directly to the linker at compile time.
list-line-len	Specifies the line length for the listing generated when option list is specified.

list-page-len	Specifies the page length for the listing generated when option list is specified.
list	Tells the compiler to create a listing of the source file.
logo	Displays the compiler version information.
l	Tells the linker to search for a specified library when linking.
L	Tells the linker to search for libraries in a specified directory before searching the standard directories.
m32, m64 , Qm32, Qm64	Tells the compiler to generate code for a specific architecture.
m80387	Specifies whether the compiler can use x87 instructions.
map-opts, Qmap-opts	Maps one or more compiler options to their equivalent on a different operating system.
map	Tells the linker to generate a link map file.
march	Tells the compiler to generate code for processors that support certain features.
masm	Tells the compiler to generate the assembler output file using a selected dialect.
mauto-arch, Qauto-arch	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit.
mbranches-within-32B-boundaries, Qbranches-within-32B-boundaries	Tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.
mcmodel	Tells the compiler to use a specific memory model to generate code and store data.
mconditional-branch, Qconditional-branch	Lets you identify and fix code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction.
MDs	Tells the linker to search for unresolved references in a single-threaded, dynamic-link run-time library. This is a deprecated option. There is no replacement option.
MD	Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.
mdynamic-no-pic	Generates code that is not position-independent but has position-independent external references.
minstruction, Qinstruction	Determines whether MOVBE instructions are generated for certain Intel processors.
mkl, Qmkl	Tells the compiler to link to certain libraries in the Intel® Math Kernel Library (Intel® MKL). On Windows systems, you must specify this option at compile time.
module	Specifies the directory where module files should be placed when created and where they should be searched for.
momit-leaf-frame-pointer	Determines whether the frame pointer is omitted or kept in leaf functions.
mp1, Qprec	Improves floating-point precision and consistency.

mstringop-inline-threshold, Qstringop-inline-threshold	Tells the compiler to not inline calls to buffer manipulation functions such as memcpy and memset when the number of bytes the functions handle are known at compile time and greater than the specified value.
mstringop-strategy, Qstringop-strategy	Lets you override the internal decision heuristic for the particular algorithm used when implementing buffer manipulation functions such as memcpy and memset.
m	Tells the compiler which features it may target, including which instruction sets it may generate.
MT	Tells the linker to search for unresolved references in a multithreaded, static run-time library.
mtune, tune	Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march).
multiple-processes, MP	Creates multiple processes that can be used to compile large numbers of source files at the same time.
names	Specifies how source code identifiers and external names are interpreted.
no-bss-init, Qnobss-init	Tells the compiler to place in the DATA section any uninitialized variables and explicitly zero-initialized variables.
nodefaultlibs	Prevents the compiler from using standard libraries when linking.
nofor-main	Specifies that the main program is not written in Fortran.
nolib-inline	Disables inline expansion of standard library or intrinsic functions.
nostartfiles	Prevents the compiler from using standard startup files when linking.
nostdlib	Prevents the compiler from using standard libraries and startup files when linking.
object	Specifies the name for an object file.
Od	Disables all optimizations.
Ofast	Sets certain aggressive options to improve the speed of your application.
Os	Enables optimizations that do not increase code size; it produces smaller code size than O2.
O	Specifies the code optimization for applications.
o	Specifies the name for an output file.
Ot	Enables all speed optimizations.
pad, Qpad	Enables the changing of the variable and array memory layout.
pad-source, Qpad-source	Specifies padding for fixed-form source records.
par-affinity, Qpar-affinity	Specifies thread affinity.
parallel, Qparallel	Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.
parallel-source-info, Qparallel-source-info	Enables or disables source location emission when OpenMP* or auto-parallelism code is generated.
par-num-threads, Qpar-num-threads	Specifies the number of threads to use in a parallel region.

par-runtime-control, Qpar-runtime-control	Generates code to perform run-time checks for loops that have symbolic loop bounds.
par-schedule, Qpar-schedule	Lets you specify a scheduling algorithm for loop iterations.
par-threshold, Qpar-threshold	Sets a threshold for the auto-parallelization of loops.
pc, Qpc	Enables control of floating-point significand precision.
p	Compiles and links for function profiling with gprof(1).
pdbfile	Lets you specify the name for a program database (PDB) file created by the linker.
pie	Determines whether the compiler generates position-independent code that will be linked into an executable.
prec-div, Qprec-div	Improves precision of floating-point divides.
prec-sqrt, Qprec-sqrt	Improves precision of square root implementations.
preprocess-only	Causes the Fortran preprocessor to send output to a file.
print-multi-lib	Prints information about where system libraries should be found.
print-sysroot	Prints the target sysroot directory that is used during compilation.
prof-data-order, Qprof-data-order	Enables or disables data ordering if profiling information is enabled.
prof-dir, Qprof-dir	Specifies a directory for profiling information output files.
prof-file, Qprof-file	Specifies an alternate file name for the profiling summary files.
prof-func-groups	Enables or disables function grouping if profiling information is enabled.
prof-func-order, Qprof-func-order	Enables or disables function ordering if profiling information is enabled.
prof-gen, Qprof-gen	Produces an instrumented object file that can be used in profile guided optimization.
prof-gen-sampling	Tells the compiler to generate debug discriminators in debug output. This aids in developing more precise sampled profiling output.
prof-hotness-threshold, Qprof-hotness-threshold	Lets you set the hotness threshold for function grouping and function ordering.
prof-src-dir, Qprof-src-dir	Determines whether directory information of the source file under compilation is considered when looking up profile data records.
prof-src-root, Qprof-src-root	Lets you use relative directory paths when looking up profile data and specifies a directory as the base.
prof-src-root-cwd, Qprof-src-root-cwd	Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.
prof-use, Qprof-use	Enables the use of profiling information during optimization.
prof-use-sampling	Lets you use data files produced by hardware profiling to produce an optimized executable.
prof-value-profiling, Qprof-value-profiling	Controls which values are value profiled.
pthread	Tells the compiler to use pthreads library for multithreading support.

qcf-protection, Qcf-protection	Enables Control-flow Enforcement Technology (CET) protection, which defends your program from certain attacks that exploit vulnerabilities. This option offers preliminary support for CET.
Qcov-dir	Specifies a directory for profiling information output files that can be used with the codecov or tselect tool.
Qcov-file	Specifies an alternate file name for the profiling summary files that can be used with the codecov or tselect tool.
Qcov-gen	Produces an instrumented object file that can be used with the codecov or tselect tool.
Qinline-dllimport	Determines whether dllimport functions are inlined.
Qinstall	Specifies the root directory where the compiler installation was performed.
Qlocation	Specifies the directory for supporting tools.
qoffload	Lets you specify the mode for offloading or tell the compiler to ignore language constructs for offloading. This is a deprecated option. There is no replacement option.
qopenmp, Qopenmp	Enables the parallelizer to generate multi-threaded code based on OpenMP* directives.
qopenmp-lib, Qopenmp-lib	Lets you specify an OpenMP* run-time library to use for linking.
qopenmp-link	Controls whether the compiler links to static or dynamic OpenMP* run-time libraries.
qopenmp-offload	Enables or disables OpenMP* offloading compilation for the TARGET directives .
qopenmp-simd, Qopenmp-simd	Enables or disables OpenMP* SIMD compilation.
qopenmp-stubs, Qopenmp-stubs	Enables compilation of OpenMP* programs in sequential mode.
qopenmp-threadprivate, Qopenmp-threadprivate	Lets you specify an OpenMP* threadprivate implementation.
qopt-args-in-regs, Qopt-args-in-regs	Determines whether calls to routines are optimized by passing arguments in registers instead of on the stack.
qopt-assume-safe-padding, Qopt-assume-safe-padding	Determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of the object.
qopt-block-factor, Qopt-block-factor	Lets you specify a loop blocking factor.
qopt-dynamic-align, Qopt-dynamic-align	Enables or disables dynamic data alignment optimizations.
Qoption	Passes options to a specified tool.
qopt-jump-tables, Qopt-jump-tables	Enables or disables generation of jump tables for switch statements.
qopt-malloc-options	Lets you specify an alternate algorithm for malloc().
qopt-matmul, Qopt-matmul	Enables or disables a compiler-generated Matrix Multiply (matmul) library call.

qopt-mem-layout-trans, Qopt-mem-layout-trans	Controls the level of memory layout transformations performed by the compiler.
qopt-multi-version-aggressive, Qopt-multi-version-aggressive	Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.
qopt-prefetch, Qopt-prefetch	Enables or disables prefetch insertion optimization.
qopt-prefetch-distance, Qopt-prefetch-distance	Specifies the prefetch distance to be used for compiler-generated prefetches inside loops.
qopt-prefetch-issue-excl-hint, Qopt-prefetch-issue-excl-hint	Supports the prefetchW instruction in Intel® microarchitecture code name Broadwell and later.
qopt-ra-region-strategy, Qopt-ra-region-strategy	Selects the method that the register allocator uses to partition each routine into regions.
qopt-report, Qopt-report	Tells the compiler to generate an optimization report.
qopt-report-annotate, Qopt-report-annotate	Enables the annotated source listing feature and specifies its format.
qopt-report-annotate-position, Qopt-report-annotate-position	Enables the annotated source listing feature and specifies the site where optimization messages appear in the annotated source in inlined cases of loop optimizations.
qopt-report-embed, Qopt-report-embed	Determines whether special loop information annotations will be embedded in the object file and/or the assembly file when it is generated.
qopt-report-file, Qopt-report-file	Specifies that the output for the optimization report goes to a file, stderr, or stdout.
qopt-report-filter, Qopt-report-filter	Tells the compiler to find the indicated parts of your application, and generate optimization reports for those parts of your application.
qopt-report-format, Qopt-report-format	Specifies the format for an optimization report.
qopt-report-help, Qopt-report-help	Displays the optimizer phases available for report generation and a short description of what is reported at each level.
qopt-report-names, Qopt-report-names	Specifies whether mangled or unmangled names should appear in the optimization report.
qopt-report-per-object, Qopt-report-per-object	Tells the compiler that optimization report information should be generated in a separate file for each object.
qopt-report-phase, Qopt-report-phase	Specifies one or more optimizer phases for which optimization reports are generated.
qopt-report-routine, Qopt-report-routine	Tells the compiler to generate an optimization report for each of the routines whose names contain the specified substring.
qopt-streaming-stores, Qopt-streaming-stores	Enables generation of streaming stores for optimization.
qopt-subscript-in-range, Qopt-subscript-in-range	Determines whether the compiler assumes that there are no "large" integers being used or being computed inside loops.
qopt-zmm-usage, Qopt-zmm-usage	Defines a level of zmm registers usage.

qoverride-limits, Qoverride-limits	Lets you override certain internal compiler limits that are intended to prevent excessive memory usage or compile times for very large, complex compilation units.
Qpar-adjust-stack	Tells the compiler to generate code to adjust the stack size for a fiber-based main thread.
Qpatchable-addresses	Tells the compiler to generate code such that references to statically assigned addresses can be patched.
Qsfalign	Specifies stack alignment for functions.
qsimd-honor-fp-model, Qsimd-honor-fp-model	Tells the compiler to obey the selected floating-point model when vectorizing SIMD loops.
qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction	Tells the compiler to serialize floating-point reduction when vectorizing SIMD loops.
Quse-msasm-symbols	Tells the compiler to use a dollar sign ("$") when producing symbol names.
Qvc	Specifies compatibility with Microsoft*Visual C++* or Microsoft Visual Studio*.
rcd, Qrcd	Enables fast float-to-integer conversions. This is a deprecated option. There is no replacement option.
real-size	Specifies the default KIND for real and complex declarations, constants, functions, and intrinsics.
recursive	Tells the compiler that all routines should be compiled for possible recursive execution.
reentrancy	Tells the compiler to generate reentrant code to support a multithreaded application.
safe-cray-ptr, Qsafe-cray-ptr	Tells the compiler that Cray* pointers do not alias other variables.
save, Qsave	Causes variables to be placed in static memory.
save-temps, Qsave-temps	Tells the compiler to save intermediate files created during compilation.
scalar-rep, Qscalar-rep	Enables or disables the scalar replacement optimization done by the compiler as part of loop transformations.
S	Causes the compiler to compile to an assembly file only and not link.
shared-intel	Causes Intel-provided libraries to be linked in dynamically.
shared-libgcc	Links the GNU libgcc library dynamically.
shared	Tells the compiler to produce a dynamic shared object instead of an executable.
show	Controls the contents of the listing generated when option list is specified.
simd, Qsimd	Enables or disables compiler interpretation of SIMD directives .
sox	Tells the compiler to save the compilation options and version number in the executable file. It also lets you choose whether to include lists of certain routines .
standard-realloc-lhs	Determines whether the compiler uses the current Fortran Standard rules or the old Fortran 2003 rules when interpreting assignment statements.

standard-semantics	Determines whether the current Fortran Standard behavior of the compiler is fully implemented.
stand	Tells the compiler to issue compile-time messages for nonstandard language elements.
static-intel	Causes Intel-provided libraries to be linked in statically.
static-libgcc	Links the GNU libgcc library statically.
staticlib	Invokes the libtool command to generate static libraries.
static-libstdc++	Links the GNU libstdc++ library statically.
static	Prevents linking with shared libraries.
syntax-only	Tells the compiler to check only for correct syntax.
sysroot	Specifies the root directory where headers and libraries are located.
tcollect, Qtcollect	Inserts instrumentation probes calling the Intel® Trace Collector API.
tcollect-filter, Qtcollect-filter	Lets you enable or disable the instrumentation of specified functions. You must also specify option [Q]tcollect.
Tf	Tells the compiler to compile the file as a Fortran source file.
threads	Tells the linker to search for unresolved references in a multithreaded run-time library.
traceback	Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.
T	Tells the linker to read link commands from a file.
u (Windows*)	Undefines all previously defined preprocessor values.
undef	Disables all predefined symbols .
unroll , Qunroll	Tells the compiler the maximum number of times to unroll loops.
unroll-aggressive, Qunroll-aggressive	Determines whether the compiler uses more aggressive unrolling for certain loops.
use-asm, Quse-asm	Tells the compiler to produce objects through the assembler. This is a deprecated option. There is no replacement option.
U	Undefines any definition currently in effect for the specified symbol .
vec, Qvec	Enables or disables vectorization.
vecabi, Qvecabi	Determines which vector function application binary interface (ABI) the compiler uses to create or call vector functions.
vec-guard-write, Qvec-guard-write	Tells the compiler to perform a conditional check in a vectorized loop.
vec-threshold, Qvec-threshold	Sets a threshold for the vectorization of loops.
vms	Causes the run-time system to behave like HP* Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).
v	Specifies that driver tool commands should be displayed and executed.
Wa	Passes options to the assembler for processing.

warn	Specifies diagnostic messages to be issued by the compiler.
watch	Tells the compiler to display certain information to the console output window.
WB	Turns a compile-time bounds check into a warning.
what	Tells the compiler to display its detailed version string.
winapp	Tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.
Winline	Warns when a function that is declared as inline is not inlined.
Wl	Passes options to the linker for processing.
Wp	Passes options to the preprocessor.
wrap-margin	Provides a way to disable the right margin wrapping that occurs in Fortran list-directed output.
x, Qx	Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.
xHost, QxHost	Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.
Xlinker	Passes a linker option directly to the linker.
X	Removes standard directories from the include file search path.
zero, Qzero	Initializes to zero variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are not yet initialized. This is a deprecated option. The replacement option is /Qinit:[no]zero or -init=[no]zero.
Zi, Z7	Tells the compiler to generate full debugging information in either an object (.obj) file or a project database (PDB) file.
Zo	Enables or disables generation of enhanced debugging information for optimized code.

Deprecated and Removed Compiler Options

This topic lists deprecated and removed compiler options and suggests replacement options, if any are available.

For more information on compiler options, see the detailed descriptions of the individual option descriptions in this section.

Deprecated Options

Occasionally, compiler options are marked as "deprecated." Deprecated options are still supported in the current release, but are planned to be unsupported in future releases.

The following two tables list options that are currently deprecated.

Note that deprecated options are not limited to these lists.

Deprecated Linux* and macOS*Options	Suggested Replacement
`-axS`	`-axSSE4.1`
`-axT`	Linux*: `-axSSSE3`

Deprecated Linux* and macOS*Options	Suggested Replacement
	macOS*: `-axSSSE3`
`-cpp`	`-fpp`
`-march=pentiumii`	None
`-march=pentiumiii`	`-march=pentium3`
`-mcpu`	`-mtune`
`-msse`	Linux* only: `-mia32`
`-rcd`	None
`-stand f15`	`-stand f18`
`-use-asm`	None
`-xS`	`-xSSE4.1`
`-xT`	Linux: `-xSSSE3` macOS*: `-xSSSE3`
`-[no]zero`	`-init=[no]zero`

Deprecated Windows* Options	Suggested Replacement
`/arch:SSE`	`/arch:IA32`
`/Ge`	`/Gs0`
`/MDs[d]`	None
`/QaxS`	`/QaxSSE4.1`
`/QaxT`	`/QaxSSSE3`
`/QIfist`	`/Qrcd`
`/Qrcd`	None
`/Qsox`	None
`/Quse-asm`	None
`/QxS`	`/QxSSE4.1`
`/QxT`	`/QxSSSE3`
`/Qzero[-]`	`/Qinit:[no]zero`
`/stand f15`	`/stand f18`
`/unroll`	`/Qunroll`

Removed Options

Some compiler options are no longer supported and have been removed. If you use one of these options, the compiler issues a warning, ignores the option, and then proceeds with compilation.

The following two tables list options that are no longer supported.

Note that removed options are not limited to these lists.

Removed Linux* and macOS*Options	Suggested Replacement
`-1`	`-f66`
`-66`	`-f66`
`-automatic`	`-auto`
`-axB`	`-axSSE2`
`-axH`	`-axSSE4.2`
`-axi`	None
`-axK`	No exact replacement; upgrade to `-msse2`
`-axM`	None
`-axN`	Linux*: `-axSSE2` macOS*: None
`-axP`	Linux: `-axSSE3` macOS*: None
`-axW`	`-msse2`
`-cm`	`-warn nousage`
`-cxxlib-gcc[=dir]`	`-cxxlib[=dir]`
`-cxxlib-icc`	None
`-dps`	`-altparam`
`-F`	`-preprocess-only` or `-P`
`-falign-stack=mode`	None; this option is only removed on macOS*
`-fp`	`-fno-omit-frame-pointer`
`-fpstkchk`	`-fp-stack-check`
`-func-groups`	`-prof-func-groups`
`-fvisibility=internal`	`-fvisibility=hidden`
`-gcc-version`	No exact replacement; use `-gcc-name`
`-guide-profile`	None
`-i-dynamic`	`-shared-intel`
`-i-static`	`-static-intel`

Removed Linux* and macOS*Options	Suggested Replacement
-inline-debug-info	-debug inline-debug-info
-ipo-obj (and -ipo_obj)	None
-Kpic, -KPIC	-fpic
-lowercase	-names lowercase
-mp	-fp-model
-nobss-init	-no-bss-init
-no-standard-semantics	No exact replacement; negate specific options separately
-nus	-assume nounderscore
-Ob	-inline-level
-onetrip	-f66
-openmp	-qopenmp
-openmp-lib	-qopenmp-lib
-openmp-lib legacy	None
-openmp-link and -qopenmp-link	None
-openmpP	-qopenmp
-openmp-profile	None
-openmp-report	-qopt-report-phase=openmp
-openmpS	-qopenmp-stubs
-openmp-simd	-qopenmp-simd
-openmp-stubs	-qopenmp-stubs
-openmp-threadprivate	-qopenmp-threadprivate
-opt-args-in-regs	-qopt-args-in-regs
-opt-assume-safe-padding	-qopt-assume-safe-padding
-opt-block-factor	-qopt-block-factor
-opt-dynamic-align	-qopt-dynamic-align
-opt-gather-scatter-unroll	None
-opt-jump-tables	-qopt-jump-tables
-opt-malloc-options	-qopt-malloc-options
-opt-matmul	-qopt-matmul

Removed Linux* and macOS*Options	Suggested Replacement
-opt-mem-layout-trans	-qopt-mem-layout-trans
-opt-multi-version-aggressive	-qopt-multi-version-aggressive
-opt-prefetch	-qopt-prefetch
-opt-prefetch-distance	-qopt-prefetch-distance
-opt-ra-region-strategy	-qopt-ra-region-strategy
-opt-report	-qopt-report
-opt-report-embed	-qopt-report-embed
-opt-report-file	-qopt-report-file
-opt-report-filter	-qopt-report-filter
-opt-report-format	-qopt-report-format
-opt-report-help	-qopt-report-help
-opt-report-level	-qopt-report
-opt-report-per-object	-qopt-report-per-object
-opt-report-phase	-qopt-report-phase
-opt-report-routine	-qopt-report-routine
-opt-streaming-cache-evict	None
-opt-streaming-stores	-qopt-streaming-stores
-opt-subscript-in-range	-qopt-subscript-in-range
-par-report	-qopt-report-phase=par
-prefetch	-qopt-prefetch
-prof-format-32	None
-prof-gen-sampling	None
-prof-genx	-prof-gen=srcpos
-profile-functions	None
-profile-loops	None
-profile-loops-report	None
-qoffload-arch	None
-qoffload-attribute-target	None
-qoffload-option	None
-qopenmp-report	-qopt-report-phase=openmp

Removed Linux* and macOS*Options	Suggested Replacement
-qopenmp-task	None
-qp	-p
-rct	None
-shared-libcxa	-shared-libgcc
-ssp	None
-static-libcxa	-static-libgcc
-syntax	-syntax-only or -fsyntax-only
-tcheck	None
-tpp1	None
-tpp2	-mtune=itanium2
-tpp5	None
-tpp6	None
-tpp7	-mtune=pentium4
-tprofile	None
-tune	-x<code>
-uppercase	-names uppercase
-us	-assume underscore
-vec-report	-qopt-report-phase=vec
-xB	-xSSE2
-xi	None
-xK	No exact replacement; upgrade to -msse2
-xM	None
-xN	Linux: -xSSE2 macOS*: None
-xO	-msse3
-xP	Linux: -xSSE3 macOS*: None
-xSSE3_ATOM	-xATOM_SSSE3
-xSSSE3_ATOM	-xATOM_SSSE3
-xW	-msse2

Removed Windows* Options	Suggested Replacement
/1	/f66
/4ccD (and /4ccd)	None
/4Nb	/check:none
/4Yb	/check:all
/architecture	/arch
/asmattr:none, /noasmattr	/FA
/asmattr:machine	/FAc
/asmattr:source	/FAs
/asmattr:all	/FAcs
/asmfile	/Fa
/automatic	/auto
/cm	/warn:nousage
/debug:parallel	None
/debug:partial	None
/Fm	/map
/G1	None
/G5	None
/G6 (or /GB)	None
/G7	None
/Gf	/GF
/ML[d]	Upgrade to /MT[d]
/Og	/O1, /O2, or /O3
/Op	/fltconsistency
/optimize:0, /nooptimize	/Od
/optimize:1, /optimize:2	/O1
/optimize:3, /optimize:4	/O2
/optimize:5	/O3
/QaxB	/QaxSSE2
/QaxH	/QaxSSE4.2
/Qaxi	None

Removed Windows* Options	Suggested Replacement
/QaxK	Upgrade to /arch:SSE2
/QaxM	None
/QaxN	/QaxSSE2
/QaxP	/QaxSSE3
/QaxW	/arch:SSE2
/Qcpp	/fpp
/Qdps	/altparam
/Qextend-source	/extend-source
/Qfpp[0 | 1 | 2 | 3]	/fpp
/Qfpstkchk	/Qfp-stack-check
/Qguide-profile	None
/Qinline-debug-info	/debug:inline-debug-info
/Qipo-obj (and /Qipo_obj)	None
/Qlowercase	/names:lowercase
/Qonetrip	/f66
/Qopenmp-lib:legacy	None
/Qopenmp-link	None
/Qopenmp-profile	None
/Qopenmp-report	/Qopt-report-phase:openmp
/Qopt-report-level	/Qopt-report
/Qpar-report	/Qopt-report-phase:par
/Qprefetch	/Qopt-prefetch
/Qprof-format-32	None
/Qprof-gen-sampling	None
/Qprof-genx	/Qprof-gen=srcpos
/Qprofile-functions	None
/Qprofile-loops	None
/Qprofile-loops-report	None
/Qrct	None
/Qssp	None

Removed Windows* Options	Suggested Replacement
/Qtprofile	None
/Qtcheck	None
/Quppercase	/names:uppercase
/Quse-vcdebug	None
/Qvc11	None
/Qvc10	
/Qvc9 and earlier	
/Qvec-report	/Qopt-report-phase:vec
/Qvms	/vms
/QxB	/QxSSE2
/Qxi	None
/QxK	Upgrade to /arch:SSE2
/QxM	None
/QxN	/QxSSE2
/QxO	/arch:SSE3
/QxP	/QxSSE3
/QxSSE3_ATOM	/QxATOM_SSSE3
/QxSSSE3_ATOM	/QxATOM_SSSE3
/QxW	/arch:SSE2
/source	/Tf
/standard-semantics-	No exact replacement; negate specific options separately
/tune	/Qx<code>
/unix	None
/us	/assume:underscore
/w90, /w95	None
/Zd	/debug:minimal

Optimization Notice
dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Ways to Display Certain Option Information

This section describes how you can get general information about compiler options on the command line.

Displaying General Option Information From the Command Line

To display a list of all available compiler options, specify option `help` on the command line.

To display functional groupings of compiler options, specify a functional category for option `help`. For example, to display a list of options that affect diagnostic messages, enter one of the following commands:

```
-help diagnostics     ! Linux and macOS*systems
```

```
/help diagnostics     ! Windows systems
```

For details on other categories you can specify, see help.

Compiler Option Details

This section contains the full details about compiler options, including descriptions of each compiler option.

In this section compiler options are listed within their categories. To see an alphabetical list of compiler options, see Alphabetical List of Compiler Options .

General Rules for Compiler Options

This section describes general rules for compiler options and it contains information about how we refer to compiler option names in descriptions.

General Rules for Compiler Options

You cannot combine options with a single dash (Linux* and macOS*) or slash (Windows*). For example:

- On Linux* and macOS* systems: This is incorrect: `-Ec`; this is correct: `-E -c`
- On Windows* systems: This is incorrect: `/Ec`; this is correct: `/E /c`

All Linux* and macOS* compiler options are case sensitive. Many Windows* options are case sensitive. Some options have different meanings depending on their case; for example, option "c" prevents linking, but option "C" checks for certain conditions at run time.

Options specified on the command line apply to all files named on the command line.

Options can take arguments in the form of file names, strings, letters, or numbers. If a string includes spaces, the string must be enclosed in quotation marks. For example:

- On Linux* and macOS* systems, `-dynamic-linker` *mylink* (file name) or `-U`*macro3* (string)
- On Windows* systems, `/Famyfile.s` (file name) or `/V`*"version 5.0"* (string)

Compiler options can appear in any order.

On Windows* systems, all compiler options must precede `/link` options, if any, on the command line.

Unless you specify certain options, the command line will both compile and link the files you specify.

You can abbreviate some option names, entering as many characters as are needed to uniquely identify the option.

Certain options accept one or more keyword arguments following the option name. For example, the `ax` option accepts several keywords.

To specify multiple keywords, you typically specify the option multiple times.

NOTE
On Windows* systems, you can sometimes use a comma to separate keywords. For example, the following is valid:

```
ifort /warn:usage,declarations test.f90
```

On these systems, you can use an equals sign (=) instead of the colon:

```
ifort /warn=usage,declarations test.f90
```

Compiler options remain in effect for the whole compilation unless overridden by a compiler directive.

To disable an option, specify the negative form of the option.

On Windows* systems, you can also disable one or more optimization options by specifying option `/Od` last on the command line.

NOTE
On Windows* systems, the `/Od` option is part of a mutually-exclusive group of options that includes `/Od`, `/O1`, `/O2`, `/O3`, and `/Ox`. The last of any of these options specified on the command line will override the previous options from this group.

If there are enabling and disabling versions of an option on the command line, the last one on the command line takes precedence.

How We Refer to Compiler Option Names in Descriptions

The following conventions are used as shortcuts when referencing compiler option names in descriptions:

- Many options have names that are the same on Linux*, macOS*, and Windows*, except that the Windows form starts with an initial / and the Linux and macOS* form starts with an initial -. Within text, such option names are shown without the initial character; for example, `check`.
- Many options have names that are the same on Linux*, macOS*, and Windows*, except that the Windows form starts with an initial Q. Within text, such option names are shown as `[Q]option-name`.

  For example, if you see a reference to `[Q]ipo`, the Linux* and macOS* form of the option is `-ipo` and the Windows form of the option is `/Qipo`.
- Several compiler options have similar names except that the Linux* and macOS* forms start with an initial q and the Windows form starts with an initial Q. Within text, such option names are shown as `[q or Q]option-name`.

  For example, if you see a reference to `[q or Q]opt-report`, the Linux* and macOS* form of the option is `-qopt-report` and the Windows form of the option is `/Qopt-report`.

Compiler option names that are more dissimilar are shown in full.

What Appears in the Compiler Option Descriptions

This section contains details about what appears in the option descriptions.

Following sections include individual descriptions of all the current compiler options. The option descriptions are arranged by functional category. Within each category, the option names are listed in alphabetical order.

Each option description contains the following information:

- The primary name for the option and a short description of the option.
- Architecture Restrictions
  This section only appears if there is a known architecture restriction for the option.

  Restrictions may appear for any of the following architectures:

  - IA-32 architecture
  - Intel® 64 architecture

  Certain operating systems are not available on all the above architectures. For the latest information, please check your Release Notes.
- Syntax
  This shows the syntax on Linux* and macOS* systems and the syntax on Windows* systems. If the option has no syntax on one of these systems, that is, the option is not valid on a particular operating system, it will specify "None".
- Arguments
  This shows any arguments (parameters) that are related to the option. If the option has no arguments, it will specify "None".
- Default
  This shows the default setting for the option.
- Description
  This shows the full description of the option. It may also include further information on any applicable arguments.
- IDE Equivalent
  This shows information related to the integrated development environment (IDE) Property Pages on Linux*, macOS*, and Windows* systems. It shows on which Property Page the option appears, and under what category it's listed. The Windows* IDE is Microsoft* Visual Studio* .NET; the Linux* IDE is Eclipse*; the macOS* IDE is Xcode*. If the option has no IDE equivalent, it will specify "None". Note that in this release, there is no IDE support for Fortran on Linux*.
- Alternate Options
  This lists any options that are synonyms for the described option. If there are no alternate option names, it will show "None".
  Some of the alternate option names are deprecated and may be removed in future releases.
  Many options have an older spelling where underscores ("_") instead of hyphens ("-") connect the main option names. The older spelling is a valid alternate option name.

Some option descriptions may also have the following:

- Example
  This shows a short example that includes the option
- See Also
  This shows where you can get further information on the option or related options.

Offload Options (Linux* only)

qoffload

Lets you specify the mode for offloading or tell the compiler to ignore language constructs for offloading. This is a deprecated option. There is no replacement option.

Syntax

Linux OS:

`-qoffload[=keyword]`

`-qno-offload`

macOS:

None

Windows OS:

None

Arguments

keyword Specifies the mode for offloading or it disables offloading. Possible values are:

`none` Tells the compiler to ignore language constructs for offloading. Warnings are issued by the compiler. This is equivalent to the negative form of the option.

`mandatory` Specifies that offloading is mandatory (required). If the target is not available, one of the following occurs:

- If no STATUS clause is specified for the OFFLOAD directive, the program fails with an error message.
- If the STATUS clause is specified, the program continues execution on the CPU.

`optional` Specifies that offloading is optional (requested). If the target is not available, the program is executed on the CPU, not the target.

Default

mandatory The compiler recognizes language constructs for offloading if they are specified. If option `-qoffload` is specified with no *keyword*, the default is `mandatory`.

Description

This option lets you specify the mode for offloading or tell the compiler to ignore language constructs for offloading.

Option `-q[no-]offload` is the replacement option for `-[no-]offload`, which is deprecated.

If no `-qoffload` option appears on the command line, then OFFLOAD directives are processed and:

- The MANDATORY or OPTIONAL clauses are obeyed if present
- If no MANDATORY or OPTIONAL clause is present, the offload is mandatory

If `-qoffload=none` or `-qno-offload` appears on the command line, then OFFLOAD directives are ignored:

However, OpenMP* directives for processor control (for example, !$OMP TARGET) are recognized if the `[q or Q]openmp` option is specified, regardless of whether or not OFFLOAD directives are recognized or ignored.

If *keyword* `mandatory` or `optional` appears for `-qoffload`, then OFFLOAD directives are processed and:

- The MANDATORY or OPTIONAL clauses are obeyed, regardless of the `-qoffload` *keyword* specified.
- If no MANDATORY or OPTIONAL clause is present, then the `-qoffload` *keyword* is obeyed.

If the STATUS clause is specified for an OFFLOAD directive, it affects run-time behavior.

IDE Equivalent

None

Alternate Options

None

See Also

Supported Environment Variables

Optimization Options

falias, Oa

Specifies whether or not a procedure call may have hidden aliases of local variables not supplied as actual arguments.

Syntax

Linux OS and macOS:

`-falias`

`-fno-alias`

Windows OS:

`/Oa`

`/Oa-`

Arguments

None

Default

`-fno-alias` or `/Oa`	Procedure calls do not alias local variables.

Description

This option specifies whether or not the compiler can assume that during a procedure call, local variables in the caller that are not present in the actual argument list and not visible by host association, are not referenced or redefined due to hidden aliasing. The Fortran standard generally prohibits such aliasing.

If you specify `-falias` (Linux* and macOS*) or `/Oa-` (Windows*), aliasing during a procedure call is assumed; this can possibly affect performance.

If you specify `-fno-alias` or `/Oa` (the default), aliasing during a procedure call is not assumed.

IDE Equivalent

None

Alternate Options

None

See Also

`ffnalias` compiler option

fast

Maximizes speed across the entire program.

Syntax

Linux OS:

`-fast`

macOS:

`-fast`

Windows OS:

`/fast`

Arguments

None

Default

OFF The optimizations that maximize speed are not enabled.

Description

This option maximizes speed across the entire program.

It sets the following options:

- On macOS* systems: `-ipo`, `-mdynamic-no-pic`,`-O3`, `-no-prec-div`,`-fp-model fast=2`, and `-xHost`
- On Windows* systems: `/O3`, `/Qipo`, `/Qprec-div-`, `/fp:fast=2`, and `/QxHost`
- On Linux* systems: `-ipo`, `-O3`, `-no-prec-div`,`-static`, `-fp-model fast=2`, and `-xHost`

When option `fast` is specified, you can override the `[Q]xHost` option setting by specifying a different processor-specific `[Q]x` option on the command line. However, the last option specified on the command line takes precedence.

For example:

- On Linux* systems, if you specify option `-fast -xSSE3`, option `-xSSE3` takes effect. However, if you specify `-xSSE3 -fast`, option `-xHost` takes effect.
- On Windows* systems, if you specify option `/fast /QxSSE3`, option`/QxSSE3` takes effect. However, if you specify `/QxSSE3 /fast`, option `/QxHost` takes effect.

For implications on non-Intel processors, refer to the `[Q]xHost` documentation.

> **NOTE**
> Option `fast` sets some aggressive optimizations that may not be appropriate for all applications. The resulting executable may not run on processor types different from the one on which you compile. You should make sure that you understand the individual optimization options that are enabled by option `fast`.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

IDE Equivalent

None

Alternate Options

None

See Also

fp-model, fp compiler option

xHost, QxHost
compiler option

x, Qx
compiler option

ffnalias, Ow

Determines whether aliasing is assumed within functions.

Syntax

Linux OS and macOS:

`-ffnalias`

`-fno-fnalias`

Windows OS:

`/Ow`

`/Ow-`

Arguments

None

Default

`-ffnalias` or `/Ow`	Aliasing is assumed within functions.

Description

This option determines whether aliasing is assumed within functions.

If you specify `-fno-fnalias` or `/Ow-`, aliasing is not assumed within functions, but it is assumed across calls.

If you specify `-ffnalias` or `/Ow`, aliasing is assumed within functions.

IDE Equivalent

None

Alternate Options

None

See Also

falias compiler option

foptimize-sibling-calls

Determines whether the compiler optimizes tail recursive calls.

Syntax

Linux OS:

`-foptimize-sibling-calls`

`-fno-optimize-sibling-calls`

macOS:

`-foptimize-sibling-calls`

`-fno-optimize-sibling-calls`

Windows OS:

`None`

Arguments

None

Default

`-foptimize-sibling-calls` The compiler optimizes tail recursive calls.

Description

This option determines whether the compiler optimizes tail recursive calls. It enables conversion of tail recursion into loops.

If you do not want to optimize tail recursive calls, specify `-fno-optimize-sibling-calls`.

Tail recursion is a special form of recursion that doesn't use stack space. In tail recursion, a recursive call is converted to a GOTO statement that returns to the beginning of the function. In this case, the return value of the recursive call is only used to be returned. It is not used in another expression. The recursive function is converted into a loop, which prevents modification of the stack space used.

IDE Equivalent

None

Alternate Options

None

fprotect-parens, Qprotect-parens

Determines whether the optimizer honors parentheses when floating-point expressions are evaluated.

Syntax

Linux OS and macOS:

`-fprotect-parens`

`-fno-protect-parens`

Windows OS:

`/Qprotect-parens`

`/Qprotect-parens-`

Arguments

None

Default

`-fno-protect-parens` or `/Qprotect-parens-`	Parentheses are ignored when determining the order of expression evaluation.

Description

This option determines whether the optimizer honors parentheses when determining the order of floating-point expression evaluation.

When option `-fprotect-parens` (Linux* and macOS*) or `/Qprotect-parens` (Windows*) is specified, the optimizer will maintain the order of evaluation imposed by parentheses in the code.

When option `-fno-protect-parens` (Linux* and macOS*) or `/Qprotect-parens-` (Windows*) is specified, the optimizer may reorder floating-point expressions without regard for parentheses if it produces faster executing code.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-assume protect_parens`

Windows: `/assume:protect_parens`

Example

Consider the following expression:

```
A+(B+C)
```

By default, the parentheses are ignored and the compiler is free to re-order the floating-point operations based on the optimization level, the setting of option `-fp-model` (Linux* and macOS*) or `/fp` (Windows*), etc. to produce faster code. Code that is sensitive to the order of operations may produce different results (such as with some floating-point computations).

However, if `-fprotect-parens` (Linux* and macOS*) or `/Qprotect-parens` (Windows*) is specified, parentheses around floating-point expressions (including complex floating-point and decimal floating-point) are honored and the expression will be interpreted following the normal precedence rules, that is, B+C will be computed first and then added to A.

This may produce slower code than when parentheses are ignored. If floating-point sensitivity is a specific concern, you should use option `-fp-model precise` (Linux* and macOS*) or `/fp:precise` (Windows*) to ensure precision because it controls all optimizations that may affect precision.

See Also

`fp-model, fp` compiler option

GF

Enables read-only string-pooling optimization.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/GF`

Arguments

None

Default

OFF Read/write string-pooling optimization is enabled.

Description

This option enables read only string-pooling optimization.

IDE Equivalent

None

Alternate Options

None

nolib-inline

Disables inline expansion of standard library or intrinsic functions.

Syntax

Linux OS:

`-nolib-inline`

macOS:

`-nolib-inline`

Windows OS:

None

Arguments

None

Default

OFF The compiler inlines many standard library and intrinsic functions.

Description

This option disables inline expansion of standard library or intrinsic functions. It prevents the unexpected results that can arise from inline expansion of these functions.

IDE Equivalent

None

Alternate Options

None

O

Specifies the code optimization for applications.

Syntax

Linux OS:

`-O[n]`

macOS:

`-O[n]`

Windows OS:

`/O[n]`

Arguments

n	Is the optimization level. Possible values are 1, 2, or 3. On Linux* and macOS* systems, you can also specify 0.

Default

`O2` Optimizes for code speed. This default may change depending on which other compiler options are specified. For details, see below.

Description

This option specifies the code optimization for applications.

Option	Description
`O` (Linux* and macOS*)	This is the same as specifying `O2`.
`O0` (Linux and macOS*)	Disables all optimizations.

Option	Description
	This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. This option causes certain `warn` options to be ignored. This is the default if you specify option `-debug` (with no keyword).
`O1`	Enables optimizations for speed and disables some optimizations that increase code size and affect speed. To limit code size, this option: • Enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling. This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. The `O1` option may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.

`O2`	Enables optimizations for speed. This is the generally recommended optimization level.
	Vectorization is enabled at `O2` and higher levels.
	
	On systems using IA-32 architecture: Some basic loop optimizations such as Distribution, Predicate Opt, Interchange, multi-versioning, and scalar replacements are performed.
	
	This option also enables:
	• Inlining of intrinsics
	• Intra-file interprocedural optimization, which includes:
	  • inlining
	  • constant propagation
	  • forward substitution
	  • routine attribute propagation
	  • variable address-taken analysis
	  • dead static function elimination
	  • removal of unreferenced variables
	• The following capabilities for performance gain:
	  • constant propagation
	  • copy propagation
	  • dead-code elimination
	  • global register allocation
	  • global instruction scheduling and control speculation
	  • loop unrolling
	  • optimized code selection
	  • partial redundancy elimination
	  • strength reduction/induction variable simplification
	  • variable renaming
	  • exception handling optimizations

Option	Description
	• tail recursions • peephole optimizations • structure assignment lowering and optimizations • dead store elimination This option may set other options, especially options that optimize for code speed. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. On Windows* systems, this option is the same as the `Ox` option. On Linux* and macOS* systems, if `-g` is specified, `O2` is turned off and `O0` is the default unless `O2` (or `O1` or `O3`) is explicitly specified in the command line together with `-g`. On Linux systems, the `-debug inline-debug-info` option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`). Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.
`O3`	Performs `O2` optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release. When `O3` is used with options `-ax` or `-x` (Linux) or with options `/Qax` or `/Qx` (Windows), the compiler performs more aggressive data dependency analysis than for `O2`, which may result in longer compilation times. The `O3` optimizations may not cause higher performance unless loop and memory access transformations take place. The optimizations may slow down code in some cases compared to `O2` optimizations. The `O3` option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

The last `O` option specified on the command line takes precedence over any others.

IDE Equivalent

Visual Studio: **General > Optimization** (`/Od`, `/O1`, `/O2`, `/O3`, `/fast`)

Optimization > Optimization (`/Od`, `/O1`, `/O2`, `/O3`, `/fast`)

Eclipse: None

Xcode: **General > Optimization Level** (`-O`)

Alternate Options

O2	Linux and macOS*: None Windows: /Ox

See Also

Od compiler option

fltconsistency compiler option

fast compiler option

Od

Disables all optimizations.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/Od

Arguments

None

Default

OFF	The compiler performs default optimizations.

Description

This option disables all optimizations. It can be used for selective optimizations, such as a combination of /Od and /Ob1 (disables all optimizations, but enables inlining).

This option also causes certain /warn options to be ignored.

On IA-32 architecture, this option sets the /Oy- option.

IDE Equivalent

Visual Studio: **Optimization > Optimization**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS*: -O0

Windows: /optimize:0

See Also

O compiler option (see O0)

Ofast

Sets certain aggressive options to improve the speed of your application.

Syntax

Linux OS:

`-Ofast`

macOS:

`-Ofast`

Windows OS:

`None`

Arguments

None

Default

OFF The aggressive optimizations that improve speed are not enabled.

Description

This option improves the speed of your application.

It sets compiler options `-O3`, `-no-prec-div`, and -fp-model fast=2.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

See Also

`O` compiler option

`prec-div, Qprec-div` compiler option

`fast` compiler option

`fp-model, fp` compiler option

Os

Enables optimizations that do not increase code size; it produces smaller code size than O2.

Syntax

Linux OS:

`-Os`

macOS:

`-Os`

Windows OS:

`/Os`

Arguments

None

Default

OFF Optimizations are made for code speed. However, if `O1` is specified, `Os` is the default.

Description

This option enables optimizations that do not increase code size; it produces smaller code size than `O2`. It disables some optimizations that increase code size for a small speed benefit.

This option tells the compiler to favor transformations that reduce code size over transformations that produce maximum performance.

IDE Equivalent

Visual Studio: **Optimization > Favor Size or Speed**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`O` compiler option

`Ot` compiler option

Ot

Enables all speed optimizations.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Ot`

Arguments

None

Default

`/Ot` Optimizations are made for code speed.

If `Od` is specified, all optimizations are disabled. If `O1` is specified, `Os` is the default.

Description

This option enables all speed optimizations.

IDE Equivalent

Visual Studio: **Optimization > Favor Size or Speed (`/Ot, /Os`)**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

- `O` compiler option
- `Os` compiler option

Code Generation Options

arch

Tells the compiler which features it may target, including which instruction sets it may generate.

Syntax

Linux OS and macOS:

`-arch` *code*

Windows OS:

`/arch:`*code*

Arguments

code

Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`AMBERLAKE`
`BROADWELL`
`CANNONLAKE`
`CASCADELAKE`
`COFFEELAKE`
`GOLDMONT`
`GOLDMONT-PLUS`
`HASWELL`
`ICELAKE-CLIENT` (or `ICELAKE`)
`ICELAKE-SERVER`
`IVYBRIDGE`
`KABYLAKE`
`KNL`
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name.
	
	Keyword `ICELAKE` is deprecated and may be removed in a future release.

KNM SANDYBRIDGE SILVERMONT SKYLAKE SKYLAKE-AVX512 TREMONT WHISKEYLAKE	
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
CORE-AVX-I	May generate the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
AVX	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
SSE4.2	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
SSE4.1	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
SSSE3	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
SSE3	May generate Intel® SSE3, SSE2, and SSE instructions.
SSE2	May generate Intel® SSE2 and SSE instructions. This value is only available on Linux* and Windows* systems.
SSE	This option has been deprecated; it is now the same as specifying IA32.
IA32	Generates x86/x87 generic code that is compatible with IA-32 architecture. Disables any default extended instruction settings, and any previously set extended instruction settings. It also disables all feature-specific optimizations and instructions. This value is only available on Linux* and Windows* systems using IA-32 architecture.

Default

Windows and Linux systems: `SSE2` macOS* systems: `SSSE3`	For more information on the default values, see Arguments above.

Description

This option tells the compiler which features it may target, including which instruction sets it may generate.

Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

Options `/arch` and `/Qx` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

If you specify both the `/Qax` and `/arch` options, the compiler will not generate Intel-specific instructions.

For compatibility with Compaq* Visual Fortran, the compiler allows the following keyword values. However, you should use the suggested replacements.

Compatibility Value	Suggested Replacement on Linux* and Windows*
`pn1`	`-mia32` or `/arch:IA32`
`pn2`	`-mia32` or `/arch:IA32`
`pn3`	`-mia32` or `/arch:IA32`
`pn4`	`-msse2` or `/arch:SSE2`

IDE Equivalent

Visual Studio: **Code Generation > Enable Enhanced Instruction Set**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS*: `-m`

Windows: None

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`m` compiler option
`m32, m64` compiler option

ax, Qax

Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit.

Syntax

Linux OS:

`-ax`*code*

macOS:

`-ax`*code*

Windows OS:

`/Qax`*code*

Arguments

code

Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. The following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`AMBERLAKE`
`BROADWELL`
`CANNONLAKE`
`CASCADELAKE`
`COFFEELAKE`
`GOLDMONT`
`GOLDMONT-PLUS`
`HASWELL`
`ICELAKE-CLIENT` (or `ICELAKE`)
`ICELAKE-SERVER`
`IVYBRIDGE`
`KABYLAKE`
`KNL`
`KNM`
`SANDYBRIDGE`
`SILVERMONT`
`SKYLAKE`
`SKYLAKE-AVX512`
`TREMONT`
`WHISKEYLAKE`
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name.
	Keywords `KNL` and `SILVERMONT` are only available on Windows* and Linux* systems.
	Keyword `ICELAKE` is deprecated and may be removed in a future release.

`COMMON-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, as well as the instructions enabled with CORE-AVX2.

`CORE-AVX512`	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Doubleword and Quadword instructions, Intel® AVX-512 Byte and Word instructions and Intel® AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2.
`CORE-AVX2`	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`CORE-AVX-I`	May generate the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`AVX`	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`SSE4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel processors.
`SSE4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors.
`SSSE3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions for Intel® processors. For macOS* systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions for Intel® processors. This value is not available on macOS* systems.
`SSE2`	May generate Intel® SSE2 and SSE instructions for Intel® processors. This value is not available on macOS* systems.

Default

OFF No auto-dispatch code is generated. Feature-specific code is generated and is controlled by the setting of the following compiler options:

- Linux*: `-m` and `-x`
- Windows*: `/arch` and `/Qx`
- macOS*: `-x`

Description

This option tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. It also generates a baseline code path. The Intel feature-specific auto-dispatch path is usually more optimized than the baseline path. Other options, such as `O3`, control how much optimization is performed on the baseline path.

The baseline code path is determined by the architecture specified by options `-m` or `-x` (Linux* and macOS*) or options `/arch` or `/Qx` (Windows*). While there are defaults for the `[Q]x` option that depend on the operating system being used, you can specify an architecture and optimization level for the baseline code that is higher or lower than the default. The specified architecture becomes the effective minimum architecture for the baseline code path.

If you specify both the `[Q]ax` and `[Q]x` options, the baseline code will only execute on Intel® processors compatible with the setting specified for the `[Q]x`.

If you specify both the `-ax` and `-m` options (Linux and macOS*) or the `/Qax` and `/arch` options (Windows), the baseline code will execute on non-Intel processors compatible with the setting specified for the `-m` or `/arch` option.

If you specify both the `-ax` and `-march` options (Linux and macOS*), or the `/Qax` and `/arch` options (Windows), the compiler will not generate Intel-specific instructions.

The `[Q]ax` option tells the compiler to find opportunities to generate separate versions of functions that take advantage of features of the specified instruction features.

If the compiler finds such an opportunity, it first checks whether generating a feature-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a feature-specific version of a function and a baseline version of the function. At run time, one of the versions is chosen to execute, depending on the Intel® processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older processors and non-Intel processors. A non-Intel processor always executes the baseline code path.

You can use more than one of the feature values by combining them. For example, you can specify `-axSSE4.1,SSSE3` (Linux and macOS*) or `/QaxSSE4.1,SSSE3` (Windows). You cannot combine the old style, deprecated options and the new options. For example, you cannot specify `-axSSE4.1,T` (Linux and macOS*) or `/QaxSSE4.1,T` (Windows).

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

IDE Equivalent

Visual Studio: **Code Generation > Add Processor-Optimized Code Path**

Eclipse: **Code Generation > Add Processor-Optimized Code Path**

Xcode: **Code Generation > Add Processor-Optimized Code Path**

Alternate Options

None

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`march` compiler option
`arch` compiler option
`m` compiler option

fasynchronous-unwind-tables

Determines whether unwind information is precise at an instruction boundary or at a call boundary.

Syntax

Linux OS:

```
-fasynchronous-unwind-tables
-fno-asynchronous-unwind-tables
```

macOS:

```
-fasynchronous-unwind-tables
-fno-asynchronous-unwind-tables
```

Windows OS:

```
None
```

Arguments

None

Default

Intel® 64 architecture: `-fasynchronous-unwind-tables`	The unwind table generated is precise at an instruction boundary, enabling accurate unwinding at any instruction.
IA-32 architecture (Linux* only): `-fno-asynchronous-unwind-tables`	The unwind table generated is precise at call boundaries only.

Description

This option determines whether unwind information is precise at an instruction boundary or at a call boundary. The compiler generates an unwind table in DWARF2 or DWARF3 format, depending on which format is supported on your system.

If `-fno-asynchronous-unwind-tables` is specified, the unwind table is precise at call boundaries only. In this case, the compiler will avoid creating unwind tables for routines such as the following:

- A C++ routine that does not declare objects with destructors and does not contain calls to routines that might throw an exception.
- A C/C++ or Fortran routine compiled without `-fexceptions`, and on Intel® 64 architecture, without `-traceback`.
- A C/C++ or Fortran routine compiled with `-fexceptions` that does not contain calls to routines that might throw an exception.

IDE Equivalent

None

Alternate Options

None

See Also

fexceptions compiler option

fexceptions

Enables exception handling table generation.

Syntax

Linux OS:

`-fexceptions`

`-fno-exceptions`

macOS:

`-fexceptions`

`-fno-exceptions`

Windows OS:

`None`

Arguments

None

Default

-fno-exceptions	Exception handling table generation is disabled.

Description

This option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines. The `-fno-exceptions` option disables C++ exception handling table generation, resulting in smaller code. When this option is used, any use of C++ exception handling constructs (such as try blocks and throw statements) when a Fortran routine is in the call chain will produce an error.

IDE Equivalent

None

Alternate Options

None

fomit-frame-pointer, Oy

Determines whether EBP is used as a general-purpose register in optimizations.

Architecture Restrictions

Option `/Oy[-]` is only available on IA-32 architecture

Syntax

Linux OS:

`-fomit-frame-pointer`

`-fno-omit-frame-pointer`

macOS:

`-fomit-frame-pointer`

`-fno-omit-frame-pointer`

Windows OS:

`/Oy`

`/Oy-`

Arguments

None

Default

`-fomit-frame-pointer` or `/Oy`	EBP is used as a general-purpose register in optimizations. However, on Linux* and macOS* systems, the default is `-fno-omit-frame-pointer` if option `-O0` or `-g` is specified. On Windows* systems, the default is `/Oy-` if option `/Od` is specified.

Description

These options determine whether EBP is used as a general-purpose register in optimizations. Options `-fomit-frame-pointer` and `/Oy` allow this use. Options `-fno-omit-frame-pointer` and `/Oy-` disallow it.

Some debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-fno-omit-frame-pointer` and `/Oy-` options direct the compiler to generate code that maintains and uses EBP as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without doing the following:

- For `-fno-omit-frame-pointer`: turning off optimizations with `-O0`
- For `/Oy-`: turning off `/O1`, `/O2`, or `/O3` optimizations

The `-fno-omit-frame-pointer` option is set when you specify option `-O0` or the `-g` option. The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3`.

The `/Oy` option is set when you specify the `/O1`, `/O2`, or `/O3` option. Option `/Oy-` is set when you specify the `/Od` option.

Using the `-fno-omit-frame-pointer` or `/Oy-` option reduces the number of available general-purpose registers by 1, and can result in slightly less efficient code.

IDE Equivalent

Visual Studio: **Optimization > Omit Frame Pointers**

Eclipse: None

Xcode: **Optimization > Provide Frame Pointer**

Alternate Options

Linux and macOS*: `-fp` (this is a deprecated option)

Windows: None

See Also

momit-leaf-frame-pointer compiler option

guard

Enables the control flow protection mechanism.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/guard:`*`keyword`*

Arguments

`keyword` Specifies the the control flow protection mechanism. Possible values are:

`cf[-]` Tells the compiler to analyze control flow of valid targets for indirect calls and to insert code to verify the targets at runtime.

To explicitly disable this option, specify `/guard:cf-`.

Default

`OFF` The control flow protection mechanism is disabled.

Description

This option enables the control flow protection mechanism. It tells the compiler to analyze control flow of valid targets for indirect calls and inserts a call to a checking routine before each indirect call to verify the target of the given indirect call.

The `/guard:cf` option must be passed to both the compiler and linker.

Code compiled using `/guard:cf` can be linked to libraries and object files that are not compiled using the option.

This option has been added for Microsoft compatibility. It uses the Microsoft implementation.

IDE Equivalent

None

Alternate Options

None

hotpatch

Tells the compiler to prepare a routine for hotpatching.

Syntax

Linux OS:

`-hotpatch[=n]`

macOS:

`None`

Windows OS:

`/hotpatch[:n]`

Arguments

n	An integer specifying the number of bytes the compiler should add before the function entry point.

Default

OFF	The compiler does not prepare routines for hotpatching.

Description

This option tells the compiler to prepare a routine for hotpatching. The compiler inserts nop padding around function entry points so that the resulting image is hot patchable.

Specifically, the compiler adds nop bytes after each function entry point and enough nop bytes before the function entry point to fit a direct jump instruction on the target architecture.

If *n* is specified, it overrides the default number of bytes that the compiler adds before the function entry point.

IDE Equivalent

None

Alternate Options

None

m

Tells the compiler which features it may target, including which instruction sets it may generate.

Syntax

Linux OS and macOS:

`-mcode`

Windows OS:

`None`

Arguments

code

Indicates to the compiler a feature set that it may target, including which instruction sets it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (SSSE). Possible values are:

`avx`	May generate Intel® Advanced Vector Extensions (Intel® AVX), SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`sse4.2`	May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`sse4.1`	May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`ssse3`	May generate SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`sse3`	May generate Intel® SSE3, SSE2, and SSE instructions.
`sse2`	May generate Intel® SSE2 and SSE instructions. This value is only available on Linux systems.
`sse`	This option has been deprecated; it is now the same as specifying `ia32`.
`ia32`	Generates x86/x87 generic code that is compatible with IA-32 architecture. Disables any default extended instruction settings, and any previously set extended instruction settings. It also disables all feature-specific optimizations and instructions. This value is only available on Linux* systems using IA-32 architecture.

This compiler option also supports many of the `-m` option settings available with gcc. For more information on gcc `-m` settings, see the gcc documentation.

Default

Linux* systems:
`-msse2`
macOS* systems:
`-mssse3`

For more information on the default values, see Arguments above.

Description

This option tells the compiler which features it may target, including which instruction sets it may generate.

Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

Options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

Linux* systems: For compatibility with gcc, the compiler allows the following options but they have no effect. You will get a warning error, but the instructions associated with the name will not be generated. You should use the suggested replacement options.

gcc Compatibility Option (Linux*)	Suggested Replacement Option
`-mfma`	`-march=core-avx2`
`-mbmi, -mavx2, -mlzcnt`	`-march=core-avx2`
`-mmovbe`	`-march=atom -minstruction=movbe`
`-mcrc32, -maes, -mpclmul, -mpopcnt`	`-march=corei7`
`-mvzeroupper`	`-march=corei7-avx`
`-mfsgsbase, -mrdrnd, -mf16c`	`-march=core-avx-i`

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/arch`

See Also

`x, Qx` compiler option
`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`m32, m64` compiler option

m32, m64, Qm32, Qm64

Tells the compiler to generate code for a specific architecture.

Syntax

Linux OS:

`-m32`

`-m64`

macOS:

`-m64`

Windows OS:

`/Qm32`

/Qm64

Arguments

None

Default

OFF The compiler's behavior depends on the host system.

Description

These options tell the compiler to generate code for a specific architecture.

Option	Description
-m32 or /Qm32	Tells the compiler to generate code for IA-32 architecture.
-m64 or /Qm64	Tells the compiler to generate code for Intel® 64 architecture.

The -m64 option is the same as macOS* option -arch x86_64. This option is not related to the Intel®Fortran Compiler option arch.

On Linux* systems, these options are provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

m80387

Specifies whether the compiler can use x87 instructions.

Syntax

Linux OS:

-m80387

-mno-80387

macOS:

-m80387

-mno-80387

Windows OS:

None

Arguments

None

Default

-m80387 The compiler may use x87 instructions.

Description

This option specifies whether the compiler can use x87 instructions.

If you specify option `-mno-80387`, it prevents the compiler from using x87 instructions. If the compiler is forced to generate x87 instructions, it issues an error message.

IDE Equivalent

None

Alternate Options

-m[no-]x87

march

Tells the compiler to generate code for processors that support certain features.

Syntax

Linux OS:

`-march=`*`processor`*

macOS:

`-march=`*`processor`*

Windows OS:

`None`

Arguments

processor

Indicates to the compiler the code it may generate. Possible values are:

`amberlake`
`broadwell`
`cannonlake`
`cascadelake`
`coffeelake`
`goldmont`
`goldmont-plus`
`haswell`
`icelake-client` (or `icelake`)
`icelake-server`
`ivybridge`
`kabylake`
`knl`
`knm`
`sandybridge`
`silvermont`
`skylake`
`skylake-avx512`
`tremont`
`whiskeylake`
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name.
	Keywords `knl` and `silvermont` are only available on Linux* systems.
	Keyword `icelake` is deprecated and may be removed in a future release.

`core-avx2`	Generates code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`core-avx-i`	Generates code for processors that support the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7-avx`	Generates code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7`	Generates code for processors that support Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.
`atom`	Generates code for processors that support MOVBE instructions, depending on the setting of option `-minstruction` (Linux* and macOS*) or `/Qinstruction` (Windows*). May also generate code for SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`core2`	Generates code for the Intel® Core™2 processor family.
`pentium4m`	Generates for Intel® Pentium® 4 processors with MMX technology.

`pentium-m`
`pentium4`
`pentium3`
`pentium`
	Generates code for Intel® Pentium® processors. Value `pentium3` is only available on Linux* systems.

Default

`pentium4`	If no architecture option is specified, value `pentium4` is used by the compiler to generate code.

Description

This option tells the compiler to generate code for processors that support certain features.

If you specify both the `-ax` and `-march` options, the compiler will not generate Intel-specific instructions.

Specifying `-march=pentium4` sets `-mtune=pentium4`.

For compatibility, a number of historical *processor* values are also supported, but the generated code will not differ from the default.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

IDE Equivalent

None

Alternate Options

`-march=pentium3`	Linux: `-xSSE` macOS*: None Windows: None
`-march=pentium4` `-march=pentium-m`	Linux: `-xSSE2` macOS*: None Windows: None
`-march=core2`	Linux: `-xSSSE3` macOS*: None Windows: None

See Also

`xHost, QxHost` compiler option
`x, Qx` compiler option
`ax, Qax` compiler option
`arch` compiler option
`minstruction, Qinstruction` compiler option
`m` compiler option

masm

Tells the compiler to generate the assembler output file using a selected dialect.

Syntax

Linux OS:

`-masm=`*`dialect`*

macOS:

`None`

Windows OS:

`None`

Arguments

dialect	Is the dialect to use for the assembler output file. Possible values are:
`att`	Tells the compiler to generate the assembler output file using AT&T* syntax.
`intel`	Tells the compiler to generate the assembler output file using Intel syntax.

Default

`-masm=att`	The compiler generates the assembler output file using AT&T* syntax.

Description

This option tells the compiler to generate the assembler output file using a selected dialect.

IDE Equivalent

None

Alternate Options

None

mauto-arch, Qauto-arch

Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit.

Syntax

Linux OS and macOS:

`-mauto-arch=`*`value`*

Windows OS:

`/Qauto-arch:`*`value`*

Arguments

`value`	Is any setting you can specify for option `[Q]ax`.

Default

OFF	No additional execution path is generated.

Description

This option tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit. It also generates a baseline code path.

This option cannot be used together with any options that may require Intel-specific optimizations (such as `[Q]x` or `[Q]ax`).

IDE Equivalent

None

Alternate Options

None

See Also

ax, Qax compiler option

mbranches-within-32B-boundaries, Qbranches-within-32B-boundaries

Tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.

Syntax

Linux OS:

`-mbranches-within-32B-boundaries`

`-mno-branches-within-32B-boundaries`

macOS:

`-mbranches-within-32B-boundaries`

`-mno-branches-within-32B-boundaries`

Windows OS:

`/Qbranches-within-32B-boundaries`

`/Qbranches-within-32B-boundaries-`

Arguments

None

Default

`-mno-branches-within-32B-boundaries` or `/Qbranches-within-32B-boundaries-`	Branches and fused branches are not aligned on 32-byte boundaries.

Description

This option tells the compiler to align branches and fused branches on 32-byte boundaries for better performance.

NOTE
When you use this option, it may affect binary utilities usage experience, such as debugability.

IDE Equivalent

None

Alternate Options

None

mconditional-branch, Qconditional-branch

Lets you identify and fix code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction.

Syntax

Linux OS:

`-mconditional-branch=`*keyword*

macOS:

`-mconditional-branch=`*keyword*

Windows OS:

`/Qconditional-branch:`*keyword*

Arguments

keyword Indicates to the compiler what action to take. Possible values are:

`keep`	Tells the compiler to not attempt any vulnerable code detection or fixing. This is equivalent to not specifying the `-mconditional-branch` option.
`pattern-report`	Tells the compiler to perform a search of vulnerable code patterns in the compilation and report all occurrences to stderr.
`pattern-fix`	Tells the compiler to perform a search of vulnerable code patterns in the compilation and generate code to ensure that the identified data accesses are not executed speculatively. It will also report any fixed patterns to stderr. This setting does not guarantee total mitigation, it only fixes cases where all components of the vulnerability can be seen or determined by the compiler. The pattern detection will be more complete if advanced optimization options are specified or are in effect, such as option `O3` and option `-ipo` (or `/Qipo`).
`all-fix`	Tells the compiler to fix all of the vulnerable code so that it is either not executed speculatively, or there is no observable side-channel created from their speculative execution. Since it is a complete mitigation against Spectre variant 1 attacks, this setting will have the most run-time performance cost. In contrast to the `pattern-fix` setting, the compiler will not attempt to identify the exact conditional branches that may have led to the mis-speculated execution.
`all-fix-lfence`	This is the same as specifying setting `all-fix`.
`all-fix-cmov`	Tells the compiler to treat any path where speculative execution of a memory load creates vulnerability (if mispredicted). The compiler automatically adds mitigation code along any vulnerable paths found, but it uses a different method then the one used for `all-fix` (or `all-fix-lfence`).

This method uses CMOVcc instruction execution, which constrains speculative execution. Thus, it is used for keeping track of the predicate value, which is updated on each conditional branch.

To prevent Spectre v.1 attack, each memory load that is potentially vulnerable is bitwise ORed with the predicate to mask out the loaded value if the code is on a mispredicted path.

This is analogous to the Clang compiler's option to do Speculative Load Hardening.

This setting is only supported on Intel® 64 architecture-based systems.

Default

`-mconditional-branch=keep` and `/Qconditional-branch:keep`	The compiler does not attempt any vulnerable code detection or fixing.

Description

This option lets you identify code that may be vulnerable to speculative execution side-channel attacks, which can leak your secure data as a result of bad speculation of a conditional branch direction. Depending on the setting you choose, vulnerabilities may be detected and code may be generated to attempt to mitigate the security risk.

IDE Equivalent

Visual Studio: **Code Generation > Spectre Variant 1 Mitigation**

Eclipse: None

Xcode: None

Alternate Options

None

minstruction, Qinstruction

Determines whether MOVBE instructions are generated for certain Intel processors.

Syntax

Linux OS and macOS:

`-minstruction=[no]movbe`

Windows OS:

`/Qinstruction:[no]movbe`

Arguments

None

Default

`-minstruction=nomovbe` or `/Qinstruction:nomovbe`	The compiler does not generate MOVBE instructions for Intel Atom® processors.

Description

This option determines whether MOVBE instructions are generated for Intel Atom® processors. To use this option, you must also specify `[Q]xATOM_SSSE3` or `[Q]xATOM_SSE4.2`.

If `-minstruction=movbe` or `/Qinstruction:movbe` is specified, the following occurs:

- MOVBE instructions are generated that are specific to the Intel Atom® processor.
- Generated executables can only be run on Intel Atom® processors or processors that support Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) or Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) and MOVBE.

If `-minstruction=nomovbe` or `/Qinstruction:nomovbe` is specified, the following occurs:

- The compiler optimizes code for the Intel Atom® processor, but it does not generate MOVBE instructions.
- Generated executables can be run on non-Intel Atom® processors that support Intel® SSE3 or Intel® SSE4.2.

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option

momit-leaf-frame-pointer

Determines whether the frame pointer is omitted or kept in leaf functions.

Syntax

Linux OS:

```
-momit-leaf-frame-pointer
-mno-omit-leaf-frame-pointer
```

macOS:

```
-momit-leaf-frame-pointer
-mno-omit-leaf-frame-pointer
```

Windows OS:

```
None
```

Arguments

None

Default

Varies — If you specify option `-fomit-frame-pointer` (or it is set by default), the default is `-momit-leaf-frame-pointer`. If you specify option `-fno-omit-frame-pointer`, the default is `-mno-omit-leaf-frame-pointer`.

Description

This option determines whether the frame pointer is omitted or kept in leaf functions. It is related to option `-f[no-]omit-frame-pointer` and the setting for that option has an effect on this option.

Consider the following option combinations:

Option Combination	Result
`-fomit-frame-pointer -momit-leaf-frame-pointer` or `-fomit-frame-pointer -mno-omit-leaf-frame-pointer`	Both combinations are the same as specifying `-fomit-frame-pointer`. Frame pointers are omitted for all routines.
`-fno-omit-frame-pointer -momit-leaf-frame-pointer`	In this case, the frame pointer is omitted for leaf routines, but other routines will keep the frame pointer. This is the intended effect of option `-momit-leaf-frame-pointer`.
`-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer`	In this case, `-mno-omit-leaf-frame-pointer` is ignored since `-fno-omit-frame-pointer` retains frame pointers in all routines . This combination is the same as specifying `-fno-omit-frame-pointer`.

This option is provided for compatibility with gcc.

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **Optimization > Provide Frame Pointer For Leaf Routines**

Alternate Options

None

See Also

fomit-frame-pointer, Oy compiler option

mstringop-inline-threshold, Qstringop-inline-threshold

Tells the compiler to not inline calls to buffer manipulation functions such as memcpy and memset when the number of bytes the functions handle are known at compile time and greater than the specified value.

Syntax

Linux OS and macOS:

`-mstringop-inline-threshold=val`

Windows OS:

`/Qstringop-inline-threshold:val`

Arguments

val	Is a positive 32-bit integer. If the size is greater than *val*, the compiler will never inline it.

Default

OFF	The compiler uses its own heuristics to determine the default.

Description

This option tells the compiler to not inline calls to buffer manipulation functions such as memcpy and memset when the number of bytes the functions handle are known at compile time and greater than the specified *val*.

IDE Equivalent

None

Alternate Options

None

See Also

mstringop-strategy, Qstringop-strategy compiler option

mstringop-strategy, Qstringop-strategy

Lets you override the internal decision heuristic for the particular algorithm used when implementing buffer manipulation functions such as memcpy and memset.

Syntax

Linux OS and macOS:

`-mstringop-strategy=alg`

Windows OS:

`/Qstringop-strategy:alg`

Arguments

alg	Specifies the algorithm to use. Possible values are:	
	`const_size_loop`	Tells the compiler to expand the string operations into an inline loop when the size is known at compile time and it is not greater than threshold value. Otherwise, the compiler uses its own heuristics to decide how to implement the string operation.

`libcall`	Tells the compiler to use a library call when implementing string operations.
`rep`	Tells the compiler to use its own heuristics to decide what form of `rep movs | stos` to use when inlining string operations.

Default

varies	If optimization option `Os` is specified, the default is `rep`. Otherwise, the default is `const_size_loop`.

Description

This option lets you override the internal decision heuristic for the particular algorithm used when implementing buffer manipulation functions such as memcpy and memset.

This option may have no effect on compiler-generated string functions, for example, a call to memcpy generated by the compiler to implement an array copy or structure copy.

IDE Equivalent

None

Alternate Options

None

See Also

mstringop-inline-threshold, Qstringop-inline-threshold compiler option

Os compiler option

mtune, tune

Performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike -march).

Syntax

Linux OS and macOS:

`-mtune=`*`processor`*

Windows OS:

`/tune:`*`processor`*

Arguments

processor

Is the processor for which the compiler should perform optimizations. Possible values are:

`generic`	Optimizes code for the compiler's default behavior.

`amberlake`
`broadwell`
`cannonlake`
`cascadelake`
`coffeelake`
	Optimizes code for processors that support the specified Intel® processor or microarchitecture code name.
	Keywords `knl` and `silvermont` are only available on Windows* and Linux* systems.

`goldmont`
`goldmont-plus`
`haswell`
`icelake-client` (or `icelake`)
`icelake-server`
`ivybridge`
`kabylake`
`knl`
`knm`
`sandybridge`
`silvermont`
`skylake`
`skylake-avx512`
`tremont`
`whiskeylake`
	Keyword `icelake` is deprecated and may be removed in a future release.

`core-avx2`	Optimizes code for processors that support Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`core-avx-i`	Optimizes code for processors that support the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7-avx`	Optimizes code for processors that support Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions.
`corei7`	Optimizes code for processors that support Intel® SSE4 Efficient Accelerated String and Text Processing instructions. May also generate code for Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSE3, SSE2, SSE, and SSSE3 instructions.
`atom`	Optimizes code for processors that support MOVBE instructions, depending on the setting of option `-minstruction` (Linux* and macOS*) or `/Qinstruction` (Windows*). May also generate code for SSSE3 instructions and Intel® SSE3, SSE2, and SSE instructions.
`core2`	Optimizes for the Intel® Core™2 processor family, including support for MMX™, Intel® SSE, SSE2, SSE3 and SSSE3 instruction sets.
`pentium-mmx`	Optimizes for Intel® Pentium® with MMX technology.

`pentiumpro`	Optimizes for Intel® Pentium® Pro, Intel Pentium II, and Intel Pentium III processors.
`pentium4m`	Optimizes for Intel® Pentium® 4 processors with MMX technology.

`pentium-m`
`pentium4`
`pentium3`
`pentium`
	Optimizes code for Intel® Pentium® processors. Value `pentium3` is only available on Linux* systems.

Default

generic	Code is generated for the compiler's default behavior.

Description

This option performs optimizations for specific processors but does not cause extended instruction sets to be used (unlike `-march`).

The resulting executable is backwards compatible and generated code is optimized for specific processors. For example, code generated with `-mtune=core2` or `/tune:core2` will run correctly on 4th Generation Intel® Core™ processors, but it might not run as fast as if it had been generated using `-mtune=haswell` or `/tune:haswell`. Code generated with `-mtune=haswell` (`/tune:haswell`) or `-mtune=core-avx2` (`/tune:core-avx2`) will also run correctly on Intel® Core™2 processors, but it might not run as fast as if it had been generated using `-mtune=core2` or `/tune:core2`. This is in contrast to code generated with `-march=core-avx2` or `/arch:core-avx2`, which will not run correctly on older processors such as Intel® Core™2 processors.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

IDE Equivalent

Visual Studio: **Code Generation >Intel Processor Microarchitecture-Specific Optimization**

Eclipse: None

Xcode: **Code Generation > Intel Processor Microarchitecture-Specific Optimization**

Alternate Options

`-mtune`	Linux: `-mcpu` (this is a deprecated option) macOS*: None Windows: None

See Also

`march` compiler option

qcf-protection, Qcf-protection

Enables Control-flow Enforcement Technology (CET) protection, which defends your program from certain attacks that exploit vulnerabilities. This option offers preliminary support for CET.

Syntax

Linux OS:

`-qcf-protection[=keyword]`

macOS:

`None`

Windows OS:

`/Qcf-protection[:keyword]`

Arguments

keyword Specifies the level of protection the compiler should perform. Possible values are:

`shadow_stack`	Enables shadow stack protection.
`branch_tracking`	Enables endbranch (EB) generation.
`full`	Enables both shadow stack protection and endbranch (EB) generation. This is the same as specifying the `[q or Q]cf-protection` option with no *keyword*.
`none`	Disables Control-flow Enforcement Technology (CET) protection.

Default

`-qcf-protection=none` No Control-flow Enforcement protection is performed.

or
`/Qcf-protection:none`

Description

This option enables Control-flow Enforcement Technology (CET) protection, which defends your program from certain attacks that exploit vulnerabilities.

CET protections are enforced on processors that support CET. They are ignored on processors that do not support CET, so they are safe to use in programs that might run on a variety of processors.

Specifying `shadow_stack` helps to protect your program from return-oriented programming (ROP). Return-oriented programming (ROP) is a technique to exploit computer security defenses such as non-executable memory and code signing by gaining control of the call stack to modify program control flow and then execute certain machine instruction sequences.

Specifying `branch_tracking` helps to protect your program from call/jump-oriented programming (COP/JOP). Jump-oriented programming (JOP) is a variant of ROP that uses indirect jumps and calls to emulate return instructions. Call-oriented programming (COP) is a variant of ROP that employs indirect calls.

To get both protections, specify `[q or Q]cf-protection` with no *keyword*, or specify `-qcf-protection=full` (Linux*) or `/Qcf-protection:full` (Windows*).

> **NOTE**
> On Linux and macOS* systems, you can also specify gcc option `-fcf-protection` to enable CET features. For more information about that option, see the gcc documentation.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-fcf-protection` (supported gcc option)

Windows: None

See Also

guard compiler option

Qpatchable-addresses

Tells the compiler to generate code such that references to statically assigned addresses can be patched.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qpatchable-addresses`

Arguments

None

Default

OFF The compiler does not generate patchable addresses.

Description

This option tells the compiler to generate code such that references to statically assigned addresses can be patched with arbitrary 64-bit addresses.

Normally, the Windows* system compiler that runs on Intel® 64 architecture uses 32-bit relative addressing to reference statically allocated code and data. That assumes the code or data is within 2GB of the access point, an assumption that is enforced by the Windows object format.

However, in some patching systems, it is useful to have the ability to replace a global address with some other arbitrary 64-bit address, one that might not be within 2GB of the access point.

This option causes the compiler to avoid 32-bit relative addressing in favor of 64-bit direct addressing so that the addresses can be patched in place without additional code modifications. This option causes code size to increase, and since 32-bit relative addressing is usually more efficient than 64-bit direct addressing, you may see a performance impact.

IDE Equivalent

None

Alternate Options

None

x, Qx

Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.

Syntax

Linux OS and macOS:

`-x`*code*

Windows OS:

`/Qx`*code*

Arguments

code

Indicates to the compiler a feature set that it may target, including which instruction sets and optimizations it may generate. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`AMBERLAKE`
`BROADWELL`
`CANNONLAKE`
`CASCADELAKE`
`COFFEELAKE`
`GOLDMONT`
`GOLDMONT-PLUS`
`HASWELL`
`ICELAKE-CLIENT` (or `ICELAKE`)
`ICELAKE-SERVER`
`IVYBRIDGE`
`KABYLAKE`
`KNL`
`KNM`
`SANDYBRIDGE`
	May generate instructions for processors that support the specified Intel® processor or microarchitecture code name. Optimizes for the specified Intel® processor or microarchitecture code name.
	
	Keywords `KNL` and `SILVERMONT` are only available on Windows* and Linux* systems.
	
	Keyword `ICELAKE` is deprecated and may be removed in a future release.

SILVERMONT SKYLAKE SKYLAKE-AVX512 TREMONT WHISKEYLAKE	
COMMON-AVX512	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
CORE-AVX512	May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Doubleword and Quadword instructions, Intel® AVX-512 Byte and Word instructions and Intel® AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
CORE-AVX2	May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel® processors that support Intel® AVX2 instructions.
CORE-AVX-I	May generate the RDRND instruction, Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel® processors that support the RDRND instruction.
AVX	May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel processors that support Intel® AVX instructions.
SSE4.2	May generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions, Intel® SSE4 Vectorizing Compiler and Media Accelerator, and Intel® SSE3, SSE2, SSE, and SSSE3 instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE4.2 instructions.

`SSE4.1`	May generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel® processors. May generate Intel® SSE4.1, SSE3, SSE2, SSE, and SSSE3 instructions for Intel processors that support Intel® SSE4.1 instructions.
`ATOM_SSE4.2`	May generate MOVBE instructions for Intel® processors, depending on the setting of option `-minstruction` (Linux* and macOS*) or `/Qinstruction` (Windows*). May also generate Intel® SSE4.2, SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for Intel Atom® processors that support Intel® SSE4.2 and MOVBE instructions. This keyword is only available on Windows* and Linux* systems.
`ATOM_SSSE3`	May generate MOVBE instructions for Intel® processors, depending on the setting of option `-minstruction` (Linux* and macOS*) or `/Qinstruction` (Windows*). May also generate SSSE3, Intel® SSE3, SSE2, and SSE instructions for Intel processors. Optimizes for Intel Atom® processors that support Intel® SSE3 and MOVBE instructions. This keyword is only available on Windows* and Linux* systems.
`SSSE3`	May generate SSSE3 and Intel® SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel processors that support SSSE3 instructions. For macOS* systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
`SSE3`	May generate Intel® SSE3, SSE2, and SSE instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE3 instructions. This value is not available on macOS* systems.
`SSE2`	May generate Intel® SSE2 and SSE instructions for Intel® processors. Optimizes for Intel processors that support Intel® SSE2 instructions. This value is not available on macOS* systems.

You can also specify `Host`. For more information, see option `[Q]xHost`.

Default

Windows* systems: None Linux* systems: None macOS* systems: `SSSE3`	On Windows systems, if neither `/Qx` nor `/arch` is specified, the default is `/arch:SSE2`. On Linux systems, if neither `-x` nor `-m` is specified, the default is `-msse2`.

Description

This option tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate. It also enables optimizations in addition to Intel feature-specific optimizations.

The specialized code generated by this option may only run on a subset of Intel® processors.

The resulting executables created from these option *code* values can only be run on Intel® processors that support the indicated instruction set.

The binaries produced by these *code* values will run on Intel® processors that support the specified features.

Do not use *code* values to create binaries that will execute on a processor that is not compatible with the targeted processor. The resulting program may fail with an illegal instruction exception or display other unexpected behavior.

Compiling the main program with any of the *code* values produces binaries that display a fatal run-time error if they are executed on unsupported processors, including all non-Intel processors.

Compiler options `m` and `arch` produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel® processors.

The `-x` and `/Qx` options enable additional optimizations not enabled with options `-m` or `/arch` (nor with options `-ax` and `/Qax`).

On Windows systems, options `/Qx` and /arch are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning. Similarly, on Linux and macOS* systems, options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

NOTE
All settings except SSE2 do a CPU check. However, if you specify option -O0 (Linux* or macOS*) or option /Od (Windows*), no CPU check is performed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

IDE Equivalent

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

Eclipse: None

Xcode: **Code Generation > Intel Processor-Specific Optimization**

Alternate Options

None

See Also

`xHost, QxHost` compiler option
`ax, Qax` compiler option
`arch` compiler option
`march` compiler option
`minstruction, Qinstruction` compiler option
`m` compiler option

xHost, QxHost

Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

Syntax

Linux OS and macOS:

`-xHost`

Windows OS:

`/QxHost`

Arguments

None

Default

Windows* systems: None Linux* systems: None macOS* systems: `-xSSSE3`	On Windows systems, if neither `/Qx` nor `/arch` is specified, the default is `/arch:SSE2`. On Linux systems, if neither `-x` nor `-m` is specified, the default is `-msse2`.

Description

This option tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.

The instructions generated by this compiler option differ depending on the compilation host processor.

The following table describes the effects of specifying the `[Q]xHost` option and it tells whether the resulting executable will run on processors different from the host processor.

Descriptions in the table refer to Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® Advanced Vector Extensions (Intel® AVX), Intel® Streaming SIMD Extensions (Intel® SSE), and Supplemental Streaming SIMD Extensions (SSSE).

Instruction Set of Host Processor	Effects When the -xHost or /QxHost Compiler Option is Specified
Intel® AVX2	When compiling on Intel® processors: Corresponds to option `[Q]xCORE-AVX2`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® AVX2 instructions. When compiling on non-Intel processors: Corresponds to option `-march=core-avx2` (Linux* and macOS*) or `/arch:CORE-AVX2` (Windows*). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® AVX2 instructions.. You may see a run-time error if the run-time processor does not support Intel® AVX2 instructions.
Intel® AVX	When compiling on Intel® processors: Corresponds to option `[Q]xAVX`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® AVX instructions. When compiling on non-Intel processors: Corresponds to option `-mavx` (Linux and macOS*) or `/arch:AVX` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® AVX instructions. You may see a run-time error if the run-time processor does not support Intel® AVX instructions.
Intel® SSE4.2	When compiling on Intel® processors: Corresponds to option `[Q]xSSE4.2`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE4.2 instructions. When compiling on non-Intel processors: Corresponds to option `-msse4.2` (Linux and macOS*) or `/arch:SSE4.2` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE4.2 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE4.2 instructions.
Intel® SSE4.1	When compiling on Intel® processors: Corresponds to option `[Q]xSSE4.1`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE4.1 instructions. When compiling on non-Intel processors: Corresponds to option `-msse4.1` (Linux and macOS*) or `/arch:SSE4.1` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE4.1 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE4.1 instructions.
SSSE3	When compiling on Intel® processors:

Instruction Set of Host Processor	Effects When the -xHost or /QxHost Compiler Option is Specified
	Corresponds to option `[Q]xSSSE3`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support SSSE3 instructions. When compiling on non-Intel processors: Corresponds to option `-mssse3` (Linux and macOS*) or `/arch:SSSE3` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least SSSE3 instructions. You may see a run-time error if the run-time processor does not support SSSE3 instructions.
Intel® SSE3	When compiling on Intel® processors: Corresponds to option `[Q]xSSE3`. The generated executable will not run on non-Intel processors and it will not run on Intel® processors that do not support Intel® SSE3 instructions. When compiling on non-Intel processors: Corresponds to option `-msse3` (Linux and macOS*) or `/arch:SSE3` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE3 instructions. You may see a warning run-time error if the run-time processor does not support Intel® SSE3 instructions.
Intel® SSE2	When compiling on Intel® processors or non-Intel processors: Corresponds to option `-msse2` (Linux and macOS*) or `/arch:SSE2` (Windows). The generated executable will run on Intel® processors and non-Intel processors that support at least Intel® SSE2 instructions. You may see a run-time error if the run-time processor does not support Intel® SSE2 instructions.

For more information on other settings for option `[Q]x`, see that option description.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

IDE Equivalent

Visual Studio: **Code Generation > Intel Processor-Specific Optimization**

Eclipse: None

Xcode: **Code Generation > Intel Processor-Specific Optimization**

Alternate Options

None

See Also

`x, Qx` compiler option
`ax, Qax` compiler option
`m` compiler option
`arch` compiler option

Interprocedural Optimization (IPO) Options

ffat-lto-objects

Determines whether a fat link-time optimization (LTO) object, containing both intermediate language and object code, is generated during an interprocedural optimization compilation (-c –ipo).

Syntax

Linux OS:

`-ffat-lto-objects`

`-fno-fat-lto-objects`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-ffat-lto-objects` When `-c -ipo` is specified, the compiler generates a fat link-time optimization (LTO) object that has both a true object and a discardable intermediate language section.

Description

This option determines whether a fat link time optimization (LTO) object, containing both intermediate language and object code, is generated during an interprocedural optimization compilation (`-c -ipo`).

During an interprocedural optimization compilation (`-c -ipo`), the following occurs:

- If you specify `-ffat-lto-objects`, the compiler generates a fat link-time optimization (LTO) object that has both a true object and a discardable intermediate language section. This enables both link-time optimization (LTO) linking and normal linking.
- If you specify `-fno-fat-lto-objects`, the compiler generates a fat link-time optimization (LTO) object that only has a discardable intermediate language section; no true object is generated. This option may improve compilation time.

Note that these files will be inserted into archives in the form in which they were created.

This option is provided for compatibility with gcc. For more information about this option, see the gcc documentation.

NOTE
Intel's intermediate language representation is not compatible with gcc's intermediate language representation.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

ip, Qip

Determines whether additional interprocedural optimizations for single-file compilation are enabled.

Syntax

Linux OS and macOS:

`-ip`

`-no-ip`

Windows OS:

`/Qip`

`/Qip-`

Arguments

None

Default

OFF Some limited interprocedural optimizations occur, including inline function expansion for calls to functions defined within the current source file. These optimizations are a subset of full intra-file interprocedural optimizations. Note that this setting is not the same as `-no-ip` (Linux* and macOS*) or `/Qip-` (Windows*).

Description

This option determines whether additional interprocedural optimizations for single-file compilation are enabled.

The `[Q]ip` option enables additional interprocedural optimizations for single-file compilation.

Options `-no-ip` (Linux and macOS*) and `/Qip-` (Windows) may not disable inlining. To ensure that inlining of user-defined functions is disabled, specify `-inline-level=0`or `-fno-inline` (Linux and macOS*), or specify `/Ob0` (Windows).

IDE Equivalent

Visual Studio: **Optimization > Interprocedural Optimization**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`finline-functions` compiler option

ip-no-inlining, Qip-no-inlining

Disables full and partial inlining enabled by interprocedural optimization options.

Syntax

Linux OS and macOS:

`-ip-no-inlining`

Windows OS:

`/Qip-no-inlining`

Arguments

None

Default

OFF Inlining enabled by interprocedural optimization options is performed.

Description

This option disables full and partial inlining enabled by the following interprocedural optimization options:

- On Linux* and macOS* systems: `-ip` or `-ipo`
- On Windows* systems: `/Qip`, `/Qipo`, or `/Ob2`

It has no effect on other interprocedural optimizations.

On Windows systems, this option also has no effect on user-directed inlining specified by option `/Ob1`.

IDE Equivalent

None

Alternate Options

None

ip-no-pinlining, Qip-no-pinlining

Disables partial inlining enabled by interprocedural optimization options.

Syntax

Linux OS and macOS:

`-ip-no-pinlining`

Windows OS:

`/Qip-no-pinlining`

Arguments

None

Default

OFF Inlining enabled by interprocedural optimization options is performed.

Description

This option disables partial inlining enabled by the following interprocedural optimization options:

- On Linux* and macOS* systems: `-ip` or `-ipo`
- On Windows* systems: `/Qip` or `/Qipo`

It has no effect on other interprocedural optimizations.

IDE Equivalent

None

Alternate Options

None

ipo, Qipo

Enables interprocedural optimization between files.

Syntax

Linux OS:

`-ipo[n]`

`-no-ipo`

macOS:

`-ipo[n]`

`-no-ipo`

Windows OS:

`/Qipo[n]`

`/Qipo-`

Arguments

n	Is an optional integer that specifies the number of object files the compiler should create. The integer must be greater than or equal to 0.

Default

`-no-ipo` or `/Qipo-` Multifile interprocedural optimization is not enabled.

Description

This option enables interprocedural optimization between files. This is also called multifile interprocedural optimization (multifile IPO) or Whole Program Optimization (WPO).

When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

You cannot specify the names for the files that are created.

If *n* is 0, the compiler decides whether to create one or more object files based on an estimate of the size of the application. It generates one object file for small applications, and two or more object files for large applications.

If *n* is greater than 0, the compiler generates *n* object files, unless *n* exceeds the number of source files (*m*), in which case the compiler generates only *m* object files.

If you do not specify *n*, the default is 0.

NOTE
When you specify option `[Q]ipo` with option `[q or Q]opt-report`, IPO information is generated in the compiler optimization report at link time. After linking, you will see a report named ipo_out.optrpt in the folder where you linked all of the object files.

IDE Equivalent

Visual Studio: **Optimization > Interprocedural Optimization**

General > Whole Program Optimization

Eclipse: None

Xcode: None

Alternate Options

None

ipo-c, Qipo-c

Tells the compiler to optimize across multiple files and generate a single object file.

Syntax

Linux OS and macOS:

`-ipo-c`

Windows OS:

`/Qipo-c`

Arguments

None

Default

OFF The compiler does not generate a multifile object file.

Description

This option tells the compiler to optimize across multiple files and generate a single object file (named ipo_out.o on Linux* and macOS* systems; ipo_out.obj on Windows* systems).

It performs the same optimizations as the `[Q]ipo` option, but compilation stops before the final link stage, leaving an optimized object file that can be used in further link steps.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

ipo-jobs, Qipo-jobs

Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).

Syntax

Linux OS and macOS:

`-ipo-jobs`*n*

Windows OS:

`/Qipo-jobs:`*n*

Arguments

n	Is the number of commands (jobs) to run simultaneously. The number must be greater than or equal to 1.

Default

`-ipo-jobs1` or `/Qipo-jobs:1`	One command (job) is executed in an interprocedural optimization parallel build.

Description

This option specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO). It should only be used if the link-time compilation is generating more than one object. In this case, each object is generated by a separate compilation, which can be done in parallel.

This option can be affected by the following compiler options:

- `[Q]ipo` when applications are large enough that the compiler decides to generate multiple object files.
- `[Q]ipo`*n* when *n* is greater than 1.
- `[Q]ipo-separate`

Caution
Be careful when using this option. On a multi-processor system with lots of memory, it can speed application build time. However, if *n* is greater than the number of processors, or if there is not enough memory to avoid thrashing, this option can increase application build time.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

`ipo-separate, Qipo-separate` compiler option

ipo-S, Qipo-S

Tells the compiler to optimize across multiple files and generate a single assembly file.

Syntax

Linux OS and macOS:

`-ipo-S`

Windows OS:

`/Qipo-S`

Arguments

None

Default

OFF The compiler does not generate a multifile assembly file.

Description

This option tells the compiler to optimize across multiple files and generate a single assembly file (named ipo_out.s on Linux* and macOS* systems; ipo_out.asm on Windows* systems).

It performs the same optimizations as the `[Q]ipo` option, but compilation stops before the final link stage, leaving an optimized assembly file that can be used in further link steps.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

ipo-separate, Qipo-separate

Tells the compiler to generate one object file for every source file.

Syntax

Linux OS:

`-ipo-separate`

macOS:

`None`

Windows OS:

`/Qipo-separate`

Arguments

None

Default

OFF The compiler decides whether to create one or more object files.

Description

This option tells the compiler to generate one object file for every source file. It overrides any `[Q]ipo` option specification.

IDE Equivalent

None

Alternate Options

None

See Also

`ipo, Qipo` compiler option

Advanced Optimization Options

ansi-alias, Qansi-alias

Tells the compiler to assume certain rules of the Fortran standard regarding aliasing and array bounds.

Syntax

Linux OS and macOS:

`-ansi-alias`

`-no-ansi-alias`

Windows OS:

`/Qansi-alias`

`/Qansi-alias-`

Arguments

None

Default

`-ansi-alias` Programs adhere to the Fortran standard's rules regarding aliasing and array bounds.
or
`/Qansi-alias`

Description

This option tells the compiler to assume certain rules of the Fortran standard regarding aliasing and array bounds.

It tells the compiler to assume that the program adheres to the following rules of the Fortran standard:

- Arrays cannot be accessed outside of declared bounds.
- A dummy argument may have its definition status changed only through that dummy argument, unless it has the TARGET attribute.

This option is similar to option `assume nodummy_aliases` with the additional restriction on array bounds.

If `-no-ansi-alias` (Linux* and macOS*) or `/Qansi-alias-` (Windows*) is specified, the compiler assumes that the program might not follow the Fortran standard's rules regarding dummy argument aliasing and array bounds; this can possibly affect performance.

IDE Equivalent

None

Alternate Options

None

See Also

`assume` compiler option, setting [no]dummy_aliases

coarray, Qcoarray

Enables the coarray feature.

Syntax

Linux OS:

`-coarray[=keyword]`

macOS:

`None`

Windows OS:

`/Qcoarray[:keyword]`

Arguments

keyword Specifies the memory system where the coarrays will be implemented. Possible values are:

`shared`	Indicates a shared memory system. This is the default.
`distributed`	Indicates a distributed memory system.
`single`	Indicates a configuration where the image does not contain self-replication code. This results in an executable with a single running image. This configuration can be useful for debugging purposes, even though there are no inter-image interactions.

Default

OFF Coarrays are not enabled without specifying this option.

Description

This option enables the coarray feature of the Fortran 2008 Standard. It enables any coarray syntax in your program. If this option is not specified, coarray syntax is rejected.

It also tells the driver to link against appropriate libraries, and to create the appropriate executables.

Only one *keyword* can be in effect. If you specify more than one *keyword*, the last one specified takes precedence. However, if *keyword* single is specified anywhere on the command line, it takes precedence.

You can specify option `[Q]coarray-num-images` to specify the default number of images that can be used to run a coarray executable. If you do not specify that option, you get the number of execution units on the current system.

You can specify the `[Q]coarray-config-file` option to specify the name of a Message Passing Interface (MPI) configuration file.

Options `[Q]coarray-num-images` and `[Q]coarray-config-file` are valid for all *keyword* values.

NOTE
32-bit coarrays are deprecated and will be removed in a future release.

IDE Equivalent

Visual Studio: **Language > Enable Coarrays**

Eclipse: None

Xcode: None

Alternate Options

None

Example

The following command runs a coarray program on shared memory using *n* images:

```
/Qcoarray /Qcoarray-num-images:n          ! Windows systems
```

```
-coarray -coarray-num-images=n            ! Linux systems
```

The following command runs a coarray program on distributed memory using *n* images:

```
/Qcoarray:distributed /Qcoarray-num-images:n          ! Windows systems
```

```
-coarray=distributed -coarray-num-images=n            ! Linux systems
```

The following command runs a coarray program on shared memory using the MPI configuration file specified by *filename*:

```
/Qcoarray:shared /Qcoarray-config-file:filename        ! Windows systems
```

```
-coarray=shared -coarray-config-file=filename          ! Linux systems
```

The following commands illustrate precedence:

Linux* systems:

```
-coarray=single -coarray=shared               ! single takes precedence (single always takes
precedence)
```

Windows* systems:

```
/Qcoarray:distributed /Qcoarray:shared        ! shared takes precedence (last one specified)
```

```
/Qcoarray:single /Qcoarray:shared             ! single takes precedence (single always takes
precedence)
```

See Also

`coarray-num-images, Qcoarray-num-images` compiler option

`coarray-config-file, Qcoarray-config-file` compiler option

Coarrays
Using Coarrays

coarray-config-file, Qcoarray-config-file

Specifies the name of a Message Passing Interface (MPI) configuration file.

Syntax

Linux OS:

`-coarray-config-file=`*`filename`*

macOS:

`None`

Windows OS:

`/Qcoarray-config-file:`*`filename`*

Arguments

filename	Is the name of the MPI configuration file. You can specify a path.

Default

OFF	When coarrays are enabled, the compiler uses default settings for MPI.

Description

This option specifies the name of a Message Passing Interface (MPI) configuration file. This file is used by the compiler when coarrays are processed; it configures the MPI for multi-node operations.

This option has no affect unless you also specify the `[Q]coarray` option, which is required to create the coarray executable.

Note that when a setting is specified in environment variable FOR_COARRAY_CONFIG_FILE, it overrides the compiler option setting.

IDE Equivalent

Visual Studio: **Language > MPI Configuration File**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`coarray, Qcoarray` compiler option

coarray-num-images, Qcoarray-num-images

Specifies the default number of images that can be used to run a coarray executable.

Syntax

Linux OS:

`-coarray-num-images=n`

macOS:

`None`

Windows OS:

`/Qcoarray-num-images:n`

Arguments

n	Is the default number of images. The limit is determined from the system configuration.

Default

OFF	The number of images is determined at run-time.

Description

This option specifies the default number of images that can be used to run a coarray executable.

This option has no affect unless you also specify the `[Q]coarray` option. This option is required to create the coarray executable.

You can specify option `[Q]coarray-num-images` to specify the default number of images that can be used to run a coarray executable. If you do not specify that option, you get the number of execution units on the current system.

Note that when a setting is specified in environment variable FOR_COARRAY_NUM_IMAGES, it overrides the compiler option setting.

IDE Equivalent

Visual Studio: **Language > Coarray Images**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`coarray, Qcoarray` compiler option

complex-limited-range, Qcomplex-limited-range

Determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

Syntax

Linux OS and macOS:

`-complex-limited-range`

`-no-complex-limited-range`

Windows OS:

`/Qcomplex-limited-range`

`/Qcomplex-limited-range-`

Arguments

None

Default

`-no-complex-limited-range` or `/Qcomplex-limited-range-`	Basic algebraic expansions of some arithmetic operations involving data of type COMPLEX are disabled.

Description

This option determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

When the option is enabled, this can cause performance improvements in programs that use a lot of COMPLEX arithmetic. However, values at the extremes of the exponent range may not compute correctly.

IDE Equivalent

Visual Studio: **Floating point > Limit COMPLEX Range**

Eclipse: None

Xcode: **Floating point > Limit COMPLEX Range**

Alternate Options

None

guide, Qguide

Lets you set a level of guidance for auto-vectorization, auto parallelism, and data transformation.

Syntax

Linux OS and macOS:

`-guide[=n]`

Windows OS:

`/Qguide[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for parallelism, vectorization, and data transformation.

Description

This option lets you set a level of guidance (advice) for auto-vectorization, auto parallelism, and data transformation. It causes the compiler to generate messages suggesting ways to improve these optimizations.

When this option is specified, the compiler does not produce any objects or executables.

You must also specify the `[Q]parallel` option to receive auto parallelism guidance.

You can set levels of guidance for the individual guide optimizations by specifying one of the following options:

`[Q]guide-data-trans`	Provides guidance for data transformation.
`[Q]guide-par`	Provides guidance for auto parallelism.
`[Q]guide-vec`	Provides guidance for auto-vectorization.

If you specify the `[Q]guide` option and also specify one of the options setting a level of guidance for an individual guide optimization, the value set for the individual guide optimization will override the setting specified in `[Q]guide`.

If you do not specify `[Q]guide`, but specify one of the options setting a level of guidance for an individual guide optimization, option `[Q]guide` is enabled with the greatest value passed among any of the three individual guide optimizations specified.

In debug mode, this option has no effect unless option `O2` (or higher) is explicitly specified in the same command line.

NOTE
The compiler speculatively performs optimizations as part of guide analysis. As a result, when you use guided auto-parallelism options with options that produce vectorization or auto-parallelizer reports (such as option `[q or Q]opt-report`), the compiler generates "LOOP WAS VECTORIZED" or similar messages as if the compilation was performed with the recommended changes.

When compilation is performed with the `[Q]guide` option, you should use extra caution when interpreting vectorizer diagnostics and auto-parallelizer diagnostics.

NOTE
You can specify `[Q]diag-disable` to prevent the compiler from issuing one or more diagnostic messages.

IDE Equivalent

Visual Studio: **Diagnostics > Guided Auto Parallelism > Guided Auto Parallelism Analysis**

Eclipse: None

Xcode: **Diagnostics > Enable Guided Auto Parallelism Analysis**

Alternate Options

None

See Also

`guide-data-trans, Qguide-data-trans` compiler option

guide-par, Qguide-par compiler option
guide-vec, Qguide-vec compiler option
guide-file, Qguide-file compiler option
guide-file-append, Qguide-file-append compiler option
guide-opts, Qguide-opts compiler option
diag, Qdiag compiler option
qopt-report, Qopt-report compiler option

guide-data-trans, Qguide-data-trans

Lets you set a level of guidance for data transformation.

Syntax

Linux OS and macOS:

`-guide-data-trans[=n]`

Windows OS:

`/Qguide-data-trans[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for data transformation.

Description

This option lets you set a level of guidance for data transformation. It causes the compiler to generate messages suggesting ways to improve that optimization.

IDE Equivalent

None

Alternate Options

None

See Also

guide, Qguide compiler option
guide-par, Qguide-par compiler option
guide-vec, Qguide-vec compiler option
guide-file, Qguide-file compiler option

guide-file, Qguide-file

Causes the results of guided auto parallelism to be output to a file.

Syntax

Linux OS and macOS:

`-guide-file[=filename]`

Windows OS:

`/Qguide-file[:filename]`

Arguments

filename	Is the name of the file for output. It can include a path.

Default

OFF	Messages that are generated by guided auto parallelism are output to stderr.

Description

This option causes the results of guided auto parallelism to be output to a file.

This option is ignored unless you also specify one or more of the following options:

- `[Q]guide`
- `[Q]guide-vec`
- `[Q]guide-data-trans`
- `[Q]guide-par`

If you do not specify a path, the file is placed in the current working directory.

If there is already a file named *filename*, it will be overwritten.

You can include a file extension in *filename*. For example, if *file.txt* is specified, the name of the output file is *file.txt*. If you do not provide a file extension, the name of the file is *filename.guide*.

If you do not specify *filename*, the name of the file is *name-of-the-first-source-file.guide*. This is also the name of the file if the name specified for *filename* conflicts with a source file name provided in the command line.

NOTE
If you specify the `[Q]guide-file` option and you also specify option `[Q]guide-file-append`, the last option specified on the command line takes precedence.

IDE Equivalent

Visual Studio: **Diagnostics > Guided Auto Parallelism > Emit Guided Auto Parallelism Diagnostics to File**

Diagnostics > Guided Auto Parallelism Diagnostics File

Eclipse: None

Xcode: **Diagnostics > Emit Guided Auto Parallelism diagnostics to File**

Diagnostics > Guided Auto Parallelism Report File

Alternate Options

None

Example

The following example shows how to cause guided auto parallelism messages to be output to a file named *my_guided_autopar.guide*:

```
-guide-file=my_guided_autopar       ! Linux and macOS* systems
/Qguide-file:my_guided_autopar      ! Windows systems
```

See Also

`guide, Qguide` compiler option

`guide-file-append, Qguide-file-append` compiler option

guide-file-append, Qguide-file-append

Causes the results of guided auto parallelism to be appended to a file.

Syntax

Linux OS and macOS:

`-guide-file-append[=`*filename*`]`

Windows OS:

`/Qguide-file-append[:`*filename*`]`

Arguments

filename	Is the name of the file to be appended to. It can include a path.

Default

OFF	Messages that are generated by guided auto parallelism are output to stderr.

Description

This option causes the results of guided auto parallelism to be appended to a file.

This option is ignored unless you also specify one or more of the following options:

- `[Q]guide`
- `[Q]guide-vec`
- `[Q]guide-data-trans`
- `[Q]guide-par`

If you do not specify a path, the compiler looks for *filename* in the current working directory.

If *filename* is not found, then a new file with that name is created in the current working directory.

If you do not specify a file extension, the name of the file is *filename.guide*.

If the name specified for *filename* conflicts with a source file name provided in the command line, the name of the file is *name-of-the-first-source-file.guide*.

NOTE
If you specify the `[Q]guide-file-append` option and you also specify option `[Q]guide-file`, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to cause guided auto parallelism messages to be appended to a file named *my_messages.txt*:

```
-guide-file-append=my_messages.txt       ! Linux and macOS* systems
/Qguide-file-append:my_messages.txt      ! Windows systems
```

See Also

`guide, Qguide` compiler option

`guide-file, Qguide-file` compiler option

guide-opts, Qguide-opts

Tells the compiler to analyze certain code and generate recommendations that may improve optimizations.

Syntax

Linux OS and macOS:

`-guide-opts=`*`string`*

Windows OS:

`/Qguide-opts:`*`string`*

Arguments

string

Is the text denoting the code to analyze. The string must appear within quotes. It can take one or more of the following forms:

filename
filename, *routine*
filename, *range* [, *range*]...
filename, *routine*, *range* [, *range*]...

If you specify more than one of the above forms in a string, a semicolon must appear between each form. If you specify more than one *range* in a string, a comma must appear between each *range*. Optional blanks can follow each parameter in the forms above and they can also follow each form in a string.

filename

Specifies the name of a file to be analyzed. It can include a path.

If you do not specify a path, the compiler looks for filename in the current working directory.

routine	Specifies the name of a routine to be analyzed. You can include an identifying argument. The name, including any argument, must be enclosed in single quotes. The compiler tries to uniquely identify the routine that corresponds to the specified routine name. It may select multiple routines to analyze, especially if the following is true: • More than one routine has the specified routine name, so the routine cannot be uniquely identified. • No argument information has been specified to narrow the number of routines selected as matches.
range	Specifies a range of line numbers to analyze in the file or routine specified. The *range* must be specified in integers in the form: *first_line_number-last_line_number* The hyphen between the line numbers is required.

Default

OFF	You do not receive guidance on how to improve optimizations. However, if you specify the `[Q]guide` option, the compiler analyzes and generates recommendations for all the code in an application

Description

This option tells the compiler to analyze certain code and generate recommendations that may improve optimizations.

This option is ignored unless you also specify one or more of the following options:

- `[Q]guide`
- `[Q]guide-vec`
- `[Q]guide-data-trans`
- `[Q]guide-par`

When the `[Q]guide-opts` option is specified, a message is output that includes which parts of the input files are being analyzed. If a routine is selected to be analyzed, the complete routine name will appear in the generated message.

When inlining is involved, you should specify callee line numbers. Generated messages also use callee line numbers.

IDE Equivalent

Visual Studio: **Diagnostics > Guided Auto Parallelism > Guided Auto Parallelism Code Selection Options**

Eclipse: None

Xcode: **Diagnostics > Guided Auto Parallelism Code Selection**

Alternate Options

None

Example

Consider the following:

```
Linux*: -guide-opts="m.f, 1-10; m2.f90, 1-40, 50-90, 100-200; m5.f, 300-400; x.f, 'funca(j)',
22-44, 55-77, 88-99; y.f, 'subrb'"
```

```
Windows*: /Quide-opts="m.f, 1-10; m2.f90, 1-40, 50-90, 100-200; m5.f, 300-400; x.f, 'funca(j)',
22-44, 55-77, 88-99; y.f, 'subrb'"
```

The above command causes the following to be analyzed:

file m.f, line numbers 1 to 10
file m2.f90, line numbers 1 to 40, 50 to 90, and 100 to 200
file m5.f, line numbers 300 to 400
file x.f, function funca with argument (j), line numbers 22 to 44, 55 to 77, and 88 to 99
file y.f, subroutine subrb

See Also

`guide, Qguide` compiler option

`guide-data-trans, Qguide-data-trans` compiler option

`guide-par, Qguide-par` compiler option

`guide-vec, Qguide-vec` compiler option

`guide-file, Qguide-file` compiler option

guide-par, Qguide-par

Lets you set a level of guidance for auto parallelism.

Syntax

Linux OS and macOS:

`-guide-par[=n]`

Windows OS:

`/Qguide-par[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for parallelism.

Description

This option lets you set a level of guidance for auto parallelism. It causes the compiler to generate messages suggesting ways to improve that optimization.

You must also specify the `[Q]parallel` option to receive auto parallelism guidance.

IDE Equivalent

None

Alternate Options

None

See Also

`guide, Qguide` compiler option
`guide-data-trans, Qguide-data-trans` compiler option
`guide-vec, Qguide-vec` compiler option
`guide-file, Qguide-file` compiler option

guide-vec, Qguide-vec

Lets you set a level of guidance for auto-vectorization.

Syntax

Linux OS and macOS:

`-guide-vec[=n]`

Windows OS:

`/Qguide-vec[:n]`

Arguments

n	Is an optional value specifying the level of guidance to be provided. The values available are 1 through 4. Value 1 indicates a standard level of guidance. Value 4 indicates the most advanced level of guidance. If *n* is omitted, the default is 4.

Default

OFF	You do not receive guidance about how to improve optimizations for vectorization.

Description

This option lets you set a level of guidance for auto-vectorization. It causes the compiler to generate messages suggesting ways to improve that optimization.

IDE Equivalent

None

Alternate Options

None

See Also

`guide, Qguide` compiler option

`guide-data-trans, Qguide-data-trans` compiler option

`guide-par, Qguide-par` compiler option

`guide-file, Qguide-file` compiler option

heap-arrays

Puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.

Syntax

Linux OS and macOS:

`-heap-arrays [size]`

`-no-heap-arrays`

Windows OS:

`/heap-arrays[:size]`

`/heap-arrays-`

Arguments

size	Is an integer value representing the size of the arrays in kilobytes. Arrays smaller than *size* are put on the stack.

Default

`-no-heap-arrays` The compiler puts automatic arrays and temporary arrays in the stack storage area.

or
`/heap-arrays-`

Description

This option puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.

When this option is specified, automatic (temporary) arrays that have a compile-time size greater than the value specified for *size* are put on the heap, rather than on the stack. If the compiler cannot determine the size at compile time, it always puts the automatic array on the heap.

If *size* is specified, the value is only used when the total size of the temporary array or automatic array can be determined at compile time, using compile-time constants. Any arrays known at compile-time to be larger than *size* are allocated on the heap instead of the stack. For example, if 10 is specified for *size*:

- All automatic and temporary arrays equal to or larger than 10 KB are put on the heap.
- All automatic and temporary arrays smaller than 10 KB are put on the stack.

If *size* is omitted, and the size of the temporary array or automatic array cannot be determined at compile time, it is assumed that the total size is greater than *size* and the array is allocated on the heap.

On Windows, you can use compiler option `/F` to tell the linker to increase the size of the run-time stack to allow for large objects on the stack.

On Linux and macOS*, you can use the shell command `unlimit` to increase the size of the run-time stack before execution.

IDE Equivalent

Visual Studio: **Optimization > Heap Arrays**

Eclipse: None

Xcode: **Data > Allocate Automatics to the Heap**

Alternate Options

None

Example

In Fortran, an automatic array gets its size from a run-time expression. For example:

```
RECURSIVE SUBROUTINE F( N )
INTEGER :: N
REAL :: X ( N )     ! an automatic array
REAL :: Y ( 1000 )  ! an explicit-shape local array on the stack
```

Array X in the example above is affected by the `heap-array` option; array Y is not.

Temporary arrays are often created before making a routine call, or when an array operation detects overlap. For example:

```
integer a(10000)
a(2:) = a(1:ubound(a,dim=1)-1)
```

In this case, the array assignment uses a temporary intermediate array because there is clearly an overlap between the right hand side and the left hand side of the assignment.

If you specify the `heap-arrays` option, the compiler creates the temporary array on the heap.

If you specify the `heap-arrays` option with *size* 50, the compiler creates the temporary array on the stack. This is because the size of the temporary intermediate array can be determined at compile time (40Kb), and it's size is less than the *size* value.

In the following example, a contiguous array is created from the array slice declaration and passed on:

```
call somesub(a(1:10000:2))
```

If you specify the `heap-arrays` option, the compiler creates the temporary array on the heap.

If you specify the `heap-arrays` option with *size* 25, the compiler creates the temporary array on the stack. This is because the size of the temporary intermediate array at compile time is only 20Kb.

See Also

`F` compiler option

mkl, Qmkl

Tells the compiler to link to certain libraries in the Intel® Math Kernel Library (Intel® MKL). On Windows systems, you must specify this option at compile time.

Syntax

Linux OS:

`-mkl[=lib]`

macOS:

`-mkl[=lib]`

Windows OS:

`/Qmkl[:lib]`

Arguments

lib	Indicates which Intel® MKL library files should be linked. Possible values are:	
	`parallel`	Tells the compiler to link using the threaded libraries in the Intel® MKL. This is the default if the option is specified with no *lib*.
	`sequential`	Tells the compiler to link using the sequential libraries in the Intel® MKL.
	`cluster`	Tells the compiler to link using the cluster-specific libraries and the sequential libraries in the Intel® MKL. Cluster-specific libraries are not available for macOS*.

Default

OFF	The compiler does not link to the Intel® MKL.

Description

This option tells the compiler to link to certain libraries in the Intel® Math Kernel Library (Intel® MKL).

On Linux* and macOS* systems, dynamic linking is the default when you specify `-mkl`. To link with Intel® MKL statically, you must specify:

```
-mkl -static-intel
```

On Windows* systems, static linking is the default when you specify `/Qmkl`. To link with Intel® MKL dynamically, you must specify:

```
/Qmkl /libs:dll or /Qmkl /MD
```

For more information about using MKL libraries, see the article in Intel® Developer Zone titled: *Intel® Math Kernel Library Link Line Advisor*, which is located in https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor .

NOTE
On Windows* systems, this option adds directives to the compiled code, which the linker then reads without further input from the driver. On Linux* and macOS* systems, the driver must add the library names explicitly to the link command.

IDE Equivalent

Visual Studio: **Libraries > Use Intel(R) Math Kernel Library**

Eclipse: None

Xcode: **Libraries > Use Intel(R) Math Kernel Library**

Alternate Options

None

See Also

`static-intel` compiler option
`MD` compiler option
`libs` compiler option

pad, Qpad

Enables the changing of the variable and array memory layout.

Syntax

Linux OS and macOS:

`-pad`

`-nopad`

Windows OS:

`/Qpad`

`/Qpad-`

Arguments

None

Default

`-nopad` Variable and array memory layout is performed by default methods.
or `/Qpad-`

Description

This option enables the changing of the variable and array memory layout.

This option is effectively not different from the `align` option when applied to structures and derived types. However, the scope of `pad` is greater because it applies also to common blocks, derived types, sequence types, and structures.

IDE Equivalent

None

Alternate Options

None

See Also

`align` compiler option

qopt-args-in-regs, Qopt-args-in-regs

Determines whether calls to routines are optimized by passing arguments in registers instead of on the stack.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-qopt-args-in-regs[=keyword]`

macOS:

`None`

Windows OS:

`/Qopt-args-in-regs[:keyword]`

Arguments

keyword Specifies whether the optimization should be performed and under what conditions. Possible values are:

`none`	The optimization is not performed. No arguments are passed in registers. They are put on the stack.
`seen`	Causes arguments to be passed in registers when they are passed to routines whose definition can be seen in the same compilation unit.
`all`	Causes arguments to be passed in registers, whether they are passed to routines whose definition can be seen in the same compilation unit, or not. This value is only available on Linux* systems.

Default

`-qopt-args-in-regs=seen` or `/Qopt-args-in-regs:seen` Arguments are passed in registers when they are passed to routines whose definition is seen in the same compilation unit.

Description

This option determines whether calls to routines are optimized by passing arguments in registers instead of on the stack. It also indicates the conditions when the optimization will be performed.

This option can improve performance for Application Binary Interfaces (ABIs) that require arguments to be passed in memory and compiled without interprocedural optimization (IPO).

Note that on Linux* systems, if `all` is specified, a small overhead may be paid when calling "unseen" routines that have not been compiled with the same option. This is because the call will need to go through a "thunk" to ensure that arguments are placed back on the stack where the callee expects them.

IDE Equivalent

None

Alternate Options

None

qopt-assume-safe-padding, Qopt-assume-safe-padding

Determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of the object.

Architecture Restrictions

Only available on all architectures that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions

Syntax

Linux OS and macOS:

`-qopt-assume-safe-padding`

`-qno-opt-assume-safe-padding`

Windows OS:

`/Qopt-assume-safe-padding`

`/Qopt-assume-safe-padding-`

Arguments

None

Default

`-qno-opt-assume-safe-padding` or `/Qopt-assume-safe-padding-`	The compiler will not assume that variables and dynamically allocated memory are padded past the end of the object. It will adhere to the sizes specified in your program.

Description

This option determines whether the compiler assumes that variables and dynamically allocated memory are padded past the end of the object.

When you specify option `[q or Q]opt-assume-safe-padding`, the compiler assumes that variables and dynamically allocated memory are padded. This means that code can access up to 64 bytes beyond what is specified in your program.

The compiler does not add any padding for static and automatic objects when this option is used, but it assumes that code can access up to 64 bytes beyond the end of the object, wherever the object appears in the program. To satisfy this assumption, you must increase the size of static and automatic objects in your program when you use this option.

This option may improve performance of memory operations.

IDE Equivalent

None

Alternate Options

None

qopt-block-factor, Qopt-block-factor

Lets you specify a loop blocking factor.

Syntax

Linux OS and macOS:

`-qopt-block-factor=n`

Windows OS:

`/Qopt-block-factor:n`

Arguments

n	Is the blocking factor. It must be an integer. The compiler may ignore the blocking factor if the value is 0 or 1.

Default

OFF	The compiler uses default heuristics for loop blocking.

Description

This option lets you specify a loop blocking factor.

IDE Equivalent

None

Alternate Options

None

qopt-dynamic-align, Qopt-dynamic-align

Enables or disables dynamic data alignment optimizations.

Syntax

Linux OS and macOS:

`-qopt-dynamic-align`

`-qno-opt-dynamic-align`

Windows OS:

`/Qopt-dynamic-align`

`/Qopt-dynamic-align-`

Arguments

None

Default

`-qopt-dynamic-align` or `/Qopt-dynamic-align`	The compiler may generate code dynamically dependent on alignment. It may do optimizations based on data location for the best performance. The result of execution on some algorithms may depend on data layout.

Description

This option enables or disables dynamic data alignment optimizations.

If you specify `-qno-opt-dynamic-align` or `/Qopt-dynamic-align-`, the compiler generates no code dynamically dependent on alignment. It will not do any optimizations based on data location and results will depend on the data values themselves.

When you specify `[q or Q]qopt-dynamic-align`, the compiler may implement conditional optimizations based on dynamic alignment of the input data. These dynamic alignment optimizations may result in different bitwise results for aligned and unaligned data with the same values.

Dynamic alignment optimizations can improve the performance of vectorized code, especially for long trip count loops. Disabling such optimizations can decrease performance, but it may improve bitwise reproducibility of results, factoring out data location from possible sources of discrepancy.

IDE Equivalent

None

Alternate Options

None

qopt-jump-tables, Qopt-jump-tables

Enables or disables generation of jump tables for switch statements.

Syntax

Linux OS and macOS:

`-qopt-jump-tables=`*keyword*

`-qno-opt-jump-tables`

Windows OS:

`/Qopt-jump-tables:`*keyword*

`/Qopt-jump-tables-`

Arguments

keyword

Is the instruction for generating jump tables. Possible values are:

`never`	Tells the compiler to never generate jump tables. All switch statements are implemented as chains of if-then-elses. This is the same as specifying `-qno-opt-jump-tables` (Linux* and macOS*) or `/Qopt-jump-tables-` (Windows*).

`default`	The compiler uses default heuristics to determine when to generate jump tables.
`large`	Tells the compiler to generate jump tables up to a certain pre-defined size (64K entries).
`n`	Must be an integer. Tells the compiler to generate jump tables up to *n* entries in size.

Default

`-qopt-jump-tables=default` or `/Qopt-jump-tables:default`	The compiler uses default heuristics to determine when to generate jump tables for switch statements.

Description

This option enables or disables generation of jump tables for switch statements. When the option is enabled, it may improve performance for programs with large switch statements.

IDE Equivalent

None

Alternate Options

None

qopt-malloc-options

Lets you specify an alternate algorithm for malloc().

Syntax

Linux OS and macOS:

`-qopt-malloc-options=`*n*

Windows OS:

`None`

Arguments

n

Specifies the algorithm to use for malloc(). Possible values are:

`0`	Tells the compiler to use the default algorithm for malloc(). This is the default.
`1`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=2 and M_TRIM_THRESHOLD=0x10000000.
`2`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=2 and M_TRIM_THRESHOLD=0x40000000.
`3`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=0 and M_TRIM_THRESHOLD=-1.

4	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=0, M_TRIM_THRESHOLD=-1, M_TOP_PAD=4096.

Default

`-qopt-malloc-options=0`	The compiler uses the default algorithm when malloc() is called. No call is made to mallopt().

Description

This option lets you specify an alternate algorithm for malloc().

If you specify a non-zero value for *n*, it causes alternate configuration parameters to be set for how malloc() allocates and frees memory. It tells the compiler to insert calls to mallopt() to adjust these parameters to malloc() for dynamic memory allocation. This may improve speed.

IDE Equivalent

None

Alternate Options

None

See Also

malloc(3) man page

mallopt function (defined in malloc.h)

qopt-matmul, Qopt-matmul

Enables or disables a compiler-generated Matrix Multiply (matmul) library call.

Syntax

Linux OS:

`-qopt-matmul`

`-qno-opt-matmul`

macOS:

`None`

Windows OS:

`/Qopt-matmul`

`/Qopt-matmul-`

Arguments

None

Default

`-qno-opt-matmul` or `/Qopt-matmul-`	The matmul library call optimization does not occur unless this option is enabled or certain other compiler options are specified (see below).

Description

This option enables or disables a compiler-generated Matrix Multiply (MATMUL) library call.

The `[q or Q]opt-matmul` option tells the compiler to identify matrix multiplication loop nests (if any) and replace them with a matmul library call for improved performance. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors.

> **NOTE**
> This option is dependent upon the OpenMP* library. If your product does not support OpenMP, this option will have no effect.

This option is enabled by default if options `O3` and `[Q]parallel` are specified. To disable this optimization, specify `-qno-opt-matmul` or `/Qopt-matmul-`.

This option has no effect unless option `O2` or higher is set.

> **NOTE**
> Many routines in the MATMUL library are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

IDE Equivalent

Visual Studio: **Optimization > Enable Matrix Multiply Library Call**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

- `O` compiler option

qopt-mem-layout-trans, Qopt-mem-layout-trans

Controls the level of memory layout transformations performed by the compiler.

Syntax

Linux OS and macOS:

`-qopt-mem-layout-trans[=n]`

`-qno-opt-mem-layout-trans`

Windows OS:

`/Qopt-mem-layout-trans[:n]`

`/Qopt-mem-layout-trans-`

Arguments

n	Is the level of memory layout transformations. Possible values are:

`0`	Disables memory layout transformations. This is the same as specifying `-qno-opt-mem-layout-trans` (Linux* and macOS*) or `/Qopt-mem-layout-trans-` (Windows*).
`1`	Enables basic memory layout transformations.
`2`	Enables more memory layout transformations. This is the same as specifying `[q or Q]opt-mem-layout-trans` with no argument.
`3`	Enables more memory layout transformations like copy-in/copy-out of structures for a region of code. You should only use this setting if your system has more than 4GB of physical memory per core.
`4`	Enables more aggressive memory layout transformations. You should only use this setting if your system has more than 4GB of physical memory per core.

Default

`-qopt-mem-layout-trans=2` The compiler performs moderate memory layout transformations.

or
`/Qopt-mem-layout-trans:2`

Description

This option controls the level of memory layout transformations performed by the compiler. This option can improve cache reuse and cache locality.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

IDE Equivalent

None

Alternate Options

None

qopt-multi-version-aggressive, Qopt-multi-version-aggressive

Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.

Syntax

Linux OS and macOS:

`-qopt-multi-version-aggressive`

`-qno-opt-multi-version-aggressive`

Windows OS:

`/Qopt-multi-version-aggressive`

`/Qopt-multi-version-aggressive-`

Arguments

None

Default

`-qno-opt-multi-version-aggressive` or `/Qopt-multi-version-aggressive-`	The compiler uses default heuristics when checking for pointer aliasing and scalar replacement.

Description

This option tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement. This option may improve performance.

The performance can be affected by certain options, such as `/arch` or `/Qx` (Windows*) or `-m` or `-x` (Linux* and macOS*).

IDE Equivalent

None

Alternate Options

None

qopt-prefetch, Qopt-prefetch

Enables or disables prefetch insertion optimization.

Syntax

Linux OS and macOS:

`-qopt-prefetch[=`*n*`]`

`-qno-opt-prefetch`

Windows OS:

`/Qopt-prefetch[:`*n*`]`

`/Qopt-prefetch-`

Arguments

n	Is the level of software prefetching optimization desired. Possible values are:
`0`	Disables software prefetching. This is the same as specifying `-qno-opt-prefetch` (Linux* and macOS*) or `/Qopt-prefetch-` (Windows*).
`1` to `5`	Enables different levels of software prefetching. If you do not specify a value for *n*, the default is `2`. Use lower values to reduce the amount of prefetching.

Default

`-qno-opt-prefetch` or `/Qopt-prefetch-`	Prefetch insertion optimization is disabled.

Description

This option enables or disables prefetch insertion optimization. The goal of prefetching is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache.

This option enables prefetching when higher optimization levels are specified.

IDE Equivalent

Visual Studio: **Optimization > Prefetch Insertion**

Eclipse: None

Xcode: **Optimization > Enable Prefetch Insertion**

Alternate Options

None

See Also

qopt-prefetch-distance, Qopt-prefetch-distance compiler option

qopt-prefetch-distance, Qopt-prefetch-distance

Specifies the prefetch distance to be used for compiler-generated prefetches inside loops.

Syntax

Linux OS:

`-qopt-prefetch-distance=`*n1*`[, `*n2*`]`

macOS:

`None`

Windows OS:

`/Qopt-prefetch-distance:`*n1*`[, `*n2*`]`

Arguments

n1, n2	Is the prefetch distance in terms of the number of (possibly-vectorized) iterations. Possible values are non-negative numbers >=0. *n2* is optional. *n1* = 0 turns off all compiler issued prefetches from memory to L2. *n2* = 0 turns off all compiler issued prefetches from L2 to L1. If *n2* is specified and *n1* > 0, *n1* should be >= *n2*.

Default

OFF	The compiler uses default heuristics to determine the prefetch distance.

Description

This option specifies the prefetch distance to be used for compiler-generated prefetches inside loops. The unit (*n1* and optionally *n2*) is the number of iterations. If the loop is vectorized by the compiler, the unit is the number of vectorized iterations.

The value of *n1* will be used as the distance for prefetches from memory to L2 (for example, the vprefetch1 instruction). If *n2* is specified, it will be used as the distance for prefetches from L2 to L1 (for example, the vprefetch0 instruction).

This option is ignored if option `-qopt-prefetch=0` (Linux*) or `/Qopt-prefetch:0` (Windows*) is specified.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following Linux* examples:

```
-qopt-prefetch-distance=64,32
```

The above causes the compiler to use a distance of 64 iterations for memory to L2 prefetches, and a distance of 32 iterations for L2 to L1 prefetches.

```
-qopt-prefetch-distance=24
```

The above causes the compiler to use a distance of 24 iterations for memory to L2 prefetches. The distance for L2 to L1 prefetches will be determined by the compiler.

```
-qopt-prefetch-distance=0,4
```

The above turns off all memory to L2 prefetches inserted by the compiler inside loops. The compiler will use a distance of 4 iterations for L2 to L1 prefetches.

```
-qopt-prefetch-distance=16,0
```

The above causes the compiler to use a distance of 16 iterations for memory to L2 prefetches. No L2 to L1 loop prefetches are issued by the compiler.

See Also

qopt-prefetch, Qopt-prefetch compiler option
PREFETCH directive

qopt-prefetch-issue-excl-hint, Qopt-prefetch-issue-excl-hint

Supports the prefetchW instruction in Intel® microarchitecture code name Broadwell and later.

Syntax

Linux OS:

`-qopt-prefetch-issue-excl-hint`

macOS:

`None`

Windows OS:

`/Qopt-prefetch-issue-excl-hint`

Arguments

None

Default

OFF	The compiler does not support the PREFETCHW instruction for this microarchitecture.

Description

This option supports the PREFETCHW instruction in Intel® microarchitecture code name Broadwell and later.

When you specify this option, you must also specify option `[q or Q]opt-prefetch`.

The prefetch instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates any other cached copy in anticipation of the line being written to in the future.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-prefetch/Qopt-prefetch` compiler option

qopt-ra-region-strategy, Qopt-ra-region-strategy

Selects the method that the register allocator uses to partition each routine into regions.

Syntax

Linux OS and macOS:

`-qopt-ra-region-strategy[=`*keyword*`]`

Windows OS:

`/Qopt-ra-region-strategy[:`*keyword*`]`

Arguments

keyword Is the method used for partitioning. Possible values are:

`routine`	Creates a single region for each routine.
`block`	Partitions each routine into one region per basic block.
`trace`	Partitions each routine into one region per trace.
`loop`	Partitions each routine into one region per loop.
`default`	The compiler determines which method is used for partitioning.

Default

`-qopt-ra-region-strategy=default` or `/Qopt-ra-region-strategy:default`

The compiler determines which method is used for partitioning. This is also the default if `keyword` is not specified.

Description

This option selects the method that the register allocator uses to partition each routine into regions.

When setting `default` is in effect, the compiler attempts to optimize the tradeoff between compile-time performance and generated code performance.

This option is only relevant when optimizations are enabled (option `O1` or higher).

IDE Equivalent

None

Alternate Options

None

See Also

- `O` compiler option

qopt-streaming-stores, Qopt-streaming-stores

Enables generation of streaming stores for optimization.

Syntax

Linux OS and macOS:

`-qopt-streaming-stores=`*keyword*

Windows OS:

`/Qopt-streaming-stores:`*keyword*

Arguments

keyword Specifies whether streaming stores are generated. Possible values are:

`always`	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound. When this option setting is specified, it is your responsibility to also insert any memory barriers (fences) as required to ensure correct memory ordering within a thread or across threads. See the Examples section for one way to do this.
`never`	Disables generation of streaming stores for optimization. Normal stores are performed.
`auto`	Lets the compiler decide which instructions to use.

Default

`-qopt-streaming-stores=auto` or `/Qopt-streaming-stores:auto`	The compiler decides whether to use streaming stores or normal stores.

Description

This option enables generation of streaming stores for optimization. This method stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.

This option may be useful for applications that can benefit from streaming stores.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows one way to insert memory barriers (fences) when specifying `-qopt-streaming-stores=always` or `/Qopt-streaming-stores:always`. It uses the procedure interface for_sfence from the module IFCORE, which maps to the C/C++ function _mm_sfence:

```
subroutine sub1(a, b, c, len, n1, n2)
use IFCORE, only : for_sfence
integer len, n1, n2, i, j
real(8) a(len), b(len), c(len), d(len)

  !$omp parallel do
  do j = 1, n1
     a(j) = 1.0
     b(j) = 2.0
     c(j) = 3.0
  enddo
  !$omp end parallel do

call ftn_sfence()

  !$omp parallel do
```

```
  do i = 1, n2
     a(i) = a(i) + b(i) * c(i)
  enddo
  !$omp end parallel do

end
```

See Also

ax, Qax compiler option

x, Qx compiler option

qopt-subscript-in-range, Qopt-subscript-in-range

Determines whether the compiler assumes that there are no "large" integers being used or being computed inside loops.

Syntax

Linux OS and macOS:

`-qopt-subscript-in-range`

`-qno-opt-subscript-in-range`

Windows OS:

`/Qopt-subscript-in-range`

`/Qopt-subscript-in-range-`

Arguments

None

Default

`-qno-opt-subscript-in-range` or `/Qopt-subscript-in-range-`	The compiler assumes there are "large" integers being used or being computed within loops.

Description

This option determines whether the compiler assumes that there are no "large" integers being used or being computed inside loops.

If you specify `[q or Q]opt-subscript-in-range`, the compiler assumes that there are no "large" integers being used or being computed inside loops. A "large" integer is typically > 2^{31}.

This feature can enable more loop transformations.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how these options can be useful. Variable m is declared as type integer(kind=8) (64-bits) and all other variables inside the subscript are declared as type integer(kind=4) (32-bits):

```
A[ i  + j  + ( n  + k) * m ]
```

qopt-zmm-usage, Qopt-zmm-usage

Defines a level of zmm registers usage.

Syntax

Linux OS and macOS:

`-qopt-zmm-usage=`*keyword*

Windows OS:

`/Qopt-zmm-usage:`*keyword*

Arguments

keyword	Specifies the level of zmm registers usage. Possible values are:	
	`low`	Tells the compiler that the compiled program is unlikely to benefit from zmm registers usage. It specifies that the compiler should avoid using zmm registers unless it can prove the gain from their usage.
	`high`	Tells the compiler to generate zmm code without restrictions.

Default

varies	The default is low when you specify [Q]xCORE-AVX512. The default is high when you specify [Q]xCOMMON-AVX512.

Description

This option may provide better code optimization for Intel® processors that are on the Intel® microarchitecture formerly code-named Skylake.

This option defines a level of zmm registers usage. The `low` setting causes the compiler to generate code with zmm registers very carefully, only when the gain from their usage is proven. The `high` setting causes the compiler to use much less restrictive heuristics for zmm code generation.

It is not always easy to predict whether the `high` or the `low` setting will yield better performance. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to use zmm registers. However, some programs that use zmm registers may not gain as much or may even lose performance. We recommend that you try both option values to measure the performance of your programs.

This option is ignored if you do not specify an option that enables Intel® AVX-512, such as `[Q]xCORE-AVX512` or option `[Q]xCOMMON-AVX512`.

This option has no effect on loops that use directive SIMD SIMDLEN(n) or on functions that are generated by vector specifications specific to CORE-AVX512.

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option
SIMD Directive (OpenMP* API) clause SIMDLEN

qoverride-limits, Qoverride-limits

Lets you override certain internal compiler limits that are intended to prevent excessive memory usage or compile times for very large, complex compilation units.

Syntax

Linux OS and macOS:

`-qoverride-limits`

Windows OS:

`/Qoverride-limits`

Arguments

None

Default

OFF	Certain internal compiler limits are not overridden. These limits are determined by default heuristics.

Description

This option provides a way to override certain internal compiler limits that are intended to prevent excessive memory usage or compile times for very large, complex compilation units.

If this option is not used and your program exceeds one of these internal compiler limits, some optimizations will be skipped to reduce the memory footprint and compile time. If you chose to create an optimization report by specifying `[q or Q]opt-report`, you may see a related diagnostic remark as part of the report.

Specifying this option may substantially increase compile time and/or memory usage.

NOTE
If you use this option, it is your responsibility to ensure that sufficient memory is available. If there is not sufficient available memory, the compilation may fail.

This option should only be used where there is a specific need; it is not recommended for inexperienced users.

IDE Equivalent

None

Alternate Options

None

reentrancy

Tells the compiler to generate reentrant code to support a multithreaded application.

Syntax

Linux OS and macOS:

`-reentrancy` *`keyword`*

`-noreentrancy`

Windows OS:

`/reentrancy:`*`keyword`*

`/noreentrancy`

Arguments

`keyword` Specifies details about the program. Possible values are:

`none`	Tells the run-time library (RTL) that the program does not rely on threaded or asynchronous reentrancy. The RTL will not guard against such interrupts inside its own critical regions. This is the same as specifying noreentrancy.
`async`	Tells the run-time library (RTL) that the program may contain asynchronous (AST) handlers that could call the RTL. This causes the RTL to guard against AST interrupts inside its own critical regions.
`threaded`	Tells the run-time library (RTL) that the program is multithreaded, such as programs using the POSIX threads library. This causes the RTL to use thread locking to guard its own critical regions.

Default

`threaded` The compiler tells the run-time library (RTL) that the program is multithreaded.

Description

This option tells the compiler to generate reentrant code to support a multithreaded application.

If you do not specify a keyword for reentrancy, it is the same as specifying `reentrancy threaded`.

To ensure that a threadsafe and/or reentrant run-time library is linked and correctly initialized, option `reentrancy threaded` should also be used for the link step and for the compilation of the main routine.

Note that if option threads is specified, it sets option `reentrancy threaded`, since multithreaded code must be reentrant.

IDE Equivalent

Visual Studio: **Code Generation > Generate Reentrant Code**

Eclipse: None

Xcode: **Code Generation > Generate Reentrant Code**

Alternate Options

None

See Also

`threads` compiler option

safe-cray-ptr, Qsafe-cray-ptr

Tells the compiler that Cray pointers do not alias other variables.*

Syntax

Linux OS and macOS:

`-safe-cray-ptr`

Windows OS:

`/Qsafe-cray-ptr`

Arguments

None

Default

OFF The compiler assumes that Cray pointers alias other variables.

Description

This option tells the compiler that Cray pointers do not alias (that is, do not specify sharing memory with) other variables.

IDE Equivalent

Visual Studio: **Data > Assume CRAY Pointers Do Not Share Memory Locations** (/Qsafe-cray-ptr)

Eclipse: None

Xcode: **Data > Assume Cray Pointers Do Not Share Memory Locations**

Alternate Options

None

Example

Consider the following:

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1
enddo
```

By default, the compiler assumes that b and a are aliased. To prevent such an assumption, specify the `-safe-cray-ptr` (Linux* and macOS*) or `/Qsafe-cray-ptr` (Windows*) option, and the compiler will treat b(i) and a(i) as independent of each other.

However, if the variables are intended to be aliased with Cray pointers, using the option produces incorrect results. In the following example, you should not use the option:

```
pointer (pb, b)
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1
enddo
```

scalar-rep, Qscalar-rep

Enables or disables the scalar replacement optimization done by the compiler as part of loop transformations.

Syntax

Linux OS and macOS:

`-scalar-rep`

`-no-scalar-rep`

Windows OS:

`/Qscalar-rep`

`/Qscalar-rep-`

Arguments

None

Default

`-scalar-rep` or `/Qscalar-rep`	Scalar replacement is performed during loop transformation at optimization levels of O2 and above.

Description

This option enables or disables the scalar replacement optimization done by the compiler as part of loop transformations. This option takes effect only if you specify an optimization level of `O2` or higher.

IDE Equivalent

None

Alternate Options

None

See Also

`O` compiler option

simd, Qsimd

Enables or disables compiler interpretation of SIMD directives.

Syntax

Linux OS and macOS:

`-simd`

`-no-simd`

Windows OS:

`/Qsimd`

`/Qsimd-`

Arguments

None

Default

`-simd` or `/Qsimd`	SIMD directives are enabled.

Description

This option enables or disables compiler interpretation of SIMD directives.

To disable interpretation of SIMD directives, specify `-no-simd` (Linux* and macOS*) or `/Qsimd-` (Windows*). Note that the compiler may still vectorize loops based on its own heuristics (leading to generation of SIMD instructions) even when `-no-simd` (or `/Qsimd-`) is specified.

To disable all compiler vectorization, use the "`-no-vec -no-simd`" (Linux* and macOS*) or "`/Qvec- /Qsimd-`" (Windows*) compiler options. The option `-no-vec` (and `/Qvec-`) disables all auto-vectorization, including vectorization of array notation statements. The option `-no-simd` (and `/Qsimd-`) disables vectorization of loops that have SIMD directives.

NOTE
If you specify option `-mia32` (Linux*) or option `/arch:IA32` (Windows*), SIMD directives are disabled by default and vector instructions cannot be used. Therefore, you cannot explicitly enable SIMD directives by specifying option `[Q]simd`.

IDE Equivalent

None

Alternate Options

None

See Also

`vec, Qvec` compiler option
SIMD Directive

unroll, Qunroll

Tells the compiler the maximum number of times to unroll loops.

Syntax

Linux OS:

`-unroll[=n]`

macOS:

`-unroll[=n]`

Windows OS:

`/Qunroll[:n]`

Arguments

n	Is the maximum number of times a loop can be unrolled. To disable loop enrolling, specify 0.

Default

`-unroll` or `/Qunroll` The compiler uses default heuristics when unrolling loops.

Description

This option tells the compiler the maximum number of times to unroll loops.

If you do not specify *n*, the optimizer determines how many times loops can be unrolled.

IDE Equivalent

Visual Studio: **Optimization > Loop Unroll Count**

Eclipse: None

Xcode: **Optimization > Loop Unroll Count**

Alternate Options

Linux and macOS*: `-funroll-loops`

Windows: `/unroll` (this is a deprecated option)

unroll-aggressive, Qunroll-aggressive

Determines whether the compiler uses more aggressive unrolling for certain loops.

Syntax

Linux OS:

`-unroll-aggressive`

`-no-unroll-aggressive`

macOS:

`-unroll-aggressive`

`-no-unroll-aggressive`

Windows OS:

`/Qunroll-aggressive`

`/Qunroll-aggressive-`

Arguments

None

Default

`-no-unroll-aggressive` or `/Qunroll-aggressive-`	The compiler uses default heuristics when unrolling loops.

Description

This option determines whether the compiler uses more aggressive unrolling for certain loops. The positive form of the option may improve performance.

This option enables aggressive, complete unrolling for loops with small constant trip counts.

IDE Equivalent

None

Alternate Options

None

vec, Qvec

Enables or disables vectorization.

Syntax

Linux OS:

`-vec`

`-no-vec`

macOS:

`-vec`

`-no-vec`

Windows OS:

`/Qvec`

`/Qvec-`

Arguments

None

Default

`-vec` or `/Qvec`	Vectorization is enabled if option `O2` or higher is in effect.

Description

This option enables or disables vectorization.

To disable vectorization, specify `-no-vec` (Linux* and macOS*) or `/Qvec-` (Windows*).

To disable interpretation of SIMD directives, specify `-no-simd` (Linux* and macOS*) or `/Qsimd-` (Windows*).

To disable all compiler vectorization, use the "`-no-vec -no-simd`" (Linux* and macOS*) or "`/Qvec- /Qsimd-`" (Windows*) compiler options. The option `-no-vec` (and `/Qvec-`) disables all auto-vectorization, including vectorization of array notation statements. The option `-no-simd` (and `/Qsimd-`) disables vectorization of loops that have SIMD directives.

NOTE
Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows), `-m` (Linux and macOS*), or `[Q]x`.

IDE Equivalent

None

Alternate Options

None

See Also

`simd, Qsimd` compiler option
`ax, Qax` compiler option
`x, Qx` compiler option
`vec-guard-write, Qvec-guard-write` compiler option
`vec-threshold, Qvec-threshold` compiler option

vec-guard-write, Qvec-guard-write

Tells the compiler to perform a conditional check in a vectorized loop.

Syntax

Linux OS and macOS:

`-vec-guard-write`

`-no-vec-guard-write`

Windows OS:

`/Qvec-guard-write`

`/Qvec-guard-write-`

Arguments

None

Default

`-vec-guard-write` or `/Qvec-guard-write`	The compiler performs a conditional check in a vectorized loop.

Description

This option tells the compiler to perform a conditional check in a vectorized loop. This checking avoids unnecessary stores and may improve performance.

IDE Equivalent

None

Alternate Options

None

vec-threshold, Qvec-threshold

Sets a threshold for the vectorization of loops.

Syntax

Linux OS and macOS:

`-vec-threshold[n]`

Windows OS:

`/Qvec-threshold[[:]n]`

Arguments

n	Is an integer whose value is the threshold for the vectorization of loops. Possible values are 0 through 100. If *n* is 0, loops get vectorized always, regardless of computation work volume. If *n* is 100, loops get vectorized when performance gains are predicted based on the compiler analysis data. Loops get vectorized only if profitable vector-level parallel execution is almost certain. The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to vectorize only if there is a 50% probability of the code speeding up if executed in vector form.

Default

`-vec-threshold100` or `/Qvec-threshold100`	Loops get vectorized only if profitable vector-level parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the vectorization of loops based on the probability of profitable execution of the vectorized loop in parallel.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

IDE Equivalent

Visual Studio: **Optimization > Threshold For Vectorization**

Eclipse: None

Xcode: **Optimization > Enable Maximum Vector-level Parallelism**

Alternate Options

None

vecabi, Qvecabi

Determines which vector function application binary interface (ABI) the compiler uses to create or call vector functions.

Syntax

Linux OS:

`-vecabi=`*keyword*

macOS:

`-vecabi=`*keyword*

Windows OS:

`/Qvecabi:`*keyword*

Arguments

keyword

Specifies which vector function ABI to use. Possible values are:

`compat`	Tells the compiler to use the compatibility vector function ABI. This ABI includes Intel®-specific features.
`cmdtarget`	Tells the compiler to generate an extended set of vector functions. The option is very similar to setting `compat`. However, for `compat`, only one vector function is created, while for `cmdtarget`, several vector functions are created for each vector specification. Vector variants are created for targets specified by compiler options `[Q]x` and/or `[Q]ax`. No change is made to the source code.
`gcc`	Tells the compiler to use the gcc vector function ABI. Use this setting only in cases when you want to link with modules compiled by gcc. This setting is not available on Windows* systems.
`legacy`	Tells the compiler to use the legacy vector function ABI. Use this setting if you need to keep the generated vector function binary backward compatible with the vectorized binary generated by older versions of the Intel® compilers (V13.1 or older).

Default

compat	The compiler uses the compatibility vector function ABI.

Description

This option determines which vector function application binary interface (ABI) the compiler uses to create or call vector functions.

> **NOTE**
> To avoid possible link-time and run-time errors, use identical `[Q]vecabi` settings when compiling all files in an application that define or use vector functions, including libraries. If setting `cmdtarget` is specified, options `[Q]x` and/or `[Q]ax` must have identical values.
>
> Be careful using setting `cmdtarget` with libraries or program modules/routines with vector function definitions that cannot be recompiled. In such cases, setting `cmdtarget` may cause link errors.

On Linux* systems, since the default is `compat`, you must specify `legacy` if you need to keep the generated vector function binary backward compatible with the vectorized binary generated by the previous version of Intel® compilers.

When `cmdtarget` is specified, the additional vector function versions are created by copying each vector specification and changing target processor in the copy. The number of vector functions is determined by the settings specified in options `[Q]x` and/or `[Q]ax`.

For example, suppose we have the following function declaration:

```
interface
integer function foo(a)
!dir$ attributes vector:(processor(core_2_duo_sse4_1)) :: foo
    integer a
end function
end interface
```

and the following options are specified: `-axAVX,CORE-AVX2`

The following table shows the different results for the above declaration and option specifications when setting `compat` or setting `cmdtarget` is used:

compat	cmdtarget
One vector version is created for Intel® SSE4.1 (by vector function specification).	Four vector versions are created for the following targets: • Intel® SSE2 (default because no `-x` option is used) • Intel® SSE4.1 (by vector function specification) • Intel® AVX (by the first `-ax` option value) • Intel® AVX2 (by the second `-ax` option value)

For more information about the Intel®-compatible vector functions ABI, see the article titled: Vector (SIMD) Function ABI, which is located in https://software.intel.com/en-us/articles/vector-simd-function-abi/

For more information about the GCC vector functions ABI, see the item Libmvec - vector math library document in the GLIBC wiki at sourceware.org.

Optimization Notice
dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

IDE Equivalent

None

Alternate Options

None

Profile Guided Optimization (PGO) Options

finstrument-functions, Qinstrument-functions

Determines whether routine entry and exit points are instrumented.

Syntax

Linux OS and macOS:

`-finstrument-functions`

`-fno-instrument-functions`

Windows OS:

`/Qinstrument-functions`

`/Qinstrument-functions-`

Arguments

None

Default

`-fno-instrument-functions` Routine entry and exit points are not instrumented.
or
`/Qinstrument-functions-`

Description

This option determines whether routine entry and exit points are instrumented. It may increase execution time.

The following profiling functions are called with the address of the current routine and the address of where the routine was called (its "call site"):

- This function is called upon routine entry:

```
void __cyg_profile_func_enter (void *this_fn,
void *call_site);
```

- This function is called upon routine exit:

```
void __cyg_profile_func_exit (void *this_fn,
void *call_site);
```

These functions can be used to gather more information, such as profiling information or timing information. Note that it is the user's responsibility to provide these profiling functions.

If you specify `-finstrument-functions` (Linux* and macOS*) or `/Qinstrument-functions` (Windows*), routine inlining is disabled. If you specify `-fno-instrument-functions` or `/Qinstrument-functions-`, inlining is not disabled.

This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

fnsplit, Qfnsplit

Enables function splitting.

Syntax

Linux OS:

`-fnsplit[=n]`

`-no-fnsplit`

macOS:

`None`

Windows OS:

`/Qfnsplit[:n]`

`/Qfnsplit-`

Arguments

n	Is an optional positive integer indicating the threshold number. The blocks can be placed into a different code segment if they are only reachable via a conditional branch whose taken probability is less than the specified *n*. Branch taken probability is heuristically calculated by the compiler and can be observed in assembly listings. The range for *n* is 0 <= n <= 100.

Default

OFF	Function splitting is not enabled. However, function grouping is still enabled.

Description

This option enables function splitting. If you specify `[Q]fnsplit` with no *n*, you must also specify option `[Q]prof-use`, or the option will have no effect and no function splitting will occur.

If you specify *n*, function splitting is enabled and you do not need to specify option `[Q]prof-use`.

To disable function splitting when you use option `[Q]prof-use`, specify `/Qfnsplit-` (Windows*) or `-no-fnsplit` (Linux*).

NOTE
Function splitting is generally not supported when exception handling is turned on for C/C++ routines in the stack of called routines. See also `-fexceptions` (Linux*).

IDE Equivalent

None

Alternate Options

Linux: `-freorder-blocks-and-partition` (a gcc option)

Windows: None

p

Compiles and links for function profiling with gprof(1).

Syntax

Linux OS and macOS:

`-p`

Windows OS:

`None`

Arguments

None

Default

OFF Files are compiled and linked without profiling.

Description

This option compiles and links for function profiling with gprof(1).

When you specify this option, inlining is disabled. However, you can override this by specifying directive FORCEINLINE, or a compiler option such as `[Q]inline-forceinline`.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-pg`,`-qp` (this is a deprecated option)

Windows: None

prof-data-order, Qprof-data-order

Enables or disables data ordering if profiling information is enabled.

Syntax

Linux OS:

`-prof-data-order`

`-no-prof-data-order`

macOS:

`None`

Windows OS:

`/Qprof-data-order`

`/Qprof-data-order-`

Arguments

None

Default

`-no-prof-data-order` Data ordering is disabled.
or
`/Qprof-data-order-`

Description

This option enables or disables data ordering if profiling information is enabled. It controls the use of profiling information to order static program data items.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify option `[Q]prof-gen setting globdata`.
- For feedback compilation, you must specify the `[Q]prof-use` option. You must not use multi-file optimization by specifying options such as `[Q]ipo` or `[Q]ipo-c`.

IDE Equivalent

None

Alternate Options

None

See Also

prof-gen, Qprof-gen
compiler option
prof-use, Qprof-use
compiler option
prof-func-order, Qprof-func-order
compiler option

prof-dir, Qprof-dir

Specifies a directory for profiling information output files.

Syntax

Linux OS and macOS:

`-prof-dir dir`

Windows OS:

`/Qprof-dir:dir`

Arguments

dir	Is the name of the directory. You can specify a relative pathname or an absolute pathname.

Default

OFF	Profiling output files are placed in the directory where the program is compiled.

Description

This option specifies a directory for profiling information output files (*.dyn and *.dpi). The specified directory must already exist.

You should specify this option using the same directory name for both instrumentation and feedback compilations. If you move the .dyn files, you need to specify the new path.

Option `/Qprof-dir` is equivalent to option `/Qcov-dir`. If you specify both options, the last option specified on the command line takes precedence.

IDE Equivalent

Visual Studio: **Output Files > Profile Directory**

Eclipse: None

Xcode: None

Alternate Options

None

prof-file, Qprof-file

Specifies an alternate file name for the profiling summary files.

Syntax

Linux OS and macOS:

`-prof-file filename`

Windows OS:

`/Qprof-file:filename`

Arguments

filename	Is the name of the profiling summary file.

Default

OFF	The profiling summary files have the file name pgopti.*

Description

This option specifies an alternate file name for the profiling summary files. The *filename* is used as the base name for files created by different profiling passes.

If you add this option to profmerge, the .dpi file will be named *filename*.dpi instead of pgopti.dpi.

If you specify this option with option `[Q]prof-use`, the .dpi file will be named *filename*.dpi instead of pgopti.dpi.

Option `/Qprof-file` is equivalent to option `/Qcov-file`. If you specify both options, the last option specified on the command line takes precedence.

> **NOTE**
> When you use option `[Q]prof-file`, you can only specify a file name. If you want to specify a path (relative or absolute) for *filename*, you must also use option `[Q]prof-dir`.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-gen, Qprof-gen` compiler option
`prof-use, Qprof-use` compiler option
`prof-dir, Qprof-dir` compiler option

prof-func-groups

Enables or disables function grouping if profiling information is enabled.

Syntax

Linux OS:

`-prof-func-groups`

`-no-prof-func-groups`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-no-prof-func-groups`	Function grouping is disabled.

Description

This option enables or disables function grouping if profiling information is enabled.

A "function grouping" is a profiling optimization in which entire routines are placed either in the cold code section or the hot code section.

If profiling information is enabled by option `-prof-use`, option `-prof-func-groups` is set and function grouping is enabled. However, if you explicitly enable `-prof-func-order`, function ordering is performed instead of function grouping.

If you want to disable function grouping when profiling information is enabled, specify `-no-prof-func-groups`.

To set the hotness threshold for function grouping, use option `-prof-hotness-threshold`.

IDE Equivalent

None

See Also

`prof-use, Qprof-use` compiler option

`prof-func-order, Qprof-func-order` compiler option

`prof-hotness-threshold, Qprof-hotness-threshold` compiler option

prof-func-order, Qprof-func-order

Enables or disables function ordering if profiling information is enabled.

Syntax

Linux OS:

`-prof-func-order`

`-no-prof-func-order`

macOS:

`None`

Windows OS:

`/Qprof-func-order`

`/Qprof-func-order-`

Arguments

None

Default

`-no-prof-func-order` Function ordering is disabled.
or
`/Qprof-func-order-`

Description

This option enables or disables function ordering if profiling information is enabled.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify option `[Q]prof-gen setting srcpos`.

- For feedback compilation, you must specify `[Q]prof-use`. You must not use multi-file optimization by specifying options such as `[Q]ipo` or `[Q]ipo-c`.

If you enable profiling information by specifying option `[Q]prof-use`, option `[Q]prof-func-groups` is set and function grouping is enabled. However, if you explicitly enable the `[Q]prof-func-order` option, function ordering is performed instead of function grouping.

On Linux* systems, this option is only available for Linux linker 2.15.94.0.1, or later.

To set the hotness threshold for function grouping and function ordering, use option `[Q]prof-hotness-threshold`.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to use this option on a Windows system:

```
ifort /Qprof-gen:globdata file1.f90 file2.f90 /exe:instrumented.exe
    ./instrumented.exe
ifort /Qprof-use /Qprof-func-order file1.f90 file2.f90 /exe:feedback.exe
```

The following example shows how to use this option on a Linux system:

```
ifort -prof-gen:globdata file1.f90 file2.f90 -o instrumented
    ./instrumented.exe
ifort -prof-use -prof-func-order file1.f90 file2.f90 -o feedback
```

See Also

`prof-hotness-threshold, Qprof-hotness-threshold`
compiler option

`prof-gen, Qprof-gen`
compiler option

`prof-use, Qprof-use`
compiler option

`prof-data-order, Qprof-data-order`
compiler option

`prof-func-groups`
compiler option

prof-gen, Qprof-gen

Produces an instrumented object file that can be used in profile guided optimization.

Syntax

Linux OS and macOS:

`-prof-gen[=keyword[, keyword],...]`

`-no-prof-gen`

Windows OS:

`/Qprof-gen[:keyword[,keyword],...]`

`/Qprof-gen-`

Arguments

keyword

Specifies details for the instrumented file. Possible values are:

`default`	Produces an instrumented object file. This is the same as specifying the `[Q]prof-gen` option with no keyword.
`srcpos`	Produces an instrumented object file that includes extra source position information.
`globdata`	Produces an instrumented object file that includes information for global data layout.
`[no]threadsafe`	Produces an instrumented object file that includes the collection of PGO data on applications that use a high level of parallelism. If `[Q]prof-gen` is specified with no keyword, the default is `nothreadsafe`.

Default

`-no-prof-gen` or `/Qprof-gen-` Profile generation is disabled.

Description

This option produces an instrumented object file that can be used in profile guided optimization. It gets the execution count of each basic block.

You can specify more than one setting for `[Q]prof-gen`. For example, you can specify the following:

```
-prof-gen=srcpos -prof-gen=threadsafe (Linux* and macOS*)
-prof-gen=srcpos, threadsafe (this is equivalent to the above)
```

```
/Qprof-gen:srcpos /Qprof-gen:threadsafe (Windows*)
/Qprof-gen:srcpos, threadsafe (this is equivalent to the above)
```

If you specify keyword `srcpos` or `globdata`, a static profile information file (.spi) is created. These settings may increase the time needed to do a parallel build using -prof-gen, because of contention writing the .spi file.

These options are used in phase 1 of the Profile Guided Optimizer (PGO) to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution.

When the `[Q]prof-gen` option is used to produce an instrumented binary file for profile generation, some optimizations are disabled. Those optimizations are not disabled for any subsequent profile-guided compilation with option `[Q]prof-use` that makes use of the generated profiles.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-use, Qprof-use`
compiler option

Profile an Application with Instrumentation

prof-gen-sampling

Tells the compiler to generate debug discriminators in debug output. This aids in developing more precise sampled profiling output.

Syntax

Linux OS:

`-prof-gen-sampling`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not generate debug discriminators in the debug output.

Description

This option tells the compiler to generate debug discriminators in debug output. Debug discriminators are used to distinguish code from different basic blocks that have the same source position information. This aids in developing more precise sampled hardware profiling output.

To build an executable suitable for generating hardware profiled sampled output, compile with the following options:

```
    -prof-gen-sampling -g
```

To use the data files produced by hardware profiling, compile with option `-prof-use-sampling`.

IDE Equivalent

None

Alternate Options

None

See Also

prof-use-sampling compiler option

g compiler option

Profile an Application with Instrumentation

prof-hotness-threshold, Qprof-hotness-threshold

Lets you set the hotness threshold for function grouping and function ordering.

Syntax

Linux OS:

`-prof-hotness-threshold=n`

macOS:

`None`

Windows OS:

`/Qprof-hotness-threshold:n`

Arguments

n	Is the hotness threshold. *n* is a percentage having a value between 0 and 100 inclusive. If you specify 0, there will be no hotness threshold setting in effect for function grouping and function ordering.

Default

OFF	The compiler's default hotness threshold setting of 10 percent is in effect for function grouping and function ordering.

Description

This option lets you set the hotness threshold for function grouping and function ordering.

The "hotness threshold" is the percentage of functions in the application that should be placed in the application's hot region. The hot region is the most frequently executed part of the application. By grouping these functions together into one hot region, they have a greater probability of remaining resident in the instruction cache. This can enhance the application's performance.

For this option to take effect, you must specify option `[Q]prof-use` and one of the following:

- On Linux systems: `-prof-func-groups` or `-prof-func-order`
- On Windows systems: `/Qprof-func-order`

IDE Equivalent

None

Alternate Options

None

See Also

prof-use, Qprof-use
compiler option

prof-func-groups
compiler option

prof-func-order, Qprof-func-order
compiler option

prof-src-dir, Qprof-src-dir

Determines whether directory information of the source file under compilation is considered when looking up profile data records.

Syntax

Linux OS and macOS:

`-prof-src-dir`

`-no-prof-src-dir`

Windows OS:

`/Qprof-src-dir`

`/Qprof-src-dir-`

Arguments

None

Default

[Q]prof-src-dir	Directory information is used when looking up profile data records in the .dpi file.

Description

This option determines whether directory information of the source file under compilation is considered when looking up profile data records in the .dpi file. To use this option, you must also specify the `[Q]prof-use` option.

If the option is enabled, directory information is considered when looking up the profile data records within the .dpi file. You can specify directory information by using one of the following options:

- Linux and macOS*: `-prof-src-root` or `-prof-src-root-cwd`
- Windows: `/Qprof-src-root` or `/Qprof-src-root-cwd`

If the option is disabled, directory information is ignored and only the name of the file is used to find the profile data record.

Note that option `[Q]prof-src-dir` controls how the names of the user's source files get represented within the .dyn or .dpi files. Option `[Q]prof-dir` specifies the location of the .dyn or the .dpi files.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-use, Qprof-use`
compiler option

`prof-src-root, Qprof-src-root`
compiler option

`prof-src-root-cwd, Qprof-src-root-cwd`
compiler option

prof-src-root, Qprof-src-root

Lets you use relative directory paths when looking up profile data and specifies a directory as the base.

Syntax

Linux OS and macOS:

`-prof-src-root=`*dir*

Windows OS:

`/Qprof-src-root:`*dir*

Arguments

dir	Is the base for the relative paths.

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It lets you specify a directory as the base. The paths are relative to a base directory specified during the `[Q]prof-gen` compilation phase.

This option is available during the following phases of compilation:

- Linux* and macOS* systems: `-prof-gen` and `-prof-use` phases
- Windows* systems: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `[Q]prof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `[Q]prof-use` phase, it specifies a root directory that replaces the root directory specified at the `[Q]prof-gen` phase for forming the lookup keys.

To be effective, this option or option `[Q]prof-src-root-cwd` must be specified during the `[Q]prof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

IDE Equivalent

None

Alternate Options

None

Example

Consider the initial `[Q]prof-gen` compilation of the source file c:\user1\feature_foo\myproject\common\glob.f90 shown below:

```
Linux* and macOS*: icc -prof-gen -prof-src-root=c:\user1\feature_foo\myproject -c common\glob.c
```

```
Windows*: ifort /Qprof-gen /Qprof-src-root=c:\user1\feature_foo\myproject -c common\glob.f90
```

```
Linux* and macOS*: ifort -prof-gen -prof-src-root=c:\user1\feature_foo\myproject -c common
\glob.f90
```

For the [Q]prof-use phase, the file glob.f90 could be moved into the directory c:\user2\feature_bar \myproject\common\glob.f90 and profile information would be found from the .dpi when using the following:

```
Windows*: ifort /Qprof-use /Qprof-src-root=c:\user2\feature_bar\myproject -c common\glob.f90
```

```
Linux* and macOS*: ifort -prof-use -prof-src-root=c:\user2\feature_bar\myproject -c common
\glob.f90
```

If you do not use option [Q]prof-src-root during the [Q]prof-gen phase, by default, the [Q]prof-use compilation can only find the profile data if the file is compiled in the c:\user1\feature_foo\my_project \common directory.

See Also

prof-gen, Qprof-gen
compiler option

prof-use, Qprof-use
compiler option

prof-src-dir, Qprof-src-dir
compiler option

prof-src-root-cwd, Qprof-src-root-cwd
compiler option

prof-src-root-cwd, Qprof-src-root-cwd

Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.

Syntax

Linux OS and macOS:

`-prof-src-root-cwd`

Windows OS:

`/Qprof-src-root-cwd`

Arguments

None

Default

OFF The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It specifies the current working directory as the base. To use this option, you must also specify option [Q]prof-use.

This option is available during the following phases of compilation:

- Linux* and macOS* systems: -prof-gen and -prof-use phases
- Windows* systems: /Qprof-gen and /Qprof-use phases

When this option is specified during the [Q]prof-gen phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the [Q]prof-use phase, it specifies a root directory that replaces the root directory specified at the [Q]prof-gen phase for forming the lookup keys.

To be effective, this option or option `[Q]prof-src-root` must be specified during the `[Q]prof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-gen, Qprof-gen`
compiler option

`prof-use, Qprof-use`
compiler option

`prof-src-dir, Qprof-src-dir`
compiler option

`prof-src-root, Qprof-src-root`
compiler option

prof-use, Qprof-use

Enables the use of profiling information during optimization.

Syntax

Linux OS and macOS:

`-prof-use[=keyword]`

`-no-prof-use`

Windows OS:

`/Qprof-use[:keyword]`

`/Qprof-use-`

Arguments

keyword
Specifies additional instructions. Possible values are:

`weighted`
Tells the profmerge utility to apply a weighting to the .dyn file values when creating the .dpi file to normalize the data counts when the training runs have differentexecution durations. This argument only has an effect when the compiler invokes the profmerge utility to create the .dpi file. This argument does not have an effect if the .dpi file was previously created without weighting.

`[no]merge`
Enables or disables automatic invocation of the profmerge utility. The default is `merge`. Note that you cannot specify both `weighted`

	and `nomerge`. If you try to specify both values, a warning will be displayed and `nomerge` takes precedence.
`default`	Enables the use of profiling information during optimization. The profmerge utility is invoked by default. This value is the same as specifying `[Q]prof-use` with no argument.

Default

`-no-prof-use` Profiling information is not used during optimization.

or
`/Qprof-use-`

Description

This option enables the use of profiling information (including function splitting and function grouping) during optimization. It enables option /Qfnsplit (Windows*) and -fnsplit (Linux* and macOS*) .

This option instructs the compiler to produce a profile-optimized executable and it merges available profiling output files into a pgopti.dpi file.

Note that there is no way to turn off function grouping if you enable it using this option.

To set the hotness threshold for function grouping and function ordering, use option `[Q]prof-hotness-threshold`.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-hotness-threshold, Qprof-hotness-threshold`
compiler option

`prof-gen, Qprof-gen`
compiler option

Profile an Application with Instrumentation

prof-use-sampling

Lets you use data files produced by hardware profiling to produce an optimized executable.

Syntax

Linux OS:

`-prof-use-sampling=`*`list`*

macOS:

`None`

Windows OS:

`None`

Arguments

list	Is a list of one or more data files. If you specify more than one data file, they must be separated by colons.

Default

OFF	Data files produced by hardware profiling will not be used to produce an optimized executable.

Description

This option lets you use data files produced by hardware profiling to produce an optimized executable.

These data files are named and produced by using Intel® VTune™.

The executable should have been produced using the following options:

```
    -prof-gen-sampling -g
```

IDE Equivalent

None

Alternate Options

None

See Also

prof-gen-sampling
compiler option

Profile an Application with Instrumentation

prof-value-profiling, Qprof-value-profiling

Controls which values are value profiled.

Syntax

Linux OS and macOS:

`-prof-value-profiling[=keyword]`

Windows OS:

`/Qprof-value-profiling[:keyword]`

Arguments

keyword	Controls which type of value profiling is performed. Possible values are:	
	`none`	Prevents all types of value profiling.
	`nodivide`	Prevents value profiling of non-compile time constants used in division or remainder operations.
	`noindcall`	Prevents value profiling of function addresses at indirect call sites.
	`all`	Enables all types of value profiling.

You can specify more than one keyword, but they must be separated by commas.

Default

`all` All value profile types are enabled and value profiling is performed.

Description

This option controls which features are value profiled.

If this option is specified with option `[Q]prof-gen`, it turns off instrumentation of operations of the specified type. This also prevents feedback of values for the operations.

If this option is specified with option `[Q]prof-use`, it turns off feedback of values collected of the specified type.

If you specify level 2 or higher for option `[q or Q]opt-report`, the value profiling specialization information will be reported within the PGO optimization report.

IDE Equivalent

None

Alternate Options

None

See Also

`prof-gen, Qprof-gen` compiler option
`prof-use, Qprof-use` compiler option
`qopt-report, Qopt-report` compiler option

Qcov-dir

Specifies a directory for profiling information output files that can be used with the codecov or tselect tool.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qcov-dir:`*`dir`*

Arguments

dir Is the name of the directory.

Default

OFF Profiling output files are placed in the directory where the program is compiled.

Description

This option specifies a directory for profiling information output files (*.dyn and *.dpi) that can be used with the code-coverage tool (codecov) or the test prioritization tool (tselect). The specified directory must already exist.

You should specify this option using the same directory name for both instrumentation and feedback compilations. If you move the .dyn files, you need to specify the new path.

Option `/Qcov-dir` is equivalent to option `/Qprof-dir`. If you specify both options, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

See Also

Qcov-gen
compiler option

Qcov-file
compiler option

Qcov-file

Specifies an alternate file name for the profiling summary files that can be used with the codecov or tselect tool.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qcov-file:`*`filename`*

Arguments

filename	Is the name of the profiling summary file.

Default

OFF	The profiling summary files have the file name pgopti.*.

Description

This option specifies an alternate file name for the profiling summary files. The file name can be used with the code-coverage tool (codecov) or the test prioritization tool (tselect).

The *filename* is used as the base name for the set of files created by different profiling passes.

If you specify this option with option `/Qcov-gen`, the .spi and .spl files will be named *filename*.spi and *filename*.spl instead of pgopti.spi and pgopti.spl.

Option `/Qcov-file` is equivalent to option `/Qprof-file`. If you specify both options, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

See Also

Qcov-gen
compiler option
Qcov-dir
compiler option

Qcov-gen

Produces an instrumented object file that can be used with the codecov or tselect tool.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qcov-gen`

`/Qcov-gen-`

Arguments

None

Default

`/Qcov-gen` The instrumented object file is not produced.

Description

This option produces an instrumented object file that can be used with the code-coverage tool (codecov) or the test prioritization tool (tselect). The instrumented code is included in the object file in preparation for instrumented execution.

This option also creates a static profile information file (.spi) that can be used with the codecov or tselect tool.

Option `/Qcov-gen` should be used to minimize the instrumentation overhead if you are interested in using the instrumentation only for code coverage. You should use `/Qprof-gen:srcpos` if you intend to use the collected data for code coverage and profile feedback.

IDE Equivalent

None

Alternate Options

None

See Also

Qcov-dir
compiler option
Qcov-file
compiler option

Optimization Report Options

qopt-report, Qopt-report

Tells the compiler to generate an optimization report.

Syntax

Linux OS and macOS:

`-qopt-report[=n]`

Windows OS:

`/Qopt-report[:n]`

Arguments

n	(Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels *n*=1 through *n*=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify *n*, the default is level 2, which produces a medium level of detail.

Default

OFF	No optimization report is generated.

Description

This option tells the compiler to generate a collection of optimization report files, one per object; this is the same output produced by option `[q or Q]opt-report-per-object`.

If you prefer another form of output, you can specify option `[q or Q]opt-report-file`.

If you specify a level (*n*) higher than 5, a warning will be displayed and you will get a level 5 report.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

For a description of the information that each *n* level provides, see the Example section in option `[q or Q]opt-report-phase`.

IDE Equivalent

Visual Studio: **Diagnostics > Optimization Diagnostic Level**

Eclipse: None

Xcode: **Diagnostics > Optimization Diagnostic Level**

Alternate Options

None

Example

If you only want reports about certain diagnostics, you can use this option with option `[q or Q]opt-report-phase`. The phase you specify determines which diagnostics you will receive.

For example, the following examples show how to get reports about certain specific diagnostics.

To get this specific report	Specify

Auto-parallelizer diagnostics	Linux* and macOS*:
	`-qopt-report -qopt-report-phase=par`
	Windows*:
	`/Qopt-report /Qopt-report-phase:par`

OpenMP parallelizer diagnostics	Linux* and macOS*:
	`-qopt-report -qopt-report-phase=openmp`
	Windows*:
	`/Qopt-report /Qopt-report-phase:openmp`

Vectorizer diagnostics	Linux* and macOS*:
	`-qopt-report -qopt-report-phase=vec`
	Windows*:
	`/Qopt-report /Qopt-report-phase:vec`

See Also

`qopt-report-file, Qopt-report-file` compiler option
`qopt-report-per-object, Qopt-report-per-object` compiler option
`qopt-report-phase, Qopt-report-phase` compiler option

qopt-report-annotate, Qopt-report-annotate

Enables the annotated source listing feature and specifies its format.

Syntax

Linux OS and macOS:

`-qopt-report-annotate[=keyword]`

Windows OS:

`/Qopt-report-annotate[:keyword]`

Arguments

keyword Specifies the format for the annotated source listing. You can specify one of the following:

`text` Indicates that the listing should be in text format. This is the default if you do not specify *keyword*.

`html` Indicates that the listing should be in html format.

Default

OFF No annotated source listing is generated

Description

This option enables the annotated source listing feature and specifies its format. The feature annotates source files with compiler optimization reports.

By default, one annotated source file is output per object. The annotated file is written to the same directory where the object files are generated. If the object file is a temporary file and an executable is generated, annotated files are placed in the directory where the executable is placed. You cannot generate annotated files to a directory of your choosing.

However, you can output annotated listings to stdout, stderr, or to a file if you also specify option `[q or Q]opt-report-file`.

By default, this option sets option `[q or Q]opt-report` with default level 2.

The following shows the file extension and listing details for the two possible *keyword*s.

Format	Listing Details
`text`	The annotated source listing has an .annot extension. It includes line numbers and compiler diagnostics placed after correspondent lines. IPO footnotes are inserted at the end of annotated file.
`html`	The annotated source listing has an .annot.html extension. It includes line numbers and compiler diagnostics placed after correspondent lines (as the text format does). It also provides hyperlinks in compiler messages and quick navigation with the routine list. IPO footnotes are displayed as tooltips.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-file, Qopt-report-file` compiler option

`qopt-report-annotate-position, Qopt-report-annotate-position` compiler option

qopt-report-annotate-position, Qopt-report-annotate-position

Enables the annotated source listing feature and specifies the site where optimization messages appear in the annotated source in inlined cases of loop optimizations.

Syntax

Linux OS and macOS:

`-qopt-report-annotate-position=`*`keyword`*

Windows OS:

`/Qopt-report-annotate-position:`*`keyword`*

Arguments

`keyword` Specifies the site where optimization messages appear in the annotated source. You can specify one of the following:

`caller` Indicates that the messages should appear in the caller site.

`callee` Indicates that the messages should appear in the callee site.

`both` Indicates that the messages should appear in both the caller and the callee sites.

Default

OFF No annotated source listing is generated

Description

This option enables the annotated source listing feature and specifies the site where optimization messages appear in the annotated source in inlined cases of loop optimizations.

This option enables option `[q or Q]opt-report-annotate` if it is not explicitly specified.

If annotated source listing is enabled and this option is not passed to compiler, loop optimizations are placed in caller position by default.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-annotate, Qopt-report-annotate` compiler option

qopt-report-embed, Qopt-report-embed

Determines whether special loop information annotations will be embedded in the object file and/or the assembly file when it is generated.

Syntax

Linux OS and macOS:

`-qopt-report-embed`

`-qno-opt-report-embed`

Windows OS:

`/Qopt-report-embed`

`/Qopt-report-embed-`

Arguments

None

Default

OFF When an assembly file is being generated, special loop information annotations will not be embedded in the assembly file.

However, if option `-g` (Linux* and macOS*) or `/Zi` (Windows*) is specified, special loop information annotations will be embedded in the assembly file unless option `-qno-opt-report-embed` (Linux and macOS*) or `/Qopt-report-embed-` (Windows) is specified.

Description

This option determines whether special loop information annotations will be embedded in the object file and/or the assembly file when it is generated. Specify the positive form of the option to include the annotations in the assembly file.

If an object file (or executable) is being generated, the annotations will be embedded in the object file (or executable).

If you use this option, you do not have to specify option `[q or Q]opt-report`.

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-file, Qopt-report-file

Specifies that the output for the optimization report goes to a file, stderr, or stdout.

Syntax

Linux OS and macOS:

`-qopt-report-file=`*`keyword`*

Windows OS:

`/Qopt-report-file:`*`keyword`*

Arguments

`keyword` Specifies the output for the report. You can specify one of the following:

`filename`	Specifies the name of the file where the output should go.
`stderr`	Indicates that the output should go to stderr.
`stdout`	Indicates that the output should go to stdout.

Default

OFF No optimization report is generated.

Description

This option specifies that the output for the optimization report goes to a file, stderr, or stdout.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

Visual Studio: **Diagnostics > Emit Optimization Diagnostics to File**

Diagnostics > Optimization Diagnostic File

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-filter, Qopt-report-filter

Tells the compiler to find the indicated parts of your application, and generate optimization reports for those parts of your application.

Syntax

Linux OS and macOS:

`-qopt-report-filter=`*`string`*

Windows OS:

`/Qopt-report-filter:`*`string`*

Arguments

string

Is the information to search for. The *string* must appear within quotes. It can take one or more of the following forms:

filename
filename, *routine*
filename, *range* [, *range*]...
filename, *routine*, *range* [, *range*]...

If you specify more than one of the above forms in a string, a semicolon must appear between each form. If you specify more than one *range* in a string, a comma must appear between each *range*. Optional blanks can follow each parameter in the forms above and they can also follow each form in a string.

filename

Specifies the name of a file to be found. It can include a path.

If you do not specify a path, the compiler looks for the filename in the current working directory.

routine

Specifies the name of a routine to be found. You can include an identifying argument.

The name, including any argument, must be enclosed in single quotes.

The compiler tries to uniquely identify the routine that corresponds to the specified routine name.

	It may select multiple routines to analyze, especially if more than one routine has the specified routine name, so the routine cannot be uniquely identified.
range	Specifies a range of line numbers to be found in the file or routine specified. The *range* must be specified in integers in the form: *first_line_number-last_line_number* The hyphen between the line numbers is required.

Default

OFF	No optimization report is generated.

Description

This option tells the compiler to find the indicated parts of your application, and generate optimization reports for those parts of your application. Optimization reports will only be generated for the routines that contain the specified *string*.

On Linux* and macOS*, if you specify both `-qopt-report-routine=string1` and `-qopt-report-filter=string2`, it is treated as `-qopt-report-filter=string1;string2`. On Windows*, if you specify both `/Qopt-report-routine:string1` and `/Qopt-report-filter:string2`, it is treated as`/Qopt-report-filter:string1;string2`.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-format, Qopt-report-format

Specifies the format for an optimization report.

Syntax

Linux OS and macOS:

`-qopt-report-format=`*`keyword`*

Windows OS:

`/Qopt-report-format:`*`keyword`*

Arguments

keyword Specifies the format for the report. You can specify one of the following:

`text` Indicates that the report should be in text format.

`vs` Indicates that the report should be in Visual Studio* (IDE) format. The Visual Studio IDE uses the information to visualize the optimization report in the context of your program source code.

Default

OFF No optimization report is generated.

Description

This option specifies the format for an optimization report. If you use this option, you must specify either `text` or `vs`.

If you do not specify this option and another option causes an optimization report to be generated, the default format is `text`.

If the `[q or Q]opt-report-file` option is also specified, it will affect where the output goes:

- If `filename` is specified, output goes to the specified file.
- If `stdout` is specified, output goes to stdout.
- If `stderr` is specified, output goes to stderr.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

`qopt-report-file, Qopt-report-file` compiler option

qopt-report-help, Qopt-report-help

Displays the optimizer phases available for report generation and a short description of what is reported at each level.

Syntax

Linux OS and macOS:

```
-qopt-report-help
```

Windows OS:

```
/Qopt-report-help
```

Arguments

None

Default

OFF No optimization report is generated.

Description

This option displays the optimizer phases available for report generation using `[q or Q]opt-report-phase`, and a short description of what is reported at each level. No compilation is performed.

To indicate where output should go, you can specify one of the following options:

- `[q or Q]opt-report-file`
- `[q or Q]opt-report-per-object`

If you use this option, you do not have to specify option `[q or Q]opt-report`.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option
`qopt-report-phase, Qopt-report-phase` compiler option
`qopt-report-file, Qopt-report-file` compiler option
`qopt-report-per-object, Qopt-report-per-object` compiler option

qopt-report-per-object, Qopt-report-per-object

Tells the compiler that optimization report information should be generated in a separate file for each object.

Syntax

Linux OS and macOS:

```
-qopt-report-per-object
```

Windows OS:

```
/Qopt-report-per-object
```

Arguments

None

Default

OFF No optimization report is generated.

Description

This option tells the compiler that optimization report information should be generated in a separate file for each object.

If you specify this option for a single-file compilation, a file with a .optrpt extension is produced for every object file or assembly file that is generated by the compiler. For a multifile Interprocedural Optimization (IPO) compilation, one file is produced for each of the N true objects generated in the compilation. If only one true object file is generated, the optimization report file generated is called ipo_out.optrpt. If multiple true object files are generated (N>1), the names used are ipo_out1.optprt, ipo_out2.optrpt, … ipo_outN.optrpt.

The .optrpt files are written to the target directory of the compilation process. If an object or assembly file is explicitly generated, the corresponding .optrpt file is written to the same directory where the object file is generated. If the object file is just a temporary file and an executable is generated, the corresponding .optrpt files are placed in the directory in which the executable is placed.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

qopt-report-phase, Qopt-report-phase

Specifies one or more optimizer phases for which optimization reports are generated.

Syntax

Linux OS and macOS:

`-qopt-report-phase[=`*list*`]`

Windows OS:

`/Qopt-report-phase[:`*list*`]`

Arguments

list

(Optional) Specifies one or more phases to generate reports for. If you specify more than one phase, they must be separated with commas. The values you can specify are:

`cg`	The phase for code generation
`ipo`	The phase for Interprocedural Optimization
`loop`	The phase for loop nest optimization
`openmp`	The phase for OpenMP
`par`	The phase for auto-parallelization
`pgo`	The phase for Profile Guided Optimization
`tcollect`	The phase for trace collection

`vec`	The phase for vectorization
`all`	All optimizer phases. This is the default if you do not specify *list*.

Default

OFF No optimization report is generated.

Description

This option specifies one or more optimizer phases for which optimization reports are generated.

For certain phases, you also need to specify other options:

- If you specify phase `cg`, you must also specify option `O1`, `O2` (default), or `O3`.
- If you specify phase `ipo`, you must also specify option `[Q]ipo`.
- If you specify phase `loop`, you must also specify option `O2` (default) or `O3`.
- If you specify phase `openmp`, you must also specify option `[q or Q]openmp`.
- If you specify phase `par`, you must also specify option `[Q]parallel`.
- If you specify phase `pgo`, you must also specify option `[Q]prof-use`.
- If you specify phase `tcollect`, you must also specify option `[Q]tcollect`.
- If you specify phase `vec`, you must also specify option `O2` (default) or `O3`. If you are interested in explicit vectorization by OpenMP* SIMD, you must also specify option `[q or Q]openmp`.

To find all phase possibilities, specify option `[q or Q]opt-report-help`.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

However, if you want to get more details for each phase, specify option `[q or Q]opt-report=`*n* along with this option and indicate the level of detail you want by specifying an appropriate value for *n*. (See also the Example section below.)

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

Visual Studio: **Diagnostics > Optimization Diagnostic Phase**

Eclipse: None

Xcode: **Diagnostics > Optimization Diagnostic Phase**

Alternate Options

None

Example

The following shows examples of the details you may receive when you specify one of the optimizer phases and a particular level (*n*) for option `[q or Q]opt-report`. Note that details may change in future releases.

Optimizer phase	The level specified in option `[q or Q]opt-report`	Description
`cg`	1	Generates a list of which intrinsics were lowered and which memcall optimizations were performed.

Optimizer phase	The level specified in option [q or Q]opt-report	Description
ipo	1	For each compiled routine, generates a list of the routines that were inlined into the routine, called directly by the routine, and whose calls were deleted.
	2	Generates level 1 details, values for important inlining command line options, and a list of the routines that were discovered to be dead and eliminated.
	3	Generates level 2 details, whole program information, the sizes of inlined routines, and the reasons routines were not inlined.
	4	Generates level 3 details, detailed footnotes on the reasons why routines are not inlined, and what action the user can take to get them inlined.
loop	1	Reports high-level details about which optimizations have been performed on the loop nests (along with the line number). Most of the loop optimizations (like fusion, unroll, unroll & jam, collapsing, rerolling etc) only support this level of detail.
	2	Generates level 1 details, and provides more detail on the metrics and types of references (like prefetch distance, indirect prefetches etc) used in optimizations. Only a few optimizations (like prefetching, loop classification framework etc) support these extra details.
openmp	1	Reports loops, regions, sections, and tasks successfully parallelized.
	2	Generates level 1 details, and messages indicating successful handling of MASTER constructs, SINGLE constructs, CRITICAL constructs, ORDERED constructs, ATOMIC directives, and so forth.

Optimizer phase	The level specified in option [q or Q]opt-report	Description
par	1	Reports which loops were parallelized.
	2	Generates level 1 details, and reports which loops were not parallelized along with a short reason.
	3	Generates level 2 details, and prints the memory locations that are categorized as private, shared, reduction, etc..
	4	For this phase, this is the same as specifying level 3.
	5	Generates level 4 details, and dependency edges that inhibit parallelization.
pgo	1	During profile feedback, generates report status of feedback (such as, profile used, no profile available, or unable to use profile) for each routine compiled.
	2	Generates level 1 details, and reports which value profile specializations took place for indirect calls and arithmetic operations.
	3	Generates level 2 details, and reports which indirect calls had profile data, but did not meet the internal threshold limits for the percentage or execution count.
tcollect	1	Generates a list of routines and whether each was selected for trace collection.
vec	1	Reports which loops were vectorized.
	2	Generates level 1 details and reports which loops were not vectorized along with short reason.
	3	Generates level 2 details, and vectorizer loop summary information.

Optimizer phase	The level specified in option [q or Q]opt-report	Description
	4	Generates level 3 details, and greater detail about vectorized and non-vectorized loops.
	5	Generates level 4 details, and details about any proven or assumed data dependences.

See Also

qopt-report, Qopt-report compiler option

qopt-report-help, Qopt-report-help compiler option

qopt-report-routine, Qopt-report-routine

Tells the compiler to generate an optimization report for each of the routines whose names contain the specified substring.

Syntax

Linux OS and macOS:

`-qopt-report-routine=`*substring*

Windows OS:

`/Qopt-report-routine:`*substring*

Arguments

substring	Is the text (string) to look for.

Default

OFF	No optimization report is generated.

Description

This option tells the compiler to generate an optimization report for each of the routines whose names contain the specified substring.

You can also specify a sequence of substrings separated by commas. If you do this, the compiler will generate an optimization report for each of the routines whose name contains one or more of these substrings.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

Visual Studio: **Diagnostics > Optimization Diagnostic Routine**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

qopt-report, Qopt-report compiler option

qopt-report-names, Qopt-report-names

Specifies whether mangled or unmangled names should appear in the optimization report.

Syntax

Linux OS and macOS:

`-qopt-report-names=`*keyword*

Windows OS:

`/Qopt-report-names:`*keyword*

Arguments

keyword	Specifies the form for the names. You can specify one of the following:	
	`mangled`	Indicates that the optimization report should contain mangled names.
	`unmangled`	Indicates that the optimization report should contain unmangled names.

Default

OFF	No optimization report is generated.

Description

This option specifies whether mangled or unmangled names should appear in the optimization report. If you use this option, you must specify either `mangled` or `unmangled`.

If this option is not specified, unmangled names are used by default.

If you specify `mangled`, encoding (also known as decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the assembly listing.

If you specify `unmangled`, no encoding (or decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the source listing.

If you use this option, you do not have to specify option `[q or Q]opt-report`.

When optimization reporting is enabled, the default is `-qopt-report-phase=all` (Linux* and macOS*) or `/Qopt-report-phase:all` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

qopt-report, Qopt-report compiler option

tcollect, Qtcollect

Inserts instrumentation probes calling the Intel® Trace Collector API.

Syntax

Linux OS:

`-tcollect[lib]`

macOS:

`None`

Windows OS:

`/Qtcollect[:lib]`

Arguments

lib	Is one of the Intel® Trace Collector libraries; for example, VT, VTcs, VTmc, or VTfs. If you do not specify *lib*, the default library is VT.

Default

OFF	Instrumentation probes are not inserted into compiled applications.

Description

This option inserts instrumentation probes calling the Intel® Trace Collector API.

This trace analyzing/collecting feature requires installation of another product. For more information, see Feature Requirements.

This option provides a flexible and convenient way of instrumenting functions of a compiled application. For every function, the entry and exit points are instrumented at compile time to let the Intel® Trace Collector record functions beyond the default MPI calls. For non-MPI applications (for example, threaded or serial), you must ensure that the Intel® Trace Collector is properly initialized (VT_initialize/VT_init).

Caution
Be careful with full instrumentation because this feature can produce very large trace files.

IDE Equivalent

None

Alternate Options

None

See Also

tcollect-filter, Qtcollect-filter
compiler option

tcollect-filter, Qtcollect-filter

Lets you enable or disable the instrumentation of specified functions. You must also specify option [Q]tcollect.

Syntax

Linux OS:

`-tcollect-filter` *filename*

macOS:

None

Windows OS:

`/Qtcollect-filter:`*filename*

Arguments

filename	Is a configuration file that lists filters, one per line. Each filter consists of a regular expression string and a switch. Strings with leading or trailing white spaces must be quoted. Other strings do not have to be quoted. The switch value can be ON, on, OFF, or off.

Default

OFF	Functions are not instrumented. However, if option `-tcollect` (Linux) or `/Qtcollect` (Windows) is specified, the filter setting is ".* ON" and all functions get instrumented.

Description

This option lets you enable or disable the instrumentation of specified functions.

To get instrumentation with a specified filter (or filters), you must specify both option `[Q]tcollect` and option `[Q]tcollect-filter`.

During instrumentation, the regular expressions in the file are matched against the function names. The switch specifies whether matching functions are to be instrumented or not. Multiple filters are evaluated from top to bottom with increasing precedence.

The names of the functions to match against are formatted as follows:

- The source file name is followed by a colon-separated function name. Source file names should contain the full path, if available. For example:

```
/home/joe/src/file.f:FOO_bar
```

- Classes and function names are separated by double colons. For example:

```
/home/joe/src/file.fpp:app::foo::bar
```

You can use option `[q or Q]opt-report` to get a full list of file and function names that the compiler recognizes from the compilation unit. This list can be used as the basis for filtering in the configuration file.

This trace analyzing/collecting feature requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

None

Alternate Options

None

Consider the following filters in a configuration file:

```
'.*' OFF '.*vector.*' ON
```

The above will cause instrumentation of only those functions having the string 'vector' in their names. No other function will be instrumented. Note that reversing the order of the two lines will prevent instrumentation of all functions.

To get a list of the file or routine strings that can be matched by the regular expression filters, generate an optimization report with tcollect information. For example:

```
Windows: ifort /Qtcollect /Qopt-report /Qopt-report-phase:tcollect
```

```
Linux: ifort -tcollect -qopt-report -qopt-report-phase=tcollect
```

See Also

tcollect, Qtcollect
compiler option

qopt-report, Qopt-report
compiler option

OpenMP* Options and Parallel Processing Options

fmpc-privatize

Enables or disables privatization of all static data for the MultiProcessor Computing environment (MPC) unified parallel runtime.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`-fmpc-privatize`

`-fno-mpc-privatize`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-fno-mpc-privatize`	The privatization of all static data for the MPC unified parallel runtime is disabled.

Description

This option enables or disables privatization of all static data for the MultiProcessor Computing environment (MPC) unified parallel runtime.

Option `-fmpc-privatize` causes calls to extended thread-local-storage (TLS) resolution, run-time routines that are not supported on standard Linux* distributions.

This option requires installation of another product. For more information, see Feature Requirements.

IDE Equivalent

None

Alternate Options

None

par-affinity, Qpar-affinity

Specifies thread affinity.

Syntax

Linux OS:

`-par-affinity=[`*modifier*`,...]`*type*`[,`*permute*`][,`*offset*`]`

macOS:

`None`

Windows OS:

`/Qpar-affinity:[`*modifier*`,...]`*type*`[,`*permute*`][,`*offset*`]`

Arguments

modifier	Is one of the following values: `granularity={fine|thread|core|tile}`, `[no]respect`, `[no]verbose`, `[no]warnings`, `proclist=proc_list`. The default is `granularity=core, respect, and noverbose`. For information on value `proclist`, see Thread Affinity Interface.
type	Indicates the thread affinity. This argument is required and must be one of the following values: `compact`, `disabled`, `explicit`, `none`, `scatter`, `logical`, `physical`. The default is `none`. Values `logical` and `physical` are deprecated. Use `compact` and `scatter`, respectively, with no *permute* value.
permute	Is a positive integer. You cannot use this argument with *type* setting `explicit`, `none`, or `disabled`. The default is 0.
offset	Is a positive integer. You cannot use this argument with *type* setting `explicit`, `none`, or `disabled`. The default is 0.

Default

OFF	The thread affinity is determined by the run-time environment.

Description

This option specifies thread affinity, which binds threads to physical processing units. It has the same effect as environment variable KMP_AFFINITY.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- You have specified option `[Q]parallel` or option `[q or Q]openmp` (or both).
- You are compiling the main program.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

See Also

`parallel, Qparallel` compiler option
`qopt-report, Qopt-report` compiler option

par-num-threads, Qpar-num-threads

Specifies the number of threads to use in a parallel region.

Syntax

Linux OS and macOS:

`-par-num-threads=`*n*

Windows OS:

`/Qpar-num-threads:`*n*

Arguments

n	Is the number of threads to use. It must be a positive integer.

Default

OFF	The number of threads to use is determined by the run-time environment.

Description

This option specifies the number of threads to use in a parallel region. It has the same effect as environment variable OMP_NUM_THREADS.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- You have specified option `[Q]parallel` or option `[q or Q]openmp` (or both).
- You are compiling the main program.

IDE Equivalent

None

Alternate Options

None

See Also

`parallel, Qparallel` compiler option

`qopt-report, Qopt-report` compiler option

par-runtime-control, Qpar-runtime-control

Generates code to perform run-time checks for loops that have symbolic loop bounds.

Syntax

Linux OS and macOS:

`-par-runtime-control[n]`

`-no-par-runtime-control`

Windows OS:

`/Qpar-runtime-control[n]`

`/Qpar-runtime-control-`

Arguments

n

Is a value denoting what kind of runtime checking to perform. Possible values are:

`0`	Performs no runtime check based on auto-parallelization. This is the same as specifying `-no-par-runtime-control` (Linux* and macOS*) or `/Qpar-runtime-control-` (Windows*).
`1`	Generates runtime check code under conservative mode. This is the default if you do not specify *n*.
`2`	Generates runtime check code under heuristic mode.
`3`	Generates runtime check code under aggressive mode.

Default

`-no-par-runtime-control` or `/Qpar-runtime-control-`	The compiler uses default heuristics when checking loops.

Description

This option generates code to perform run-time checks for loops that have symbolic loop bounds.

If the granularity of a loop is greater than the parallelization threshold, the loop will be executed in parallel.

If you do not specify this option, the compiler may not parallelize loops with symbolic loop bounds if the compile-time granularity estimation of a loop can not ensure it is beneficial to parallelize the loop.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

par-schedule, Qpar-schedule

Lets you specify a scheduling algorithm for loop iterations.

Syntax

Linux OS and macOS:

`-par-schedule-`*keyword*`[=`*n*`]`

Windows OS:

`/Qpar-schedule-`*keyword*`[[:]`*n*`]`

Arguments

keyword

Specifies the scheduling algorithm or tuning method. Possible values are:

`auto`	Lets the compiler or run-time system determine the scheduling algorithm.
`static`	Divides iterations into contiguous pieces.
`static-balanced`	Divides iterations into even-sized chunks.
`static-steal`	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
`dynamic`	Gets a set of iterations dynamically.
`guided`	Specifies a minimum number of iterations.
`guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution.
`runtime`	Defers the scheduling decision until run time.

n

Is the size of the chunk or the number of iterations for each chunk. This setting can only be specified for static, dynamic, and guided. For more information, see the descriptions of each keyword below.

Default

static-balanced

Iterations are divided into even-sized chunks and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

Description

This option lets you specify a scheduling algorithm for loop iterations. It specifies how iterations are to be divided among the threads of the team.

This option is only useful when specified with option `[Q]parallel`.

This option affects performance tuning and can provide better performance during auto-parallelization. It does nothing if it is used with option `[q or Q]openmp`.

Option	Description
`[Q]par-schedule-auto`	Lets the compiler or run-time system determine the scheduling algorithm. Any possible mapping may occur for iterations to threads in the team.
`[Q]par-schedule-static`	Divides iterations into contiguous pieces (chunks) of size *n*. The chunks are assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. If no *n* is specified, the iteration space is divided into chunks that are approximately equal in size, and each thread is assigned at most one chunk.
`[Q]par-schedule-static-balanced`	Divides iterations into even-sized chunks. The chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
`[Q]par-schedule-static-steal`	Divides iterations into even-sized chunks, but when a thread completes its chunk, it can steal parts of chunks assigned to neighboring threads. Each thread keeps track of L and U, which represent the lower and upper bounds of its chunks respectively. Iterations are executed starting from the lower bound, and simultaneously, L is updated to represent the new lower bound.
`[Q]par-schedule-dynamic`	Can be used to get a set of iterations dynamically. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations. Each chunk contains *n* iterations, except for the last chunk to be assigned, which may have fewer iterations. If no *n* is specified, the default is 1.
`[Q]par-schedule-guided`	Can be used to specify a minimum number of iterations. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk of size 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1.

Option	Description
	For an *n* with value *k* (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). If no *n* is specified, the default is 1.
`[Q]par-schedule-guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution. The method depends on run-time implementation. Loop bounds are calculated with faster synchronization and chunks are dynamically dispatched at run time by threads in the team.
`[Q]par-schedule-runtime`	Defers the scheduling decision until run time. The scheduling algorithm and chunk size are then taken from the setting of environment variable OMP_SCHEDULE.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

None

Alternate Options

None

par-threshold, Qpar-threshold

Sets a threshold for the auto-parallelization of loops.

Syntax

Linux OS and macOS:

`-par-threshold[n]`

Windows OS:

`/Qpar-threshold[[:]n]`

Arguments

n

Is an integer whose value is the threshold for the auto-parallelization of loops. Possible values are 0 through 100.

If *n* is 0, loops get auto-parallelized always, regardless of computation work volume.

If *n* is 100, loops get auto-parallelized when performance gains are predicted based on the compiler analysis data. Loops get auto-parallelized only if profitable parallel execution is almost certain.

The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to parallelize only if there is a 50% probability of the code speeding up if executed in parallel.

Default

`-par-threshold100` or `/Qpar-threshold100`	Loops get auto-parallelized only if profitable parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. To use this option, you must also specify option `[Q]parallel`.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

NOTE
This option may behave differently on Intel® microprocessors than on non-Intel microprocessors.

IDE Equivalent

Visual Studio: **Optimization > Threshold For Auto-Parallelization**

Eclipse: None

Xcode: **Optimization > Threshold For Auto-Parallelization**

Alternate Options

None

parallel, Qparallel

Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

Syntax

Linux OS and macOS:

`-parallel`

Windows OS:

`/Qparallel(or /Qpar)`

Arguments

None

Default

OFF	Multithreaded code is not generated for loops that can be safely executed in parallel.

Description

This option tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

To use this option, you must also specify option `O2` or `O3`.

This option sets option `[q or Q]opt-matmul` if option `O3` is also specified.

NOTE
On macOS* systems, when you enable automatic parallelization, you must also set the DYLD_LIBRARY_PATH environment variable within Xcode* or an error will be displayed.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` or `/Qx` (Windows*) or `-m` or `-x` (Linux* and macOS*).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

IDE Equivalent

Visual Studio: **Optimization > Parallelization**

Eclipse: None

Xcode: **Optimization > Parallelization**

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option
`par-affinity, Qpar-affinity` compiler option
`par-num-threads, Qpar-num-threads` compiler option
`par-runtime-control, Qpar-runtime-control` compiler option
`par-schedule, Qpar-schedule` compiler option
`qopt-matmul, Qopt-matmul` compiler option
`O` compiler option

parallel-source-info, Qparallel-source-info

Enables or disables source location emission when OpenMP or auto-parallelism code is generated.*

Syntax

Linux OS and macOS:

`-parallel-source-info[=n]`

`-no-parallel-source-info`

Windows OS:

`/Qparallel-source-info`

`/Qparallel-source-info-[:n]`

Arguments

n

Is the level of source location emission. Possible values are:

`0`	Disables the emission of source location information when OpenMP* code or auto-parallelism code is generated. This is the same as specifying `-no-parallel-source-info` (Linux* and macOS*) or `/Qparallel-source-info-` (Windows*).
`1`	Tells the compiler to emit routine name and line information. This is the same as specifying `[Q]parallel-source-info` with no *n*.
`2`	Tells the compiler to emit path, file, routine name, and line information.

Default

`-parallel-source-info=1` or `/Qparallel-source-info:1`	When OpenMP* code or auto-parallelism code is generated, the routine name and line information is emitted.

Description

This option enables or disables source location emission when OpenMP code or auto-parallelism code is generated. It also lets you set the level of emission.

IDE Equivalent

None

Alternate Options

None

qopenmp, Qopenmp

Enables the parallelizer to generate multi-threaded code based on OpenMP directives.*

Syntax

Linux OS and macOS:

`-qopenmp`

`-qno-openmp`

Windows OS:

`/Qopenmp`

`/Qopenmp-`

Arguments

None

Default

`-qno-openmp` No OpenMP* multi-threaded code is generated by the compiler.

or
`/Qopenmp-`

Description

This option enables the parallelizer to generate multi-threaded code based on OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

If you use this option, multithreaded libraries are used, but option `fpp` is not automatically invoked.

This option sets option `automatic`.

This option works with any optimization level. Specifying no optimization (`-O0` on Linux* or `/Od` on Windows*) helps to debug OpenMP applications.

To ensure that a threadsafe and/or reentrant run-time library is linked and correctly initialized, option `[q or Q]openmp` should also be used for the link step and for the compilation of the main routine.

NOTE
On macOS* systems, when you enable OpenMP* API, you must also set the DYLD_LIBRARY_PATH environment variable within Xcode* or an error will be displayed.

NOTE
Options that use OpenMP* API are available for both Intel® microprocessors and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors versus non-Intel microprocessors include: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, thread affinity, and binding.

IDE Equivalent

Visual Studio: **Language > Process OpenMP Directives**

Eclipse: None

Xcode: **Language > Process OpenMP Directives**

Alternate Options

Linux and macOS*: `-fopenmp`

Windows: `/openmp`

See Also

`qopenmp-stubs, Qopenmp-stubs` compiler option

qopenmp-lib, Qopenmp-lib

Lets you specify an OpenMP run-time library to use for linking.*

Syntax

Linux OS:

`-qopenmp-lib=`*type*

macOS:

`-qopenmp-lib=`*type*

Windows OS:

`/Qopenmp-lib:`*type*

Arguments

type

Specifies the type of library to use; it implies compatibility levels. Currently, the only possible value is:

`compat`

Tells the compiler to use the compatibility OpenMP* run-time library (libiomp). This setting provides compatibility with object files created using Microsoft* and GNU* compilers.

Default

`-qopenmp-lib=compat` or `/Qopenmp-lib:compat`	The compiler uses the compatibility OpenMP* run-time library (libiomp).

Description

This option lets you specify an OpenMP* run-time library to use for linking.

The compatibility OpenMP run-time libraries are compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) or the GNU OpenMP run-time library (libgomp).

To use the compatibility OpenMP run-time library, compile and link your application using the compat setting for option `[q or Q]openmp-lib`. To use this option, you must also specify one of the following compiler options:

- Linux* systems: `-qopenmp` or `-qopenmp-stubs`
- Windows* systems: `/Qopenmp` or `/Qopenmp-stubs`

On Windows* systems, the compatibility OpenMP* run-time library lets you combine OpenMP* object files compiled with the Microsoft* C/C++ compiler with OpenMP* object files compiled with the Intel® C, Intel® C++, or Intel® Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

On Linux* systems, the compatibility Intel OpenMP* run-time library lets you combine OpenMP* object files compiled with the GNU* gcc or gfortran compilers with similar OpenMP* object files compiled with the Intel® C, Intel® C++, or Intel® Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

You cannot link object files generated by the Intel® Fortran compiler to object files compiled by the GNU Fortran compiler, regardless of the presence or absence of the `[Q]openmp` compiler option. This is because the Fortran run-time libraries are incompatible.

NOTE
The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compilers earlier than 10.0.

NOTE
On Windows* systems, this option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker. On Linux* and macOS* systems, this option is processed by the `ifort` command that initiates linking, adding library names explicitly to the link command.

IDE Equivalent

None

Alternate Options

None

See Also

`qopenmp, Qopenmp` compiler option

`qopenmp-stubs, Qopenmp-stubs` compiler option

qopenmp-link

Controls whether the compiler links to static or dynamic OpenMP run-time libraries.*

Syntax

Linux OS:

`-qopenmp-link=`*`library`*

macOS:

`-qopenmp-link=`*`library`*

Windows OS:

`None`

Arguments

library

Specifies the OpenMP library to use. Possible values are:

`static`	Tells the compiler to link to static OpenMP run-time libraries. Note that static OpenMP libraries are deprecated.
`dynamic`	Tells the compiler to link to dynamic OpenMP run-time libraries.

Default

`-qopenmp-link=dynamic`

The compiler links to dynamic OpenMP* run-time libraries. However, if Linux* option `-static` is specified, the compiler links to static OpenMP run-time libraries.

Description

This option controls whether the compiler links to static or dynamic OpenMP* run-time libraries.

To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify `-qopenmp-link=static`. However, we strongly recommend you use the default setting, `-qopenmp-link=dynamic`.

NOTE
Compiler options `-static-intel` and `-shared-intel` (Linux* and macOS*) have no effect on which OpenMP run-time library is linked.

NOTE
On Linux* systems, `-qopenmp-link=dynamic` cannot be used in conjunction with option `-static`. If you try to specify both options together, an error will be displayed.

NOTE
On Linux systems, the OpenMP runtime library depends on using libpthread and libc (libgcc when compiled with gcc). Libpthread and libc (libgcc) must both be static or both be dynamic. If both libpthread and libc (libgcc) are static, then the static version of the OpenMP runtime should be used. If both libpthread and libc (libgcc) are dynamic, then either the static or dynamic version of the OpenMP runtime may be used.

IDE Equivalent

None

Alternate Options

None

qopenmp-offload

Enables or disables OpenMP offloading compilation for the TARGET directives.*

Syntax

Linux OS:

`-qopenmp-offload[=device]`

`-qno-openmp-offload`

macOS:

`None`

Windows OS:

`None`

Arguments

device Specifies the default device for target pragmas. Possible values are:

`host`	OpenMP* offloading constructs are ignored. For Openmp* combined offload constructs, only the offloading part is ignored.

None

Default

`-qno-openmp-offload`	OpenMP* offloading compilation is disabled. However, if option `qopenmp` is specified, the default is ON and OpenMP offloading compilation is enabled.

Description

This option enables or disables OpenMP* offloading compilation for the TARGET directives. When enabling offloading, it lets you specify what the default target device should be for the TARGET directives. .

NOTE
The TARGET directives are only available on Linux* systems.

You can also use this option if you want to enable or disable the offloading feature with no impact on other OpenMP* features. In this case, no OpenMP runtime library is needed to link and the compiler does not need to generate OpenMP runtime initialization code.

If you specify this option with the `qopenmp` option, it can impact other OpenMP* features.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following:

```
-qno-openmp -qopenmp-offload
```

The above is equivalent to specifying only `qopenmp-offload`. In this case, only the offload library is linked, not the OpenMP* library, and only the !$OMP directives for TARGET are processed but no other !$OMP directives.

Consider the following:

```
-qopenmp -qopenmp-offload
```

In this case, the offload library is linked, the OpenMP library is linked, and OpenMP runtime initialization code is generated.

See Also

`qopenmp, Qopenmp` compiler option
TARGET directive
TARGET DATA directive
TARGET UPDATE directive

qopenmp-simd, Qopenmp-simd

Enables or disables OpenMP SIMD compilation.*

Syntax

Linux OS and macOS:

`-qopenmp-simd`

`-qno-openmp-simd`

Windows OS:

`/Qopenmp-simd`

`/Qopenmp-simd-`

Arguments

None

Default

`-qopenmp-simd` or `/Qopenmp-simd`	OpenMP* SIMD compilation is enabled if option `O2` or higher is in effect.

OpenMP* SIMD compilation is always disabled at optimization levels of `O1` or lower.

When option `O2` or higher is in effect, OpenMP SIMD compilation can only be disabled by specifying option `-qno-openmp-simd` or `/Qopenmp-simd-`. It is not disabled by specifying option `-qno-openmp` or `/Qopenmp-`.

Description

This option enables or disables OpenMP* SIMD compilation.

You can use this option if you want to enable or disable the SIMD support with no impact on other OpenMP features. In this case, no OpenMP runtime library is needed to link and the compiler does not need to generate OpenMP runtime initialization code.

If you specify this option with the `[q or Q]openmp` option, it can impact other OpenMP features.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following:

```
-qno-openmp -qopenmp-simd    ! Linux or macOS*
/Qopenmp- /Qopenmp-simd      ! Windows
```

The above is equivalent to specifying only `[q or Q]openmp-simd`. In this case, only SIMD support is provided, the OpenMP* library is not linked, and only the !$OMP directives related to SIMD are processed.

Consider the following:

```
-qopenmp -qopenmp-simd       ! Linux or macOS*
/Qopenmp /Qopenmp-simd       ! Windows
```

In this case, SIMD support is provided, the OpenMP library is linked, and OpenMP runtime initialization code is generated.

See Also

`qopenmp, Qopenmp` compiler option
`O` compiler option
SIMD Directive (OpenMP* API) directive

qopenmp-stubs, Qopenmp-stubs

Enables compilation of OpenMP programs in sequential mode.*

Syntax

Linux OS:

`-qopenmp-stubs`

macOS:

`-qopenmp-stubs`

Windows OS:

`/Qopenmp-stubs`

Arguments

None

Default

OFF The library of OpenMP* function stubs is not linked.

Description

This option enables compilation of OpenMP* programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

IDE Equivalent

Visual Studio: **Language > Process OpenMP Directives**

Eclipse: None

Xcode: **Language > Process OpenMP Directives**

Alternate Options

None

See Also

`qopenmp, Qopenmp` compiler option

qopenmp-threadprivate, Qopenmp-threadprivate

Lets you specify an OpenMP threadprivate implementation.*

Syntax

Linux OS:

`-qopenmp-threadprivate=`*type*

macOS:

`None`

Windows OS:

`/Qopenmp-threadprivate:`*type*

Arguments

type	Specifies the type of threadprivate implementation. Possible values are:	
	`legacy`	Tells the compiler to use the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel® compiler. This setting does not provide compatibility with the implementation used by other compilers.

	`compat`	Tells the compiler to use the compatibility OpenMP* threadprivate implementation based on applying the thread-local attribute to each threadprivate variable. This setting provides compatibility with the implementation provided by the Microsoft* and GNU* compilers.

Default

`-qopenmp-threadprivate=legacy` or `/Qopenmp-threadprivate:legacy`	The compiler uses the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel compiler.

Description

This option lets you specify an OpenMP* threadprivate implementation.

The threadprivate implementation of the legacy OpenMP run-time library may not be compatible with object files created using OpenMP run-time libraries supported in other compilers.

To use this option, you must also specify one of the following compiler options:

- Linux* systems: `-qopenmp` or `-qopenmp-stubs`
- Windows* systems: `/Qopenmp` or `/Qopenmp-stubs`

The value specified for this option is independent of the value used for the `[q or Q]openmp-lib` option.

NOTE
On Windows* systems, if you specify option `/Qopenmp-threadprivate:compat`, the compiler does not generate threadsafe code for common blocks in an !$OMP THREADPRIVATE directive unless at least one element in the common block is explicitly initialized. For more information, see the article titled: /Qopenmp-threadprivate:compat doesn't work with uninitialized threadprivate common blocks, which is located in http://intel.ly/1aHhsjc

NOTE
On macOS* systems, legacy is the only type of threadprivate supported. Option `-qopenmp-threadprivate` is not recognized by the compiler.

IDE Equivalent

None

Alternate Options

None

Qpar-adjust-stack

Tells the compiler to generate code to adjust the stack size for a fiber-based main thread.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qpar-adjust-stack:`*n*

Arguments

n	Is the stack size (in bytes) for the fiber-based main thread. It must be a number equal to or greater than zero.

Default

`/Qpar-adjust-stack:0`	No adjustment is made to the main thread stack size.

Description

This option tells the compiler to generate code to adjust the stack size for a fiber-based main thread. This can reduce the stack size of threads.

For this option to be effective, you must also specify option `/Qparallel`.

IDE Equivalent

None

Alternate Options

None

See Also

`parallel, Qparallel` compiler option

Floating-Point Options

fast-transcendentals, Qfast-transcendentals

Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.

Syntax

Linux OS and macOS:

`-fast-transcendentals`

`-no-fast-transcendentals`

Windows OS:

`/Qfast-transcendentals`

`/Qfast-transcendentals-`

Arguments

None

Default

depends on the setting of `-fp-model` (Linux* and macOS*) or `/fp` (Windows*)	If you do not specify option `-[no-]fast-transcendentals`or option /Qfast-transcendentals[-]: • The default is ON if option `-fp-model fast` or `/fp:fast` is specified or is in effect. • The default is OFF if a value-safe setting is specified for `-fp-model` or `/fp` (such as "precise", "source", etc.).

Description

This option enables the compiler to replace calls to transcendental functions with implementations that may be faster but less precise.

It allows the compiler to perform certain optimizations on transcendental functions, such as replacing individual calls to sine in a loop with a single call to a less precise vectorized sine library routine. These optimizations can cause numerical differences that would not otherwise exist if you are also compiling with a value-safe option such as `-fp-model precise` (Linux* and macOS*) or `/fp:precise` (Windows).

For example, you may get different results if you specify option `O0` versus option `O2`, or you may get different results from calling the same function with the same input at different points in your program. If these kinds of numerical differences are problematic, consider using option `-fimf-use-svml` (Linux* and macOS*) or `/Qimf-use-svml` (Windows) as an alternative. When used with a value-safe option such as `-fp-model precise` or `/fp:precise`, option `-fimf-use-svml` or `/Qimf-use-svml` provides many of the positive performance benefits of `[Q]fast-transcendentals` without negatively affecting numeric consistency. For more details, see the description of option `-fimf-use-svml` and `/Qimf-use-svml`.

This option does not affect explicit Short Vector Math Library (SVML) intrinsics. It only affects scalar calls to the standard math library routines.

You cannot use option `-fast-transcendentals` with option `-fp-model strict` and you cannot use option `/Qfast-transcendentals` with option `/fp:strict`.

This option determines the setting for the maximum allowable relative error for math library function results (max-error) if none of the following options are specified:

- `-fimf-accuracy-bits` (Linux* and macOS*) or `/Qimf-accuracy-bits` (Windows*)
- `-fimf-max-error` (Linux and macOS*) or `/Qimf-max-error` (Windows)
- `-fimf-precision` (Linux and macOS*) or `/Qimf-precision` (Windows)

This option enables extra optimization that only applies to Intel® processors.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option
`fimf-use-svml, Qimf-use-svml` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option

fimf-absolute-error, Qimf-absolute-error

Defines the maximum allowable absolute error for math library function results.

Syntax

Linux OS:

`-fimf-absolute-error=`*value*`[:`*funclist*`]`

macOS:

`-fimf-absolute-error=`*value*`[:`*funclist*`]`

Windows OS:

`/Qimf-absolute-error:`*value*`[:`*funclist*`]`

Arguments

value

Is a positive, floating-point number. Errors in math library function results may exceed the maximum relative error (max-error) setting if the absolute-error is less than or equal to *value*.

The format for the number is [digits] [.digits] [{ e | E }[sign]digits]

funclist

Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-absolute-error=0.00001:sin,sinf` (or `/Qimf-absolute-error:0.00001:sin,sinf`) to specify the maximum allowable absolute error for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision

divides. For example you can specify
`-fimf-absolute-error=0.00001:/`
or `/Qimf-absolute-error: 0.00001:/`.

Default

Zero ("0")	An absolute-error setting of 0 means that the function is bound by the relative error setting. This is the default behavior.

Description

This option defines the maximum allowable absolute error for math library function results.

This option can improve run-time performance, but it may decrease the accuracy of results.

This option only affects functions that have zero as a possible return value, such as log, sin, asin, etc.

The relative error requirements for a particular function are determined by options that set the maximum relative error (max-error) and precision. The return value from a function must have a relative error less than the max-error value, or an absolute error less than the absolute-error value.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-absolute-error=0.00001:sin` or `/Qimf-absolute-error:0.00001:sin`, or `-fimf-absolute-error=0.00001:sqrtf` or `/Qimf-absolute-error:0.00001:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to define the maximum allowable absolute error for the single-precision sine function, you would specify `-fimf-absolute-error=sinf` (or `/Qimf-absolute-error:sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-accuracy-bits, Qimf-accuracy-bits

Defines the relative error for math library function results, including division and square root.

Syntax

Linux OS:

`-fimf-accuracy-bits=`*`bits`*`[:`*`funclist`*`]`

macOS:

`-fimf-accuracy-bits=`*`bits`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-accuracy-bits:`*`bits`*`[:`*`funclist`*`]`

Arguments

bits	Is a positive, floating-point number indicating the number of correct bits the compiler should use. The format for the number is [digits] [.digits] [{ e | E }[sign]digits].
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-accuracy-bits=23:sin,sinf` (or `/Qimf-accuracy-bits:23:sin,sinf`) to specify the relative error for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-accuracy-bits=10.0:/f` or `/Qimf-accuracy-bits:10.0:/f`.

Default

-fimf-precision=medium or /Qimf-precision:medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option defines the relative error, measured by the number of correct bits, for math library function results.

The following formula is used to convert bits into ulps: ulps = $2^{p-1-bits}$, where p is the number of the target format mantissa bits (24, 53, and 113 for single, double, and quad precision, respectively).

This option can affect run-time performance and the accuracy of results.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in the following:

- `-fimf-accuracy-bits=23:sinf,cosf,logf` or `/Qimf-accuracy-bits:23:sinf,cosf,logf`
- `-fimf-accuracy-bits=52:sqrt,/,trunc` or `/Qimf-accuracy-bits:52:sqrt,/,trunc`
- `-fimf-accuracy-bits=10:powf` or `/Qimf-accuracy-bits:10:powf`

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

- `-fimf-precision` (Linux* and macOS*) or `/Qimf-precision` (Windows*)
- `-fimf-max-error` (Linux* and macOS*) or `/Qimf-max-error` (Windows*)
- `-fimf-accuracy-bits` (Linux and macOS*) or `/Qimf-accuracy-bits` (Windows)

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `[Q]fast-transcendentals`
- `[Q]prec-div`
- `[Q]prec-sqrt`

- `-fp-model` (Linux and macOS*) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to request the relative error for the single-precision sine function, you would specify `-fimf-accuracy-bits=sinf` (or `/Qimf-accuracy-bits:sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-arch-consistency, Qimf-arch-consistency

Ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture.

Syntax

Linux OS:

`-fimf-arch-consistency=`*value*`[:`*funclist*`]`

macOS:

`-fimf-arch-consistency=`*value*`[:`*funclist*`]`

Windows OS:

`/Qimf-arch-consistency:`*value*`[:`*funclist*`]`

Arguments

value	Is one of the logical values "true" or "false".
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-arch-consistency=true:sin,sinf` (or `/Qimf-arch-consistency:true:sin,sinf`) to specify consistent results for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-arch-consistency=true:/` or `/Qimf-arch-consistency:true:/`.

Default

false	Implementations of some math library functions may produce slightly different results on implementations of the same architecture.

Description

This option ensures that the math library functions produce consistent results across different microarchitectural implementations of the same architecture (for example, across different microarchitectural implementations of IA-32 architecture). Consistency is only guaranteed for a single binary. Consistency is not guaranteed across different architectures. For example, consistency is not guaranteed across IA-32 architecture and Intel® 64 architecture.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-arch-consistency=true:sin` or `/Qimf-arch-consistency:true:sin`, or `-fimf-arch-consistency=false:sqrtf` or `/Qimf-arch-consistency:false:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

The `-fimf-arch-consistency` (Linux* and macOS*) and `/Qimf-arch-consistency` (Windows*) option may decrease run-time performance, but the option will provide bit-wise consistent results on all Intel® processors and compatible, non-Intel processors, regardless of micro-architecture. This option may not provide bit-wise consistent results between different architectures.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to ensure consistent results for the single-precision sine function, you would specify `-fimf-arch-consistency=sinf` (or `/Qimf-arch-consistency:sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fimf-precision, Qimf-precision` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-domain-exclusion, Qimf-domain-exclusion

Indicates the input arguments domain on which math functions must provide correct results.

Syntax

Linux OS:

`-fimf-domain-exclusion=`*`classlist`*`[:`*`funclist`*`]`

macOS:

`-fimf-domain-exclusion=`*`classlist`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-domain-exclusion:`*`classlist`*`[:`*`funclist`*`]`

Arguments

classlist

Is one of the following:

- One or more of the following floating-point value classes you can exclude from the function domain without affecting the correctness of your program. The supported class names are:

`extremes`	This class is for values which do not lie within the usual domain of arguments for a given function.
`nans`	This means "x=Nan".
`infinities`	This means "x=infinities".
`denormals`	This means "x=denormal".
`zeros`	This means "x=0".

Each *classlist* element corresponds to a power of two. The exclusion attribute is the logical or of the associated powers of two (that is, a bitmask).

The following shows the current mapping from *classlist* mnemonics to numerical values:

extremes	1
nans	2
infinities	4
denormals	8

zeros	16
none	0
all	31
common	15
other combinations	bitwise OR of the used values

You must specify the integer value that corresponds to the class that you want to exclude.

Note that on excluded values, unexpected results may occur.

- One of the following short-hand tokens:

`none` This means that none of the supported classes are excluded from the domain. To indicate this token, specify 0, as in `-fimf-domain-exclusion=0` (or `/Qimf-domain-exclusion:0`).

`all` This means that all of the supported classes are excluded from the domain. To indicate this token, specify 31, as in `-fimf-domain-exclusion=31` (or `/Qimf-domain-exclusion:31`).

`common` This is the same as specifying extremes,nans,infinities,subnormals. To indicate this token, specify 15 (1 + 2+ 4 + 8), as in `-fimf-domain-exclusion=15` (or `/Qimf-domain-exclusion:15`)

funclist

Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-domain-exclusion=4:sin,sinf` (or `/Qimf-domain-exclusion:4:sin,sinf`) to specify infinities for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example, you can specify:

`-fimf-domain-exclusion=4` or `/Qimf-domain-exclusion:4`

`-fimf-domain-exclusion=5:/,powf`
or `/Qimf-domain-exclusion:5:/,powf`

`-fimf-domain-exclusion=23:log,logf,/,sin,cosf`
or `/Qimf-domain-exclusion:23:log,logf,/,sin,cosf`

If you don't specify argument *funclist*, the domain restrictions apply to all math library functions.

Default

Zero ("0") The compiler uses default heuristics when calling math library functions.

Description

This option indicates the input arguments domain on which math functions must provide correct results. It specifies that your program will function correctly if the functions specified in *funclist* do not produce standard conforming results on the number classes.

This option can affect run-time performance and the accuracy of results. As more classes are excluded, faster code sequences can be used.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-domain-exclusion=subnormals:sin` or `/Qimf-domain-exclusion:subnormals:sin`, or `-fimf-domain-exclusion=extremes:sqrtf` or `/Qimf-domain-exclusion:extremes:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to indicate the input arguments domain for the single-precision sine function, you would specify `-fimf-domain-exclusion=sinf` (or `/Qimf-domain-exclusion:sinf`).

Optimization Notice
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

IDE Equivalent

None

Alternate Options

None

Example

Consider the following single-precision sequence for function exp2f:

Operation:	y = exp2f(x)
Accuracy:	1.014 ulp
Instructions:	4 (2 without fix-up)

The following shows the 2-instruction sequence without the fix-up:

```
vcvtfxpntps2dq  zmm1 {k1}, zmm0, 0x50      // zmm1 <-- rndToInt(2^24 * x)
vexp223ps       zmm1 {k1}, zmm1            // zmm1 <-- exp2(x)
```

However, the above 2-instruction sequence will not correctly process NaNs. To process Nans correctly, the following fix-up must be included following the above instruction sequence:

```
vpxord          zmm2, zmm2, zmm2             // zmm2 <-- 0
vfixupnanps     zmm1 {k1}, zmm0, zmm2 {aaaa} // zmm1 <-- QNaN(x) if x is NaN <F>
```

If the `vfixupnanps` instruction is not included, the sequence correctly processes any arguments except NaN values. For example, the following options generate the 2-instruction sequence:

```
-fimf-domain-exclusion=2:exp2f      <- NaN’s are excluded (2 corresponds to NaNs)
-fimf-domain-exclusion=6:exp2f      <- NaN’s and infinities are excluded (4 corresponds to
infinities; 2 + 4 = 6)
-fimf-domain-exclusion=7:exp2f      <- NaN’s, infinities, and extremes are excluded (1
corresponds to extremes; 2 + 4 + 1 = 7)
-fimf-domain-exclusion=15:exp2f     <- NaN’s, infinities, extremes, and subnormals are excluded
(8 corresponds to subnormals;  2 + 4 + 1 + 8=15)
```

If the `vfixupnanps` instruction is included, the sequence correctly processes any arguments including NaN values. For example, the following options generate the 4-instruction sequence:

```
-fimf-domain-exclusion=1:exp2f      <- only extremes are excluded (1 corresponds to extremes)
-fimf-domain-exclusion=4:exp2f      <- only infinities are excluded (4 corresponds to infinities)
-fimf-domain-exclusion=8:exp2f      <- only subnormals are excluded (8 corresponds to subnormals)
-fimf-domain-exclusion=13:exp2f     <- only extremes, infinities and subnormals are excluded (1
+ 4 + 8 = 13)
```

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-arch-consistency, Qimf-arch-consistency` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-use-svml_Qimf-use-svml` compiler option

fimf-force-dynamic-target, Qimf-force-dynamic-target

Instructs the compiler to use run-time dispatch in calls to math functions.

Syntax

Linux OS:

`-fimf-force-dynamic-target[=`*`funclist`*`]`

macOS:

`-fimf-force-dynamic-target[=`*`funclist`*`]`

Windows OS:

`/Qimf-force-dynamic-target[:`*`funclist`*`]`

Arguments

funclist

Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-dynamic-target=sin,sinf` (or `/Qimf-dynamic-target:sin,sinf`) to specify run-time dispatch for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example, you can specify `-fimf-dynamic-target=/` or `/Qimf-dynamic-target:/`.

Default

OFF

Run-time dispatch is not forced in math libraries calls. The compiler can choose to call a CPU-specific version of a math function if one is available.

Description

This option instructs the compiler to use run-time dispatch in calls to math functions. When this option set to ON, it lets you force run-time dispatch in math libraries calls.

By default, when this option is set to OFF, the compiler often optimizes math library calls using the target CPU architecture-specific information available at compile time through the `[Q]x` and `arch` compiler options.

If you want to target multiple CPU families with a single application or you prefer to choose a target CPU at run time, you can force run-time dispatch in math libraries by using this option.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to use run-time dispatch in calls to the single-precision sine function, you would specify `-fimf-force-dynamic-target=sinf` (or `/Qimf-force-dynamic-target:sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`x, Qx` compiler option
`arch` compiler option
`mtune, tune` compiler option

fimf-max-error, Qimf-max-error

Defines the maximum allowable relative error for math library function results, including division and square root.

Syntax

Linux OS:

`-fimf-max-error=`*`ulps`*`[:`*`funclist`*`]`

macOS:

`-fimf-max-error=`*`ulps`*`[:`*`funclist`*`]`

Windows OS:

`/Qimf-max-error:`*`ulps`*`[:`*`funclist`*`]`

Arguments

ulps	Is a positive, floating-point number indicating the maximum allowable relative error the compiler should use. The format for the number is [digits] [.digits] [{ e | E }[sign]digits].
funclist	Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-max-error=4.0:sin,sinf` (or `/Qimf-max-error=4.0:sin,sinf`) to specify the maximum allowable relative error for both the single-precision and double-precision sine functions. You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-max-error=4.0:/` or `/Qimf-max-error:4.0:/`.

Default

-fimf-precision=medium or /Qimf-precision:medium	The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option defines the maximum allowable relative error, measured in ulps, for math library function results.

This option can affect run-time performance and the accuracy of results.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-max-error=4.0:sin` or `/Qimf-max-error:4.0:sin`, or `-fimf-max-error=4.0:sqrtf` or `/Qimf-max-error:4.0:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

- `-fimf-precision` (Linux* and macOS*) or `/Qimf-precision` (Windows*)
- `-fimf-max-error` (Linux* and macOS*) or `/Qimf-max-error` (Windows*)
- `-fimf-accuracy-bits` (Linux and macOS*) or `/Qimf-accuracy-bits` (Windows)

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `[Q]fast-transcendentals`
- `[Q]prec-div`
- `[Q]prec-sqrt`
- `-fp-model` (Linux and macOS*) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to define the maximum allowable relative error for the single-precision sine function, you would specify `-fimf-max-error=sinf` (or `/Qimf-max-error:sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-precision, Qimf-precision` compiler option

`fimf-use-svml_Qimf-use-svml` compiler option

fimf-precision, Qimf-precision

Lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.

Syntax

Linux OS:

`-fimf-precision[=value[:funclist]]`

macOS:

`-fimf-precision[=value[:funclist]]`

Windows OS:

`/Qimf-precision[:value[:funclist]]`

Arguments

value

Is one of the following values denoting the desired accuracy:

`high`	This is equivalent to max-error = 1.0.
`medium`	This is equivalent to max-error = 4; this is the default setting if the option is specified and *value* is omitted.
`low`	This is equivalent to accuracy-bits = 11 for single-precision functions; accuracy-bits = 26 for double-precision functions.

In the above explanations, max-error means option `-fimf-max-error` (Linux* and macOS*) or `/Qimf-max-error` (Windows*); accuracy-bits means option `-fimf-accuracy-bits` (Linux* and macOS*) or `/Qimf-accuracy-bits` (Windows*).

funclist

Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas.

Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-precision=high:sin,sinf` (or `/Qimf-precision:high:sin,sinf`) to specify high precision for both the single-precision and double-precision sine functions.

You also can specify the symbol /f to denote single-precision divides, symbol / to denote double-precision divides, symbol /l to denote extended-precision divides, and symbol /q to denote quad-precision divides. For example you can specify `-fimf-precision=low:/` or `/Qimf-precision:low:/` and `-fimf-precision=low:/f` or `/Qimf-precision:low:/f`.

Default

medium The compiler uses medium precision when calling math library functions. Note that other options can affect precision; see below for details.

Description

This option lets you specify a level of accuracy (precision) that the compiler should use when determining which math library functions to use.

This option can be used to improve run-time performance if reduced accuracy is sufficient for the application, or it can be used to increase the accuracy of math library functions selected by the compiler.

In general, using a lower precision can improve run-time performance and using a higher precision may reduce run-time performance.

If you need to define the accuracy for a math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sinf, as in `-fimf-precision=low:sin` or `/Qimf-precision:low:sin`, or `-fimf-precision=high:sqrtf` or `/Qimf-precision:high:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

There are three options you can use to express the maximum relative error. They are as follows:

- `-fimf-precision` (Linux* and macOS*) or `/Qimf-precision` (Windows*)
- `-fimf-max-error` (Linux* and macOS*) or `/Qimf-max-error` (Windows*)
- `-fimf-accuracy-bits` (Linux and macOS*) or `/Qimf-accuracy-bits` (Windows)

If more than one of these options are specified, the default value for the maximum relative error is determined by the last one specified on the command line.

If none of the above options are specified, the default values for the maximum relative error are determined by the setting of the following options:

- `[Q]fast-transcendentals`
- `[Q]prec-div`
- `[Q]prec-sqrt`
- `-fp-model` (Linux and macOS*) or `/fp` (Windows)

NOTE
Many routines in libraries LIBM (Math Library) and SVML (Short Vector Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to specify a level of accuracy for the single-precision sine function, you would specify `-fimf-precision=sinf` (or `/Qimf-precision:sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`fimf-absolute-error, Qimf-absolute-error` compiler option
`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option
`fimf-arch-consistency, Qimf-arch-consistency` compiler option
`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option
`fimf-max-error, Qimf-max-error` compiler option
`fast-transcendentals, Qfast-transcendentals` compiler option
`prec-div, Qprec-div` compiler option
`prec-sqrt, Qprec-sqrt` compiler option
`fp-model, fp` compiler option
`fimf-use-svml_Qimf-use-svml` compiler option

fimf-use-svml, Qimf-use-svml

Instructs the compiler to use the Short Vector Math Library (SVML) rather than the Intel® Math Library (LIBM) to implement math library functions.

Syntax

Linux OS:

`-fimf-use-svml=`*value*`[:`*funclist*`]`

macOS:

`-fimf-use-svml=`*value*`[:`*funclist*`]`

Windows OS:

`/Qimf-use-svml:`*value*`[:`*funclist*`]`

Arguments

funclist	Is an optional list of one or more math library functions to which the attribute should be applied. Do not specify the standard Fortran name of the math function; you must specify the actual math library name. If you specify more than one function, they must be separated with commas. Precision-specific variants like sin and sinf are considered different functions, so you would need to use `-fimf-use-svmlt=true:sin,sinf` (or `/Qimf-use-svml:true:sin,sinf`) to specify that both the single-precision and double-precision sine functions should use SVML.

Default

false	Math library functions are implemented using the Intel® Math Library, though other compiler options such as `-fast-transcendentals` or `/Qfast-transcendentals` may give the compiler the flexibility to implement math library functions with either LIBM or SVML.

Description

This option instructs the compiler to implement math library functions using the Short Vector Math Library (SVML). When you specify `-fimf-use-svml=true` or `/Qimf-use-svml:true`, the specific SVML variant chosen is influenced by other compiler options such as `-fimf-precision` (Linux* and macOS*) or `/Qimf-precision` (Windows*) and `-fp-model` (Linux and macOS*) or `/fp` (Windows). This option has no effect on math library functions that are implemented in LIBM but not in SVML.

In value-safe settings of option `-fp-model` (Linux and macOS*) or option `/fp` (Windows) such as precise, this option causes a slight decrease in the accuracy of math library functions, because even the high accuracy SVML functions are slightly less accurate than the corresponding functions in LIBM. Additionally, the SVML functions might not accurately raise floating-point exceptions, do not maintain errno, and are designed to work correctly only in round-to-nearest-even rounding mode.

The benefit of using `-fimf-use-svml=true` or `/Qimf-use-svml:true` with value-safe settings of `-fp-model` (Linux and macOS*) or /fp (Windows) is that it can significantly improve performance by enabling the compiler to efficiently vectorize loops containing calls to math library functions.

If you need to use SVML for a specific math function of a certain precision, specify the function name of the precision that you need. For example, if you want double precision, you can specify :sin; if you want single precision, you can specify :sqrtf, as in `-fimf-use-svml=true:sin` or `/Qimf-use-svml:true:sin`, or `-fimf-use-svml =false:sqrtf` or `/Qimf-use-svml:false:sqrtf`.

If you do not specify any function names, then the setting applies to all functions (and to all precisions). However, as soon as you specify an individual function name, the setting applies only to the function of corresponding precision. So, for example, sinf applies only to the single-precision sine function, sin applies only to the double-precision sine function, sinl applies only to the extended-precision sine function, etc.

NOTE
If you specify option `-mia32` (Linux*) or option `/arch:IA32` (Windows*), vector instructions cannot be used. Therefore, you cannot use Linux* option `-mia32` with option `-fimf-use-svml=true`, and you cannot use Windows* option `/arch:IA32` with option `/Qimf-use-svml:true`.

NOTE
Since SVML functions may raise unexpected floating-point exceptions, be cautious about using features that enable trapping on floating-point exceptions. For example, be cautious about specifying option `-fimf-use-svml=true` with option `-fp-trap`, or option `/Qimf-use-svml:true` with option `/Qfp-trap`. For some inputs to some math library functions, such option combinations may cause your program to trap unexpectedly.

NOTE
The standard Fortran names for the various math intrinsic functions *do not* match the math library names of the math intrinsic functions. You must find the actual math library name that is generated for the relevant Fortran math intrinsic.

One way to do this is to generate assembly code by using options `/Fa` or `/S` on Windows, or option `-S` on Linux. The assembly code will show the actual math library name.

For example, if you create a program that contains a call to SIN(x) where x is declared as REAL(KIND=4) and then use option `/S` on Windows to produce assembly code for the program, the assembly code will show a call to sinf.

Therefore, to request the use of SVML for the single-precision sine function, you would specify `-fimf-use-svml=true: sinf` (or `/Qimf-use-svml:true: sinf`).

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option
`m` compiler option
`arch` compiler option

fltconsistency

Enables improved floating-point consistency.

Syntax

Linux OS and macOS:

```
-fltconsistency
-nofltconsistency
```

Windows OS:

```
/fltconsistency
/nofltconsistency
```

Arguments

None

Default

`nofltconsistency` Improved floating-point consistency is not enabled. This setting provides better accuracy and run-time performance at the expense of less consistent floating-point results.

Description

This option enables improved floating-point consistency and may slightly reduce execution speed. It limits floating-point optimizations and maintains declared precision. It also disables inlining of math library functions.

Floating-point operations are not reordered and the result of each floating-point operation is stored in the target variable rather than being kept in the floating-point processor for use in a subsequent calculation.

For example, the compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. This change can alter the results of floating-point division computations slightly.

Floating-point intermediate results are kept in full 80 bits internal precision. Additionally, all spills/reloads of the X87 floating point registers are done using the internal formats; this prevents accidental loss of precision due to spill/reload behavior over which you have no control.

Specifying this option has the following effects on program compilation:

- Floating-point user variables are not assigned to registers.
- Floating-point arithmetic comparisons conform to IEEE 754.
- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.
- The compiler performs floating-point operations in the order specified without reassociation.

- The compiler does not perform constant folding on floating-point values. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point arithmetic is not performed to ensure that floating-point exceptions are also maintained.
- Whenever an expression is spilled, it is spilled as 80 bits (extended precision), not 64 bits (DOUBLE PRECISION). When assignments to type REAL and DOUBLE PRECISION are made, the precision is rounded from 80 bits down to 32 bits (REAL) or 64 bits (DOUBLE PRECISION). When you do not specify `/fltconsistency`, the extra bits of precision are not always rounded away before the variable is reused.
- Even if vectorization is enabled by the `[Q]x` option, the compiler does not vectorize reduction loops (loops computing the dot product) and loops with mixed precision types. Similarly, the compiler does not enable certain loop transformations. For example, the compiler does not transform reduction loops to perform partial summation or loop interchange.

This option causes performance degradation relative to using default floating-point optimization flags.

On Windows systems, an alternative is to use the `/Qprec` option, which should provide better than default floating-point precision while still delivering good floating-point performance.

The recommended method to control the semantics of floating-point calculations is to use option `-fp-model` (Linux* and macOS*) or `/fp` (Windows*).

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **Floating Point > Improve Floating-Point Consistency**

Alternate Options

`fltconsistency`
Linux and macOS*: `-mieee-fp`
Windows: None

`nofltconsistency`
Linux and macOS*: `-mno-ieee-fp`
Windows: None

See Also

`mp1, Qprec` compiler option
`fp-model, fp` compiler option

fma, Qfma

Determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor.

Syntax

Linux OS:

`-fma`

`-no-fma`

macOS:

`-fma`

`-no-fma`

Windows OS:

`/Qfma`

`/Qfma-`

Arguments

None

Default

`-fma` or `/Qfma`	If the instructions exist on the target processor, the compiler generates fused multiply-add (FMA) instructions. However, if you specify `-fp-model strict` (Linux* and macOS*) or `/fp:strict` (Windows*), but do not explicitly specify `-fma` or `/Qfma`, the default is `-no-fma` or `/Qfma-`.

Description

This option determines whether the compiler generates fused multiply-add (FMA) instructions if such instructions exist on the target processor. When the `[Q]fma` option is specified, the compiler may generate FMA instructions for combining multiply and add operations. When the negative form of the `[Q]fma` option is specified, the compiler must generate separate multiply and add instructions with intermediate rounding.

This option has no effect unless setting CORE-AVX2 or higher is specified for option [Q]x,`-march` (Linux and macOS*), or `/arch` (Windows).

IDE Equivalent

None

See Also

`fp-model, fp` compiler option
`x, Qx` compiler option
`ax, Qax` compiler option
`march` compiler option
`arch` compiler option

fp-model, fp

Controls the semantics of floating-point calculations.

Syntax

Linux OS:

`-fp-model` *keyword*

macOS:

`-fp-model` *keyword*

Windows OS:

`/fp:`*keyword*

Arguments

keyword	Specifies the semantics to be used. Possible values are:

`precise`	Disables optimizations that are not value-safe on floating-point data and rounds intermediate results to source-defined precision.
`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.
`consistent`	The compiler uses default heuristics to determine results for different optimization levels or between different processors of the same architecture.
`strict`	Enables precise and except, disables contractions, and enables the property that allows modification of the floating-point environment.
`source`	Rounds intermediate results to source-defined precision.
`[no-]except` (Linux* and macOS*) or `except[-]` (Windows*)	Determines whether strict floating-point exception semantics are honored.

Default

`-fp-model fast=1` or `/fp:fast=1`	The compiler uses more aggressive optimizations on floating-point calculations.

Description

This option controls the semantics of floating-point calculations.

The *keywords* can be considered in groups:

- Group A: `precise`, `fast`, `strict`
- Group B: `source`
- Group C: `except` (or negative forms `-no-except` or `/except-`)
- Group D: `consistent`

You can specify more than one *keyword*. However, the following rules apply:

- You cannot specify `fast` and `except` together in the same compilation. You can specify any other combination of group A, group B, and group C.
  Since `fast` is the default, you must not specify `except` without a group A or group B *keyword*.
- You should specify only one *keyword* from group A and only one *keyword* from group B. If you try to specify more than one *keyword* from either group A or group B, the last (rightmost) one takes effect.
- If you specify `except` more than once, the last (rightmost) one takes effect.
- If you specify `consistent` and any other keyword from another group, the last (rightmost) one may not fully override the heuristics set by `consistent`.

The floating-point (FP) environment is a collection of registers that control the behavior of FP machine instructions and indicate the current FP status. The floating-point environment may include rounding-mode controls, exception masks, flush-to-zero controls, exception status flags, and other floating-point related features.

Option	Description
`-fp-model precise` or `/fp:precise`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the reproducibility of floating-point computations for serial code, including code vectorized or auto-parallelized by the compiler, but they may slow performance. They do not ensure value safety or run-to-run reproducibility of other parallel code. Run-to-run reproducibility for floating-point reductions in OpenMP* code may be obtained for a fixed number of threads through the KMP_DETERMINISTIC_REDUCTION environment variable. For more information about this environment variable, see topic "Supported Environment Variables". The compiler assumes the default floating-point environment; you are not allowed to modify it. Note that option `fp-model precise` implies `fp-model source` and option `fp:precise` implies `fp:source`. Floating-point exception semantics are disabled by default. To enable these semantics, you must also specify `-fp-model except` or `/fp:except`.
`-fp-model fast[=1|2]` or `/fp:fast[=1|2]`	Tells the compiler to use more aggressive optimizations when implementing floating-point calculations. These optimizations increase speed, but may affect the accuracy or reproducibility of floating-point computations. Specifying fast is the same as specifying `fast=1`. `fast=2` may produce faster and less accurate results. Floating-point exception semantics are disabled by default and they cannot be enabled because you cannot specify fast and except together in the same compilation. To enable exception semantics, you must explicitly specify another keyword (see other keyword descriptions for details). To enable exception semantics, you must explicitly specify another keyword (see other keyword descriptions for details).
`-fp-model consistent` or `/fp:consistent`	The compiler uses default heuristics to generate code that will determine results for different optimization levels or between different processors of the same architecture .

Option	Description
	For more information, see the article titled: *Consistency of Floating-Point Results using the Intel® Compiler*, which is located in http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/
`-fp-model source` or `/fp:source`	This option causes intermediate results to be rounded to the precision defined in the source code. It also implies keyword `precise` unless it is overridden by a keyword from Group A. The compiler assumes the default floating-point environment; you are not allowed to modify it.
`-fp-model except` or `/fp:except`	Tells the compiler to follow strict floating-point exception semantics.

The `-fp-model` and `/fp` options determine the setting for the maximum allowable relative error for math library function results (max-error) if none of the following options are specified:

- `-fimf-accuracy-bits` (Linux* and macOS*) or `/Qimf-accuracy-bits` (Windows*)
- `-fimf-max-error` (Linux and macOS*) or `/Qimf-max-error` (Windows)
- `-fimf-precision` (Linux and macOS*) or `/Qimf-precision` (Windows)
- `[Q]fast-transcendentals`

Option `-fp-model fast` (and `/fp:fast`) sets option `-fimf-precision=medium` (`/Qimf-precision:medium`) and option `-fp-model precise` (and `/fp:precise`) implies `-fimf-precision=high` (and `/Qimf-precision:high`). Option `-fp-model fast=2` (and `/fp:fast2`) sets option `-fimf-precision=medium` (and `/Qimf-precision:medium`) and option `-fimf-domain-exclusion=15` (and `/Qimf-domain-exclusion=15`).

NOTE
This option cannot be used to change the default (source) precision for the calculation of intermediate results.

NOTE
In Microsoft* Visual Studio, when you create a Visual Studio* Fortran project, option `/fp:fast` is set by default. It sets the floating-point model to use more aggressive optimizations when implementing floating-point calculations, which increase speed, but may affect the accuracy or reproducibility of floating-point computations. `/fp:fast` is the general default for the IDE project property for Floating Point Model.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

IDE Equivalent

Visual Studio: **Floating Point > Floating Point Model**

Floating Point > Reliable Floating Point Exceptions Model

Eclipse: None

Xcode: **Floating Point > Floating Point Model**

Floating Point > Reliable Floating Point Exceptions Model

Alternate Options

None

See Also

`O` compiler option (specifically O0)

`Od` compiler option

`mp1, Qprec` compiler option

`fimf-absolute-error, Qimf-absolute-error` compiler option

`fimf-accuracy-bits, Qimf-accuracy-bits` compiler option

`fimf-max-error, Qimf-max-error` compiler option

`fimf-precision, Qimf-precision` compiler option

`fimf-domain-exclusion, Qimf-domain-exclusion` compiler option

`fast-transcendentals, Qfast-transcendentals` compiler option

Supported Environment Variables

The article titled: Consistency of Floating-Point Results using the Intel® Compiler, which is located in http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/

fp-port, Qfp-port

Rounds floating-point results after floating-point operations.

Syntax

Linux OS:

```
-fp-port
-no-fp-port
```

macOS:

```
-fp-port
-no-fp-port
```

Windows OS:

```
/Qfp-port
/Qfp-port-
```

Arguments

None

Default

`-no-fp-port` or `/Qfp-port-`	The default rounding behavior depends on the compiler's code generation decisions and the precision parameters of the operating system.

Description

This option rounds floating-point results after floating-point operations.

This option is designed to be used with the `-mia32` (Linux*) or `/arch:IA32` (Windows*) option on a 32-bit compiler. Under those conditions, the compiler implements floating-point calculations using the x87 instruction set, which uses an internal precision that may be higher than the precision specified in the program.

By default, the compiler may keep results of floating-point operations in this higher internal precision. Rounding to program precision occurs at unspecified points. This provides better performance, but the floating-point results are less deterministic. The `[Q]fp-port` option rounds floating-point results to user-specified precision at assignments and type conversions. This has some impact on speed.

When compiling for newer architectures, the compiler implements floating-point calculations with different instructions, such as Intel® SSE and SSE2. These Intel® Streaming SIMD Extensions round directly to single precision or double precision at every instruction. In these cases, option `[Q]fp-port` has no effect.

IDE Equivalent

Visual Studio: **Floating-Point > Round Floating-Point Results**

Eclipse: None

Xcode: **Floating-Point > Round Floating-Point Results**

Alternate Options

None

See Also

Understanding Floating-point Operations

fp-speculation, Qfp-speculation

Tells the compiler the mode in which to speculate on floating-point operations.

Syntax

Linux OS:

`-fp-speculation=`*mode*

macOS:

`-fp-speculation=`*mode*

Windows OS:

`/Qfp-speculation:`*mode*

Arguments

mode	Is the mode for floating-point operations. Possible values are:

`fast`	Tells the compiler to speculate on floating-point operations.
`safe`	Tells the compiler to disable speculation if there is a possibility that the speculation may cause a floating-point exception.
`strict`	Tells the compiler to disable speculation on floating-point operations.
`off`	This is the same as specifying strict.

Default

`-fp-speculation=fast` or `/Qfp-speculation:fast`	The compiler speculates on floating-point operations. This is also the behavior when optimizations are enabled. However, if you specify no optimizations (`-O0` on Linux*; `/Od` on Windows*), the default is `-fp-speculation=safe` (Linux*) or `/Qfp-speculation:safe` (Windows*).

Description

This option tells the compiler the mode in which to speculate on floating-point operations.

Disabling speculation may prevent the vectorization of some loops containing conditionals. For an example, see the article titled: *Diagnostic 15326: loop was not vectorized: implied FP exception model prevents vectorization*, which is located in https://software.intel.com/en-us/articles/fdiag15326 .

IDE Equivalent

Visual Studio: **Floating Point > Floating-Point Speculation**

Eclipse: None

Xcode: **Floating Point > Floating-Point Speculation**

Alternate Options

None

fp-stack-check, Qfp-stack-check

Tells the compiler to generate extra code after every function call to ensure that the floating-point stack is in the expected state.

Syntax

Linux OS and macOS:

`-fp-stack-check`

Windows OS:

`/Qfp-stack-check`

Arguments

None

Default

OFF	There is no checking to ensure that the floating-point (FP) stack is in the expected state.

Description

This option tells the compiler to generate extra code after every function call to ensure that the floating-point (FP) stack is in the expected state.

By default, there is no checking. So when the FP stack overflows, a NaN value is put into FP calculations and the program's results differ. Unfortunately, the overflow point can be far away from the point of the actual bug. This option places code that causes an access violation exception immediately after an incorrect call occurs, thus making it easier to locate these issues.

IDE Equivalent

Visual Studio: **Floating-Point > Check Floating-Point Stack**

Eclipse: None

Xcode: **Floating-Point > Check Floating-point Stack**

Alternate Options

None

fpe

Allows some control over floating-point exception handling for the main program at run-time.

Syntax

Linux OS and macOS:

`-fpe`*n*

Windows OS:

`/fpe:`*n*

Arguments

n Specifies the floating-point exception handling level. Possible values are:

`0` Floating-point invalid, divide-by-zero, and overflow exceptions are enabled throughout the application when the main program is compiled with this value. If any such exceptions occur, execution is aborted. This option causes subnormal floating-point results to be set to zero. Underflow results will also be set to zero, unless you override this by explicitly specifying option `-no-ftz` or `-fp-model precise` (Linux* and macOS*) or option `/Qftz-` or `/fp:precise` (Windows*).

Underflow results from SSE instructions, as well as x87 instructions, will be set to zero. By contrast, option `[Q]ftz` only sets SSE underflow results to zero.

Sets option `-fp-speculation=strict` (Linux* and macOS*) or `/Qfp-speculation:strict` (Windows*) for any program unit compiled with `-fpe0` (Linux* and macOS*) or `/fpe:0` (Windows*). This disables certain optimizations in cases where speculative execution of floating-point operations could lead to floating-point exceptions that would not occur in the absence of speculation. For example, this may prevent the vectorization of some loops containing conditionals.

Disables certain optimizations that generate calls to the Short Vector Math Library that could lead to floating-point exceptions for extreme input arguments that would not occur if libm was called instead. For example, this may prevent the vectorization of some loops containing calls to transcendental math functions.

To get more detailed location information about where the error occurred, use option `traceback`.

`1` All floating-point exceptions are disabled.

Underflow results from SSE instructions, as well as x87 instructions, will be set to zero.

`3` All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero, such as `[Q]ftz`, `O3`, or `O2`. This setting provides full IEEE support.

Default

`-fpe3` or `/fpe:3` All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero.

Description

This option allows some control over floating-point exception handling at run-time. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported.

The `fpe` option affects how the following conditions are handled:

- When floating-point calculations result in a divide by zero, overflow, or invalid operation.
- When floating-point calculations result in an underflow.
- When a subnormal number or other exceptional number (positive infinity, negative infinity, or a NaN) is present in an arithmetic expression.

When enabled exceptions occur, execution is aborted and the cause of the abort reported to the user. If compiler option `traceback` is specified at compile time, detailed information about the location of the abort is also reported.

This option does not enable underflow exceptions, input subnormal exceptions, or inexact exceptions.

IDE Equivalent

Visual Studio: **Floating-Point > Floating-Point Exception Handling**

Eclipse: None

Xcode: **Floating-Point > Floating-Point Exception Handling**

Alternate Options

None

See Also

`fpe-all` compiler option
`ftz, Qftz` compiler option
`fp-model, fp` compiler option
`fp-speculation, Qfp-speculation` compiler option
`traceback` compiler option

fpe-all

Allows some control over floating-point exception handling for each routine in a program at run-time.

Syntax

Linux OS and macOS:

`-fpe-all=`*n*

Windows OS:

`/fpe-all:`*n*

Arguments

n Specifies the floating-point exception handling level. Possible values are:

`0` Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted. This option sets the `[Q]ftz` option; therefore underflow results will be set to zero unless you explicitly specify `-no-ftz` (Linux and macOS*) or `/Qftz-` (Windows).

To get more detailed location information about where the error occurred, use option `traceback`.

`1` All floating-point exceptions are disabled.

Underflow results from SSE instructions, as well as x87 instructions, will be set to zero.

`3` All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero, such as [Q]ftz, `O3`, or `O2`. This setting provides full IEEE support.

Default

`-fpe-all=3` All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero.

or `/fpe-all:3`

or the setting of `fpe` that the main program was compiled with

Description

This option allows some control over floating-point exception handling for each routine in a program at run-time. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported.

The `fpe-all` option affects how the following conditions are handled:

- When floating-point calculations result in a divide by zero, overflow, or invalid operation.
- When floating-point calculations result in an underflow.
- When a subnormal number or other exceptional number (positive infinity, negative infinity, or a NaN) is present in an arithmetic expression.

The current settings of the floating-point exception and status flags are saved on each routine entry and restored on each routine exit. This may incur some performance overhead.

When option `fpe-all` is applied to a main program, it has the same effect as when option `fpe` is applied to the main program.

When enabled exceptions occur, execution is aborted and the cause of the abort reported to the user. If compiler option `traceback` is specified at compile time, detailed information about the location of the abort is also reported.

This option does not enable underflow exceptions, input subnormal exceptions, or inexact exceptions.

Option `fpe-all` sets option `assume ieee_fpe_flags`.

IDE Equivalent

None

Alternate Options

None

See Also

`assume` compiler option
`fpe` compiler option
`ftz, Qftz` compiler option
`traceback` compiler option

ftz, Qftz

Flushes subnormal results to zero.

Syntax

Linux OS and macOS:

`-ftz`

`-no-ftz`

Windows OS:

`/Qftz`

`/Qftz-`

Arguments

None

Default

`-ftz` or `/Qftz`	Subnormal results are flushed to zero. Every optimization option `O` level, except `O0`, sets `[Q]ftz`. Value 0 for the `[Q]fpe` option sets `[Q]ftz`.

Description

This option flushes subnormal results to zero when the application is in the gradual underflow mode. It may improve performance if the subnormal values are not critical to your application's behavior.

The `[Q]ftz` option has no effect during compile-time optimization.

The [Q]ftz option sets or resets the FTZ and the DAZ hardware flags. If FTZ is ON, subnormal results from floating-point calculations will be set to the value zero. If FTZ is OFF, subnormal results remain as is. If DAZ is ON, subnormal values used as input to floating-point instructions will be treated as zero. If DAZ is OFF, subnormal instruction inputs remain as is. Systems using Intel® 64 architecture have both FTZ and DAZ. FTZ and DAZ are not supported on all IA-32 architectures.

When the [Q]ftz option is used in combination with an SSE-enabling option on systems using IA-32 architecture (for example, the [Q]xSSE2 option), the compiler will insert code in the main routine to set FTZ and DAZ. When [Q]ftz is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a run-time processor check.

If you specify option -no-ftz (Linux and macOS*) or option /Qftz- (Windows), it prevents the compiler from inserting any code that might set FTZ or DAZ.

Option [Q]ftz only has an effect when the main program is being compiled. It sets the FTZ/DAZ mode for the process. The initial thread and any threads subsequently created by that process will operate in FTZ/DAZ mode.

If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ/DAZ mode off by specifying -no-ftz or /Qftz- in the command line while still benefiting from the O3 optimizations.

NOTE
Option [Q]ftz is a performance option. Setting this option does not guarantee that all subnormals in a program are flushed to zero. The option only causes subnormals generated at run time to be flushed to zero.

IDE Equivalent

Visual Studio: IA-32 architecture: **Floating Point > Flush Subnormal Results to Zero**

Intel® 64 architecture: None

Eclipse: None

Xcode: **Floating Point > Flush Subnormal Results to Zero**

Alternate Options

None

See Also

x, Qx compiler option
Setting the FTZ and DAZ Flags

Ge

Enables stack-checking for all functions. This is a deprecated option. The replacement option is /Gs0.

Syntax

Linux OS and macOS:

None

Windows OS:

/Ge

Arguments

None

Default

OFF Stack-checking for all functions is disabled.

Description

This option enables stack-checking for all functions.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/Gs0`

mp1, Qprec

Improves floating-point precision and consistency.

Syntax

Linux OS:

`-mp1`

macOS:

`-mp1`

Windows OS:

`/Qprec`

Arguments

None

Default

OFF The compiler provides good accuracy and run-time performance at the expense of less consistent floating-point results.

Description

This option improves floating-point consistency. It ensures the out-of-range check of operands of transcendental functions and improves the accuracy of floating-point compares.

This option prevents the compiler from performing optimizations that change NaN comparison semantics and causes all values to be truncated to declared precision before they are used in comparisons. It also causes the compiler to use library routines that give better precision results compared to the X87 transcendental instructions.

This option disables fewer optimizations and has less impact on performance than option `fltconsistency`.

This option disables fewer optimizations and has less impact on performance than option `fltconsistency`,`-fp-model precise` (Linux* and macOS*), or option `/fp:precise` (Windows*).

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`fltconsistency` compiler option

pc, Qpc

Enables control of floating-point significand precision.

Syntax

Linux OS:

`-pc`*n*

macOS:

`-pc`*n*

Windows OS:

`/Qpc`*n*

Arguments

n	Is the floating-point significand precision. Possible values are:	
	`32`	Rounds the significand to 24 bits (single precision).
	`64`	Rounds the significand to 53 bits (double precision).
	`80`	Rounds the significand to 64 bits (extended precision).

Default

`-pc80` or `/Qpc64`	On Linux* and macOS* systems, the floating-point significand is rounded to 64 bits. On Windows* systems, the floating-point significand is rounded to 53 bits.

Description

This option enables control of floating-point significand precision.

Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the this option.

Note that a change of the default precision control or rounding mode, for example, by using the `[Q]pc32` option or by user intervention, may affect the results returned by some of the mathematical functions.

IDE Equivalent

None

Alternate Options

None

prec-div, Qprec-div

Improves precision of floating-point divides.

Syntax

Linux OS and macOS:

`-prec-div`

`-no-prec-div`

Windows OS:

`/Qprec-div`

`/Qprec-div-`

Arguments

None

Default

OFF	Default heuristics are used. The default is not as accurate as full IEEE division, but it is slightly more accurate than would be obtained when `/Qprec-div-` or `-no-prec-div` is specified. If you need full IEEE precision for division, you should specify `[Q]prec-div`.

Description

This option improves precision of floating-point divides. It has a slight impact on speed.

At default optimization levels, the compiler may change floating-point division computations into multiplication by the reciprocal of the denominator. For example, A/B is computed as A * (1/B) to improve the speed of the computation.

However, sometimes the value produced by this transformation is not as accurate as full IEEE division. When it is important to have fully precise IEEE division, use this option to disable the floating-point division-to-multiplication optimization. The result is more accurate, with some loss of performance.

If you specify `-no-prec-div` (Linux* and macOS*) or `/Qprec-div-` (Windows*), it enables optimizations that give slightly less precise results than full IEEE division.

Option `[Q]prec-div` is implied by option `-fp-model precise` (Linux* and macOS*) and option `/fp:precise` (Windows*).

IDE Equivalent

None

Alternate Options

None

See Also

`fp-model, fp` compiler option

prec-sqrt, Qprec-sqrt

Improves precision of square root implementations.

Syntax

Linux OS and macOS:

`-prec-sqrt`

`-no-prec-sqrt`

Windows OS:

`/Qprec-sqrt`

`/Qprec-sqrt-`

Arguments

None

Default

`-no-prec-sqrt` or `/Qprec-sqrt-`	The compiler uses a faster but less precise implementation of square root. However, the default is `-prec-sqrt` or `/Qprec-sqrt` if any of the following options are specified: `/Od`, `/fltconsistency`, or `/Qprec` on Windows* systems; `-O0`, `-fltconsistency`, or `-mp1` on Linux* and macOS* systems.

Description

This option improves precision of square root implementations. It has a slight impact on speed.

This option inhibits any optimizations that can adversely affect the precision of a square root computation. The result is fully precise square root implementations, with some loss of performance.

IDE Equivalent

None

Alternate Options

None

qsimd-honor-fp-model, Qsimd-honor-fp-model

Tells the compiler to obey the selected floating-point model when vectorizing SIMD loops.

Syntax

Linux OS:

`-qsimd-honor-fp-model`

`-qno-simd-honor-fp-model`

macOS:

`-qsimd-honor-fp-model`

`-qno-simd-honor-fp-model`

Windows OS:

`/Qsimd-honor-fp-model`

`/Qsimd-honor-fp-model-`

Arguments

None

Default

`-qno-simd-honor-fp-model` or `/Qsimd-honor-fp-model-`	The compiler performs vectorization of SIMD loops even if it breaks the floating-point model setting.

Description

The OpenMP* SIMD specification and the setting of compiler option `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) can contradict in requirements. When contradiction occurs, the default behavior of the compiler is to follow the OpenMP* specification and therefore vectorize the loop.

This option lets you override this default behavior - it causes the compiler to follow the `-fp-model` (or `/fp`) specification. This means that the compiler will serialize the loop.

> **NOTE**
> This option does not affect automatic vectorization of loops. By default, the compiler uses `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) settings for this.

IDE Equivalent

None

Alternate Options

None

See Also

`qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction` compiler option
`fp-model, fp` compiler option
SIMD Loop Directive directive
SIMD Directive (OpenMP* API) directive

qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction

Tells the compiler to serialize floating-point reduction when vectorizing SIMD loops.

Syntax

Linux OS:

`-qsimd-serialize-fp-reduction`

`-qno-simd-serialize-fp-reduction`

macOS:

`-qsimd-serialize-fp-reduction`

`-qno-simd-serialize-fp-reduction`

Windows OS:

`/Qsimd-serialize-fp-reduction`

`/Qsimd-serialize-fp-reduction-`

Arguments

None

Default

`-qno-simd-serialize-fp-reduction` or `/Qsimd-serialize-fp-reduction-`	The compiler does not attempt to serialize floating-point reduction in SIMD loops.

Description

The OpenMP* SIMD reduction specification and the setting of compiler option `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) can contradict in requirements. When contradiction occurs, the default behavior of the compiler is to follow OpenMP* specification and therefore vectorize the loop, including floating-point reduction.

This option lets you override this default behavior - it causes the compiler to follow the `-fp-model` (or `/fp`) specification. This means that the compiler will serialize the floating-point reduction while vectorizing the rest of the loop.

NOTE
When `[q or Q]simd-honor-fp-model` is specified and OpenMP* SIMD reduction specification is the only thing causing serialization of the entire loop, addition of option `[q or Q]simd-serialize-fp-reduction` will result in vectorization of the entire loop except for reduction calculation, which will be serialized.

NOTE
This option does not affect automatic vectorization of loops. By default, the compiler uses `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) settings for this.

IDE Equivalent

None

Alternate Options

None

See Also

`qsimd-honor-fp-model, Qsimd-honor-fp-model` compiler option
`fp-model, fp` compiler option
SIMD Loop Directive directive
SIMD Directive (OpenMP* API) directive

rcd, Qrcd

Enables fast float-to-integer conversions. This is a deprecated option. There is no replacement option.

Syntax

Linux OS:

`-rcd`

macOS:

`-rcd`

Windows OS:

`/Qrcd`

Arguments

None

Default

OFF Floating-point values are truncated when a conversion to an integer is involved.

Description

This option enables fast float-to-integer conversions. It can improve the performance of code that requires floating-point-to-integer conversions.

The system default floating-point rounding mode is round-to-nearest. However, the Fortran language requires floating-point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point-to-integer conversion and change it back afterwards.

This option disables the change to truncation of the rounding mode for all floating-point calculations, including floating point-to-integer conversions. This option can improve performance, but floating-point conversions to integer will not conform to Fortran semantics.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/QIfist` (this is a deprecated option)

recursive

Tells the compiler that all routines should be compiled for possible recursive execution.

Syntax

Linux OS and macOS:

`-recursive`

`-norecursive`

Windows OS:

`/recursive`

`/norecursive`

Arguments

None

Default

`norecursive` Routines are not compiled for possible recursive execution.

Description

This option tells the compiler that all routines should be compiled for possible recursive execution. It sets the `automatic` option.

> **NOTE**
> This option will be deprecated in a future release. We recommend you use its replacement option: `assume [no]recursion`.

IDE Equivalent

Visual Studio: **Code Generation > Enable Recursive Routines**

Eclipse: None

Xcode: **Code Generation > Enable Recursive Routines**

Alternate Options

Linux and macOS*: `-assume [no]recursion`

Windows: `/assume:[no]recursion`

See Also

`auto` compiler option

`assume` compiler option, setting [no]recursion

Inlining Options

finline

Tells the compiler to inline functions declared with !DIR$ ATTRIBUTES FORCEINLINE.

Syntax

Linux OS:

`-finline`

`-fno-inline`

macOS:

`-finline`

`-fno-inline`

Windows OS:

None

Arguments

None

Default

`-fno-inline`	The compiler does not inline functions declared with !DIR$ ATTRIBUTES FORCEINLINE.

Description

This option tells the compiler to inline functions declared with !DIR$ ATTRIBUTES FORCEINLINE.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-inline-level`

Windows: `/Ob`

finline-functions

Enables function inlining for single file compilation.

Syntax

Linux OS:

`-finline-functions`

`-fno-inline-functions`

macOS:

`-finline-functions`

`-fno-inline-functions`

Windows OS:

None

Arguments

None

Default

`-finline-functions`	Interprocedural optimizations occur. However, if you specify `-O0`, the default is OFF.

Description

This option enables function inlining for single file compilation.

It enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

The compiler applies a heuristic to perform the function expansion. To specify the size of the function to be expanded, use the `-finline-limit` option.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-inline-level=2`

Windows: `/Ob2`

See Also

`ip, Qip` compiler option

`finline-limit` compiler option

finline-limit

Lets you specify the maximum size of a function to be inlined.

Syntax

Linux OS and macOS:

`-finline-limit=`*n*

Windows OS:

`None`

Arguments

n	Must be an integer greater than or equal to zero. It is the maximum number of lines the function can have to be considered for inlining.

Default

OFF	The compiler uses default heuristics when inlining functions.

Description

This option lets you specify the maximum size of a function to be inlined. The compiler inlines smaller functions, but this option lets you inline large functions. For example, to indicate a large function, you could specify 100 or 1000 for *n*.

Note that parts of functions cannot be inlined, only whole functions.

This option is a modification of the `-finline-functions` option, whose behavior occurs by default.

IDE Equivalent

None

Alternate Options

None

See Also

`finline-functions` compiler option

inline

Specifies the level of inline function expansion.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/inline[:keyword]`

Arguments

keyword Is the level of inline function expansion. Possible values are:

`none` Disables inlining of user-defined functions. This is the same as specifying `manual`.

`manual` Disables inlining of user-defined functions. Fortran statement functions are always inlined.

size Enables inlining of any function. However, the compiler decides which functions are inlined.

This option enables interprocedural optimizations and most speed optimizations.

`speed` Enables inlining of any function. This is the same as specifying `all`.

`all` Enables inlining of any function. However, the compiler decides which functions are inlined.

This option enables interprocedural optimizations and all speed optimizations. This is the same as specifying `inline` with no *keyword*.

Default

OFF The compiler inlines certain functions by default.

Description

This option specifies the level of inline function expansion.

IDE Equivalent

None

Alternate Options

`inline all`
or
`inline speed` Linux and macOS*: None
Windows: `/Ob2/Ot`

`inline size` Linux and macOS*: None
Windows: `/Ob2/Os`

`inline manual` Linux and macOS*: None
Windows: `/Ob0`

`inline none` Linux and macOS*: None
Windows: `/Ob0`

See Also

`finline-functions` compiler option

inline-factor, Qinline-factor

Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.

Syntax

Linux OS and macOS:

`-inline-factor=`*n*

`-no-inline-factor`

Windows OS:

`/Qinline-factor:`*n*

`/Qinline-factor-`

Arguments

n	Is a positive integer specifying the percentage value. The default value is 100 (a factor of 1).

Default

`-inline-factor=100` or `/Qinline-factor:100`	The compiler uses a percentage multiplier of 100.

Description

This option specifies the percentage multiplier that should be applied to all inlining options that define upper limits:

- `[Q]inline-max-size`
- `[Q]inline-max-total-size`
- `[Q]inline-max-per-routine`
- `[Q]inline-max-per-compile`

The `[Q]inline-factor` option takes the default value for each of the above options and multiplies it by *n* divided by 100. For example, if 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2. This option is useful if you do not want to individually increase each option limit.

If you specify `-no-inline-factor` (Linux* and macOS*) or `/Qinline-factor-` (Windows*), the following occurs:

- Every function is considered to be a small or medium function; there are no large functions.
- There is no limit to the size a routine may grow when inline expansion is performed.
- There is no limit to the number of times some routine may be inlined into a particular routine.
- There is no limit to the number of times inlining can be applied to a compilation unit.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase default limits, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-max-size, Qinline-max-size` compiler option
`inline-max-total-size, Qinline-max-total-size` compiler option
`inline-max-per-routine, Qinline-max-per-routine` compiler option
`inline-max-per-compile, Qinline-max-per-compile` compiler option
`qopt-report, Qopt-report` compiler option

inline-forceinline, Qinline-forceinline

Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.

Syntax

Linux OS and macOS:

`-inline-forceinline`

Windows OS:

`/Qinline-forceinline`

Default

OFF The compiler uses default heuristics for inline routine expansion.

Description

This option instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.

Without this option, the compiler treats functions declared with an INLINE attribute as merely being recommended for inlining. When this option is used, it is as if they were declared with the directive !DIR$ ATTRIBUTES FORCEINLINE.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to change the meaning of inline to "forceinline", the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

qopt-report, Qopt-report compiler option

inline-level, Ob

Specifies the level of inline function expansion.

Syntax

Linux OS and macOS:

`-inline-level=n`

Windows OS:

`/Obn`

Arguments

n	Is the inline function expansion level. Possible values are 0, 1, and 2.

Default

`-inline-level=2` or `/Ob2`	This is the default if option `O2` is specified or is in effect by default. On Windows* systems, this is also the default if option `O3` is specified.
`-inline-level=0` or `/Ob0`	This is the default if option `-O0` (Linux* and macOS*) or `/Od` (Windows*) is specified.

Description

This option specifies the level of inline function expansion. Inlining procedures can greatly improve the run-time performance of certain programs.

Option	Description
`-inline-level=0` or `/Ob0`	Disables inlining of user-defined functions. Note that statement functions are always inlined.
`-inline-level=1` or `/Ob1`	Enables inlining when an inline keyword or an inline directive is specified.
`-inline-level=2` or `/Ob2`	Enables inlining of any function at the compiler's discretion.

IDE Equivalent

Visual Studio: **Optimization > Inline Function Expansion**

Eclipse: None

Xcode: **Optimization > Inline Function Expansion**

Alternate Options

None

See Also

inline compiler option

inline-max-per-compile, Qinline-max-per-compile

Specifies the maximum number of times inlining may be applied to an entire compilation unit.

Syntax

Linux OS and macOS:

`-inline-max-per-compile=`*n*

`-no-inline-max-per-compile`

Windows OS:

`/Qinline-max-per-compile=`*n*

`/Qinline-max-per-compile-`

Arguments

n	Is a positive integer that specifies the number of times inlining may be applied.

Default

`-no-inline-max-per-compile` or `/Qinline-max-per-compile-`	The compiler uses default heuristics for inline routine expansion.

Description

This option the maximum number of times inlining may be applied to an entire compilation unit. It limits the number of times that inlining can be applied.

For compilations using Interprocedural Optimizations (IPO), the entire compilation is a compilation unit. For other compilations, a compilation unit is a file.

If you specify `-no-inline-max-per-compile` (Linux* and macOS*) or `/Qinline-max-per-compile-` (Windows*), there is no limit to the number of times inlining may be applied to a compilation unit.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-max-per-routine, Qinline-max-per-routine

Specifies the maximum number of times the inliner may inline into a particular routine.

Syntax

Linux OS and macOS:

`-inline-max-per-routine=n`

`-no-inline-max-per-routine`

Windows OS:

`/Qinline-max-per-routine=n`

`/Qinline-max-per-routine-`

Arguments

n	Is a positive integer that specifies the maximum number of times the inliner may inline into a particular routine.

Default

`-no-inline-max-per-routine` or `/Qinline-max-per-routine-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the maximum number of times the inliner may inline into a particular routine. It limits the number of times that inlining can be applied to any routine.

If you specify `-no-inline-max-per-routine` (Linux* and macOS*) or `/Qinline-max-per-routine-` (Windows*), there is no limit to the number of times some routine may be inlined into a particular routine.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-max-size, Qinline-max-size

Specifies the lower limit for the size of what the inliner considers to be a large routine.

Syntax

Linux OS and macOS:

`-inline-max-size=n`

`-no-inline-max-size`

Windows OS:

`/Qinline-max-size=`*n*

`/Qinline-max-size-`

Arguments

n	Is a positive integer that specifies the minimum size of what the inliner considers to be a large routine.

Default

`-inline-max-size` or `/Qinline-max-size`	The compiler sets the maximum size (*n*) dynamically, based on the platform.

Description

This option specifies the lower limit for the size of what the inliner considers to be a large routine (a function or subroutine). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be medium and large-size routines.

The inliner prefers to inline small routines. It has a preference against inlining large routines. So, any large routine is highly unlikely to be inlined.

If you specify `-no-inline-max-size` (Linux* and macOS*) or `/Qinline-max-size-` (Windows*), there are no large routines. Every routine is either a small or medium routine.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-min-size, Qinline-min-size` compiler option
`inline-factor, Qinline-factor` compiler option
`qopt-report, Qopt-report` compiler option

inline-max-total-size, Qinline-max-total-size

Specifies how much larger a routine can normally grow when inline expansion is performed.

Syntax

Linux OS and macOS:

`-inline-max-total-size=`*n*

`-no-inline-max-total-size`

Windows OS:

`/Qinline-max-total-size=n`

`/Qinline-max-total-size-`

Arguments

n	Is a positive integer that specifies the permitted increase in the routine's size when inline expansion is performed.

Default

`-no-inline-max-total-size` or `/Qinline-max-total-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies how much larger a routine can normally grow when inline expansion is performed. It limits the potential size of the routine. For example, if 2000 is specified for *n*, the size of any routine will normally not increase by more than 2000.

If you specify `-no-inline-max-total-size` (Linux* and macOS*) or `/Qinline-max-total-size-` (Windows*), there is no limit to the size a routine may grow when inline expansion is performed.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

Caution
When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-factor, Qinline-factor` compiler option

`qopt-report, Qopt-report` compiler option

inline-min-caller-growth, Qinline-min-caller-growth

Lets you specify a procedure size n for which procedures of size <= n do not contribute to the estimated growth of the caller when inlined.

Syntax

Linux OS and macOS:

`-inline-min-caller-growth=n`

Windows OS:

`/Qinline-min-caller-growth=n`

Arguments

n	Is a non-negative integer. When *n* > 0, procedures with a size of *n* are treated as if they are size 0.

Default

`-inline-min-caller-growth=0` or `/Qinline-min-caller-growth=0`	The compiler treats procedures as if they have size zero.

Description

This option lets you specify a procedure size *n* for which procedures of size <= *n* do not contribute to the estimated growth of the caller when inlined. It allows you to inline procedures that the compiler would otherwise consider too large to inline.

NOTE
We recommend that you choose a value of *n* <= 10; otherwise, compile time and code size may greatly increase.

IDE Equivalent

None

Alternate Options

None

inline-min-size, Qinline-min-size

Specifies the upper limit for the size of what the inliner considers to be a small routine.

Syntax

Linux OS and macOS:

`-inline-min-size=`*n*

`-no-inline-min-size`

Windows OS:

`/Qinline-min-size=`*n*

`/Qinline-min-size-`

Arguments

n	Is a positive integer that specifies the maximum size of what the inliner considers to be a small routine.

Default

`-no-inline-min-size` or`/Qinline-min-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the upper limit for the size of what the inliner considers to be a small routine (a function or subroutine). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be small and medium-size routines.

The inliner has a preference to inline small routines. So, when a routine is smaller than or equal to the specified size, it is very likely to be inlined.

If you specify `-no-inline-min-size` (Linux* and macOS*) or `/Qinline-min-size-` (Windows*), there is no limit to the size of small routines. Every routine is a small routine; there are no medium or large routines.

To see compiler values for important inlining limits, specify option `[q or Q]opt-report`.

> **Caution**
> When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

IDE Equivalent

None

Alternate Options

None

See Also

`inline-max-size, Qinline-max-size` compiler option

`qopt-report, Qopt-report` compiler option

Qinline-dllimport

Determines whether dllimport functions are inlined.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Qinline-dllimport`

`/Qinline-dllimport-`

Arguments

None

Default

`/Qinline-dllimport` The dllimport functions are inlined.

Description

This option determines whether dllimport functions are inlined. To disable dllimport functions from being inlined, specify `/Qinline-dllimport-`.

IDE Equivalent

None

Alternate Options

None

Output, Debug, and Precompiled Header (PCH) Options

bintext

Places a text string into the object file (.obj) being generated by the compiler.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/bintext:`*`string`*

`/nobintext`

Arguments

`string`	Is the text string to go into the object file.

Default

`/nobintext`	No text string is placed in the object file.

Description

This option places a text string into the object file (.obj) being generated by the compiler. The string also gets propagated into the executable file.

For example, this option is useful if you want to place a version number or copyright information into the object and executable.

If the string contains a space or tab, the string must be enclosed by double quotation marks ("). A backslash (\) must precede any double quotation marks contained within the string.

If the command line contains multiple `/bintext` options, the last (rightmost) one is used.

IDE Equivalent

Visual Studio: **Code Generation > Object Text String**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS*: None

Windows: `/V`*string*

c

Prevents linking.

Syntax

Linux OS:

`-c`

macOS:

`-c`

Windows OS:

`/c`

Arguments

None

Default

OFF Linking is performed.

Description

This option prevents linking. Compilation stops after the object file is generated.

The compiler generates an object file for each Fortran source file.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/compile-only`, `/nolink`

debug (Linux* and macOS*)

Enables or disables generation of debugging information.

Syntax

Linux OS:

`-debug [`*keyword*`]`

macOS:

`-debug [`*keyword*`]`

Windows OS:

`None`

Arguments

keyword

Is the type of debugging information to be generated. Possible values are:

`none`	Disables generation of debugging information.
`full` or `all`	Generates complete debugging information.
`minimal`	Generates line number information for debugging.
`[no]emit_column`	Determines whether the compiler generates column number information for debugging.
`[no]inline-debug-info`	Determines whether the compiler generates enhanced debug information for inlined code.
`[no]pubnames`	Determines whether the compiler generates a DWARF debug_pubnames section.
`[no]semantic-stepping`	Determines whether the compiler generates enhanced debug information useful for breakpoints and stepping.
`[no]variable-locations`	Determines whether the compiler generates enhanced debug information useful in finding scalar local variables.
`extended`	Generates complete debugging information and also sets keyword values `semantic-stepping` and `variable-locations`.
`[no]parallel` (Linux only)	Determines whether the compiler generates parallel debug code instrumentations useful for thread data sharing and reentrant call detection.

For information on the non-default settings for these keywords, see the Description section.

Default

varies

Normally, the default is `-debug none` and no debugging information is generated. However, on Linux*, the `-debug inline-debug-info` option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`).

Description

This option enables or disables generation of debugging information.

By default, enabling debugging, will disable optimization. To enable both debugging and optimization use the `-debug` option together with one of the optimization level options (`-O3`, `-O2` or `-O3`).

Keywords `semantic-stepping`, `inline-debug-info`, `variable-locations`, and `extended` can be used in combination with each other. If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	Description
`-debug none`	Disables generation of debugging information.
`-debug full` or `-debug all`	Generates complete debugging information. It is the same as specifying `-debug` with no keyword.
`-debug minimal`	Generates line number information for debugging.
`-debug emit_column`	Generates column number information for debugging.
`-debug inline-debug-info`	Generates enhanced debug information for inlined code. On inlined functions, symbols are (by default) associated with the caller. This option causes symbols for inlined functions to be associated with the source of the called function.
`-debug pubnames`	The compiler generates a DWARF debug_pubnames section. This provides a means to list the names of global objects and functions in a compilation unit.
`-debug semantic-stepping`	Generates enhanced debug information useful for breakpoints and stepping. It tells the debugger to stop only at machine instructions that achieve the final effect of a source statement. For example, in the case of an assignment statement, this might be a store instruction that assigns a value to a program variable; for a function call, it might be the machine instruction that executes the call. Other instructions generated for those source statements are not displayed during stepping. This option has no impact unless optimizations have also been enabled.
`-debug variable-locations`	Generates enhanced debug information useful in finding scalar local variables. It uses a feature of the Dwarf object module known as "location lists". This feature allows the run-time locations of local scalar variables to be specified more accurately; that is, whether, at a given position in the code, a variable value is found in memory or a machine register.
`-debug extended`	Sets keyword values `semantic-stepping` and `variable-locations`. It also tells the compiler to include column numbers in the line information. Generates complete debugging information and also sets keyword values `semantic-stepping` and `variable-locations`. This is a more powerful setting than `-debug full` or `-debug all`.
`-debug parallel`	Generates parallel debug code instrumentations needed for the thread data sharing and reentrant call detection. For this setting to be effective, option `-qopenmp` must be set.

On Linux* systems, debuggers read debug information from executable images. As a result, information is written to object files and then added to the executable by the linker.

On macOS* systems, debuggers read debug information from object files. As a result, the executables don't contain any debug information. Therefore, if you want to be able to debug on these systems, you must retain the object files.

IDE Equivalent

None

Alternate Options

For `-debug full`, `-debug all`, or `-debug`	Linux and macOS*: `-g` Windows: `/debug:full`, `/debug:all`, or `/debug`
For `-debug variable-locations`	Linux and macOS*: `-fvar-tracking` Windows: None
For `-debug semantic-stepping`	Linux and macOS*: `-fvar-tracking-assignments` Windows: None

See Also

`debug` (Windows*) compiler option
`qopenmp, Qopenmp` compiler option

debug (Windows*)

Enables or disables generation of debugging information.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/debug[:keyword]`

`/nodebug`

Arguments

keyword

Is the type of debugging information to be generated. Possible values are:

`none`	Disables generation of debugging information.
`full` or `all`	Generates complete debugging information.
`minimal`	Generates line number information for debugging.
`[no]inline-debug-info`	Determines whether the compiler generates enhanced debug information for inlined code.

For information on the non-default settings for these keywords, see the Description section.

Default

`/debug:none`	This is the default on the command line and for a release configuration in the IDE.
`/debug:all`	This is the default for a debug configuration in the IDE.

Description

This option enables or disables generation of debugging information. It is passed to the linker.

By default, enabling debugging, will disable optimization. To enable both debugging and optimization use the `/debug` option together with one of the optimization level options (`/O3`, `/O2` or `/O3`).

If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	Description
`/debug:none`	Disables generation of debugging information. It is the same as specifying /nodebug.
`/debug:full` or `/debug:all`	Generates complete debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking. It is the same as specifying `/debug` with no keyword. If you specify `/debug:full` for an application that makes calls to C library routines and you need to debug calls into the C library, you should also specify `/dbglibs` to request that the appropriate C debug library be linked against.
`/debug:minimal`	Generates line number information for debugging.
`/debug:partial`	Generates global symbol table information needed for linking, but not local symbol table information needed for debugging. This option is deprecated and is not available in the IDE.
`/debug:inline-debug-info`	Generates enhanced debug information for inlined code. On inlined functions, symbols are (by default) associated with the caller. This option causes symbols for inlined functions to be associated with the source of the called function.

IDE Equivalent

Visual Studio: **General > Debug Information Format** (`/debug:minimal`, `/debug:full`)

Eclipse: None

Xcode: None

Alternate Options

For `/debug:all` or `/debug`	Linux and macOS*: None Windows: `/Zi`

See Also

`dbglibs` compiler option

`debug` (Linux* and macOS*) compiler option

debug-parameters

Tells the compiler to generate debug information for PARAMETERs used in a program.

Syntax

Linux OS and macOS:

`-debug-parameters [keyword]`

`-nodebug-parameters`

Windows OS:

`/debug-parameters[:keyword]`

`/nodebug-parameters`

Arguments

keyword Are the PARAMETERs to generate debug information for. Possible values are:

`none`	Generates no debug information for any PARAMETERs used in the program. This is the same as specifying `nodebug-parameters`.
`used`	Generates debug information for only PARAMETERs that have actually been referenced in the program. This is the default if you do not specify a *keyword*.
`all`	Generates debug information for all PARAMETERs defined in the program.

Default

`nodebug-parameters` The compiler generates no debug information for any PARAMETERs used in the program. This is the same as specifying *keyword* `none`.

Description

This option tells the compiler to generate debug information for PARAMETERs used in a program.

Note that if a .mod file contains PARAMETERs, debug information is only generated for the PARAMETERs that have actually been referenced in the program, even if you specify *keyword* `all`.

IDE Equivalent

Visual Studio: **Debugging > Information for PARAMETER Constants**

Eclipse: None

Xcode: **Debug > Information for PARAMETER Constants**

Alternate Options

None

exe

Specifies the name for a built program or dynamic-link library.

Syntax

Linux OS and macOS:

None

Windows OS:

/exe:{*filename* | *dir*}

Arguments

filename Is the name for the built program or dynamic-link library.

dir Is the directory where the built program or dynamic-link library should be placed. It can include *filename*.

Default

OFF The name of the file is the name of the first source file on the command line with file extension .exe, so file.f becomes file.exe.

Description

This option specifies the name for a built program (.EXE) or a dynamic-link library (.DLL).

You can use this option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the option to give the resulting file a name other than that of the first input file (source or object) on the command line.

You can use this option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the option to give the resulting file a name other than that of the first input file (source or object) on the command line.

IDE Equivalent

None

Alternate Options

Linux and macOS*: -o

Windows: /Fe

Example

The following example creates a dynamic-link library file named file.dll (note that you can use /LD in place of /dll):

```
ifort /dll /exe:file.dll a.f
```

In the following example (which uses the alternate option /Fe), the command produces an executable file named outfile.exe as a result of compiling and linking three files: one object file and two Fortran source files.

```
prompt>ifort /Feoutfile.exe file1.obj file2.for file3.for
```

By default, this command produces an executable file named file1.exe.

See Also

- compiler option

Fa

Specifies that an assembly listing file should be generated.

Syntax

Linux OS:

`-Fa[filename|dir]`

macOS:

`-Fa[filename|dir]`

Windows OS:

`/Fa[filename|dir]`

Arguments

filename	Is the name of the assembly listing file.
dir	Is the directory where the file should be placed. It can include *filename*.

Default

OFF	No assembly listing file is produced.

Description

This option specifies that an assembly listing file should be generated (optionally named *filename*).

If *filename* is not specified, the file name will be the name of the source file with an extension of `.asm`; the file is placed in the current directory.

IDE Equivalent

Visual Studio: **Output > ASM Listing Name**

Eclipse: None

Xcode: **Output Files > Filename for Generated Assembler Listing**

Output Files > Generate Assembler Listing

Alternate Options

Linux and macOS*: `-S`

Windows: `/S`, `/asmfile` (this is a deprecated option)

FA

Specifies the contents of an assembly listing file.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/FA[specifier]`

Arguments

specifier	Denotes the contents of the assembly listing file. Possible values are *c*, *s*, or *cs*.

Default

OFF	No source or machine code annotations appear in the assembly listing file, if one is produced.

Description

These options specify what information, in addition to the assembly code, should be generated in the assembly listing file.

To use this option, you must also specify option `/Fa`, which causes an assembly listing to be generated.

Option	Description
`/FA`	Produces an assembly listing without source or machine code annotations.
`/FAc`	Produces an assembly listing with machine code annotations. The assembly listing file shows the hex machine instructions at the beginning of each line of assembly code. The file cannot be assembled; the file name is the name of the source file with an extension of .cod.
`/FAs`	Produces an assembly listing with source code annotations. The assembly listing file shows the source code as interspersed comments. Note that if you use alternate option `-fsource-asm`, you must also specify the `-S` option.
`/FAcs`	Produces an assembly listing with source and machine code annotations. The assembly listing file shows the source code as interspersed comments and shows the hex machine instructions at the beginning of each line of assembly code. This file cannot be assembled.

IDE Equivalent

Visual Studio: **Output Files > Assembler Output**

Eclipse: None

Xcode: None

Alternate Options

`/FAc`	Linux and macOS*: `-fcode-asm` Windows: None
`/FAs`	Linux and macOS*: `-fsource-asm` Windows: None

fcode-asm

Produces an assembly listing with machine code annotations.

Syntax

Linux OS and macOS:

`-fcode-asm`

Windows OS:

`None`

Arguments

None

Default

OFF No machine code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with machine code annotations.

The assembly listing file shows the hex machine instructions at the beginning of each line of assembly code. The file cannot be assembled; the file name is the name of the source file with an extension of `.cod`.

To use this option, you must also specify option `-S`, which causes an assembly listing to be generated.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/FAc`

See Also

S compiler option

Fd

Lets you specify a name for a program database (PDB) file created by the compiler.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Fd[:filename]`

Arguments

filename	Is the name for the PDB file. It can include a path. If you do not specify a file extension, the extension .pdb is used.

Default

OFF	No PDB file is created unless you specify option `/Zi`. If you specify option `/Zi` and `/Fd`, the default filename is vc*x*0.pdb, where *x* represents the version of Visual C++, for example vc100.pdb.

Description

This option lets you specify a name for a program database (PDB) file that is created by the compiler.

A program database (PDB) file holds debugging and project state information that allows incremental linking of a Debug configuration of your program. A PDB file is created when you build with option `/Zi`. Option `/Fd` has no effect unless you specify option `/Zi`.

IDE Equivalent

Visual Studio: **Output Files > Program Database File Name**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

Zi, Z7 compiler option
pdbfile compiler option

feliminate-unused-debug-types, Qeliminate-unused-debug-types

Controls the debug information emitted for types declared in a compilation unit.

Syntax

Linux OS and macOS:

`-feliminate-unused-debug-types`

`-fno-eliminate-unused-debug-types`

Windows OS:

`/Qeliminate-unused-debug-types`

`/Qeliminate-unused-debug-types-`

Arguments

None

Default

`-feliminate-unused-debug-types`	The compiler emits debug information only for types that are actually used by a variable/parameter/etc..

or
`/Qeliminate-unused-debug-types`

Description

This option controls the debug information emitted for types declared in a compilation unit.

If you specify `-fno-eliminate-unused-debug-types` (Linux and macOS*) or `/Qeliminate-unused-debug-types-`, it will cause the compiler to emit debug information for all types present in the sources. This option may cause a large increase in the size of the debug information.

IDE Equivalent

None

Alternate Options

None

fmerge-constants

Determines whether the compiler and linker attempt to merge identical constants (string constants and floating-point constants) across compilation units.

Syntax

Linux OS:

`-fmerge-constants`

`-fno-merge-constants`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-fmerge-constants`	The compiler and linker attempt to merge identical constants across compilation units if the compiler and linker supports it.

Description

This option determines whether the compiler and linker attempt to merge identical constants (string constants and floating-point constants) across compilation units.

If you do not want the compiler and linker to attempt to merge identical constants across compilation units. specify `-fno-merge-constants`.

IDE Equivalent

None

Alternate Options

None

fmerge-debug-strings

Causes the compiler to pool strings used in debugging information.

Syntax

Linux OS:

`-fmerge-debug-strings`

`-fno-merge-debug-strings`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-fmerge-debug-strings` The compiler will pool strings used in debugging information.

Description

This option causes the compiler to pool strings used in debugging information. The linker will automatically retain this pooling.

This option can reduce the size of debug information, but it may produce slightly slower compile and link times.

This option is only turned on by default if you are using gcc 4.3 or later, where this setting is also the default, since the generated debug tables require binutils version 2.17 or later to work reliably.

If you do not want the compiler to pool strings used in debugging information, specify option `-fno-merge-debug-strings`.

IDE Equivalent

None

Alternate Options

None

fsource-asm

Produces an assembly listing with source code annotations.

Syntax

Linux OS and macOS:

`-fsource-asm`

Windows OS:

None

Arguments

None

Default

OFF No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with source code annotations. The assembly listing file shows the source code as interspersed comments.

To use this option, you must also specify option `-S`, which causes an assembly listing to be generated.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/FAs`

See Also

`S` compiler option

ftrapuv, Qtrapuv

Initializes stack local variables to an unusual value to aid error detection.

Syntax

Linux OS:

`-ftrapuv`

macOS:

`-ftrapuv`

Windows OS:

`/Qtrapuv`

Arguments

None

Default

OFF The compiler does not initialize local variables.

Description

This option initializes stack local variables to an unusual value to aid error detection. Normally, these local variables should be initialized in the application. It also unmasks the floating-point invalid exception.

The option sets any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors.

This option sets option `-g` (Linux* and macOS*) and `/Zi` or `/Z7` (Windows*), which changes the default optimization level from `O2` to `-O0` (Linux and macOS*) or `/Od` (Windows). You can override this effect by explicitly specifying an `O` option setting.

This option sets option `[Q]init snan`.

If option `O2` and option `-ftrapuv` (Linux and macOS*) or `/Qtrapuv` (Windows) are used together, you should specify option `-fp-speculation safe` (Linux and macOS*) or `/Qfp-speculation:safe` (Windows) to prevent exceptions resulting from speculated floating-point operations from being trapped.

For more details on using options `-ftrapuv` and `/Qtrapuv` with compiler option `O`, see the article in Intel® Developer Zone titled *Don't optimize when using -ftrapuv for uninitialized variable detection*, which is located in https://software.intel.com/en-us/articles/dont-optimize-when-using-ftrapuv-for-uninitialized-variable-detection/ .

Another way to detect uninitialized local scalar variables is by specifying keyword `uninit` for option `check`.

IDE Equivalent

Visual Studio: **Data > Initialize stack variables to an unusual value**

Eclipse: None

Xcode: **Run-Time > Initialize Stack Variables to an Unusual Value**

Alternate Options

None

See Also

g compiler option
Zi, Z7 compiler option
O compiler option
check compiler option (see setting uninit)
init, Qinit compiler option (see setting snan)
Locating Run-Time Errors

fverbose-asm

Produces an assembly listing with compiler comments, including options and version information.

Syntax

Linux OS:

```
-fverbose-asm
```

```
-fno-verbose-asm
```

macOS:

```
-fverbose-asm
```

```
-fno-verbose-asm
```

Windows OS:

```
None
```

Arguments

None

Default

`-fno-verbose-asm`	No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with compiler comments, including options and version information.

To use this option, you must also specify `-S`, which sets `-fverbose-asm`.

If you do not want this default when you specify `-S`, specify `-fno-verbose-asm`.

IDE Equivalent

None

Alternate Options

None

See Also

`S` compiler option

g

Tells the compiler to generate a level of debugging information in the object file.

Syntax

Linux OS:

`-g[n]`

macOS:

`-g[n]`

Windows OS:

`See option Zi, Z7.`

Arguments

n	Is the level of debugging information to be generated. Possible values are:
`0`	Disables generation of symbolic debug information.
`1`	Produces minimal debug information for performing stack traces.
`2`	Produces complete debug information. This is the same as specifying `-g` with no *n*.

3	Produces extra information that may be useful for some tools.

Default

-g or -g2	The compiler produces complete debug information.

Description

Option `-g` tells the compiler to generate symbolic debugging information in the object file, which increases the size of the object file.

The compiler does not support the generation of debugging information in assemblable files. If you specify this option, the resulting object file will contain debugging information, but the assemblable file will not.

This option turns off option `-O2` and makes option `-O0` the default unless option `-O2` (or higher) is explicitly specified in the same command line.

Specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option. On Linux*, the -debug inline-debug-info option will be enabled by default if you compile with optimizations (option `-O2` or higher) and debugging is enabled (option `-g`).

NOTE
When option `-g` is specified, debugging information is generated in the DWARF Version 3 format. Older versions of some analysis tools may require applications to be built with the `-gdwarf-2` option to ensure correct operation.

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **General > Generate Debug Information**

Alternate Options

Linux: None

Windows: /Zi, /Z7

See Also

gdwarf compiler option
Zi, Z7 compiler option
debug (Linux* and macOS*) compiler option

gdwarf

Lets you specify a DWARF Version format when generating debug information.

Syntax

Linux OS:

`-gdwarf-`*n*

macOS:

`-gdwarf-`*n*

Windows OS:

None

Arguments

n	Is a value denoting the DWARF Version format to use. Possible values are:
2	Generates debug information using the DWARF Version 2 format.
3	Generates debug information using the DWARF Version 3 format.
4	Generates debug information using the DWARF Version 4 format. This setting is only available on Linux*.

Default

OFF	No debug information is generated. However, if compiler option `-g` is specified, debugging information is generated in the DWARF Version 3 format.

Description

This option lets you specify a DWARF Version format when generating debug information.

Note that older versions of some analysis tools may require applications to be built with the -gdwarf-2 option to ensure correct operation.

IDE Equivalent

None

Alternate Options

None

See Also

`g` compiler option

grecord-gcc-switches

Causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.

Syntax

Linux OS:

```
-grecord-gcc-switches
```

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	The command line options that were used to invoke the compiler are not appended to the DW_AT_producer attribute in DWARF debugging information.

Description

This option causes the command line options that were used to invoke the compiler to be appended to the DW_AT_producer attribute in DWARF debugging information.

The options are concatenated with whitespace separating them from each other and from the compiler version.

IDE Equivalent

None

Alternate Options

None

gsplit-dwarf

Creates a separate object file containing DWARF debug information.

Syntax

Linux OS:

```
-gsplit-dwarf
```

macOS:

```
None
```

Windows OS:

```
None
```

Arguments

None

Default

OFF	No separate object file containing DWARF debug information is created.

Description

This option creates a separate object file containing DWARF debug information. It causes debug information to be split between the generated object (.o) file and the new DWARF object (.dwo) file.

The DWARF object file is not used by the linker, so this reduces the amount of debug information the linker must process and it results in a smaller executable file.

For this option to perform correctly, you must use binutils-2.24 or later. To debug the resulting executable, you must use gdb-7.6.1 or later.

NOTE
If you use the split executable with a tool that does not support the split DWARF format, it will behave as though the DWARF debug information is absent.

IDE Equivalent

None

Alternate Options

None

list

Tells the compiler to create a listing of the source file.

Syntax

Linux OS and macOS:

`-list[=`*filename*`]`

`-no-list`

Windows OS:

`/list[:`*filename*`]`

`/list-`

Arguments

filename Is the name of the file for output. It can include a path.

Default

`-no-list` No listing is created for the source file.
or `/list-`

Description

This option tells the compiler to create a listing of the source file. The listing contains the following:

- The contents of files included with INCLUDE statements
- A symbol list with a line number cross-reference for each routine
- A list of compiler options used for the current compilation

The contents of the listing can be controlled by specifying option `show`.

The line length of the listing can be specified by using option `list-line-len`.

The page length of the listing can be specified by using option `list-page-len`.

If you do not specify *filename*, the output is written to a file in the same directory as the source. The file name is the name of the source file with an extension of .lst.

IDE Equivalent

Visual Studio: **Output Files > Source Listing** (/list)

Output Files > Source Listing File (/list:[filename])

Eclipse: None

Xcode: **Output Files > Source Listing File**

Output Files > Source Listing File with Cross Reference

Alternate Options

None

See Also

`show` compiler option
`list-line-len` compiler option
`list-page-len` compiler option

list-line-len

Specifies the line length for the listing generated when option list is specified.

Syntax

Linux OS and macOS:

`-list-line-len=`*n*

Windows OS:

`/list-line-len:`*n*

Arguments

n	Is a positive integer indicating the number of columns to show in the listing.

Default

80	When a listing is generated, the default line length is 80 columns.

Description

This option specifies the line length for the listing generated when option `list` is specified.

If you specify option `list-line-len` and do not specify option `list`, the option is ignored.

IDE Equivalent

None

Alternate Options

None

See Also

`list` compiler option
`list-page-len` compiler option

list-page-len

Specifies the page length for the listing generated when option list is specified.

Syntax

Linux OS and macOS:

`-list-page-len=`*n*

Windows OS:

`/list-page-len:`*n*

Arguments

n	Is a positive integer indicating the number of lines on a page to show in the listing.

Default

60	When a listing is generated, the default page length is 60 lines.

Description

This option specifies the page length for the listing generated when option `list` is specified.

If you specify option `list-page-len` and do not specify option `list`, the option is ignored.

IDE Equivalent

None

Alternate Options

None

See Also

`list` compiler option

`list-line-len` compiler option

map-opts, Qmap-opts

Maps one or more compiler options to their equivalent on a different operating system.

Syntax

Linux OS:

`-map-opts`

macOS:

`None`

Windows OS:

`/Qmap-opts`

Arguments

None

Default

OFF	No platform mappings are performed.

Description

This option maps one or more compiler options to their equivalent on a different operating system. The result is output to `stdout`.

On Windows systems, the options you provide are presumed to be Windows options, so the options that are output to `stdout` will be Linux equivalents.

On Linux systems, the options you provide are presumed to be Linux options, so the options that are output to `stdout` will be Windows equivalents.

The tool can be invoked from the compiler command line or it can be used directly.

No compilation is performed when the option mapping tool is used.

This option is useful if you have both compilers and want to convert scripts or makefiles.

NOTE
Compiler options are mapped to their equivalent on the architecture you are using. For example, if you are using a processor with IA-32 architecture, you will only see equivalent options that are available on processors with IA-32 architecture.

IDE Equivalent

None

Alternate Options

None

Example

The following command line invokes the option mapping tool, which maps the Linux options to Windows-based options, and then outputs the results to `stdout`:

```
ifort -map-opts -xP -O2
```

The following command line invokes the option mapping tool, which maps the Windows options to Linux-based options, and then outputs the results to `stdout`:

```
ifort /Qmap-opts /QxP /O2
```

See Also

Compiler Option Mapping Tool

o

Specifies the name for an output file.

Syntax

Linux OS:

`-o filename`

macOS:

`-o filename`

Windows OS:

`None`

Arguments

filename Is the name for the output file. The space before *filename* is optional.

Default

OFF The compiler uses the default file name for an output file.

Description

This option specifies the name for an output file as follows:

- If `-c` is specified, it specifies the name of the generated object file.
- If `-S` is specified, it specifies the name of the generated assembly listing file.
- If -preprocess-only or `-P` is specified, it specifies the name of the generated preprocessor file.

Otherwise, it specifies the name of the executable file.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/exe`

See Also

`object` compiler option
`exe` compiler option

object

Specifies the name for an object file.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/object:`*`filename`*

Arguments

`filename` Is the name for the object file. It can be a file name or a directory name. A directory name must be followed by a backslash (\). If a special character appears within the file name or directory name, the file name or directory name must appear within quotes. To be safe, you should consider any non-ASCII numeric character to be a special character.

Default

OFF An object file has the same name as the name of the first source file and a file extension of .obj.

Description

This option specifies the name for an object file.

If you specify this option and you omit `/c` or `/compile-only`, the `/object` option gives the object file its name.

On Linux and macOS* systems, this option is equivalent to specifying option `-o`*filename*`-c`.

IDE Equivalent

Visual Studio: **Output Files > Object File Name**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS*: None

Windows: `/Fo`

Example

The following command shows how to specify a directory:

```
ifort /object:directorya\ end.f
```

If you do not add the backslash following a directory name, an executable is created. For example, the following command causes the compiler to create directorya.exe:

```
ifort /object:directorya end.f
```

The following commands show how to specify a subdirectory that contains a special character:

```
ifort /object:"blank subdirectory"\ end.f
```

```
ifort /object:"c:\my_directory"\ end.f
```

See Also

- compiler option

pdbfile

Lets you specify the name for a program database (PDB) file created by the linker.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

`/pdbfile[:`*filename*`]`

Arguments

filename	Is the name for the PDB file. It can include a path. If you do not specify a file extension, the extension .pdb is used.

Default

OFF No PDB file is created unless you specify option `/Zi`. If you specify option `/Zi` the default filename is *executablename*.pdb.

Description

This option lets you specify the name for a program database (PDB) file created by the linker. This option does not affect where the compiler outputs debug information.

To use this option, you must also specify option `/debug:full` or `/Zi`.

If *filename* is not specified, the default file name used is the name of your file with an extension of .pdb.

IDE Equivalent

None

Alternate Options

None

See Also

Zi, Z7 compiler option
`debug` compiler option
`Fd` compiler option

print-multi-lib

Prints information about where system libraries should be found.

Syntax

Linux OS:

```
-print-multi-lib
```

macOS:

```
-print-multi-lib
```

Windows OS:

```
None
```

Arguments

None

Default

OFF No information is printed unless the option is specified.

Description

This option prints information about where system libraries should be found, but no compilation occurs. On Linux* systems, it is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

Quse-msasm-symbols

Tells the compiler to use a dollar sign ("$") when producing symbol names.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Quse-msasm-symbols`

Arguments

None

Default

OFF	The compiler uses a period (".") when producing symbol names

Description

This option tells the compiler to use a dollar sign ("$") when producing symbol names.

Use this option if you require symbols in your .asm files to contain characters that are accepted by the MS assembler.

IDE Equivalent

None

Alternate Options

None

S

Causes the compiler to compile to an assembly file only and not link.

Syntax

Linux OS:

`-S`

macOS:

`-S`

Windows OS:

`/S`

Arguments

None

Default

OFF Normal compilation and linking occur.

Description

This option causes the compiler to compile to an assembly file only and not link.

On Linux* and macOS* systems, the assembly file name has a .s suffix. On Windows* systems, the assembly file name has an .asm suffix.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/Fa`

See Also

Fa compiler option

show

Controls the contents of the listing generated when option list is specified.

Syntax

Linux OS and macOS:

`-show` *keyword*`[,` *keyword*`...]`

Windows OS:

`/show:`*keyword*`[,` *keyword*`...]`

Arguments

keyword Specifies the contents for the listing. Possible values are:

`[no]include` Controls whether contents of files added with INCLUDE statements are included when a listing is generated.

`[no]map` Controls whether a symbol listing with a line number cross-reference for each routine is included when a listing is generated.

`[no]options` Controls whether a list of compiler options used for the compilation is included when a listing is generated.

Default

`include`, `map`, and `options` When a listing is generated, it contains the contents of INCLUDEd files, a symbol list with a line number cross reference, and a list of compiler options used.

Description

This option controls the contents of the listing generated when option `list` is specified.

If you specify option `show` and do not specify option `list`, the option is ignored.

IDE Equivalent

None

Alternate Options

None

See Also

`list` compiler option

use-asm, Quse-asm

Tells the compiler to produce objects through the assembler. This is a deprecated option. There is no replacement option.

Syntax

Linux OS and macOS:

`-use-asm`

`-no-use-asm`

Windows OS:

`/Quse-asm`

`/Quse-asm-`

Arguments

None

Default

OFF	The compiler produces objects directly.

Description

This option tells the compiler to produce objects through the assembler.

IDE Equivalent

None

Alternate Options

None

Zi, Z7

Tells the compiler to generate full debugging information in either an object (.obj) file or a project database (PDB) file.

Syntax

Linux OS:

`See option g.`

macOS:

`See option g.`

Windows OS:

`/Zi`

`/Z7`

Arguments

None

Default

OFF No debugging information is produced.

Description

Option `/Z7` tells the compiler to generate symbolic debugging information in the object (.obj) file for use with the debugger. No .pdb file is produced by the compiler.

The `/Zi` option tells the compiler to generate symbolic debugging information in a program database (PDB) file for use with the debugger. Type information is placed in the .pdb file, and not in the .obj file, resulting in smaller object files in comparison to option `/Z7`.

When option `/Zi` is specified, two PDB files are created:

- The compiler creates the program database project.pdb. If you compile a file without a project, the compiler creates a database named vc*x*0.pdb, where *x* represents the major version of Visual C++, for example vc140.pdb.

  This file stores all debugging information for the individual object files and resides in the same directory as the project makefile. If you want to change this name, use option `/Fd`.
- The linker creates the program database executablename.pdb.

  This file stores all debug information for the .exe file and resides in the debug subdirectory. It contains full debug information, including function prototypes, not just the type information found in vc*x*0.pdb.

Both PDB files allow incremental updates. The linker also embeds the path to the .pdb file in the .exe or .dll file that it creates.

The compiler does not support the generation of debugging information in assemblable files. If you specify these options, the resulting object file will contain debugging information, but the assemblable file will not.

These options turn off option `/O2` and make option `/Od` the default unless option `/O2` (or higher) is explicitly specified in the same command line.

For more information about the `/Z7` and `/Zi` options, see the Microsoft documentation.

IDE Equivalent

Visual Studio: **General > Generate Debug Information**

Eclipse: None

Xcode: None

Alternate Options

Linux: -g

Windows: None

See Also

Fd compiler option
g compiler option
debug (Windows*) compiler option

Zo

Enables or disables generation of enhanced debugging information for optimized code.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Zo`

`/Zo-`

Arguments

None

Default

OFF The compiler does not generate enhanced debugging information for optimized code.

Description

This option enables or disables the generation of additional debugging information for local variables and inlined routines when code optimizations are enabled. It should be used with option `/Zi` or `/Z7` to allow improved debugging of optimized code.

Option `/Zo` enables generation of this enhanced debugging information. Option `/Zo-` disables this functionality.

For more information on code optimization, see option `/O`.

IDE Equivalent

None

Alternate Options

None

See Also

Zi, Z7 compiler option
debug (Windows*) compiler option
O compiler option

Preprocessor Options

B

Specifies a directory that can be used to find include files, libraries, and executables.

Syntax

Linux OS:

`-B`*dir*

macOS:

`-B`*dir*

Windows OS:

`None`

Arguments

dir	Is the directory to be used. If necessary, the compiler adds a directory separator character at the end of *dir*.

Default

OFF	The compiler looks for files in the directories specified in your PATH environment variable.

Description

This option specifies a directory that can be used to find include files, libraries, and executables.

The compiler uses *dir* as a prefix.

For include files, the *dir* is converted to `-I`/*dir*/include. This command is added to the front of the includes passed to the preprocessor.

For libraries, the *dir* is converted to `-L`/*dir*. This command is added to the front of the standard `-L` inclusions before system libraries are added.

For executables, if *dir* contains the name of a tool, such as `ld` or `as`, the compiler will use it instead of those found in the default directories.

The compiler looks for include files in *dir* /include while library files are looked for in *dir*.

IDE Equivalent

None

Alternate Options

None

D

Defines a symbol name that can be associated with an optional value.

Syntax

Linux OS:

`-Dname[=value]`

macOS:

`-Dname[=value]`

Windows OS:

`/Dname[=value]`

Arguments

name	Is the name of the symbol.
value	Is an optional integer or an optional character string delimited by double quotes; for example, `D`*name=string*.

Default

`noD` Only default symbols or macros are defined.

Description

Defines a symbol name that can be associated with an optional value. This definition is used during preprocessing in both Intel® Fortran conditional compilation directives and the fpp preprocessor. The D*name=value* will be ignored if there are any non-alphabetic, non-numeric characters in *name*.

If a *value* is not specified, *name* is defined as "1".

If you want to specify more than one definition, you must use separate `D` options.

If you specify `noD`, all preprocessor definitions apply only to fpp and not to Intel® Fortran conditional compilation directives. To use this option, you must also specify the `fpp` option.

Caution
On Linux* and macOS* systems, if you are not specifying a *value*, do not use `D` for *name*, because it will conflict with the `-DD` option.

IDE Equivalent

Visual Studio: **General > Preprocessor Definitions**

Preprocessor> Preprocessor Definitions

Preprocessor > Preprocessor Definitions to FPP only

Eclipse: None

Xcode: **Preprocessor > Preprocessor Definitions**

Preprocessor > Preprocessor Definitions to FPP only

Alternate Options

D	Linux and macOS*: None Windows: `/define:`*name[=value]*
noD	Linux and macOS*: `-nodefine`

Windows: `/nodefine`

See Also

Using Predefined Preprocessor Symbols

d-lines, Qd-lines

Compiles debug statements.

Syntax

Linux OS and macOS:

`-d-lines`

`-nod-lines`

Windows OS:

`/d-lines`

`/nod-lines`

`/Qd-lines`

Arguments

None

Default

`nod-lines` Debug lines are treated as comment lines.

Description

This option compiles debug statements. It specifies that lines in fixed-format files that contain a D in column 1 (debug statements) should be treated as source code.

IDE Equivalent

Visual Studio: **Language > Compile Lines With D in Column 1** (/d-lines)

Eclipse: None

Xcode: **Language > Compile Lines With D in Column 1**

Alternate Options

Linux and macOS*: `-DD`

Windows: None

E

Causes the preprocessor to send output to stdout.

Syntax

Linux OS:

`-E`

macOS:

`-E`

Windows OS:

`/E`

Arguments

None

Default

OFF Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`. Compilation stops when the files have been preprocessed.

When you specify this option, the compiler's preprocessor expands your source module and writes the result to `stdout`. The preprocessed source contains #line directives, which the compiler uses to determine the source file and line number.

IDE Equivalent

None

None

Alternate Options

None

EP

Causes the preprocessor to send output to stdout, omitting #line directives.

Syntax

Linux OS:

`-EP`

macOS:

`-EP`

Windows OS:

`/EP`

Arguments

None

Default

OFF Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`, omitting #line directives.

If you also specify option `preprocess-only`, option `P`, or option `F`, the preprocessor will write the results (without #line directives) to a file instead of `stdout`.

IDE Equivalent

None

Alternate Options

None

fpp

Runs the Fortran preprocessor on source files before compilation.

Syntax

Linux OS and macOS:

`-fpp`

`-nofpp`

Windows OS:

`/fpp`

`/fpp[:"`*`fpp_option`*`"]`

`/nofpp`

Arguments

fpp_option	Is a Fortran preprocessor (fpp) option; it must start with a slash (/) and appear in quotes. This argument is only allowed on Windows* systems.

Default

`nofpp`	The Fortran preprocessor is not run on files before compilation.

Description

This option runs the Fortran preprocessor on source files before they are compiled.

If you want to pass fpp options to the Fortran preprocessor, you can use any of the following methods:

- Use option `Qoption,fpp,"option"`. This is the recommended method.
- On Windows* systems, use this option (`fpp`) and include the argument *fpp_option* (for example, `fpp:"/macro=no"`).
- On Linux* and macOS* systems, use option `-Wp,fpp_option` (for example, `-Wp,-macro=no`).

To see a list of all available fpp options, specify one of the following on the command line:

```
fpp -help    ! Linux and macOS* systems
```

```
fpp /help    ! Windows systems
```

IDE Equivalent

Visual Studio: **Preprocessor > Preprocess Source File**

Eclipse: None

Xcode: **Preprocessor > Preprocess Source File**

Alternate Options

Linux and macOS*: `-cpp` (this is a deprecated option)

Windows: None

Example

The following examples show the recommended method of passing fpp options to the Fortran preprocessor:

```
ifort /fpp /Qoption,fpp,"/macro=no_com"  file.f90        ! Disables macro expansion within
comments (Windows*)
```

```
ifort -fpp -Qoption,fpp,"-undef"  file.f90               ! Undefines all predefined macros
(Linux* and macOS*)
```

See Also

Qoption
compiler option

Using Fortran Preprocessor Options

Wp
compiler option

fpp-name

Lets you specify an alternate preprocessor to use with Fortran.

Syntax

Linux OS and macOS:

`-fpp-name=`*name*

Windows OS:

`/fpp-name:`*name*

Arguments

name	Is the name of the preprocessor executable. It can include a path. See the description below for more details.

Default

OFF	No preprocessor is run on files before compilation.

Description

This option lets you specify an alternate preprocessor to use with Fortran.

The compiler invokes the user-specified Fortran preprocessor by spawning a command with the following signature:

```
alt_fpp [ [-D<define>]..] [[-I<include directory>]..] inputfile
```

where `alt_fpp` is the name of the Fortran preprocessor you want to use. Output from the preprocessor goes to STDOUT and will be captured for any further processing.

You can use option `Qoption,fpp,…` to pass options, other than the definitions (`-D` xxx) or include directories (`-I` xxx), to the preprocessor.

You can use option `Qlocation, fpp,…` to specify a directory for supporting tools.

IDE Equivalent

None

Alternate Options

None

See Also

fpp Preprocessing
`fpp` compiler option
`Qoption` compiler option
`Qlocation` compiler option

gen-dep

Tells the compiler to generate build dependencies for the current compilation.

Syntax

Linux OS and macOS:

`-gen-dep[=filename]`

`-no-gen-dep`

Windows OS:

`/gen-dep[:filename]`

`/gen-dep-`

Arguments

filename Is the name of the file for output. It can include a path.

If you specify *filename*, it is similar to specifying option [Q]MF*filename*. If you do not specify *filename*, it is similar to specifying option [Q]MD or [Q]MMD.

Default

`-no-gen-dep` The compiler does not generate build dependencies for the compilation.

or
`/gen-dep-`

Description

This option tells the compiler to generate build dependencies for the current compilation. The build dependencies include a list of all files included with INCLUDE statements or .mod files accessed with USE statements.

If you do not specify *filename*, the dependencies are written to stdout.

You can use option `gen-depformat` to specify the form of the output for the build dependencies generated.

If you specify option `gen-dep` and you do not specify option `gen-depformat`, the output format is in a form acceptable to the make utility.

Note that if option `fpp` is used to process #include files, those files will also appear in the list of build dependencies.

If you want to generate build dependencies but you do not want to compile the sources, you must also specify option `syntax-only`.

IDE Equivalent

Visual Studio: **Output Files > Build Dependencies** (/gen-dep)

Output Files > Emit Build Dependencies to File (/gen-dep:filename)

Output Files > Build Dependencies File (/gen-dep:filename)

Eclipse: None

Xcode: **Output Files > Create a List of Build Dependencies** (-gen-dep)

Output Files > Build Dependencies File (/gen-dep:filename)

Alternate Options

`gen-dep` with a *filename*	Linux and macOS*: `-MF` Windows: `/QMF`
`gen-dep` with no *filename*	Linux and macOS*: `-MD` or `-MMD` Windows: `/QMD` or `/QMMD`

See Also

`gen-depformat` compiler option

`gen-depshow` compiler option

`syntax-only` compiler option

gen-depformat

Specifies the form for the output generated when option gen-dep is specified.

Syntax

Linux OS and macOS:

`-gen-depformat=`*`form`*

Windows OS:

`/gen-depformat:`*`form`*

Arguments

form	Is the output form for the list of build dependencies. Possible values are make or nmake.

Default

make	The output form for the list of build dependencies is in a form acceptable to the make utility.

Description

This option specifies the form for the output generated when option `gen-dep` is specified.

If you specify option `gen-depformat` and do not specify option `gen-dep`, the option is ignored.

IDE Equivalent

None

Alternate Options

None

See Also

`gen-dep` compiler option

gen-depshow

Determines whether certain features are excluded from dependency analysis. Currently, it only applies to intrinsic modules.

Syntax

Linux OS and macOS:

`-gen-depshow=`*`keyword`*

Windows OS:

`/gen-depshow:`*`keyword`*

Arguments

`keyword` Specifies inclusion or exclusion from dependency analysis. Possible values are:

`[no]intr_mod` Determines whether intrinsic modules are excluded from dependency analysis.

Default

`nointr_mod` Tells the compiler to exclude Fortran intrinsic modules in dependency analysis.

Description

This option determines whether certain features are excluded from dependency analysis. Currently, it only applies to intrinsic modules.

Option	Description
`gen-depshow intr_mod`	Tells the compiler to include Fortran intrinsic modules in dependency analysis.

If you do not specify option `gen-dep`, the compiler does not generate build dependencies for the compilation.

If you specify option `gen-depshow` and do not specify option `gen-dep`, the option is ignored.

IDE Equivalent

None

Alternate Options

`gen-depshow nointr_mod` Linux and macOS*: `-MMD`

Windows: `/QMMD`

See Also

`gen-dep` compiler option

I

Specifies an additional directory for the include path.

Syntax

Linux OS and macOS:

`-I`*dir*

Windows OS:

`/I`*dir*

Arguments

dir	Is the directory to add to the include path.

Default

OFF	The default include path is used.

Description

This option specifies an additional directory for the include path, which is searched for module files referenced in USE statements and include files referenced in INCLUDE statements. To specify multiple directories on the command line, repeat the option for each directory you want to add.

For all USE statements and for those INCLUDE statements whose file name does not begin with a device or directory name, the directories are searched in this order:

1. The directory containing the first source file.

   Note that if `assume nosource_include` is specified, this directory will not be searched.
2. The current working directory where the compilation is taking place (if different from the above directory).
3. Any directory or directories specified using the I option. If multiple directories are specified, they are searched in the order specified on the command line, from left to right.
4. On Linux* and macOS* systems, any directories indicated using environment variable CPATH. On Windows* systems, any directories indicated using environment variable INCLUDE.

This option affects fpp preprocessor behavior and the USE statement.

IDE Equivalent

Visual Studio: **General > Additional Include Directories** (/I)

Preprocessor > Additional Include Directories (/I)

Eclipse: None

Xcode: **Preprocessor > Additional Include Directories** (/I)

Alternate Options

Linux and macOS*: None

Windows: `/include`

See Also

`X` compiler option

`assume` compiler option

idirafter

Adds a directory to the second include file search path.

Syntax

Linux OS:

`-idirafter`*dir*

macOS:

`-idirafter`*dir*

Windows OS:

`None`

Arguments

dir	Is the name of the directory to add.

Default

OFF	Include file search paths include certain default directories.

Description

This option adds a directory to the second include file search path (after `-I`).

IDE Equivalent

None

Alternate Options

None

isystem

Specifies a directory to add to the start of the system include path.

Syntax

Linux OS:

`-isystem`*dir*

macOS:

`-isystem`*dir*

Windows OS:

`None`

Arguments

dir	Is the directory to add to the system include path.

Default

OFF The default system include path is used.

Description

This option specifies a directory to add to the system include path. The compiler searches the specified directory for include files after it searches all directories specified by the `-I` compiler option but before it searches the standard system directories.

On Linux* systems, this option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

module

Specifies the directory where module files should be placed when created and where they should be searched for.

Syntax

Linux OS and macOS:

`-module` *path*

Windows OS:

`/module:`*path*

Arguments

path Is the directory for module files.

Default

OFF The compiler places module files in the current directory.

Description

This option specifies the directory (path) where module (.mod) files should be placed when created and where they should be searched for (USE statement).

IDE Equivalent

Visual Studio: **Output > Module Path**

Eclipse: None

Xcode: **Output Files > Module Path**

Alternate Options

None

preprocess-only

Causes the Fortran preprocessor to send output to a file.

Syntax

Linux OS and macOS:

`-preprocess-only`

Windows OS:

`/preprocess-only`

Arguments

None

Default

OFF Preprocessed source files are output to the compiler.

Description

This option causes the Fortran preprocessor to send output to a file.

The source file is preprocessed by the Fortran preprocessor, and the result for each source file is output to a corresponding .i or .i90 file.

Note that the source file is not compiled.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-P`

Windows: `/P`

u (Windows*)

Undefines all previously defined preprocessor values.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/u`

Arguments

None

Default

OFF Defined preprocessor values are in effect until they are undefined.

Description

This option undefines all previously defined preprocessor values.

To undefine specific preprocessor values, use the `/U` option.

IDE Equivalent

Visual Studio: **Preprocessor > Undefine All Preprocessor Definitions**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`U` compiler option

U

Undefines any definition currently in effect for the specified symbol.

Syntax

Linux OS:

`-U`*name*

macOS:

`-U`*name*

Windows OS:

`/U`*name*

Arguments

name	Is the name of the symbol to be undefined.

Default

OFF	Symbol definitions are in effect until they are undefined.

Description

This option undefines any definition currently in effect for the specified symbol.

On Windows systems, use the `/u` option to undefine all previously defined preprocessor values.

IDE Equivalent

Visual Studio: **Preprocessor > Undefine Preprocessor Definitions**

Eclipse: None

Xcode: **Preprocessor > Undefine Preprocessor Definitions**

Alternate Options

Linux and macOS*: None

Windows: `/undefine:`*name*

See Also

`u` (Windows) compiler option

undef

Disables all predefined symbols.

Syntax

Linux OS:

`-undef`

macOS:

`-undef`

Windows OS:

`None`

Arguments

None

Default

OFF Defined symbols are in effect until they are undefined.

Description

This option disables all predefined symbols.

IDE Equivalent

None

Alternate Options

None

X

Removes standard directories from the include file search path.

Syntax

Linux OS:

`-X`

macOS:

`-X`

Windows OS:

`/X`

Arguments

None

Default

OFF Standard directories are in the include file search path.

Description

This option removes standard directories from the include file search path. It prevents the compiler from searching the default path specified by the CPATH environment variable.

On Linux* and macOS* systems, specifying `-X` (or `-noinclude`) prevents the compiler from searching in `/usr/include` for files specified in an INCLUDE statement.

You can use this option with the `I` option to prevent the compiler from searching the default path for include files and direct it to use an alternate path.

This option affects fpp preprocessor behavior and the USE statement.

IDE Equivalent

Visual Studio: **Preprocessor > Ignore Standard Include Path** (/X)

Eclipse: None

Xcode: **Preprocessor > Ignore Standard Include Path** (-X)

Alternate Options

Linux and macOS*: `-nostdinc`

Windows: /noinclude

See Also

`I` compiler option

Component Control Options

Qinstall

Specifies the root directory where the compiler installation was performed.

Syntax

Linux OS:

`-Qinstall`*`dir`*

macOS:

`-Qinstall`*`dir`*

Windows OS:

`None`

Arguments

dir Is the root directory where the installation was performed.

Default

OFF The default root directory for compiler installation is searched for the compiler.

Description

This option specifies the root directory where the compiler installation was performed. It is useful if you want to use a different compiler or if you did not use the compilervars shell script to set your environment variables.

IDE Equivalent

None

Alternate Options

None

Qlocation

Specifies the directory for supporting tools.

Syntax

Linux OS:

`-Qlocation,`*`string`*`,`*`dir`*

macOS:

`-Qlocation,`*`string`*`,`*`dir`*

Windows OS:

`/Qlocation,`*`string`*`,`*`dir`*

Arguments

string	Is the name of the tool.
dir	Is the directory (path) where the tool is located.

Default

OFF	The compiler looks for tools in a default area.

Description

This option specifies the directory for supporting tools.

string can be any of the following:

- f - Indicates the Intel® Fortran compiler.
- fpp (or cpp) - Indicates the Intel® Fortran preprocessor or a user-specified (alternate) Fortran preprocessor.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows* systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux* and macOS* systems, the following are also available:
  - as - Indicates the assembler.
  - gas - Indicates the GNU assembler. This setting is for Linux* only.
  - ld - Indicates the loader.
  - gld - Indicates the GNU loader. This setting is for Linux* only.

- lib - Indicates an additional library.
- crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

On Windows and macOS* systems, you can also specify a tool command name.

The following shows an example on macOS* systems:

```
-Qlocation,ld,/usr/bin               ! This tells the driver to use /usr/bin/ld for the loader
-Qlocation,ld,/usr/bin/gld           ! This tells the driver to use /usr/bin/gld as the loader
```

The following shows an example on Windows* systems:

```
/Qlocation,link,"c:\Program Files\tools\"            ! This tells the driver to use c:\Program
Files\tools\link.exe for the loader
/Qlocation,link,"c:\Program Files\tools\my_link.exe" ! This tells the driver to use c:\Program
Files\tools\my_link.exe as the loader
```

IDE Equivalent

None

Alternate Options

None

Example

The following command provides the path for the fpp tool:

```
ifort -Qlocation,fpp,/usr/preproc myprog.f
```

See Also

`Qoption` compiler option

Qoption

Passes options to a specified tool.

Syntax

Linux OS:

`-Qoption,`*`string`*`,`*`options`*

macOS:

`-Qoption,`*`string`*`,`*`options`*

Windows OS:

`/Qoption,`*`string`*`,`*`options`*

Arguments

string	Is the name of the tool.
options	Are one or more comma-separated, valid options for the designated tool. Note that certain tools may require that options appear within quotation marks (" ").

Default

OFF No options are passed to tools.

Description

This option passes options to a specified tool.

If an argument contains a space or tab character, you must enclose the entire argument in quotation marks (" "). You must separate multiple arguments with commas.

string can be any of the following:

- fpp (or cpp) - Indicates the Intel® Fortran preprocessor or a user-specified (alternate) Fortran preprocessor.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows* systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux* and macOS* systems, the following are also available:
  - as - Indicates the assembler.
  - gas - Indicates the GNU assembler.
  - ld - Indicates the loader.
  - gld - Indicates the GNU loader.
  - lib - Indicates an additional library.
  - crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

IDE Equivalent

None

Alternate Options

None

Example

Examples for Linux* and macOS* systems:

The following example directs the linker to link with an alternative library:

```
ifort -Qoption,link,-L.,-Lmylib prog1.f
```

The following example passes a compiler option to the assembler to generate a listing file:

```
ifort -Qoption,as,"-as=myprogram.lst" -use-asm myprogram.f90
```

The following option passes an `fpp` option to the Fortran preprocessor:

```
ifort -Qoption,fpp,"-fpp=macro=no
```

Examples for Windows* systems:

The following example directs the linker to create a memory map when the compiler produces the executable file from the source being compiled:

```
ifort /Qoption,link,/map:prog1.map prog1.f
```

The following example passes a compiler option to the assembler:

```
ifort /Quse-asm /Qoption,masm,"/WX" myprogram.f90
```

The following option passes an `fpp` option to the Fortran preprocessor:

```
ifort /Qoption,fpp,"/fpp:macro=no"
```

See Also

`Qlocation` compiler option

Language Options

allow

Determines whether the compiler allows certain behaviors.

Syntax

Linux OS and macOS:

`-allow` *keyword*

Windows OS:

`/allow:`*keyword*

Arguments

keyword Specifies the behaviors to allow or disallow. Possible values are:

`[no]fpp_comments` Determines how the fpp preprocessor treats Fortran end-of-line comments in preprocessor directive lines.

Default

`fpp_comments` The compiler recognizes Fortran-style end-of-line comments in preprocessor lines.

Description

This option determines whether the compiler allows certain behaviors.

Option	Description
`allow nofpp_comments`	Tells the compiler to disallow Fortran-style end-of-line comments on preprocessor lines. Comment indicators have no special meaning.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following:

```
#define MAX_ELEMENTS 100  ! Maximum number of elements
```

By default, the compiler recognizes Fortran-style end-of-line comments on preprocessor lines. Therefore, the line above defines MAX_ELEMENTS to be "100" and the rest of the line is ignored. If `allow nofpp_comments` is specified, Fortran comment conventions are not used and the comment indicator "!" has no special meaning. So, in the above example, "! Maximum number of elements" is interpreted as part of the value for the MAX_ELEMENTS definition.

Option `allow nofpp_comments` can be useful when you want to have a Fortran directive as a define value; for example:

```
#define dline(routname) !dir$ attributes alias:"__routname":: routname
```

altparam

Allows alternate syntax (without parentheses) for PARAMETER statements.

Syntax

Linux OS and macOS:

`-altparam`

`-noaltparam`

Windows OS:

`/altparam`

`/noaltparam`

Arguments

None

Default

`altparam` The alternate syntax for PARAMETER statements is allowed.

Description

This option specifies that the alternate syntax for PARAMETER statements is allowed. The alternate syntax is:

```
PARAMETER c = expr [, c = expr] ...
```

This statement assigns a name to a constant (as does the standard PARAMETER statement), but there are no parentheses surrounding the assignment list.

In this alternative statement, the form of the constant, rather than implicit or explicit typing of the name, determines the data type of the variable.

IDE Equivalent

Visual Studio: **Language > Enable Alternate PARAMETER Syntax**

Eclipse: None

Xcode: **Language > Enable Alternate PARAMETER Syntax**

Alternate Options

`altparam` Linux and macOS*: None

Windows: `/4Yaltparam`

`noaltparam` Linux and macOS*: None

Windows: `/4Naltparam`

assume

Tells the compiler to make certain assumptions.

Syntax

Linux OS and macOS:

`-assume` *keyword*`[,` *keyword*`...]`

Windows OS:

`/assume:`*keyword*`[,` *keyword*`...]`

Arguments

keyword Specifies the assumptions to be made. Possible values are:

`none`	Disables all assume options.
`[no]bscc`	Determines whether the backslash character is treated as a C-style control character syntax in character literals.
`[no]buffered_io`	Determines whether data is immediately read from or written to disk or accumulated in a buffer. For variable length, unformatted files, determines whether data is buffered on input or read directly from disk to user variables.
`[no]buffered_stdout`	Determines whether data is immediately written to the standard output device or accumulated in a buffer.
`[no]byterecl`	Determines whether units for the OPEN statement RECL specifier (record length) value in unformatted files are in bytes or longwords (four-byte units).
`[no]cc_omp`	Determines whether conditional compilation as defined by the OpenMP Fortran API is enabled or disabled.
`[no]contiguous_assumed_shape`	Determines whether contiguity is assumed for assumed-shape dummy arguments.
`[no]contiguous_pointer`	Determines whether contiguity is assumed for pointers.
`[no]dummy_aliases`	Determines whether the compiler assumes that dummy arguments to procedures share memory locations with other dummy arguments or with COMMON variables that are assigned.
`[no]fpe_summary`	Determines whether a floating-point exceptions summary is displayed when a STOP or ERROR STOP statement is encountered.

`[no]ieee_fpe_flags`	Determines whether the floating-point exception and status flags are saved on routine entry and restored on routine exit.
`[no]minus0`	Determines whether the compiler uses Fortran 2003 or Fortran 90/77 standard semantics in the SIGN intrinsic when treating -0.0 and +0.0 as 0.0, and how it writes the value on formatted output.
`[no]old_boz`	Determines whether the binary, octal, and hexadecimal constant arguments in intrinsic functions INT, REAL, DBLE, and CMPLX are treated as signed integer constants.
`[no]old_inquire_recl`	Determines the value of the RECL= specifier on an INQUIRE statement for an unconnected unit or a unit connected for stream access. Prior to Fortran 2018, this behavior was undefined (`ifort` used the value 0 for an unconnected file). Fortran 2018 specifies that the *scalar-int-variable* in the RECL= specifier becomes defined with -1 if the file is unconnected, and -2 if the file is connected for stream access.
`[no]old_ldout_format`	Determines the output of integer and real values in list-directed and namelist-directed output.
`[no]old_ldout_zero`	Determines the format of a floating-point zero produced by list-directed output. `old_ldout_zero` uses exponential format; `noold_ldout_zero` uses fractional format.
`[no]old_logical_assign`	Determines the behavior in assignment statements of logical values assigned to numeric variables and numeric values assigned to logical variables.
`[no]old_logical_ldio`	Determines whether NAMELIST and list-directed input accept logical values for numeric IO-list items and numeric values for logical IO-list items.
`[no]old_maxminloc`	Determines the results of intrinsics MAXLOC and MINLOC when given an empty array as an argument or every element of the mask is false.
`[no]old_unit_star`	Determines whether unit * is treated the same as units 5 and 6, or is distinct.
`[no]old_xor`	Determines whether .XOR. is defined by the compiler as an intrinsic operator.
`[no]protect_allocates`	Determines whether memory allocation requests using the ALLOCATE statement are protected with critical sections to avoid random timing problems in a multi-threaded environment.
`[no]protect_constants`	Determines whether a constant actual argument or a copy of it is passed to a called routine.
`[no]protect_parens`	Determines whether the optimizer honors parentheses in REAL and COMPLEX expression evaluations by not reassociating operations.

[no]realloc_lhs	Determines whether the compiler uses the current Fortran Standard rules or the old Fortran 2003 rules when interpreting assignment statements.
[no]recursion	Determines whether procedures are compiled for recursion by default.
[no]source_include	Determines whether the compiler searches for USE modules and INCLUDE files in the default directory or in the directory where the source file is located.
[no]std_intent_in	Determines whether the compiler assumes that dummy arguments with the INTENT(IN) attribute in a called procedure are not modified across a call, in accordance with the Fortran standard.
[no]std_minus0_rounding	Determines whether to display a negative value that is not zero but rounds to zero on output with a leading minus sign.
[no]std_mod_proc_name	Determines whether the names of module procedures are allowed to conflict with user external symbol names.
[no]std_value	Determines whether the VALUE attribute has the effect as if the actual argument is assigned to a temporary, and the temporary is then passed to the called procedure so that subsequent changes to the value of the dummy argument do not affect the actual argument, in accordance with the Fortran standard.
[no]underscore	Determines whether the compiler appends an underscore character to external user-defined names.
[no]2underscores (Linux and macOS*)	Determines whether the compiler appends two underscore characters to external user-defined names.
[no]writeable-strings	Determines whether character constants go into non-read-only memory.

Default

nobscc	The backslash character is treated as a normal character in character literals.
nobuffered_io	Data in the internal buffer is immediately read from or written (flushed) to disk (OPEN specifier BUFFERED='NO'). Data read from variable length, unformatted files is read directly from disk to a user's variables. If you set the FORT_BUFFERED environment variable to true, the default is `assume buffered_io`.

`nobuffered_stdout` Data is not buffered for the standard output device but instead is written immediately to the device.

`nobyterecl` Units for OPEN statement RECL values with unformatted files are in four-byte (longword) units.

`nocc_omp` Conditional compilation as defined by the OpenMP Fortran API is disabled unless option `[q or Q]openmp` is specified.

If compiler option `[q or Q]openmp` is specified, the default is `assume cc_omp`.

`nocontiguous_assumed_shape` Contiguity is not assumed for assumed-shape dummy arguments.

`nocontiguous_pointer` Contiguity is not assumed for pointers.

`nodummy_aliases` Dummy arguments to procedures do not share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use.

`nofpe_summary` Suppresses a summary of floating-point exceptions from being displayed when a STOP or ERROR STOP statement is encountered.

`noieee_fpe_flags` The flags are not saved on routine entry and they are not restored on routine exit.

`nominus0` The compiler uses Fortran 90/77 standard semantics in the SIGN intrinsic to treat -0.0 and +0.0 as 0.0, and writes a value of 0.0 with no sign on formatted output.

`noold_boz` The binary, octal, and hexadecimal constant arguments in intrinsic functions INT, REAL, DBLE, and CMPLX are treated as bit strings that represent a value of the data type of the intrinsic, that is, the bits are not converted.

`old_inquire_recl` The *scalar-int-variable* in a RECL= specifier of an INQUIRE statement for an unconnected unit becomes defined with the value 0; if the unit is connected for stream access, the value is undefined.

`old_ldout_format` For list-directed and namelist-directed output, integers are written with a fixed width that is dependent on the integer kind, and zero real values are written using the E format.

`old_ldout_zero` For list-directed output of a floating-point zero, exponential format is used instead of fractional output.

`noold_logical_assign` In the assignment statement L = N, where L is a logical variable and N is a numeric value, N is converted to integer if necessary, and L is assigned the value .FALSE. if the integer value is 0, and .TRUE. if the integer value is -1 or 1 according to the setting of the compiler option `fpscomp logicals`.

In the assignment statement N = L, where N is a variable of numeric type and L is a logical value, if L is .FALSE., N is assigned the value 0, converted, if necessary, to the type of N. If L is .TRUE., N is assigned the value 1 or -1, converted, if necessary, to the type of N, according to the setting of the compiler option `fpscomp logicals`.

The compiler option `fpscomp logicals` specifies that non-zero values are treated as true and zero values are treated as false. The literal constant .TRUE. has an integer value of 1 and the literal constant .FALSE. has an integer value of 0.

The default is `fpscomp nologicals`, which specifies that odd integer values (low bit one) are treated as true and even integer values (low bit zero) are treated as false. The literal constant .TRUE. has an integer value of -1 and the literal constant .FALSE. has an integer value of 0.

`noold_logical_ldio` Tells the compiler that NAMELIST and list-directed input cannot accept logical values (T, F, etc.) for numeric (integer, real, and complex) IO-list items or numeric values for logical IO-list items. If this option is specified and a logical value is given for a numeric item or a numeric value is given for a logical item in NAMELIST and list-directed input, a run-time error will be produced.

`old_maxminloc` MAXLOC and MINLOC return 1 when given an empty array as an argument or every element of the mask is false.

`old_unit_star` The READs or WRITEs to UNIT=* go to stdin or stdout, respectively, even if UNIT=5 or 6 has been connected to another file.

`old_xor` Intrinsic operator .XOR. is defined by the compiler.

`noprotect_allocates` Memory allocation requests using the ALLOCATE statement are not protected with critical sections and may encounter random timing problems in a multi-threaded environment.

`protect_constants` A constant actual argument is passed to a called routine. Any attempt to modify it results in an error.

`noprotect_parens` The optimizer reorders REAL and COMPLEX expressions without regard for parentheses by reassociating operations if it produces faster executing code.

`realloc_lhs` Tells the compiler that when the left-hand side of an assignment is an allocatable object, it should be reallocated to the shape of the right-hand side of the assignment before the assignment occurs. This is the current Fortran Standard definition. This feature may cause extra overhead at run time. The option `standard-realloc-lhs` has the same effect as `assume realloc_lhs`.

`norecursion` Tells the compiler that all procedures are not compiled for recursion, unless declared with the RECURSIVE keyword. Fortran 2018 specifies the default mode of compilation is recursion; previous standards specified no recursion. This default will change in a future release.

`source_include` The compiler searches for USE modules and INCLUDE files in the directory where the source file is located.

`std_intent_in` The compiler assumes that dummy arguments with the INTENT(IN) attribute in a called procedure are not modified across a call, in accordance with the Fortran standard.

`std_minus0_rounding` A negative value that is not zero but rounds to zero on output is displayed with a leading minus sign. For example, the value -0.00000001 in F5.1 format will be displayed as -0.0 rather than as 0.0.

`nostd_mod_proc_name` The compiler allows the names of module procedures to conflict with user external symbol names.

`std_value` The compiler assumes that the VALUE attribute has the effect as if the actual argument is assigned to a temporary, and the temporary is then passed to the called procedure so that subsequent changes to the value of the dummy argument do not affect the actual argument, in accordance with the Fortran standard.

Windows: `nounderscore` Linux and macOS*: `underscore`	On Windows* systems, the compiler does not append an underscore character to external user-defined names. On Linux* and macOS* systems, the compiler appends an underscore character to external user-defined names.
`no2underscores` (Linux and macOS*)	The compiler does not append two underscore characters to external user-defined names that contain an embedded underscore.
`nowriteable-strings`	The compiler puts character constants into read-only memory.

Description

This option specifies assumptions to be made by the compiler.

Option	Description
`assume none`	Disables all the assume options.
`assume bscc`	Tells the compiler to treat the backslash character (\) as a C-style control (escape) character syntax in character literals. The "bscc" keyword means "BackSlashControlCharacters."
`assume buffered_io`	Tells the compiler to accumulate records in a buffer. This sets the default for opening sequential files to BUFFERED='YES', which also occurs if the FORT_BUFFERED run-time environment variable is specified. When this option is specified, the internal buffer is filled, possibly by many record input statements (READ) or output statements (WRITE), before it is read from disk, or written to disk, by the Fortran run-time system. If a file is opened for direct access, I/O buffering is ignored. Using buffered reads and writes usually makes disk I/O more efficient by handling larger blocks of data on disk less often. However, if you request buffered writes, records not yet written to disk may be lost in the event of a system failure. The OPEN statement BUFFERED specifier applies to a specific logical unit. In contrast, the `assume [no]buffered_io` option and the FORT_BUFFERED environment variable apply to all Fortran units.
`assume buffered_stdout`	Tells the Fortran run-time system to accumulate data for the standard output device in a buffer. When the buffer is full or the user executes a FLUSH on OUTPUT_UNIT in intrinsic module ISO_FORTRAN_ENV, the data is displayed on the standard output unit. Using buffered writes may be a more efficient in time and space but use `assume nobuffered_stdout` if you want data displayed immediately on the standard output device, like for an input prompt. `assume [no]buffered_stdout` does not affect and is not affected by `assume [no]buffered_io`.

Option	Description
	After compiling with this option, the default blocksize for stdout is 8 KB.
`assume byterecl`	Specifies that the units for the OPEN statement RECL specifier (record length) value are in bytes for unformatted data files, not longwords (four-byte units). For formatted files, the RECL value is always in bytes. If a file is open for unformatted data and `assume byterecl` is specified, INQUIRE returns RECL in bytes; otherwise, it returns RECL in longwords. An INQUIRE returns RECL in bytes if the unit is not open.
`assume cc_omp`	Enables conditional compilation as defined by the OpenMP Fortran API. That is, when "!$space" appears in free-form source or "c$spaces" appears in column 1 of fixed-form source, the rest of the line is accepted as a Fortran line.
`assume contiguous_assumed_shape`	Tells the compiler to assume contiguity for assumed-shape dummy arguments. This may aid in optimization. However, if you are mistaken about the contiguity of your data, it could result in run-time failures.
`assume contiguous_pointer`	Tells the compiler to assume contiguity for pointers. This may aid in optimization. However, if you are mistaken about the contiguity of your data, it could result in run-time failures.
`assume dummy_aliases`	Tells the compiler that dummy (formal) arguments to procedures share memory locations with other dummy arguments (aliases) or with variables shared through use association, host association, or common block use. Specify the option when you compile the called subprogram. The program semantics involved with dummy aliasing do not strictly obey Standard Fortran and they slow performance, so you get better run-time performance if you do not use this option. However, if a program depends on dummy aliasing and you do not specify this option, the run-time behavior of the program will be unpredictable. In such programs, the results will depend on the exact optimizations that are performed. In some cases, normal results will occur, but in other cases, results will differ because the values used in computations involving the offending aliases will differ.
`assume fpe_summary`	Causes a summary of floating-point exceptions that occurred during program execution to be displayed when a STOP or ERROR STOP statement is encountered. Counts will be shown for each exception. This is the behavior specified by the Fortran 2003 standard. Note that if there is no STOP or ERROR STOP statement, no summary is displayed.
`assume ieee_fpe_flags`	Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance because it provides extra code to save and restore the floating-point exception and status flags (and the rounding mode) on entry to and exit from every routine compiled with the option. This option can be used to get the full Fortran Standard behavior of intrinsic modules IEEE EXCEPTIONS, IEEE ARITHMETIC, and IEEE FEATURES, which require that if a flag is signaling on routine entry, the processor will set it to quiet on entry and restore it to signaling on return. If a flag signals while the routine is executing, it will not be set to quiet on routine exit. Options `fpe` and `fpe-all` can be used to set the initial state for which floating-point exceptions will signal.

Option	Description
`assume minus0`	Tells the compiler to use Fortran 95 standard semantics for the treatment of the IEEE* floating value -0.0 in the SIGN intrinsic, which distinguishes the difference between -0.0 and +0.0, and to write a value of -0.0 with a negative sign on formatted output.
`assume old_boz`	Tells the compiler that the binary, octal, and hexadecimal constant arguments in intrinsic functions INT, REAL, DBLE, and CMPLX should be treated as signed integer constants.
`assume noold_inquire_recl`	Tells the compiler to use Fortran 2018 semantics for RECL= on an INQUIRE statement. If the unit is unconnected, the *scalar-int-variable* of the RECL= specifier becomes defined with the value -1; if the unit is connected for stream access, it becomes defined with the value -2.
`assume noold_ldout_format`	Tells the compiler to use Fortran 2003 standard semantics for output of integer and real values in list-directed and namelist-directed output. Integers are written using an I0 format with a leading blank for spacing. Real and complex values are written using and E or F format with a leading blank for spacing. The format used depends on the magnitude of the value. Values that are zero are written using an F format.
`assume noold_ldout_real`	For list-directed output of a floating-point zero, fractional format is used instead of exponential output. Early versions of the Fortran standard specified exponential format should be used for list-directed output of a floating-point zero value. Newer versions of the standard specify that fractional format should be used.
`assume old_logical_assign`	In the assignment statement L = N, where L is a logical variable and N is a numeric value, N is converted to integer if necessary, and L is assigned the bit value of the integer value without conversion. In the assignment statement N = L, where N is a variable of numeric type and L is a logical value, N is assigned the bit value of the value of L without conversion.
`assume old_logical_int`	Logical values are allowed for numeric items and numeric values are allowed for logical items.
`assume noold_maxminloc`	Tells the compiler that MAXLOC and MINLOC should return 0 when given an empty array as an argument or every element of the mask is false. Compared to the default setting (`old_maxminloc`), this behavior may slow performance because of the extra code needed to check for an empty array argument or for an all-false mask.
`assume noold_unit_star`	Tells the compiler that READs or WRITEs to UNIT=* go to the file to which UNIT=5 or 6 is connected.
`assume noold_xor`	Prevents the compiler from defining .XOR. as an intrinsic operator. This lets you use .XOR. as a user-defined operator. This is a Fortran 2003 feature.
`assume protect_allocate`	Memory allocation requests using the ALLOCATE statement are protected with critical sections to avoid random timing problems in a multi-threaded environment in some distributions and configurations.

Option	Description
`assume noprotect_constants`	Tells the compiler to pass a copy of a constant actual argument. This copy can be modified by the called routine, even though the Fortran standard prohibits such modification. The calling routine does not see any modification to the constant.
`assume protect_parens`	Tells the optimizer to honor parentheses in REAL and COMPLEX expression evaluations by not reassociating operations. For example, (A+B)+C would not be evaluated as A+(B+C). If assume noprotect_parens is specified, (A+B)+C would be treated the same as A +B+C and could be evaluated as A+(B+C) if it produced faster executing code. Such reassociation could produce different results depending on the sizes and precision of the arguments. For example, in (A+B)+C, if B and C had opposite signs and were very large in magnitude compared to A, A+B could result in the value as B; adding C would result in 0.0. With reassociation, B+C would be 0.0; adding A would result in a non-zero value.
`assume norealloc_lhs`	The compiler uses the old Fortran 2003 rules when interpreting assignment statements. The left-hand side is assumed to be allocated with the correct shape to hold the right-hand side. If it is not, incorrect behavior will occur. The option `nostandard-realloc-lhs` has the same effect as `assume norealloc_lhs`.
`assume recursion`	Tells the compiler to compile procedures that are declared without the RECURSIVE or NON_RECURSIVE keyword as recursive procedures. In Fortran 2018, these procedures are compiled for recursion by default; in previous standards they were compiled as non-recursive procedures. The current default behavior is non-recursive. This will change in a future release.
`assume nosource_include`	Tells the compiler to search the default directory for module files specified by a USE statement or source files specified by an INCLUDE statement. This option affects fpp preprocessor behavior and the USE statement.
`assume nostd_intent_in`	Tells the compiler to assume that dummy arguments with the INTENT(IN) attribute in a called procedure may be modified across a call. This is not in accordance with the Fortran standard. If you specify option `standard_semantics`, it sets option `assume std_intent_in`.
`assume nostd_minus0_rounding`	Tells the compiler to use pre-Fortran 2008 standard semantics for the treatment of IEEE* floating values that are negative, non-zero, and when rounded for display are zero. The value should be printed without a leading minus sign. For example, the floating value -0.00000001 when rounded in F5.1 format will be displayed as 0.0. Use `assume std_minus0_rounding` to use Fortran 2008 standard semantics to display this value as -0.0 when rounded in F5.1 format. `assume [no]std_minus0_rounding` does not affect and is not affected by `assume [no]minus0`. The former controls printing of a minus sign for non-zero numbers while the latter controls printing of actual signed zero values.
`assume std_mod_proc_name`	Tells the compiler to revise the names of module procedures so they do not conflict with user external symbol names. For example, procedure `proc` in module `m` would be named `m_MP_proc`. The Fortran 2003 Standard requires that module procedure names not conflict with other external symbols.

Option	Description
	By default, procedure `proc` in module `m` would be named `m_mp_proc`, which could conflict with a user-defined external name `m_mp_proc`.
`assume nostd_value`	Tells the compiler to use pre-Fortran 2003 standard semantics for the VALUE attribute so that the value of the actual argument is passed to the called procedure, not the address of the actual argument nor the address of a copy of the actual argument.
`assume underscore`	Tells the compiler to append an underscore character to external user-defined names: the main program name, named common blocks, BLOCK DATA blocks, global data names in MODULEs, and names implicitly or explicitly declared EXTERNAL. The name of a blank (unnamed) common block remains _BLNK__, and Fortran intrinsic names are not affected.
`assume 2underscores` (Linux and macOS*)	Tells the compiler to append two underscore characters to external user-defined names that contain an embedded underscore: the main program name, named common blocks, BLOCK DATA blocks, global data names in MODULEs, and names implicitly or explicitly declared EXTERNAL. The name of a blank (unnamed) common block remains _BLNK__, and Fortran intrinsic names are not affected. This option does not affect external names that do not contain an embedded underscore. By default, the compiler only appends one underscore to those names. For example, if you specify `assume 2underscores` for external names `my_program` and `myprogram`, `my_program` becomes `my_program__`, but `myprogram` becomes `myprogram_`.
`assume writeable-strings`	Tells the compiler to put character constants into non-read-only memory.

IDE Equivalent

Visual Studio: **Code Generation > Enable Recursive Routines** (`/assume:[no]recursion`)

Compatibility > Treat Backslash as Normal Character in Strings (`/assume:[no]bscc`)

Data > Assume Dummy Arguments Share Memory Locations (`/assume:dummy_aliases`)

Data > Constant Actual Arguments Can Be Changed (`/assume:noprotect_constants`)

Data > Use Bytes as RECL=Unit for Unformatted Files (`/assume:byterecl`)

External Procedures > Append Underscore to External Names (`/assume:underscore`)

Floating Point > Enable IEEE Minus Zero Support (`/assume:minus0`)

Optimization > I/O Buffering (`/assume:buffered_io`)

Preprocessor > Default Include and Use Path (`/assume:nosource_include`)

Preprocessor > OpenMP Conditional Compilation (`/assume:nocc_omp`)

Eclipse: None

Xcode: **Code Generation > Enable Recursive Routines** (`-assume [no]recursion`)

Compatibility > Treat Backslash as Normal Character in Strings (`-assume [no]bscc`)

Data > Assume Dummy Arguments Share Memory Locations (`-assume [no]dummy_aliases`)

Data > Constant Actual Arguments Can Be Changed (`-assume [no]protect_constants`)

Data > Use Bytes as RECL=Unit for Unformatted Files (`-assume [no]byterecl`)

External Procedures > Append Underscore to External Names (`-assume [no]underscore`)

Floating Point > Enable IEEE Minus Zero Support (`-assume [no]minus0`)

Optimization > I/O Buffering (`-assume [no]buffered_io`)

Preprocessor > Default Include and Use Path (`-assume [no]source_include`)

Preprocessor > OpenMP Conditional Compilation (`-assume [no]cc_omp`)

Alternate Options

`assume nobsc`
Linux and macOS*: `-nbs`
Windows: `/nbs`

`assume dummy_aliases`
Linux and macOS*: `-common-args`
Windows: `/Qcommon-args`

`assume protect_parens`
Linux and macOS*: `-fprotect-parens`
Windows: `/Qprotect-parens`

`assume realloc_lhs`
Linux and macOS*: `-standard-realloc-lhs`
Windows: `/standard-realloc-lhs`

`assume recursion`
Linux and macOS*: `-recursive`
Windows: `/recursive`

See Also

`fp-model, fp` compiler option
`fpe` compiler option
`fpe-all` compiler option
`standard-semantics` compiler option
`standard-realloc-lhs` compiler option
`fpscomp` compiler option, setting logicals
Rules for Unformatted Sequential READ Statements

ccdefault

Specifies the type of carriage control used when a file is displayed at a terminal screen.

Syntax

Linux OS and macOS:

`-ccdefault` *keyword*

Windows OS:

`/ccdefault:`*keyword*

Arguments

keyword Specifies the carriage-control setting to use. Possible values are:

`none` Tells the compiler to use no carriage control processing.

`default` Tells the compiler to use the default carriage-control setting.

`fortran` Tells the compiler to use normal Fortran interpretation of the first character. For example, the character 0 causes output of a blank line before a record.

`list` Tells the compiler to output one line feed between records.

Default

`ccdefault default` The compiler uses the default carriage control setting.

Description

This option specifies the type of carriage control used when a file is displayed at a terminal screen (units 6 and *). It provides the same functionality as using the CARRIAGECONTROL specifier in an OPEN statement.

The default carriage-control setting can be affected by the `vms` option. If option `vms` is specified with `ccdefault default`, carriage control defaults to normal Fortran interpretation (`ccdefault fortran`) if the file is formatted and the unit is connected to a terminal. If option `novms` (the default) is specified with `ccdefault default`, carriage control defaults to list (`ccdefault list`).

IDE Equivalent

Visual Studio: **Run-time > Default Output Carriage Control**

Eclipse: None

Xcode: **Run-time > Default Output Carriage Control**

Alternate Options

None

check

Checks for certain conditions at run time.

Syntax

Linux OS and macOS:

`-check [keyword[, keyword...]]`

`-nocheck`

Windows OS:

`/check[:keyword[, keyword...]]`

`/nocheck`

Arguments

keyword Specifies the conditions to check. Possible values are:

`none` Disables all check options.

`[no]arg_temp_created` Determines whether checking occurs for actual arguments copied into temporary storage before routine calls.

`[no]assume`	Determines whether checking occurs to test that the *scalar-Boolean-expression* in the ASSUME directive is true, or that the addresses in the ASSUME_ALIGNED directive are aligned on the specified byte boundaries.
`[no]bounds`	Determines whether checking occurs for array subscript and character substring expressions.
`[no]contiguous`	Determines whether the compiler checks pointer contiguity at pointer-assignment time.
`[no]format`	Determines whether checking occurs for the data type of an item being formatted for output.
`[no]output_conversion`	Determines whether checking occurs for the fit of data items within a designated format descriptor field.
`[no]pointers`	Determines whether checking occurs for certain disassociated or uninitialized pointers or unallocated allocatable objects.
`[no]shape`	Determines whether array conformance checking is performed.
`[no]stack`	Determines whether checking occurs on the stack frame.
`[no]udio_iostat`	Determines whether conformance checking occurs when user-defined derived type input/output routines are executed.
`[no]uninit`	Determines whether checking occurs for uninitialized variables.
`all`	Enables all check options.

Default

`nocheck`	No checking is performed for run-time failures. Note that if option `vms` is specified, the defaults are `check format` and `check output_conversion`.

Description

This option checks for certain conditions at run time.

Option	Description
`check none`	Disables all check options (same as `nocheck`).
`check arg_temp_created`	Enables run-time checking on whether actual arguments are copied into temporary storage before routine calls. If a copy is made at run-time, an informative message is displayed.
`check assume`	Enables run-time checking on whether the *scalar-Boolean-expression* in the ASSUME directive is true and that the addresses in the ASSUME_ALIGNED directive are aligned on the specified byte boundaries. If the test is .FALSE., a run-time error is reported and the execution terminates.

Option	Description
`check bounds`	Enables compile-time and run-time checking for array subscript and character substring expressions. An error is reported if the expression is outside the dimension of the array or the length of the string. For array bounds, each individual dimension is checked. For arrays that are dummy arguments, only the lower bound is checked for a dimension whose upper bound is specified as * or where the upper and lower bounds are both 1. For some intrinsics that specify a DIM= dimension argument, such as LBOUND, an error is reported if the specified dimension is outside the declared rank of the array being operated upon. Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance. It is recommended that you do bounds checking on unoptimized code. If you use option check bounds on optimized code, it may produce misleading messages because registers (not memory locations) are used for bounds values.
`check contiguous`	Tells the compiler to check pointer contiguity at pointer-assignment time. This will help prevent programming errors such as assigning contiguous pointers to non-contiguous objects.
`check format`	Issues the run-time FORVARMIS fatal error when the data type of an item being formatted for output does not match the format descriptor being used (for example, a REAL*4 item formatted with an I edit descriptor). With `check noformat`, the data item is formatted using the specified descriptor unless the length of the item cannot accommodate the descriptor (for example, it is still an error to pass an INTEGER*2 item to an E edit descriptor).
`check output_conversion`	Issues the run-time OUTCONERR continuable error message when a data item is too large to fit in a designated format descriptor field without loss of significant digits. Format truncation occurs, the field is filled with asterisks (*), and execution continues.
`check pointers`	Enables run-time checking for disassociated or uninitialized Fortran pointers, unallocated allocatable objects, and integer pointers that are uninitialized.
`check shape`	Enables compile-time and run-time array conformance checking in contexts where it is required by the standard. These include the right-hand and left-hand side of intrinsic and elemental defined assignment, the operands of intrinsic and elemental defined binary operations, two or more array arguments to ELEMENTAL procedures, the ARRAY= and MASK= arguments to intrinsic procedures as required, and the arguments to the intrinsic module procedures IEEE_SET_FLAG and IEEE_SET_HALTING_MODE. In an ALLOCATE statement with array bounds specified for an allocate-object and with SOURCE=source specified, the allocate-object must conform with source. Note that you can specify a setting in the `warn` option to choose whether array conformance violations are diagnosed with errors or warnings.
`check stack`	Enables checking on the stack frame. The stack is checked for buffer overruns and buffer underruns. This option also enforces local variables initialization and stack pointer verification.

Option	Description
	This option disables optimization and overrides any optimization level set by option `O`.
`check udio_iostat`	Enables checking that user-defined derived type input/output routines conform to the standard's rules for assigning values to IOSTAT= and IOMSG= specifiers.
`check uninit`	Enables run-time checking for uninitialized variables. If a variable is read before it is written, a run-time error routine will be called. Only local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, and LOGICAL without the SAVE attribute are checked. To detect uninitialized arrays or array elements, please see option `[Q]init` or see the article titled: Detection of Uninitialized Floating-point Variables in Intel® Fortran, which is located in https://software.intel.com/articles/detection-of-uninitialized-floating-point-variables-in-intel-fortran
`check all`	Enables all check options. This is the same as specifying `check` with no keyword. This option disables optimization and overrides any optimization level set by option `O`.

To get more detailed location information about where an error occurred, use option `traceback`.

IDE Equivalent

Visual Studio:

Run-time > Runtime Error Checking (`/nocheck`, `/check:all`)

Run-time > Check Array and String Bounds (`/check:bounds`)

Run-time > Check Uninitialized Variables (`/check:uninit`)

Run-time > Check Edit Descriptor Data Type (`/check:format`)

Run-time > Check Edit Descriptor Data Size (`/check:output_conversion`)

Run-time > Check For Actual Arguments Using Temporary Storage (`/check:arg_temp_created`)

Run-time > Check Array Conformance (`/check:shape`)

Run-time > Check For Null Pointers and Allocatable Array References (`/check:pointers`)

Eclipse: None

Xcode: **Run-time > Runtime Error Checking** (`-check all, -check none`)

Run-time > Check Array and String Bounds (`-check [no]bounds`)

Run-time > Check Edit Descriptor Data Type (`-check [no]format`)

Run-time > Check Edit Descriptor Data Size (`-check [no]output_conversion`)

Run-time > Check For Actual Arguments Using Temporary Storage (`-check [no]arg_temp_created`)

Run-time > Check Array Conformance (`-check [no]shape`)

Run-time > Check for Uninitialized Variables (`-check [no]uninit`)

Run-time > Check For Null Pointers and Allocatable Array References (`-check [no]pointers`)

Run-time > Check Stack Frame (`-check [no]stack`)

Alternate Options

`check none` Linux and macOS*: `-nocheck`
Windows: `/nocheck`

`check bounds` Linux and macOS*: `-CB`
Windows: `/CB`

`check shape` Linux and macOS*: `-CS`
Windows: `/CS`

`check uninit` Linux and macOS*: `-CU`
Windows: `/RTCu`, `/CU`

`check all` Linux and macOS*: `-check, -C`
Windows: `/check`, `/C`

See Also

`traceback` compiler option
`init, Qinit` compiler option
`warn` compiler option
ASSUME directive
ASSUME_ALIGNED directive

extend-source

Specifies the length of the statement field in a fixed-form source file.

Syntax

Linux OS and macOS:

`-extend-source [size]`

`-noextend-source`

Windows OS:

`/extend-source[:size]`

`/noextend-source`

Arguments

size	Is the length of the statement field in a fixed-form source file. Possible values are: 72, 80, or 132.

Default

72	If you do not specify this option or you specify `noextend-source`, the statement field ends at column 72.
132	If you specify `extend_source` without *size*, the statement field ends at column 132.

Description

This option specifies the size (column number) of the statement field of a source line in a fixed-form source file. This option is valid only for fixed-form files; it is ignored for free-form files.

When size is specified, it is the last column parsed as part of the statement field. Any columns after that are treated as comments.

If you do not specify *size*, it is the same as specifying `extend_source 132`.

Option	Description
`extend-source 72`	Specifies that the statement field ends at column 72.
`extend-source 80`	Specifies that the statement field ends at column 80.
`extend-source 132`	Specifies that the statement field ends at column 132.

IDE Equivalent

Visual Studio: **Language > Fixed Form Line Length**

Eclipse: None

Xcode: **Language > Fixed Form Line Length**

Alternate Options

`extend-source 72`
Linux and macOS*: `-72`
Windows: `/4L72`

`extend-source 80`
Linux and macOS*: `-80`
Windows: `/4L80`

`extend-source 132`
Linux and macOS*: `-132`
Windows: `/4L132`

fixed

Specifies source files are in fixed format.

Syntax

Linux OS and macOS:

`-fixed`

`-nofixed`

Windows OS:

`/fixed`

`/nofixed`

Arguments

None

Default

OFF The source file format is determined from the file extension.

Description

This option specifies source files are in fixed format. If this option is not specified, format is determined as follows:

- Files with an extension of .f90, .F90, or .i90 are free-format source files.
- Files with an extension of .f, .for, .FOR, .ftn, .FTN, .fpp, .FPP, or .i are fixed-format files.

Note that on Linux* and macOS* systems, file names and file extensions are case sensitive.

IDE Equivalent

Visual Studio: **Language > Source File Format** (`/free`, `/fixed`)

Eclipse: None

Xcode: **Language > Source File Format** (`/free`, `/fixed`)

Alternate Options

Linux and macOS*: `-FI`

Windows: `/nofree`, `/FI`, `/4Nf`

free

Specifies source files are in free format.

Syntax

Linux OS and macOS:

`-free`

`-nofree`

Windows OS:

`/free`

`/nofree`

Arguments

None

Default

OFF The source file format is determined from the file extension.

Description

This option specifies source files are in free format. If this option is not specified, format is determined as follows:

- Files with an extension of .f90, .F90, or .i90 are free-format source files.
- Files with an extension of .f, .for, .FOR, .ftn, or .i are fixed-format files.

IDE Equivalent

Visual Studio: **Language > Source File Format** (`/free`, `/fixed`)

Eclipse: None

Xcode: **Language > Source File Format** (`/free`, `/fixed`)

Alternate Options

Linux and macOS*: `-FR`

Windows: `/nofixed`, `/FR`, `/4Yf`

See Also

`fixed` compiler option

iface

Specifies the default calling convention and argument-passing convention for an application.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/iface:`*`keyword`*

Arguments

keyword Specifies the calling convention or the argument-passing convention. Possible values are:

`default`	Tells the compiler to use the default calling conventions.
`cref`	Tells the compiler to use calling conventions C, REFERENCE.
`cvf`	Tells the compiler to use calling conventions compatible with Compaq Visual Fortran*. This value is only available on Windows* systems.
`[no]mixed_str_len_arg`	Determines the argument-passing convention for hidden-length character arguments.
`stdcall`	Tells the compiler to use calling convention STDCALL.
`stdref`	Tells the compiler to use calling conventions STDCALL, REFERENCE.

Default

`/iface:default` The default calling convention is used.

Description

This option specifies the default calling convention and argument-passing convention for an application.

The aspects of calling and argument passing controlled by this option are as follows:

- The calling mechanism (C or STDCALL): On IA-32 architecture, these mechanisms differ in how the stack register is adjusted when a procedure call returns. On Intel® 64 architecture, the only calling mechanism available is C; requests for the STDCALL mechanism are ignored.
- The argument passing mechanism (by value or by reference)
- Character-length argument passing (at the end of the argument list or after the argument address)

- The case of external names (uppercase or lowercase)
- The name decoration (prefix and suffix)

You can also use the ATTRIBUTES compiler directive to modify these conventions on an individual basis. Note that the effects of the ATTRIBUTES directive do not always match that of the iface option of the same name.

Option	Description
`/iface:default`	Tells the compiler to use the default calling conventions. These conventions are as follows: • The calling mechanism: C • The argument passing mechanism: by reference • Character-length argument passing: at end of argument list • The external name case: uppercase • The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 architecture; no suffix
`/iface:cref`	Tells the compiler to use the same conventions as `/iface:default` except that external names are lowercase.
`/iface:cvf`	Tells the compiler to use calling conventions compatible with Compaq Visual Fortran* and Microsoft Fortran PowerStation. This option is only available on Windows* systems. These conventions are as follows: • The calling mechanism: STDCALL on Windows* systems using IA-32 architecture • The argument passing mechanism: by reference • Character-length argument passing: following the argument address • The external name case: uppercase • The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 architecture. On Windows* systems using IA-32 architecture, @n suffix where *n* is the number of bytes to be removed from the stack on exit from the procedure. No suffix on other systems.
`/iface:mixed_str_len_arg`	Specifies argument-passing conventions for hidden-length character arguments. This option tells the compiler that the hidden length passed for a character argument is to be placed immediately after its corresponding character argument in the argument list. This is the method used by Compaq Visual Fortran*. When porting mixed-language programs that pass character arguments, either this option must be specified correctly or the order of hidden length arguments must be changed in the source code. This option can be used in addition to other `/iface` options.
`/iface:stdcall`	Tells the compiler to use the following conventions: • The calling mechanism: STDCALL • The argument passing mechanism: by value • Character-length argument passing: at the end of the argument list • The external name case: uppercase • The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 architecture. On Windows* systems using IA-32 architecture, @n suffix where *n* is the number of bytes to be removed from the stack on exit from the procedure. No suffix on other systems.
`/iface:stdref`	Tells the compiler to use the same conventions as `/iface:stdcall` except that argument passing is by reference.

Caution
On Windows systems, if you specify option `/iface:cref`, it overrides the default for external names and causes them to be lowercase. It is as if you specified "!dir$ attributes c, reference" for the external name.

If you specify option `/iface:cref` and want external names to be uppercase, you must explicitly specify option `/names:uppercase`.

Caution
On systems using IA-32 architecture, there must be agreement between the calling program and the called procedure as to which calling mechanism (C or STDCALL) is used, or unpredictable errors may occur. If you change the default mechanism to STDCALL, you must use the ATTRIBUTES DEFAULT directive to reset the calling conventions for any procedure that is:

- Specified as a FINAL procedure for a derived type
- Specified as a USEROPEN procedure in an OPEN statement
- Referenced in a variable format expression
- Specified as a comparison routine passed to the QSORT library routine from module IFPORT
- Used as a dialog callback procedure with the IFLOGM module
- Called from other contexts or languages that assume the C calling convention

Note that Visual Studio projects that are imported using the *Extract Compaq Visual Fortran project items* wizard have `/iface:cvf` applied automatically; this establishes STDCALL as the default convention.

IDE Equivalent

Visual Studio: **External Procedures > Calling Convention** (`/iface:{cref|stdref|stdcall|cvf|default}`)

External Procedures > String Length Argument Passing (`/iface:[no]mixed_str_len_arg`)

Eclipse: None

Xcode: None

Alternate Options

`/iface:cvf`
Linux and macOS*: None
Windows: `/Gm`

`/iface:mixed_str_len_arg`
Linux and macOS*: `-mixed-str-len-arg`
Windows: None

`/iface:nomixed_str_len_arg`
Linux and macOS*: `-nomixed-str-len-arg`
Windows: None

`/iface:stdcall`
Linux and macOS*: None
Windows: `/Gz`

See Also

Language Reference: ATTRIBUTES

names

Specifies how source code identifiers and external names are interpreted.

Syntax

Linux OS and macOS:

`-names keyword`

Windows OS:

`/names:keyword`

Arguments

keyword Specifies how to interpret the identifiers and external names in source code. Possible values are:

`lowercase`	Causes the compiler to ignore case differences in identifiers and to convert external names to lowercase.
`uppercase`	Causes the compiler to ignore case differences in identifiers and to convert external names to uppercase.
`as_is`	Causes the compiler to distinguish case differences in identifiers and to preserve the case of external names.

Default

`lowercase`	This is the default on Linux* and macOS* systems. The compiler ignores case differences in identifiers and converts external names to lowercase.
`uppercase`	This is the default on Windows* systems. The compiler ignores case differences in identifiers and converts external names to uppercase.

Description

This option specifies how source code identifiers and external names are interpreted. It can be useful in mixed-language programming.

This naming convention applies whether names are being defined or referenced.

You can use the ALIAS directive to specify an alternate external name to be used when referring to external subprograms.

Caution
On Windows systems, if you specify option `/iface:cref`, it overrides the default for external names and causes them to be lowercase. It is as if you specified "!dir$ attributes c, reference" for the external name.

If you specify option `/iface:cref` and want external names to be uppercase, you must explicitly specify option `/names:uppercase`.

IDE Equivalent

Visual Studio: **External Procedures > Name Case Interpretation**

Eclipse: None

Xcode: **External Procedures > Name Case Interpretation**

Alternate Options

None

See Also

`iface` compiler option

ALIAS Directive

pad-source, Qpad-source

Specifies padding for fixed-form source records.

Syntax

Linux OS and macOS:

`-pad-source`

`-nopad-source`

Windows OS:

`/pad-source`

`/nopad-source`

`/Qpad-source`

`/Qpad-source-`

Arguments

None

Default

`-nopad-source` Fixed-form source records are not padded.

or
`/Qpad-source-`

Description

This option specifies padding for fixed-form source records. It tells the compiler that fixed-form source lines shorter than the statement field width are to be padded with spaces to the end of the statement field. This affects the interpretation of character and Hollerith literals that are continued across source records.

The default value setting causes a warning message to be displayed if a character or Hollerith literal that ends before the statement field ends is continued onto the next source record. To suppress this warning message, specify setting `nousage` for option `warn`.

Specifying `[Q]pad-source` can prevent warning messages associated with setting `usage` for option `warn`.

IDE Equivalent

Visual Studio: **Language > Pad Fixed Form Source Lines**

Eclipse: None

Xcode: **Language > Pad Fixed Form Source Lines**

Alternate Options

None

See Also

`warn` compiler option

stand

Tells the compiler to issue compile-time messages for nonstandard language elements.

Syntax

Linux OS and macOS:

`-stand [keyword]`

`-nostand`

Windows OS:

`/stand[:keyword]`

`/nostand`

Arguments

keyword Specifies the language to use as the standard. Possible values are:

`none`	Issues no messages for nonstandard language elements.
`f90`	Issues messages for language elements that are not standard in Fortran 90.
`f95`	Issues messages for language elements that are not standard in Fortran 95.
`f03`	Issues messages for language elements that are not standard in Fortran 2003.
`f08`	Issues messages for language elements that are not standard in Fortran 2008.
`f18`	Issues messages for language elements that are not standard in Fortran 2018. This setting was previously named `f15`.

Default

`nostand` The compiler issues no messages for nonstandard language elements.

Description

This option tells the compiler to issue compile-time messages for nonstandard language elements.

If you do not specify a keyword for `stand`, it is the same as specifying `stand f08`.

Option	Description
`stand none`	Tells the compiler to issue no messages for nonstandard language elements. This is the same as specifying `nostand`.
`stand f90`	Tells the compiler to issue messages for language elements that are not standard in Fortran 90.
`stand f95`	Tells the compiler to issue messages for language elements that are not standard in Fortran 95.
`stand f03`	Tells the compiler to issue messages for language elements that are not standard in Fortran 2003.

Option	Description
`stand f08`	Tells the compiler to issue messages for language elements that are not standard in Fortran 2008. This option is set if you specify `warn stderrors`.
`stand f18`	Tells the compiler to issue messages for language elements that are not standard in Fortran 2018. Setting `f18` was previously named `f15`.

NOTE
When you specify this option, things that are not provided for in the Fortran standard at the specified standard level are diagnosed with warnings - this includes compiler directives recognized by ifort.

These standard compliance warnings can be ignored as they are informational only and do not affect compilation.

IDE Equivalent

Visual Studio: **Diagnostics > Warn For Nonstandard Fortran**

Eclipse: None

Xcode: **Diagnostics > Warn For Nonstandard Fortran**

Alternate Options

`stand none`	Linux and macOS*: `-nostand` Windows: `/nostand`, `/4Ns`
`stand f90`	Linux and macOS*: `-std90` Windows: `/4Ys`
`stand f95`	Linux and macOS*: `-std95` Windows: None
`stand f03`	Linux and macOS*: `-std03` Windows: None
`stand f08`	Linux and macOS*: `-std08`, `-stand`, `-std` Windows: `/stand`
`stand f18` or `stand f15` (deprecated)	Linux and macOS*: `-std18` or `-std15` (deprecated) Windows: none

See Also

`warn stderrors` compiler option

standard-realloc-lhs

Determines whether the compiler uses the current Fortran Standard rules or the old Fortran 2003 rules when interpreting assignment statements.

Syntax

Linux OS and macOS:

`-standard-realloc-lhs`

`-nostandard-realloc-lhs`

Windows OS:

`/standard-realloc-lhs`

`/nostandard-realloc-lhs`

Arguments

None

Default

`standard-realloc-lhs` The compiler uses the current Fortran Standard rules when interpreting assignment statements.

Description

This option determines whether the compiler uses the current Fortran Standard rules or the old Fortran 2003 rules when interpreting assignment statements.

Option `standard-realloc-lhs` (the default), tells the compiler that when the left-hand side of an assignment is an allocatable object, it should be reallocated to the shape of the right-hand side of the assignment before the assignment occurs. This is the current Fortran Standard definition. This feature may cause extra overhead at run time. This option has the same effect as option `assume realloc_lhs`.

If you specify `nostandard-realloc-lhs`, the compiler uses the old Fortran 2003 rules when interpreting assignment statements. The left-hand side is assumed to be allocated with the correct shape to hold the right-hand side. If it is not, incorrect behavior will occur. This option has the same effect as option `assume norealloc_lhs`.

IDE Equivalent

None

Alternate Options

None

standard-semantics

Determines whether the current Fortran Standard behavior of the compiler is fully implemented.

Syntax

Linux OS and macOS:

`-standard-semantics`

Windows OS:

`/standard-semantics`

Arguments

None

Default

OFF The compiler implements most but not all of the current Fortran Standard behavior.

Description

This option determines whether the current Fortran Standard behavior of the compiler is fully implemented.

If you specify option `standard-semantics`, it enables all of the options that implement the current Fortran Standard behavior of the compiler, which is Fortran 2008 features.

Option `standard-semantics` enables option `fpscomp logicals` and the following settings for option `assume`: byterecl, fpe_summary, ieee_fpe_flags (if the `fp-model` option setting is `strict` or `precise`), minus0, noold_inquire_recl, noold_ldout_format, noold_ldout_zero, noold_maxminloc, noold_unit_star, noold_xor, protect_parens, realloc_lhs[1], std_intent_in, std_minus0_rounding[1], std_mod_proc_name, and std_value[1].

If you specify option `standard-semantics` and also explicitly specify a different setting for an affected `assume` option, the value you specify takes effect. It overrides the settings enabled by option `standard-semantics`.

IDE Equivalent

Visual Studio: **Language > Enable F2008 Semantics**

Eclipse: None

Xcode: **Language > Enable F2008 Semantics**

Alternate Options

None

See Also

`assume` compiler option
`stand` compiler option

syntax-only

Tells the compiler to check only for correct syntax.

Syntax

Linux OS and macOS:

`-syntax-only`

Windows OS:

`/syntax-only`

Arguments

None

[1] This is the default setting for this assume option.

Default

OFF Normal compilation is performed.

Description

This option tells the compiler to check only for correct syntax. It lets you do a quick syntax check of your source file.

Compilation stops after the source file has been parsed. No code is generated, no object file is produced, and some error checking done by the optimizer is bypassed.

Warnings and messages appear on `stderr`.

IDE Equivalent

None

Alternate Options

Linux: `-y`, `-fsyntax-only`, `-syntax` (this is a deprecated option)

macOS*: `-y`, `-fsyntax-only`

Windows: `/Zs`

wrap-margin

Provides a way to disable the right margin wrapping that occurs in Fortran list-directed output.

Syntax

Linux OS:

`-wrap-margin`

`-no-wrap-margin`

macOS:

`None`

Windows OS:

`/wrap-margin`

`/wrap-margin-`

Arguments

None

Default

`wrap-margin` The right margin wraps at column 80 if the record length is greater than 80 characters.

Description

This option provides a way to disable the right margin wrapping that occurs in Fortran list-directed output. By default, when the record being written becomes longer than 80 characters, the record is wrapped to a new record at what is called the "right margin".

Specify `-no-wrap-margin` (Linux*) or `/wrap-margin-` (Windows*) to disable this behavior.

IDE Equivalent

None

Alternate Options

None

Data Options

align

Tells the compiler how to align certain data items.

Syntax

Linux OS and macOS:

`-align [keyword[, keyword...]]`

`-noalign`

Windows OS:

`/align[:keyword[, keyword...]]`

`/noalign`

Arguments

keyword Specifies the data items to align. Possible values are:

`none`	Prevents padding bytes anywhere in common blocks and structures.
`array`*n*`byte`	Specifies a starting boundary for arrays.
`[no]commons`	Affects alignment of common block entities.
`[no]dcommons`	Affects alignment of common block entities.
`[no]qcommons`	Affects alignment of common block entities.
`[no]zcommons`	Affects alignment of common block entities.
`[no]records`	Affects alignment of derived-type components and fields of record structures.
`rec`*n*`byte`	Specifies a size boundary for derived-type components and fields of record structures.
`[no]sequence`	Affects alignment of sequenced derived-type components.
`all`	Adds padding bytes whenever possible to data items in common blocks and structures.

Default

`nocommons` Adds no padding bytes for alignment of common blocks.

`nodcommmons` Adds no padding bytes for alignment of common blocks.

`noqcommmons` Adds no padding bytes for alignment of common blocks.

`nozcommmons` Adds no padding bytes for alignment of common blocks.

`records` Aligns derived-type components and record structure fields on default natural boundaries.

`nosequence` Causes derived-type components declared with the SEQUENCE statement to be packed, regardless of current alignment rules set by the user.

By default, no padding is added to common blocks but padding is added to structures.

Description

This option specifies the alignment to use for certain data items. The compiler adds padding bytes to perform the alignment.

Option	Description
`align none`	Tells the compiler not to add padding bytes anywhere in common blocks or structures. This is the same as specifying `noalign`.
`align array`*n*`byte`	Aligns the start of arrays on an *n*-byte boundary. *n* can be 8, 16, 32, 64, 128, or 256. The default value for *n* is 8. This affects the starting alignment for all arrays except for arrays in COMMON. Arrays do not have padding between their elements.
`align commons`	Aligns all common block entities on natural boundaries up to 4 bytes, by adding padding bytes as needed. The `align nocommons` option adds no padding to common blocks. In this case, unaligned data can occur unless the order of data items specified in the COMMON statement places the largest numeric data item first, followed by the next largest numeric data (and so on), followed by any character data.
`align dcommons`	Aligns all common block entities on natural boundaries up to 8 bytes, by adding padding bytes as needed. The `align nodcommons` option adds no padding to common blocks.
`align qcommons`	Aligns all common block entities on natural boundaries up to 16 bytes, by adding padding bytes as needed. The `align noqcommons` option adds no padding to common blocks.
`align zcommons`	Aligns all common block entities on natural boundaries up to 32 bytes, by adding padding bytes as needed. The `align nozcommons` option adds no padding to common blocks.
`align norecords`	Aligns components of derived types and fields within record structures on arbitrary byte boundaries with no padding.

Option	Description
	The align records option requests that multiple data items in record structures and derived-type structures without the SEQUENCE statement be naturally aligned, by adding padding as needed.
`align rec`*n*`byte`	Aligns components of derived types and fields within record structures on the smaller of the size boundary specified (*n*) or the boundary that will naturally align them. *n* can be 1, 2, 4, 8, 16, or 32. The default value for *n* is 8. When you specify this option, each structure member after the first is stored on either the size of the member type or *n*-byte boundaries, whichever is smaller. For example, to specify 16 bytes as the packing boundary (or alignment constraint) for all structures and unions in the file prog1.f, use the following command: `ifort {-align rec16byte | /align:rec16byte} prog1.f` This option does not affect whether common blocks are naturally aligned or packed.
`align sequence`	Aligns components of a derived type declared with the SEQUENCE statement (sequenced components) according to the alignment rules that are currently in use. The default alignment rules are to align unsequenced components on natural boundaries. The `align nosequence` option requests that sequenced components be packed regardless of any other alignment rules. Note that `align none` implies `align nosequence`. If you specify an option for standards checking, `align sequence` is ignored.
`align all`	Tells the compiler to add padding bytes whenever possible to obtain the natural alignment of data items in common blocks, derived types, and record structures. Specifies `align dcommons, align records, align nosequence`. This is the same as specifying `align` with no *keyword*.

IDE Equivalent

Visual Studio: **Data > Structure Member Alignment** (/align:recnbyte)

Data > Alignment of COMMON block entities (/align:[no]commons, /align:[no]dcommons, /align:[no]qcommons, /align:[no]zcommons)

Data > SEQUENCE Types Obey Alignment Rules (/align:[no]sequence)

Data > Default Array Alignment (/align:arraynbyte)

Eclipse: None

Xcode: **Data > Structure Member Alignment** (-align rec<1,2,4,8,16,32>byte)

Data > Alignment of COMMON block entities (-align [no]commons, -align [no]dcommons, -align [no]qcommons, -align [no]zcommons)

Data > SEQUENCE Types Obey Alignment Rules (-align [no]sequence)

Data > Default Array Alignment (-align array<8,16,32,64,128,256>byte)

Alternate Options

`align none`
Linux and macOS*: `-noalign`
Windows: `/noalign`

`align records`
Linux and macOS*: `-align rec16byte`, `-Zp16`

	Windows: `/align:rec16byte`, `/Zp16`
`align norecords`	Linux and macOS*: `-Zp1`, `-align rec1byte`
	Windows: `/Zp1`, `/align:rec1byte`
`align rec`*`n`*`byte`	Linux and macOS*: `-Zp{1|2|4|8|16}`
	Windows: `/Zp{1|2|4|8|16}`
`align all`	Linux and macOS*: `-align commons-align dcommons-align records-align nosequence`
	Windows: `/align:nocommons,dcommons,records,nosequence`

auto

Causes all local, non-SAVEd variables to be allocated to the run-time stack.

Syntax

Linux OS and macOS:

`-auto`

`-noauto`

Windows OS:

`/auto`

`/noauto`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`	Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL are allocated to the run-time stack. Note that the default changes to `auto` if one of the following options are specified: • `recursive` • `[q or Q]openmp`

Description

This option places local variables (scalars and arrays of all types), except those declared as SAVE, on the run-time stack. It is as if the variables were declared with the AUTOMATIC attribute.

It does not affect variables that have the SAVE attribute or ALLOCATABLE attribute, or variables that appear in an EQUIVALENCE statement or in a common block.

This option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly.

If you want to cause variables to be placed in static memory, specify option `[Q]save`. If you want only scalar variables of certain intrinsic types to be placed on the run-time stack, specify option `auto-scalar`.

NOTE
On Windows NT* systems, there is a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/[Q]auto`, because arrays are allocated on the stack along with scalars. However, with `/Qauto-scalar`, you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty. `/Qauto-scalar` enables the compiler to make better choices about which variables should be kept in registers during program execution.

IDE Equivalent

Visual Studio: **Data > Local Variable Storage**

Eclipse: None

Xcode: **Data > Local Variable Storage**

Alternate Options

`auto`	Linux and macOS*: None Windows: `/Qauto`, `/4Ya`
`noauto`	Linux and macOS*: `-save` Windows: `/Qsave`, `/4Na`

See Also

`auto-scalar, Qauto-scalar` compiler option
`save, Qsave` compiler option

auto-scalar, Qauto-scalar

Causes scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute to be allocated to the run-time stack.

Syntax

Linux OS and macOS:

`-auto-scalar`

Windows OS:

`/Qauto-scalar`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`

Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute are allocated to the run-time stack. Note that the default changes to `auto` if one of the following options are specified:

- `recursive`
- `[q or Q]openmp`

Description

This option causes allocation of scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL to the run-time stack. It is as if they were declared with the AUTOMATIC attribute.

It does not affect variables that have the SAVE attribute (which include initialized locals) or that appear in an EQUIVALENCE statement or in a common block.

This option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a SAVE statement.

You cannot specify option `save` or `auto` with this option.

> **NOTE**
> On Windows NT* systems, there is a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `[Q]auto` because arrays are allocated on the stack along with scalars. However, with `/Qauto-scalar`, you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty. `/Qauto-scalar` enables the compiler to make better choices about which variables should be kept in registers during program execution.

IDE Equivalent

Visual Studio: **Data > Local Variable Storage** (`/Qsave`, `/Qauto`, `/Qauto_scalar`)

Eclipse: None

Xcode: **Data > Local Variable Storage**

Alternate Options

None

See Also

`auto` compiler option

`save` compiler option

convert

Specifies the format of unformatted files containing numeric data.

Syntax

Linux OS and macOS:

`-convert` *keyword*

Windows OS:

`/convert:`*keyword*

Arguments

keyword Specifies the format for the unformatted numeric data. Possible values are:

`native` Specifies that unformatted data should not be converted.

`big_endian`	Specifies that the format will be big endian for integer data and big endian IEEE floating-point for real and complex data.
`cray`	Specifies that the format will be big endian for integer data and CRAY* floating-point for real and complex data.
`fdx` (Linux*, macOS*)	Specifies that the format will be little endian for integer data, and VAX processor floating-point format F_floating, D_floating, and IEEE binary128 for real and complex data.
`fgx` (Linux, macOS*)	Specifies that the format will be little endian for integer data, and VAX processor floating-point format F_floating, G_floating, and IEEE binary128 for real and complex data.
`ibm`	Specifies that the format will be big endian for integer data and IBM* System\370 floating-point format for real and complex data.
`little_endian`	Specifies that the format will be little endian for integer data and little endian IEEE floating-point for real and complex data.
`vaxd`	Specifies that the format will be little endian for integer data, and VAX* processor floating-point format F_floating, D_floating, and H_floating for real and complex data.
`vaxg`	Specifies that the format will be little endian for integer data, and VAX processor floating-point format F_floating, G_floating, and H_floating for real and complex data.

Default

`convert native` No conversion is performed on unformatted files containing numeric data.

Description

This option specifies the format of unformatted files containing numeric data.

Option	Description
`convert native`	Specifies that unformatted data should not be converted.
`convert big_endian`	Specifies that the format will be big endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and big endian IEEE floating-point for REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16, or COMPLEX*32.
`convert cray`	Specifies that the format will be big endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and CRAY* floating-point for REAL*8 or COMPLEX*16.
`convert fdx`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, D_floating for REAL*8 or COMPLEX*16, and IEEE binary128 for REAL*16 or COMPLEX*32.
`convert fgx`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, G_floating for REAL*8 or COMPLEX*16, and IEEE binary128 for REAL*16 or COMPLEX*32.

Option	Description
`convert ibm`	Specifies that the format will be big endian for INTEGER*1, INTEGER*2, or INTEGER*4, and IBM* System\370 floating-point format for REAL*4 or COMPLEX*8 (IBM short 4) and REAL*8 or COMPLEX*16 (IBM long 8).
`convert little_endian`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8 and little endian IEEE floating-point for REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16, or COMPLEX*32.
`convert vaxd`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, D_floating for REAL*8 or COMPLEX*16, and H_floating for REAL*16 or COMPLEX*32.
`convert vaxg`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, G_floating for REAL*8 or COMPLEX*16, and H_floating for REAL*16 or COMPLEX*32.

Non-native data conversion works on scalars and arrays of intrinsic types: INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8, including LOGICAL*1, LOGICAL*2, LOGICAL*4, LOGICAL*8, and REAL*4, REAL*8, REAL*16, including COMPLEX*8, COMPLEX*16, and COMPLEX*32. Conversion does not work inside defined type records on their individual fields. For example:

```
type REC
   integer(8) :: K
   real(8)    :: X
end type REC

type (REC) :: R

write (17) R%K, R%X    ! conversion will work on these objects
write (17) R           ! conversion will not work on the fields of this object
```

IDE Equivalent

Visual Studio: **Compatibility > Unformatted File Conversion**

Eclipse: None

Xcode: **Compatibility > Unformatted File Conversion**

Alternate Options

None

double-size

Specifies the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX declarations, constants, functions, and intrinsics.

Syntax

Linux OS and macOS:

`-double-size` *size*

Windows OS:

`/double-size:`*size*

Arguments

size	Specifies the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX declarations, constants, functions, and intrinsics. Possible values are: 64 (KIND=8) or 128 (KIND=16).

Default

64	DOUBLE PRECISION variables are defined as REAL*8 and DOUBLE COMPLEX variables are defined as COMPLEX*16.

Description

This option defines the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX declarations, constants, functions, and intrinsics.

Option	Description
`double-size 64`	Defines DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL(KIND=8) (REAL*8) and defines DOUBLE COMPLEX declarations, functions, and intrinsics as COMPLEX(KIND=8) (COMPLEX*16).
`double-size 128`	Defines DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL(KIND=16) (REAL*16) and defines DOUBLE COMPLEX declarations, functions, and intrinsics as COMPLEX(KIND=16) (COMPLEX*32).

The `real-size` option overrides the `double-size` option; for example, on Linux* systems, "-double-size 64 -real-size 128" acts like "-double-size 128 -real-size 128".

IDE Equivalent

Visual Studio: **Data > Default Double Precision KIND**

Eclipse: None

Xcode: **Data > Default Double Precision KIND**

Alternate Options

None

dyncom, Qdyncom

Enables dynamic allocation of common blocks at run time.

Syntax

Linux OS and macOS:

`-dyncom "`*`common1`*`,`*`common2`*`,..."`

Windows OS:

`/Qdyncom "`*`common1`*`,`*`common2`*`,..."`

Arguments

common1, *common2*, ...	Are the names of the common blocks to be dynamically allocated. The list of names must be within quotes.

Default

OFF Common blocks are not dynamically allocated at run time.

Description

This option enables dynamic allocation of the specified common blocks at run time. For example, to enable dynamic allocation of common blocks a, b, and c at run time, use this syntax:

```
/Qdyncom "a,b,c"    ! on Windows systems
-dyncom "a,b,c"     ! on Linux and macOS* systems
```

The following are some limitations that you should be aware of when using this option:

- An entity in a dynamic common cannot be initialized in a DATA statement.
- Only named common blocks can be designated as dynamic COMMON.
- An entity in a dynamic common block must not be used in an EQUIVALENCE expression with an entity in a static common block or a DATA-initialized variable.

NOTE
On macOS* systems, to successfully enable dynamic allocation of common blocks, you must specify the link-time option `-undefined dynamic_lookup` as well as option `-dyncom`.

IDE Equivalent

Visual Studio: **Data > Dynamic Common Blocks**

Eclipse: None

Xcode: **Data > Dynamic Common Blocks**

Alternate Options

None

See Also

Allocating Common Blocks
FOR__SET_FTN_ALLOC

falign-functions, Qfnalign

Tells the compiler to align procedures on an optimal byte boundary.

Syntax

Linux OS and macOS:

`-falign-functions[=n]`

`-fno-align-functions`

Windows OS:

`/Qfnalign[:n]`

`/Qfnalign-`

Arguments

n	Is an optional positive integer scalar initialization expression indicating the number of bytes for the minimum alignment boundary. It tells the compiler to align procedures on a power-of-2 byte boundary. If you do not specify *n*, the compiler aligns the start of procedures on 16-byte boundaries. The *n* must be a positive integer less than or equal to 4096. If you specify a value that is not a power of 2, *n* will be rounded up to the nearest power of 2. For example, if 23 is specified for *n*, procedures will be aligned on 32 byte boundaries.

Default

`-fno-align-functions` or `/Qfnalign-`	The compiler aligns procedures on 2-byte boundaries. This is the same as specifying `-falign-functions=2` (Linux* and macOS*) or `/Qfnalign:2` (Windows*).

Description

This option tells the compiler to align procedures on an optimal byte boundary. If you do not specify *n*, the compiler aligns the start of procedures on 16-byte boundaries.

IDE Equivalent

None

Alternate Options

None

falign-loops, Qalign-loops

Aligns loops to a power-of-two byte boundary.

Syntax

Linux OS and macOS:

`-falign-loops[=n]`

`-fno-align-loops`

Windows OS:

`/Qalign-loops[:n]`

`/Qalign-loops-`

Arguments

`n`	Is the optional number of bytes for the minimum alignment boundary. It must be a power of 2 between 1 and 4096, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on. If you specify 1 for `n`, no alignment is performed; this is the same as specifying the negative form of the option. If you do not specify `n`, the default alignment is 16 bytes.

Default

`-fno-align-loops` No special loop alignment is performed.

or
`/Qalign-loops-`

Description

This option aligns loops to a power-of-two boundary. This alignment may improve performance.

It can be affected by the directives CODE_ALIGN and ATTRIBUTES CODE_ALIGN.

If code is compiled with the `-falign-loops=m` (Linux* and macOS*) or `/Qalign-loops:m` (Windows*) option and a CODE_ALIGN:n directive precedes a loop, the loop is aligned on a MAX (m, n) byte boundary. If a procedure has the CODE_ALIGN:k attribute and a CODE_ALIGN:n directive precedes a loop, then both the procedure and the loop are aligned on a MAX (k, n) byte boundary.

IDE Equivalent

None

Alternate Options

None

Example

Consider the following code fragment in file `test_align_loops.f90`:

```
DO J = 1, N
…
END DO
```

Compiling `test_align_loops.f90` with the `-falign-loops` (Linux and macOS*) or `/Qalign-loops` (Windows) compiler option aligns the code that begins the DO J loop on a (default) 16-byte boundary. If you do not specify this compiler option, the alignment of the loop is implementation-dependent and may change from compilation to compilation.

See Also

falign-functions, Qfnalign compiler option
CODE_ALIGN directive
ATTRIBUTES CODE_ALIGN directive

falign-stack

Tells the compiler the stack alignment to use on entry to routines.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

`-falign-stack=`*mode*

macOS:

None

Windows OS:

None

Arguments

mode Is the method to use for stack alignment. Possible values are:

`assume-4-byte`	Tells the compiler to assume the stack is aligned on 4-byte boundaries. The compiler can dynamically adjust the stack to 16-byte alignment if needed.
`maintain-16-byte`	Tells the compiler to not assume any specific stack alignment, but attempt to maintain alignment in case the stack is already aligned. The compiler can dynamically align the stack if needed. This setting is compatible with gcc.
`assume-16-byte`	Tells the compiler to assume the stack is aligned on 16-byte boundaries and to continue to maintain 16-byte alignment. This setting is compatible with gcc.

Default

`-falign-stack=assume-16-byte` The compiler assumes the stack is aligned on 16-byte boundaries and continues to maintain 16-byte alignment.

Description

This option tells the compiler the stack alignment to use on entry to routines.

IDE Equivalent

None

Alternate Options

None

fcommon

Determines whether the compiler treats common symbols as global definitions.

Syntax

Linux OS:

`-fcommon`

`-fno-common`

macOS:

`-fcommon`

`-fno-common`

Windows OS:

None

Arguments

None

Default

`-fcommon` The compiler does not treat common symbols as global definitions.

Description

This option determines whether the compiler treats common symbols as global definitions and to allocate memory for each symbol at compile time.

Option `-fno-common` tells the compiler to treat common symbols as global definitions. When using this option, you can only have a common variable declared in one module; otherwise, a link time error will occur for multiple defined symbols.

On macOS*, if a library built with the `ar` utility contains objects with Fortran module data but no executable functions, the symbols corresponding to the module data may not be resolved when an object referencing them is linked against the library. You can work around this by compiling with option `-fno-common`. For more information, see the article titled: *ld: symbol(s) not found when linking library containing no executable functions*, which is located in https://software.intel.com/en-us/articles/ld-symbols-not-found-when-linking-library-containing-no-executable-functions

IDE Equivalent

None

Alternate Options

None

fkeep-static-consts, Qkeep-static-consts

Tells the compiler to preserve allocation of variables that are not referenced in the source.

Syntax

Linux OS:

```
-fkeep-static-consts
-fno-keep-static-consts
```

macOS:

```
-fkeep-static-consts
-fno-keep-static-consts
```

Windows OS:

```
/Qkeep-static-consts
/Qkeep-static-consts-
```

Arguments

None

Default

`-fno-keep-static-consts` or `/Qkeep-static-consts-`	If a variable is never referenced in a routine, the variable is discarded unless optimizations are disabled by option `-O0` (Linux* and macOS*) or `/Od` (Windows*).

Description

This option tells the compiler to preserve allocation of variables that are not referenced in the source.

The negated form can be useful when optimizations are enabled to reduce the memory usage of static data.

IDE Equivalent

None

Alternate Options

None

fmath-errno

Tells the compiler that errno can be reliably tested after calls to standard math library functions.

Syntax

Linux OS:

`-fmath-errno`

`-fno-math-errno`

macOS:

`-fmath-errno`

`-fno-math-errno`

Windows OS:

`None`

Arguments

None

Default

`-fno-math-errno`	The compiler assumes that the program does not test `errno` after calls to standard math library functions.

Description

This option tells the compiler to assume that the program tests `errno` after calls to math library functions. This restricts optimization because it causes the compiler to treat most math functions as having side effects.

Option `-fno-math-errno` tells the compiler to assume that the program does not test `errno` after calls to math library functions. This frequently allows the compiler to generate faster code. Floating-point code that relies on IEEE exceptions instead of `errno` to detect errors can safely use this option to improve performance.

IDE Equivalent

None

Alternate Options

None

fminshared

Specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.

Syntax

Linux OS and macOS:

`-fminshared`

Windows OS:

`None`

Arguments

None

Default

OFF Source files are compiled together to form a single object file.

Description

This option specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.

This option allows the compiler to optimize references to defined symbols without special visibility settings. To ensure that external and common symbol references are optimized, you need to specify visibility hidden or protected by using the `-fvisibility`, `-fvisibility-hidden`, or `-fvisibility-protected` option.

Also, the compiler does not need to generate position-independent code for the main program. It can use absolute addressing, which may reduce the size of the global offset table (GOT) and may reduce memory traffic.

IDE Equivalent

None

Alternate Options

None

See Also

`fvisibility` compiler option

fpconstant

Tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.

Syntax

Linux OS and macOS:

`-fpconstant`

`-nofpconstant`

Windows OS:

`/fpconstant`

`/nofpconstant`

Arguments

None

Default

`nofpconstant` Single-precision constants assigned to double-precision variables are evaluated in single precision according to Fortran 2003 Standard rules.

Description

This option tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.

This is extended precision. It does not comply with the Fortran 2003 standard, which requires that single-precision constants assigned to double-precision variables be evaluated in single precision.

It allows compatibility with FORTRAN 77, where such extended precision was allowed. If this option is not used, certain programs originally created for FORTRAN 77 compilers may show different floating-point results because they rely on the extended precision for single-precision constants assigned to double-precision variables.

IDE Equivalent

Visual Studio: **Floating-Point > Extend Precision of Single-Precision Constants**

Eclipse: None

Xcode: **Floating-Point > Extend Precision of Single-Precision Constants**

Alternate Options

None

Example

In the following example, if you specify `fpconstant`, identical values are assigned to D1 and D2. If you omit `fpconstant`, the compiler will obey the Fortran 2003 Standard and assign a less precise value to D1:

```
REAL (KIND=8) D1, D2
DATA D1 /2.71828182846182/ ! REAL (KIND=4) value expanded to double
DATA D2 /2.71828182846182D0/ ! Double value assigned to double
```

fpic

Determines whether the compiler generates position-independent code.

Syntax

Linux OS:

`-fpic`

`-fno-pic`

macOS:

`-fpic`

`-fno-pic`

Windows OS:

`None`

Arguments

None

Default

`-fno-pic`	The compiler does not generate position-independent code.

Description

This option determines whether the compiler generates position-independent code.

Option `-fpic` specifies full symbol preemption. Global symbol definitions as well as global symbol references get default (that is, preemptable) visibility unless explicitly specified otherwise.

Option `-fpic` must be used when building shared objects.

This option can also be specified as `-fPIC`.

IDE Equivalent

None

Alternate Options

None

fpie

Tells the compiler to generate position-independent code. The generated code can only be linked into executables.

Syntax

Linux OS:

`-fpie`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler does not generate position-independent code for an executable-only object.

Description

This option tells the compiler to generate position-independent code. It is similar to `-fpic`, but code generated by `-fpie` can only be linked into an executable.

Because the object is linked into an executable, this option causes better optimization of some symbol references.

To ensure that run-time libraries are set up properly for the executable, you should also specify option `-pie` to the compiler driver on the link command line.

Option `-fpie` can also be specified as `-fPIE`.

IDE Equivalent

None

Alternate Options

None

See Also

fpic
compiler option

pie
compiler option

fstack-protector

Enables or disables stack overflow security checks for certain (or all) routines.

Syntax

Linux OS:

`-fstack-protector[-keyword]`

`-fno-stack-protector[-keyword]`

macOS:

`-fstack-protector[-keyword]`

`-fno-stack-protector[-keyword]`

Windows OS:

`None`

Arguments

keyword	Possible values are:

`strong`	When option `-fstack-protector-strong` is specified, it enables stack overflow security checks for routines with any type of buffer.
`all`	When option `-fstack-protector-all` is specified, it enables stack overflow security checks for every routine.

If no `-keyword` is specified, option `-fstack-protector` enables stack overflow security checks for routines with a string buffer.

Default

`-fno-stack-protector`, `-fno-stack-protector-strong`	No stack overflow security checks are enabled for the relevant routines.
`-fno-stack-protector-all`	No stack overflow security checks are enabled for any routines.

Description

This option enables or disables stack overflow security checks for certain (or all) routines. A stack overflow occurs when a program stores more data in a variable on the execution stack than is allocated to the variable. Writing past the end of a string buffer or using an index for an array that is larger than the array bound could cause a stack overflow and security violations.

The `-fstack-protector` options are provided for compatibility with gcc. They use the gcc/glibc implementation when possible. If the gcc/glibc implementation is not available, they use the Intel implementation.

For an Intel-specific version of this feature, see option `-fstack-security-check`.

IDE Equivalent

None

Alternate Options

None

See Also

`fstack-security-check` compiler option

`GS` compiler option

fstack-security-check

Determines whether the compiler generates code that detects some buffer overruns.

Syntax

Linux OS:

`-fstack-security-check`

`-fno-stack-security-check`

macOS:

`-fstack-security-check`

`-fno-stack-security-check`

Windows OS:

None

Arguments

None

Default

`-fno-stack-security-check`	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.

This option always uses an Intel implementation.

For a gcc-compliant version of this feature, see option `fstack-protector`.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/GS`

See Also

`fstack-protector` compiler option

`GS` compiler option

fvisibility

Specifies the default visibility for global symbols or the visibility for symbols in a file.

Syntax

Linux OS and macOS:

`-fvisibility=`*`keyword`*

`-fvisibility-`*`keyword=filename`*

Windows OS:

None

Arguments

keyword	Specifies the visibility setting. Possible values are:
`default`	Sets visibility to default.
`extern`	Sets visibility to extern.
`hidden`	Sets visibility to hidden.

	`internal`	Sets visibility to internal.
	`protected`	Sets visibility to protected. This value is not available on macOS* systems.
filename		Is the pathname of a file containing the list of symbols whose visibility you want to set. The symbols must be separated by whitespace (spaces, tabs, or newlines).

Default

-fvisibility=default	The compiler sets visibility of symbols to default.

Description

This option specifies the default visibility for global symbols (syntax `-fvisibility=`*keyword*) or the visibility for symbols in a file (syntax `-fvisibility-`*keyword*`=`*filename*).

Visibility specified by `-fvisibility-`*keyword*`=`*filename* overrides visibility specified by `-fvisibility=`*keyword* for symbols specified in a file.

Option	Description
`-fvisibility=default` `-fvisibility-default=`*filename*	Sets visibility of symbols to default. This means other components can reference the symbol, and the symbol definition can be overridden (preempted) by a definition of the same name in another component.
`-fvisibility=extern` `-fvisibility-extern=`*filename*	Sets visibility of symbols to extern. This means the symbol is treated as though it is defined in another component. It also means that the symbol can be overridden by a definition of the same name in another component.
`-fvisibility=hidden` `-fvisibility-hidden=`*filename*	Sets visibility of symbols to hidden. This means that other components cannot directly reference the symbol. However, its address may be passed to other components indirectly.
`-fvisibility=internal` `-fvisibility-internal=`*filename*	Sets visibility of symbols to internal. This means that the symbol cannot be referenced outside its defining component, either directly or indirectly. The affected functions can never be called from another module, including through function pointers.
`-fvisibility=protected` `-fvisibility-protected=`*filename*	Sets visibility of symbols to protected. This means other components can reference the symbol, but it cannot be overridden by a definition of the same name in another component. This value is not available on macOS* systems.

If an `-fvisibility` option is specified more than once on the command line, the last specification takes precedence over any others.

If a symbol appears in more than one visibility *filename*, the setting with the least visibility takes precedence.

The following shows the precedence of the visibility settings (from greatest to least visibility):

- `extern`
- `default`
- `protected`
- `hidden`
- `internal`

Note that `extern` visibility only applies to functions. If a variable symbol is specified as `extern`, it is assumed to be `default`.

IDE Equivalent

None

Alternate Options

None

Example

A file named `prot.txt` contains symbols a, b, c, d, and e. Consider the following:

```
-fvisibility-protected=prot.txt
```

This option sets `protected` visibility for all the symbols in the file. It has the same effect as specifying `fvisibility=protected` in the declaration for each of the symbols.

fzero-initialized-in-bss, Qzero-initialized-in-bss

Determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

Syntax

Linux OS:

`-fzero-initialized-in-bss`

`-fno-zero-initialized-in-bss`

macOS:

`-fzero-initialized-in-bss`

`-fno-zero-initialized-in-bss`

Windows OS:

`/Qzero-initialized-in-bss`

`/Qzero-initialized-in-bss-`

Arguments

None

Default

`-fno-zero-initialized-in-bss` or `/Qzero-initialized-in-bss -`

Variables explicitly initialized with zeros are placed in the BSS section. This can save space in the resulting code.

Description

This option determines whether the compiler places in the DATA section any variables explicitly initialized with zeros.

If option `-fno-zero-initialized-in-bss` (Linux* and macOS*) or `/Qzero-initialized-in-bss-` (Windows*) is specified, the compiler places in the DATA section any variables that are initialized to zero.

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **Data > Place Zero-Initialized Variables in .bss**

Alternate Options

None

Gs

Lets you control the threshold at which the stack checking routine is called or not called.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/Gs[n]`

Arguments

n	Is the number of bytes that local variables and compiler temporaries can occupy before stack checking is activated. This is called the *threshold*.

Default

`/Gs`	Stack checking occurs for routines that require more than 4KB (4096 bytes) of stack space. This is also the default if you do not specify *n*.

Description

This option lets you control the threshold at which the stack checking routine is called or not called. If a routine's local stack allocation exceeds the threshold (*n*), the compiler inserts a __chkstk() call into the prologue of the routine.

IDE Equivalent

None

Alternate Options

None

GS

Determines whether the compiler generates code that detects some buffer overruns.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/GS[:*keyword*]

/GS-

Arguments

keyword Specifies the level of stack protection heuristics used by the compiler. Possible values are:

off	Tells the compiler to ignore buffer overruns. This is the same as specifying /GS-.
partial	Tells the compiler to provide a stack protection level that is compatible with Microsoft* Visual Studio 2008.
strong	Tells the compiler to provide full stack security level checking. This setting is compatible with more recent Microsoft* Visual Studio stack protection heuristics. This is the same as specifying /GS with no keyword.

Default

/GS- The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters.

This option has been added for Microsoft compatibility.

Following Visual Studio 2008, the Microsoft implementation of option /GS became more extensive (for example, more routines are protected). The performance of some programs may be impacted by the newer heuristics. In such cases, you may see better performance if you specify /GS:partial.

For more details about option /GS, see the Microsoft documentation.

IDE Equivalent

None

Alternate Options

Linux and macOS*: -fstack-security-check

Windows: None

See Also

`fstack-security-check` compiler option
`fstack-protector` compiler option

homeparams

Tells the compiler to store parameters passed in registers to the stack.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/homeparams`

Arguments

None

Default

OFF Register parameters are not written to the stack.

Description

This option tells the compiler to store parameters passed in registers to the stack.

IDE Equivalent

None

Alternate Options

None

init, Qinit

Lets you initialize a class of variables to zero or to various numeric exceptional values.

Syntax

Linux OS and macOS:

`-init=`*`keyword`* `[, `*`keyword`* `]`

Windows OS:

`/Qinit:`*`keyword`* `[, `*`keyword`* `]`

Arguments

`keyword` Specifies the initial value for a class of variables. Possible values are:

[no]arrays	Determines whether the compiler initializes variables that are arrays or scalars. Specifying arrays initializes variables that are arrays or scalars. Specifying noarrays or neither arrays or noarrays initializes only variables that are scalars. You must also specify at least one other keyword when you specify keyword noarrays.
huge or minus_huge	Determines both of the following: • whether the compiler initializes to the largest representable positive or negative real value all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated variables • whether the compiler initializes to the largest representable positive or negative integer value all uninitialized variables of intrinsic type INTEGER that are saved, local, automatic, or allocated variables
infinity or minus_infinity	Determines whether the compiler initializes to positive or negative Infinity all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated variables.
[no]snan	Determines whether the compiler initializes to signaling NaN all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated variables.
tiny or minus_tiny	Determines whether the compiler initializes to the smallest representable positive or negative real value all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated variables.
[no]zero	Determines whether the compiler initializes to zero all uninitialized variables of intrinsic type REAL, COMPLEX, INTEGER, or LOGICAL that are saved, local, automatic, or allocated variables. Option /Qinit:[no]zero replaces option /Qzero[-] (Windows*) , and option -init=[no]zero replaces option -[no]zero (Linux* and macOS*).

Default

OFF	No initializations are performed by default if you do not specify any of these options.

Description

This option lets you initialize a class of variables to zero or to one or more of the various numeric exceptional values positive or negative huge, positive or negative Infinity, signaling NaN, or positive or negative tiny.

If you only specify the keyword [minus_]huge, [minus_]infinity, snan, [minus_]tiny, or zero, option [Q]init affects only scalar variables. To apply an initialization to arrays as well, you must also specify the keyword arrays.

If you have specified an ALLOCATE statement and you specify one or more [Q]init keywords, the initializers are applied to the memory that has been allocated by the ALLOCATE statement.

Keywords are applied to the various numeric types in the following order:

- For REAL and COMPLEX variables, keywords [minus_]huge, [minus_]infinity, snan, [minus_]tiny, and zero initialize to the specified value.
- For INTEGER variables, keywords [minus_]huge and zero initialize to the specified value.
- For LOGICAL variables, keyword zero initializes to .FALSE..

The following classes of variables are initialized by the [Q]init option:

- Variables of intrinsic numeric type, that is, of type COMPLEX, INTEGER, LOGICAL, or REAL, of any KIND
- SAVEd scalar or array variables, not in the main program, that are not initialized in the source code
- Local scalars and arrays
- Module variables that are not initialized in the source code
- Automatic arrays
- Integers can be set to `zero`, `huge`, or `minus_huge`.
- In a COMPLEX variable, each of the real and imaginary parts is separately initialized as a REAL.

The following are general restrictions for this option:

- The keywords `[minus_]infinity`, `snan`, and `[minus_]tiny` only affect certain variables of REAL or COMPLEX type.
- You cannot initialize variables in equivalence groups to any of the numeric exceptional values.
- In an equivalence group, if no member of that equivalence group has an explicit initialization or a default initialization (in the case of a derived type), a variable in that equivalence group can be initialized to zero but not to any of the numeric exceptional values.
- Derived types, arrays of derived types, and their components will not be initialized.
- Dummy arguments including adjustable arrays will not be initialized.
- Variables in COMMON will not be initialized.

The following spellings all cause the same behavior, that is, they initialize certain numeric arrays and scalars to zero:

- `[Q]init zero [Q]init arrays`
- `[Q]init arrays [Q]init zero`
- `[Q]init zero, arrays`
- `[Q]init arrays, zero`

Combinations of keywords will override each other in a left-to-right fashion as follows:

- `zero` and `nozero` override each other.
- `snan` and `nosnan` override each other.
- `huge` and `minus_huge` override each other.
- `tiny` and `minus_tiny` override each other.
- `infinity`, `minus_infinity`, and `snan` will all override each other.

Because a REAL or COMPLEX variable can be initialized to `huge` or `minus_huge`, `infinity` or `minus_infinity`, `tiny` or `minus_tiny`, `snan`, or `zero`, these initializations are applied in the following order:

1. `snan`
2. `infinity` or `minus_infinity`
3. `tiny` or `minus_tiny`
4. `huge` or `minus_huge`
5. `zero`

Because an INTEGER variable can be initialized to `huge` or `minus_huge`, or `zero`, these initializations are applied in the following order:

1. `huge` or `minus_huge`
2. `zero`

For example, if the you specify [Q]init zero, minus_huge, snan when compiling the following program:

```
program test
real X
integer K
complex C
end
```

The variable X will be initialized with a signaling NaN (`snan`), variable K will be initialized to the integer value `minus_huge`, and the real and imaginary parts of variable C will be initialized to a signaling NaN (`snan`) in each.

If you specify `[Q]init snan`, the floating-point exception handling flags will be set to trap signaling NaN and halt so that when such a value is trapped at run-time, the Fortran library can catch the usage, display an error message about a possible uninitialized variable, display a traceback, and stop execution. You can use the debugger to determine where in your code this uninitialized variable is being referenced when execution stops.

Setting the option `[Q]init snan` implicitly sets the option `fpe 0`. A compile time warning will occur if you specify both option `fpe 3` and option `[Q]init snan` on the command line. In this case, `fpe 3` is ignored.

NOTE
If you build with optimization, the compiler may speculate floating-point operations, assuming the default floating-point environment in which floating-point exceptions are masked. When you add `[Q]init snan`, this speculation may result in exceptions, unrelated to uninitialized variables, that now get trapped. To avoid this, reduce the optimization level to `/O1` or `/Od` (Windows*), or `-O1` or `-O0` (Linux* and macOS*) when doing uninitialized variable detection.

If you wish to maintain optimization, you should add option `[Q]fp-speculation safe` to disable speculation when there is a possibility that the speculation may result in a floating-point exception.

NOTE
To avoid possible performance issues, you should only use `[Q]init` for debugging (for example, `[Q]init zero`) and checking for uninitialized variables (for example, `[Q]init snan`).

Use option `[Q]save` if you wish all variables to be specifically marked as SAVE.

IDE Equivalent

Visual Studio: **Data > Initialize Variables to Signaling NaN**

Data > Initialize Variables to Zero

Data > Initialize Arrays as well as Scalars

Eclipse: None

Xcode: **Data > Initialize Variables to Signaling NaN**

Data > Initialize Variables to Zero

Data > Initialize Arrays as well as Scalars

Alternate Options

None

Example

The following example shows how to initialize scalars of intrinsic type REAL and COMPLEX to signaling NaN, and scalars of intrinsic type INTEGER and LOGICAL to zero:

```
-init=snan,zero               ! Linux and macOS* systems
/Qinit:snan,zero              ! Windows systems
```

The following example shows how to initialize scalars and arrays of intrinsic type REAL and COMPLEX to signaling NaN, and scalars and arrays of intrinsic type INTEGER and LOGICAL to zero:

```
-init=zero -init=snan -init=arrays       ! Linux and macOS* systems
/Qinit:zero /Qinit:snan /Qinit:arrays    ! Windows systems
```

To see an example of how to use option `[Q]init` for detection of uninitialized floating-point variables at run-time, see the article titled *Detection of Uninitialized Floating-point Variables in Intel® Fortran*, which is located in https://software.intel.com/articles/detection-of-uninitialized-floating-point-variables-in-intel-fortran

See Also

`O` compiler option

`fp-speculation, Qfp-speculation` compiler option

Explicit-Shape Specifications for details on automatic arrays and adjustable arrays

HUGE

TINY

Data Representation Models

intconstant

Tells the compiler to use FORTRAN 77 semantics to determine the kind parameter for integer constants.

Syntax

Linux OS and macOS:

`-intconstant`

`-nointconstant`

Windows OS:

`/intconstant`

`/nointconstant`

Arguments

None

Default

`nointconstant` The compiler uses the Fortran default INTEGER type.

Description

This option tells the compiler to use FORTRAN 77 semantics to determine the kind parameter for integer constants.

With FORTRAN 77 semantics, the kind is determined by the value of the constant. All constants are kept internally by the compiler in the highest precision possible. For example, if you specify option `intconstant`, the compiler stores an integer constant of 14 internally as INTEGER(KIND=8) and converts the constant upon reference to the corresponding proper size. Fortran specifies that integer constants with no explicit KIND are kept internally in the default INTEGER kind (KIND=4 by default).

Note that the internal precision for floating-point constants is controlled by option fpconstant.

IDE Equivalent

Visual Studio: **Compatibility > Use F77 Integer Constants**

Eclipse: None

Xcode: **Compatibility > Use F77 Integer Constants**

Alternate Options

None

integer-size

Specifies the default KIND for integer and logical variables.

Syntax

Linux OS and macOS:

`-integer-size` *size*

Windows OS:

`/integer-size:`*size*

Arguments

size	Is the size for integer and logical variables. Possible values are: 16, 32, or 64.

Default

`integer-size 32`	Integer and logical variables are 4 bytes long (INTEGER(KIND=4) and LOGICAL(KIND=4)).

Description

This option specifies the default size (in bits) for integer and logical variables.

Option	Description
`integer-size 16`	Makes default integer and logical declarations, constants, functions, and intrinsics 2 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=2). Integer and logical constants of unspecified KIND are evaluated in INTEGER (KIND=2) and LOGICAL(KIND=2)respectively.
`integer-size 32`	Makes default integer and logical declarations, constants, functions, and intrinsics 4 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=4). Integer and logical constants of unspecified KIND are evaluated in INTEGER (KIND=4) and LOGICAL(KIND=4)respectively.
`integer-size 64`	Makes default integer and logical declarations, constants, functions, and intrinsics 8 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=8). Integer and logical constants of unspecified KIND are evaluated in INTEGER (KIND=8) and LOGICAL(KIND=8)respectively.

IDE Equivalent

Visual Studio: **Data > Default Integer KIND**

Eclipse: None

Xcode: **Data > Default Integer KIND**

Alternate Options

`integer-size 16`
Linux and macOS*: `-i2`
Windows: `/4I2`

`integer-size 32`
Linux and macOS*: `-i4`
Windows: `/4I4`

`integer-size 64`
Linux and macOS*: `-i8`
Windows: `/4I8`

mcmodel

Tells the compiler to use a specific memory model to generate code and store data.

Architecture Restrictions

Only available on Intel® 64 architecture

Syntax

Linux OS:

`-mcmodel=`*`mem_model`*

macOS:

`None`

Windows OS:

`None`

Arguments

mem_model
Is the memory model to use. Possible values are:

`small`	Tells the compiler to restrict code and data to the first 2GB of address space. All accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.
`medium`	Tells the compiler to restrict code to the first 2GB; it places no memory restriction on data. Accesses of code can be done with IP-relative addressing, but accesses of data must be done with absolute addressing.
`large`	Places no memory restriction on code or data. All accesses of code and data must be done with absolute addressing.

Default

`-mcmodel=small`
On systems using Intel® 64 architecture, the compiler restricts code and data to the first 2GB of address space. Instruction Pointer (IP)-relative addressing can be used to access code and data.

Description

This option tells the compiler to use a specific memory model to generate code and store data. It can affect code size and performance. If your program has COMMON blocks and local data with a total size smaller than 2GB, `-mcmodel=small` is sufficient. COMMONs larger than 2GB require`-mcmodel=medium` or `-mcmodel=large`. Allocation of memory larger than 2GB can be done with any setting of `-mcmodel`.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. IP-relative addressing is somewhat faster. So, the `small` memory model has the least impact on performance.

> **NOTE**
> When you specify option `-mcmodel=medium` or `-mcmodel=large`, it sets option `-shared-intel`. This ensures that the correct dynamic versions of the Intel run-time libraries are used.
>
> If you specify option `-static-intel` while `-mcmodel=medium` or `-mcmodel=large` is set, an error will be displayed.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to compile using `-mcmodel`:

```
ifort -shared-intel -mcmodel=medium -o  prog prog.f
```

See Also

`shared-intel` compiler option
`fpic` compiler option

mdynamic-no-pic

Generates code that is not position-independent but has position-independent external references.

Syntax

Linux OS:

`None`

macOS:

`-mdynamic-no-pic`

Windows OS:

`None`

Arguments

None

Default

OFF All references are generated as position independent.

Description

This option generates code that is not position-independent but has position-independent external references.

The generated code is suitable for building executables, but it is not suitable for building shared libraries.

This option may reduce code size and produce more efficient code. It overrides the `-fpic` compiler option.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic` compiler option

no-bss-init, Qnobss-init

Tells the compiler to place in the DATA section any uninitialized variables and explicitly zero-initialized variables.

Syntax

Linux OS and macOS:

`-no-bss-init`

Windows OS:

`/Qnobss-init`

Arguments

None

Default

OFF Uninitialized variables and explicitly zero-initialized variables are placed in the BSS section.

Description

This option tells the compiler to place in the DATA section any uninitialized variables and explicitly zero-initialized variables.

IDE Equivalent

None

Alternate Options

None

Qsfalign

Specifies stack alignment for functions.

Architecture Restrictions

Only available on IA-32 architecture

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/Qsfalign[*n*]

Arguments

n Is the byte size of aligned variables. Possible values are:

8	Specifies that alignment should occur for functions with 8-byte aligned variables. At this setting the compiler aligns the stack to 16 bytes if there is any 16-byte or 8-byte data on the stack. For 8-byte data, the compiler only aligns the stack if the alignment will produce a performance advantage.
16	Specifies that alignment should occur for functions with 16-byte aligned variables. At this setting, the compiler only aligns the stack for 16-byte data. No attempt is made to align for 8-byte data.

Default

/Qsfalign8 Alignment occurs for functions with 8-byte aligned variables.

Description

This option specifies stack alignment for functions. It lets you disable the normal optimization that aligns a stack for 8-byte data.

If you do not specify *n*, stack alignment occurs for all functions. If you specify /Qsfalign-, no stack alignment occurs for any function.

IDE Equivalent

None

Alternate Options

None

real-size

Specifies the default KIND for real and complex declarations, constants, functions, and intrinsics.

Syntax

Linux OS and macOS:

`-real-size size`

Windows OS:

`/real-size:size`

Arguments

size	Is the size for real and complex declarations, constants, functions, and intrinsics. Possible values are: 32, 64, or 128.

Default

`real-size 32`	Default real and complex declarations, constants, functions, and intrinsics are 4 bytes long (REAL(KIND=4) and COMPLEX(KIND=4)).

Description

This option specifies the default size (in bits) for real and complex declarations, constants, functions, and intrinsics.

Option	Description
`real-size 32`	Makes default real and complex declarations, constants, functions, and intrinsics 4 bytes long. REAL declarations are treated as single precision REAL (REAL(KIND=4)) and COMPLEX declarations are treated as COMPLEX (COMPLEX(KIND=4)). Real and complex constants of unspecified KIND are evaluated in single precision (KIND=4).
`real-size 64`	Makes default real and complex declarations, constants, functions, and intrinsics 8 bytes long. REAL declarations are treated as DOUBLE PRECISION (REAL(KIND=8)) and COMPLEX declarations are treated as DOUBLE COMPLEX (COMPLEX(KIND=8)). Real and complex constants of unspecified KIND are evaluated in double precision (KIND=8).
`real-size 128`	Makes default real and complex declarations, constants, functions, and intrinsics 16 bytes long. REAL declarations are treated as extended precision REAL (REAL(KIND=16)); COMPLEX and DOUBLE COMPLEX declarations are treated as extended precision COMPLEX (COMPLEX(KIND=16)). Real and complex constants of unspecified KIND are evaluated in extended precision (KIND=16).

These compiler options can affect the result type of intrinsic procedures, such as CMPLX, FLOAT, REAL, SNGL, and AIMAG, which normally produce single-precision REAL or COMPLEX results. To prevent this effect, you must explicitly declare the kind type for arguments of such intrinsic procedures.

For example, if `real-size 64` is specified, the CMPLX intrinsic will produce a result of type DOUBLE COMPLEX (COMPLEX(KIND=8)). To prevent this, you must explicitly declare any real argument to be REAL(KIND=4), and any complex argument to be COMPLEX(KIND=4).

IDE Equivalent

Visual Studio: **Data > Default Real KIND**

Eclipse: None

Xcode: **Data > Default Real KIND**

Alternate Options

`real-size 64`
Linux and macOS*: `-r8`, `-autodouble`
Windows: `/4R8`, `/Qautodouble`

`real-size 128`
Linux and macOS*: `-r16`
Windows: `/4R16`

save, Qsave

Causes variables to be placed in static memory.

Syntax

Linux OS and macOS:

`-save`

Windows OS:

`/Qsave`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`
Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL are allocated to the run-time stack. Note that the default changes to `auto` if one of the following options are specified:

- `recursive`
- `[q or Q]openmp`

Description

This option saves all variables in static allocation except local variables within a recursive routine and variables declared as AUTOMATIC.

If you want all local, non-SAVEd variables to be allocated to the run-time stack, specify option `automatic`.

IDE Equivalent

Visual Studio: **Data > Local Variable Storage**

Eclipse: None

Xcode: **Data > Local Variable Storage**

Alternate Options

Linux and macOS*: `-noauto`

Windows: `/noauto`, `/4Na`

See Also

`auto` compiler option
`auto_scalar` compiler option

zero, Qzero

Initializes to zero variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are not yet initialized. This is a deprecated option. The replacement option is /Qinit:[no]zero or -init=[no]zero.

Syntax

Linux OS and macOS:

`-zero`

`-nozero`

Windows OS:

`/Qzero`

`/Qzero-`

Arguments

None

Default

`-nozero` or `/Qzero-` Variables are not initialized to zero.

Description

This option initializes to zero variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are not yet initialized.

Use option `[Q]save` on the command line to make all local variables specifically marked as SAVE.

IDE Equivalent

None

Alternate Options

None

See Also

`init, Qinit` compiler option (see setting zero)

`save` compiler option

Compiler Diagnostic Options

diag, Qdiag

Controls the display of diagnostic information during compilation.

Syntax

Linux OS and macOS:

`-diag-`*`type=diag-list`*

Windows OS:

`/Qdiag-`*`type`*`:`*`diag-list`*

Arguments

type

Is an action to perform on diagnostics. Possible values are:

`enable`	Enables a diagnostic message or a group of messages. If you specify `-diag-enable=all` (Linux* and macOS*) or `/Qdiag-enable:all` (Windows*), all diagnostic messages shown in *diag-list* are enabled.
`disable`	Disables a diagnostic message or a group of messages. If you specify `-diag-disable=all` (Linux* and macOS*) or `/Qdiag-disable:all` (Windows*), all diagnostic messages shown in *diag-list* are disabled.
`error`	Tells the compiler to change diagnostics to errors.
`warning`	Tells the compiler to change diagnostics to warnings.
`remark`	Tells the compiler to change diagnostics to remarks (comments).

diag-list

Is a diagnostic group or ID value. Possible values are:

`driver`	Specifies diagnostic messages issued by the compiler driver.
`vec`	Specifies diagnostic messages issued by the vectorizer.
`par`	Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer).
`openmp`	Specifies diagnostic messages issued by the OpenMP* parallelizer.
`warn`	Specifies diagnostic messages that have a "warning" severity level.
`error`	Specifies diagnostic messages that have an "error" severity level.
`remark`	Specifies diagnostic messages that are remarks or comments.
`cpu-dispatch`	Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default.

`id[,id,...]`	Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each `id`.
`tag[,tag,...]`	Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each `tag`.

The diagnostic messages generated can be affected by certain options, such as `[Q]x`, `/arch` (Windows) or `-m` (Linux and macOS*).

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option controls the display of diagnostic information during compilation. Diagnostic messages are output to stderr unless the `[Q]diag-file` option is specified.

NOTE
The [Q]diag options do not control diagnostics emitted at run-time. For more information about run-time errors and diagnostics, see Handling Run-Time Errors.

To control the diagnostic information reported by the vectorizer, use options `[q or Q]opt-report` and `[q or Q]opt-report-phase`, phase `vec`.

To control the diagnostic information reported by the auto-parallelizer, use options `[q or Q]opt-report` and `[q or Q]opt-report-phase`, phase `par`.

IDE Equivalent

Visual Studio: **Diagnostics > Disable Specific Diagnostics**

Eclipse: None

Xcode: **Diagnostics > Disable Specific Diagnostics**

Alternate Options

enable vec	Linux and macOS*: `-qopt-report`; `-qopt-report -qopt-report-phase=vec` Windows: `/Qopt-report`; `/Qopt-report /Qopt-report-phase:vec`
disable vec	Linux and macOS*: `-qopt-report=0 -qopt-report-phase=vec` Windows: `/Qopt-report:0 /Qopt-report-phase:vec`
enable par	Linux and macOS*: `-qopt-report`; `-qopt-report -qopt-report-phase=par`

	Windows: `/Qopt-report`; `/Qopt-report /Qopt-report-phase:par`
disable par	Linux and macOS*: `-qopt-report=0 -qopt-report-phase=par` Windows: `/Qopt-report:0 /Qopt-report-phase:par`

Example

The following example shows how to enable diagnostic IDs 117, 230 and 450:

```
-diag-enable=117,230,450     ! Linux and macOS* systems
/Qdiag-enable:117,230,450    ! Windows systems
```

The following example shows how to change vectorizer diagnostic messages to warnings:

```
-diag-enable=vec -diag-warning=vec      ! Linux and macOS* systems
/Qdiag-enable:vec /Qdiag-warning:vec    ! Windows systems
```

Note that you need to enable the vectorizer diagnostics before you can change them to warnings.

The following example shows how to disable all auto-parallelizer diagnostic messages:

```
-diag-disable=par     ! Linux and macOS* systems
/Qdiag-disable:par    ! Windows systems
```

The following example shows how to change all diagnostic warnings and remarks to errors:

```
-diag-error=warn,remark    ! Linux and macOS* systems
/Qdiag-error:warn,remark   ! Windows systems
```

The following example shows how to get a list of *only* vectorization diagnostics:

```
-diag-dump -diag-disable=all -diag-enable=vec        ! Linux and macOS* systems
/Qdiag-dump /Qdiag-disable:all /Qdiag-enable:vec     ! Windows systems
```

See Also

`diag-dump, Qdiag-dump` compiler option
`diag-id-numbers, Qdiag-id-numbers` compiler option
`diag-file, Qdiag-file` compiler option
`qopt-report, Qopt-report` compiler option
`x, Qx` compiler option

diag-dump, Qdiag-dump

Tells the compiler to print all enabled diagnostic messages.

Syntax

Linux OS and macOS:

`-diag-dump`

Windows OS:

`/Qdiag-dump`

Arguments

None

Default

OFF The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to print all enabled diagnostic messages. The diagnostic messages are output to `stdout`.

This option prints the enabled diagnostics from all possible diagnostics that the compiler can issue, including any default diagnostics.

If *diag-list* is specified for the `[Q]diag-enable` option, the print out will include the *diag-list* diagnostics.

IDE Equivalent

None

Alternate Options

None

Example

The following example adds vectorizer diagnostic messages to the printout of default diagnostics:

```
-diag-enable vec -diag-dump          ! Linux and macOS* systems
/Qdiag-enable:vec /Qdiag-dump        ! Windows systems
```

See Also

`diag, Qdiag` compiler option

diag-error-limit, Qdiag-error-limit

Specifies the maximum number of errors allowed before compilation stops.

Syntax

Linux OS and macOS:

`-diag-error-limit=`*n*

`-no-diag-error-limit`

Windows OS:

`/Qdiag-error-limit:`*n*

`/Qdiag-error-limit-`

Arguments

n	Is the maximum number of error-level or fatal-level compiler errors allowed.

Default

30	A maximum of 30 error-level and fatal-level messages are allowed.

Description

This option specifies the maximum number of errors allowed before compilation stops. It indicates the maximum number of error-level or fatal-level compiler errors allowed for a file specified on the command line.

If you specify the negative form of the `[Q]diag-error-limit` option on the command line, there is no limit on the number of errors that are allowed.

If the maximum number of errors is reached, a warning message is issued and the next file (if any) on the command line is compiled.

IDE Equivalent

Visual Studio: **Diagnostics > Error Limit**

Eclipse: None

Xcode: **Diagnostics > Error Limit**

Alternate Options

Linux and macOS*: `-error-limit` and `-noerror-limit`

Windows: `/error-limit` and `/noerror-limit`

diag-file, Qdiag-file

Causes the results of diagnostic analysis to be output to a file.

Syntax

Linux OS:

`-diag-file[=`*`filename`*`]`

macOS:

`None`

Windows OS:

`/Qdiag-file[:`*`filename`*`]`

Arguments

filename	Is the name of the file for output.

Default

OFF	Diagnostic messages are output to stderr.

Description

This option causes the results of diagnostic analysis to be output to a file. The file is placed in the current working directory.

You can include a file extension in *filename*. For example, if *file.txt* is specified, the name of the output file is *file.txt*. If you do not provide a file extension, the name of the file is *filename.diag*.

If *filename* is not specified, the name of the file is *name-of-the-first-source-file.diag*. This is also the name of the file if the name specified for file conflicts with a source file name provided in the command line.

NOTE
If you specify the `[Q]diag-file` option and you also specify the `[Q]diag-file-append` option, the last option specified on the command line takes precedence.

IDE Equivalent

Visual Studio: **Diagnostics > Diagnostics File**

Eclipse: None

Xcode: **Diagnostics > Diagnostics File**

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be output to a file named *my_diagnostics.diag*:

```
-diag-file=my_diagnostics       ! Linux systems
/Qdiag-file:my_diagnostics      ! Windows systems
```

See Also

`diag-file-append, Qdiag-file-append` compiler option

diag-file-append, Qdiag-file-append

Causes the results of diagnostic analysis to be appended to a file.

Syntax

Linux OS:

`-diag-file-append[=`*filename*`]`

macOS:

`None`

Windows OS:

`/Qdiag-file-append[:`*filename*`]`

Arguments

filename Is the name of the file to be appended to. It can include a path.

Default

OFF Diagnostic messages are output to `stderr`.

Description

This option causes the results of diagnostic analysis to be appended to a file. If you do not specify a path, the driver will look for *filename* in the current working directory.

If *filename* is not found, then a new file with that name is created in the current working directory. If the name specified for file conflicts with a source file name provided in the command line, the name of the file is *name-of-the-first-source-file.diag*.

NOTE
If you specify the `[Q]diag-file-append` option and you also specify the `[Q]diag-file` option, the last option specified on the command line takes precedence.

IDE Equivalent

None

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be appended to a file named *my_diagnostics.txt*:

```
-diag-file-append=my_diagnostics.txt       ! Linux systems
/Qdiag-file-append:my_diagnostics.txt      ! Windows systems
```

See Also

`diag-file, Qdiag-file` compiler option

diag-id-numbers, Qdiag-id-numbers

Determines whether the compiler displays diagnostic messages by using their ID number values.

Syntax

Linux OS and macOS:

`-diag-id-numbers`

`-no-diag-id-numbers`

Windows OS:

`/Qdiag-id-numbers`

`/Qdiag-id-numbers-`

Arguments

None

Default

`-diag-id-numbers` or `/Qdiag-id-numbers`	The compiler displays diagnostic messages by using their ID number values.

Description

This option determines whether the compiler displays diagnostic messages by using their ID number values. If you specify the negative form of the `[Q]diag-id-numbers` option, mnemonic names are output for driver diagnostics only.

IDE Equivalent

None

Alternate Options

None

See Also

`diag, Qdiag` compiler option

diag-once, Qdiag-once

Tells the compiler to issue one or more diagnostic messages only once.

Syntax

Linux OS and macOS:

`-diag-once`*`id`*`[,`*`id`*`,...]`

Windows OS:

`/Qdiag-once:`*`id`*`[,`*`id`*`,...]`

Arguments

id	Is the ID number of the diagnostic message. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each *id*.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to issue one or more diagnostic messages only once.

IDE Equivalent

None

Alternate Options

None

gen-interfaces

Tells the compiler to generate an interface block for each routine in a source file.

Syntax

Linux OS and macOS:

`-gen-interfaces [[`*`no`*`]`*`source`*`]`

`-nogen-interfaces`

Windows OS:

`/gen-interfaces[:[`*`no`*`]`*`source`*`]`

`/nogen-interfaces`

Arguments

None

Default

`nogen-interfaces` The compiler does not generate interface blocks for routines in a source file.

Description

This option tells the compiler to generate an interface block for each routine (that is, for each SUBROUTINE and FUNCTION statement) defined in the source file. The compiler generates two files for each routine, a .mod file and a .f90 file, and places them in the current directory or in the directory specified by the include (`-I`) or `-module` option. The .f90 file is the text of the interface block; the .mod file is the interface block compiled into binary form. The .f90 file is for reference only and may not completely represent the generated interface used by the compiler.

If `source` is specified, the compiler creates the *procedure-name*__GENmod.f90 as well as the *procedure-name*__GENmod.mod files. If `nosource` is specified, the compiler creates the *procedure-name*__GENmod.mod but not the *procedure-name*__GENmod.f90 files. If neither is specified, it is the same as specifying setting `source` for the `gen-interfaces` option.

On Windows* systems, for a Debug configuration in a Visual Studio project, the default is /warn:interfaces.

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **Diagnostics > Generate Interface Blocks**

Alternate Options

None

traceback

Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

Syntax

Linux OS and macOS:

`-traceback`

`-notraceback`

Windows OS:

`/traceback`

`/notraceback`

Arguments

None

Default

`notraceback` No extra information is generated in the object file to produce traceback information.

Description

This option tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

When the severe error occurs, source file, routine name, and line number correlation information is displayed along with call stack hexadecimal addresses (program counter trace).

Note that when a severe error occurs, advanced users can also locate the cause of the error using a map file and the hexadecimal addresses of the stack displayed when the error occurs.

This option increases the size of the executable program, but has no impact on run-time execution speeds.

It functions independently of the debug option.

On Windows* systems, `traceback` sets the `/Oy-` option, which forces the compiler to use EBP as the stack frame pointer.

On Windows* systems, the linker places the traceback information in the executable image, in a section named ".trace". To see which sections are in an image, use the command:

```
link -dump -summary your_app_name.exe
```

To see more detailed information, use the command:

```
link -dump -headers your_app_name.exe
```

On Windows* systems, when requesting traceback, you must set Enable Incremental Linking in the VS .NET* IDE Linker Options to No. You must also set Omit Frame Pointers (the `/Oy` option) in the Optimization Options to "No."

On Linux* systems, to display the section headers in the image (including the header for the .trace section, if any), use the command:

```
objdump -h your_app_name.exe
```

On macOS* systems, to display the section headers in the image, use the command:

```
otool -l your_app_name.exe
```

IDE Equivalent

Visual Studio: **Run-time > Generate Traceback Information**

Eclipse: None

Xcode: **Run-time > Generate Traceback Information**

Alternate Options

None

warn

Specifies diagnostic messages to be issued by the compiler.

Syntax

Linux OS and macOS:

`-warn [keyword[, keyword...]]`

`-nowarn`

Windows OS:

`/warn[:keyword[, keyword...]]`

`/nowarn`

Arguments

keyword Specifies the diagnostic messages to be issued. Possible values are:

`none`	Disables all warning messages.
`[no]alignments`	Determines whether warnings occur for data that is not naturally aligned.
`[no]declarations`	Determines whether warnings occur for any undeclared names.
`[no]errors`	Determines whether warnings are changed to errors.
`[no]externals`	Determines whether warnings occur for any dummy procedures or procedure calls that have no explicit interface or have not been declared external.
`[no]general`	Determines whether warning messages and informational messages are issued by the compiler.
`[no]ignore_loc`	Determines whether warnings occur when %LOC is stripped from an actual argument.
`[no]interfaces`	Determines whether the compiler checks the interfaces of all SUBROUTINEs called and FUNCTIONs invoked in your compilation against an external set of interface blocks.
`[no]shape`	Determines whether array conformance violations are diagnosed with errors or warnings when used with the `check shape` option.
`[no]stderrors`	Determines whether warnings about Fortran standard violations are changed to errors.
`[no]truncated_source`	Determines whether warnings occur when source exceeds the maximum column width in fixed-format files.
`[no]uncalled`	Determines whether warnings occur when a statement function is never called
`[no]unused`	Determines whether warnings occur for declared variables that are never used.
`[no]usage`	Determines whether warnings occur for questionable programming practices.
`all`	Enables all warning messages except errors and stderrors.

Default

`alignments`	Warnings are issued about data that is not naturally aligned.
`general`	All information-level and warning-level messages are enabled.
`nodeclarations`	No warnings are issued for undeclared names.
`noerrors`	Warning-level messages are not changed to error-level messages.

`noexternals`	No warnings are issued when a dummy procedure or external procedure does not have an explicit interface and has not be declared external.
`noignore_loc`	No warnings are issued when %LOC is stripped from an argument.
`nointerfaces`	The compiler does not check interfaces of SUBROUTINEs called and FUNCTIONs invoked in your compilation against an external set of interface blocks.
`noshape`	Array conformance violations are issued as errors if the `check shape` option is specified.
`nostderrors`	Warning-level messages about Fortran standards violations are not changed to error-level messages.
`notruncated_source`	No warnings are issued when source exceeds the maximum column width in fixed-format files.
`nouncalled`	No warnings are issued when a statement function is not called.
`nounused`	No warnings are issued for variables that are declared but never used.
`usage`	Warnings are issued for questionable programming practices.

Description

This option specifies the diagnostic messages to be issued by the compiler.

Option	Description
`warn none`	Disables all warning messages. This is the same as specifying `nowarn`.
`warn noalignments`	Disables warnings about data that is not naturally aligned.
`warn declarations`	Enables warnings about any undeclared names. The compiler will use the default implicit data typing rules for such undeclared names. The IMPLICIT and IMPLICIT NONE statements override this option.
`warn errors`	Tells the compiler to change all warning-level messages to error-level messages; this includes warnings about Fortran standards violations.
`warn externals`	Enables warnings about dummy procedures and external procedures that do not have explicit interfaces and have not been declared with the EXTERNAL attribute.
`warn nogeneral`	Disables all informational-level and warning-level diagnostic messages.
`warn ignore_loc`	Enables warnings when %LOC is stripped from an actual argument.
`warn interfaces`	Tells the compiler to check the interfaces of all SUBROUTINEs called and FUNCTIONs invoked in your compilation against a set of interface blocks stored separately from the source being compiled. The compiler generates a compile-time message if the interface used to invoke a routine does not match the interface defined in a .mod file external to the source (that is, in a .mod generated by option `gen-interfaces` as opposed to a .mod file USEd in the source). The compiler looks for these .mods in the current directory or in the directory specified by the include (`-I`) or `-module` option. If interface mismatches occur, some will result in a compile-time error, others will only generate a warning.

Option	Description
	By default, `warn interfaces` turns on option `gen-interfaces`. You can turn off that option by explicitly specifying option `/gen-interfaces-` (Windows*) or `-no-gen-interfaces` (Linux* and macOS*).
`warn shape`	If the `check shape` option is specified, array conformance violations are issued as warnings rather than as errors.
`warn stderrors`	Tells the compiler to change all warning-level messages about Fortran standards violations to error-level messages. This option sets the stand option.
`warn truncated_source`	Enables warnings when a source line exceeds the maximum column width in fixed-format source files. The maximum column width for fixed-format files is 72, 80, or 132, depending on the setting of the `extend-source` option. The `warn truncated_source` option has no effect on truncation; lines that exceed the maximum column width are always truncated. This option does not apply to free-format source files.
`warn uncalled`	Enables warnings when a statement function is never called.
`warn unused`	Enables warnings for variables that are declared but never used.
`warn nousage`	Disables warnings about questionable programming practices. Questionable programming practices, although allowed, often are the result of programming errors; for example: a continued character or Hollerith literal whose first part ends before the statement field and appears to end with trailing spaces. Note that the `/pad-source` option can prevent this error.
`warn all`	This is the same as specifying `warn`. This option does not set options `warn errors` or `warn stderrors`. To enable all the additional checking to be performed and force the severity of the diagnostic messages to be severe enough to not generate an object file, specify `warn allwarn errors` or `warn allwarn stderrors`. On Windows systems: In the Property Pages, **Custom** means that diagnostics will be specified on an individual basis.

IDE Equivalent

Visual Studio: **General > Compile Time Diagnostics** (`/warn:all, /warn:none`)

Diagnostics > Treat Warnings as Errors (`/warn:[no]errors`)

Diagnostics > Treat Fortran Standard Warnings as Errors (`/warn:[no]stderrors`)

Diagnostics > Language Usage Warnings > Compile Time Diagnostics (`/warn:all, /warn:none`)

Diagnostics > Warn for Undeclared Symbols (`/warn:[no]declarations`)

Diagnostics > Warn for Undeclared Externals (`/warn:[no]externals`)

Diagnostics > Warn for Unused Variables (`/warn:[no]unused`)

Diagnostics > Warn When Removing %LOC (`/warn:[no]ignore_loc`)

Diagnostics > Warn When Truncating Source Line (`/warn:[no]truncated_source`)

Diagnostics > Warn for Unaligned Data (`/warn:[no]alignments`)

Diagnostics > Warn for Uncalled Statement Function (`/warn:[no]uncalled`)

Diagnostics > Warn for Array Conformance Violations (`/warn:shape`)

Diagnostics > Suppress Usage Messages (`/warn:[no]usage`)

Diagnostics > Check Routine Interfaces (`/warn:[no]interfaces`)

Eclipse: None

Xcode: **General > Compile Time Diagnostics** (`-warn all, -warn none`)

Diagnostics > Warn For Unaligned Data (`-warn [no]alignments`)

Diagnostics > Warn For Undeclared Symbols (`-warn [no]declarations`)

Diagnostics > Warn for Undeclared Externals (`-warn [no]externals`)

Diagnostics > Treat Warnings as Errors (`-warn error`)

Diagnostics > Warn When Removing %LOC (`-warn [no]ignore_loc`)

Diagnostics > Check Routine Interfaces (`-warn [no]interfaces`)

Diagnostics > Treat Fortran Standard Warnings As Errors (`-warn [no]stderrors`)

Diagnostics > Warn When Truncating Source Line (`-warn [no]truncated_source`)

Diagnostics > Warn For Uncalled Routine (`-warn [no]uncalled`)

Diagnostics > Warn For Unused Variables (`-warn [no]unused`)

Diagnostics > Warn for Array Conformance Violations (`-warn [no]shape`)

Diagnostics > Suppress Usage Messages (`-warn [no]usage`)

Alternate Options

`warn none`
Linux and macOS*: `-nowarn`, `-w`, `-W0`, `-warn nogeneral`
Windows: `/nowarn`,`/w`, `/W0`, `/warn:nogeneral`

`warn declarations`
Linux and macOS*: `-implicitnone`, `-u`
Windows: `/4Yd`

`warn nodeclarations`
Linux and macOS*: None
Windows: `/4Nd`

`warn general`
Linux and macOS*: `-W1`
Windows: `/W1`

`warn nogeneral`
Linux and macOS*: `-W0`, `-w`, `-nowarn`, `-warn none`
Windows: `/W0`, `/w`, `/nowarn`, `/warn:none`

`warn stderrors`
Linux and macOS*: `-e90`, `-e95`, `-e03`, `-e08`, `-e18`
Windows: None

`warn all`
Linux and macOS*: `-warn`
Windows: `/warn` (this is a deprecated option)

WB

Turns a compile-time bounds check into a warning.

Syntax

Linux OS and macOS:

`-WB`

Windows OS:

`/WB`

Arguments

None

Default

OFF Compile-time bounds checks are errors.

Description

This option turns a compile-time bounds check into a warning.

IDE Equivalent

None

Alternate Options

None

Winline

Warns when a function that is declared as inline is not inlined.

Syntax

Linux OS and macOS:

`-Winline`

Windows OS:

`None`

Arguments

None

Default

OFF No warning is produced when a function that is declared as inline is not inlined.

Description

This option warns when a function that is declared as inline is not inlined.

To see diagnostic messages, including a message about why a particular function was not inlined, you should generate an optimization report by specifying option `-qopt-report=5`.

IDE Equivalent

None

Alternate Options

None

See Also

`qopt-report, Qopt-report` compiler option

Compatibility Options

f66

Tells the compiler to apply FORTRAN 66 semantics.

Syntax

Linux OS and macOS:

`-f66`

Windows OS:

`/f66`

Arguments

None

Default

OFF The compiler applies Fortran 2008 semantics.

Description

This option tells the compiler to apply FORTRAN 66 semantics when interpreting language features. This causes the following to occur:

- DO loops are always executed at least once
- FORTRAN 66 EXTERNAL statement syntax and semantics are allowed
- If the OPEN statement STATUS specifier is omitted, the default changes to STATUS='NEW' instead of STATUS='UNKNOWN'
- If the OPEN statement BLANK specifier is omitted, the default changes to BLANK='ZERO' instead of BLANK='NULL'

IDE Equivalent

Visual Studio: **Language > Enable FORTRAN 66 Semantics**

Eclipse: None

Xcode: **Language > Enable FORTRAN 66 Semantics**

Alternate Options

None

f77rtl

Tells the compiler to use the run-time behavior of FORTRAN 77.

Syntax

Linux OS and macOS:

`-f77rtl`

`-nof77rtl`

Windows OS:

`/f77rtl`

`/nof77rtl`

Arguments

None

Default

`nof77rtl` The compiler uses the run-time behavior of Intel® Fortran.

Description

This option tells the compiler to use the run-time behavior of FORTRAN 77.

Specifying this option controls the following run-time behavior:

- When the unit is not connected to a file, some INQUIRE specifiers will return different values:
  - NUMBER= returns 0
  - ACCESS= returns 'UNKNOWN'
  - BLANK= returns 'UNKNOWN'
  - FORM= returns 'UNKNOWN'
- PAD= defaults to 'NO' for formatted input.
- NAMELIST and list-directed input of character strings must be delimited by apostrophes or quotes.
- When processing NAMELIST input:
  - Column 1 of each record is skipped.
  - The '$' or '&' that appears prior to the group-name must appear in column 2 of the input record.

IDE Equivalent

Visual Studio: **Compatibility > Enable F77 Run-Time Compatibility**

Eclipse: None

Xcode: **Compatibility > Enable F77 Run-Time Compatibility**

Alternate Options

None

fpscomp

Controls whether certain aspects of the run-time system and semantic language features within the compiler are compatible with Intel® Fortran or Microsoft Fortran PowerStation.*

Syntax

Linux OS and macOS:

`-fpscomp [keyword[, keyword...]]`

`-nofpscomp`

Windows OS:

`/fpscomp[:keyword[, keyword...]]`

`/nofpscomp`

Arguments

keyword Specifies the compatibility that the compiler should follow. Possible values are:

`none`	Specifies that no options should be used for compatibility.
`[no]filesfromcmd`	Determines what compatibility is used when the OPEN statement FILE= specifier is blank.
`[no]general`	Determines what compatibility is used when semantics differences exist between Fortran PowerStation and Intel® Fortran.
`[no]iofformat`	Determines what compatibility is used for list-directed formatted and unformatted I/O.
`[no]libs`	Determines whether the portability library is passed to the linker.
`[no]ldio_spacing`	Determines whether a blank is inserted at run-time after a numeric value before a character value.
`[no]logicals`	Determines what compatibility is used for representation of LOGICAL values.
`all`	Specifies that all options should be used for compatibility.

Default

`fpscomp libs` The portability library is passed to the linker.

Description

This option controls whether certain aspects of the run-time system and semantic language features within the compiler are compatible with Intel Fortran or Microsoft* Fortran PowerStation.

If you experience problems when porting applications from Fortran PowerStation, specify `fpscomp` (or `fpscomp all`). When porting applications from Intel Fortran, use `fpscomp none` or `fpscomp libs` (the default).

Option	Description
`fpscomp none`	Specifies that no options should be used for compatibility with Fortran PowerStation. This is the same as specifying `nofpscomp`. Option `fpscomp none` enables full Intel® Fortran compatibility. If you omit `fpscomp`, the default is `fpscomp libs`. You cannot use the `fpscomp` and `vms` options in the same command.
`fpscomp filesfromcmd`	Specifies Fortran PowerStation behavior when the OPEN statement FILE= specifier is blank (FILE=' '). It causes the following actions to be taken at run time:

Option	Description
	• The program reads a file name from the list of arguments (if any) in the command line that invoked the program. If any of the command-line arguments contain a null string (''), the program asks the user for the corresponding file name. Each additional OPEN statement with a blank FILE= specifier reads the next command-line argument. • If there are more nameless OPEN statements than command-line arguments, the program prompts for additional file names. • In a QuickWin application, a "File Select" dialog box appears to request file names. To prevent the run-time system from using the file name specified on the command line when the OPEN statement FILE specifier is omitted, specify `fpscomp nofilesfromcmd`. This allows the application of Intel Fortran defaults, such as the FORTn environment variable and the FORT. *n* file name (where *n* is the unit number). The `fpscomp filesfromcmd` option affects the following Fortran features: • The OPEN statement FILE specifier For example, assume a program OPENTEST contains the following statements: OPEN(UNIT = 2, FILE = ' ') OPEN(UNIT = 3, FILE = ' ') OPEN(UNIT = 4, FILE = ' ') The following command line assigns the file TEST.DAT to unit 2, prompts the user for a file name to associate with unit 3, then prompts again for a file name to associate with unit 4: opentest test.dat '' '' • Implicit file open statements such as the WRITE, READ, and ENDFILE statements Unopened files referred to in READ or WRITE statements are opened implicitly as if there had been an OPEN statement with a name specified as all blanks. The name is read from the command line.

`fpscomp general`	Specifies that Fortran PowerStation semantics should be used when a difference exists between Intel Fortran and Fortran PowerStation. The `fpscomp general` option affects the following Fortran features:
	
	• The BACKSPACE statement:
	
	  • It allows files opened with ACCESS='APPEND' to be used with the BACKSPACE statement.
	  • It allows files opened with ACCESS='DIRECT' to be used with the BACKSPACE statement.
	
	  Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be used with the BACKSPACE statement violates the Fortran 95 standard and may be removed in the future.
	• The READ statement:

Option	Description

- It causes a READ from a formatted file opened for direct access to read records that have the same record type format as Fortran PowerStation. This consists of accounting for the trailing Carriage Return/Line Feed pair (<CR><LF>) that is part of the record. It allows sequential reads from a formatted file opened for direct access.

  Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be used with the sequential READ statement violates the Fortran 95 standard and may be removed in the future.
- It allows the last record in a file opened with FORM='FORMATTED' and a record type of STREAM_LF or STREAM_CR that does not end with a proper record terminator (<line feed> or <carriage return>) to be read without producing an error.
- It allows sequential reads from an unformatted file opened for direct access.
- Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be read with the sequential READ statement violates the Fortran 95 standard and may be removed in the future.

- The INQUIRE statement:
  - The CARRIAGECONTROL specifier returns the value "UNDEFINED" instead of "UNKNOWN" when the carriage control is not known.
  - The NAME specifier returns the file name "UNKNOWN" instead of filling the file name with spaces when the file name is not known.
  - The SEQUENTIAL specifier returns the value "YES" instead of "NO" for a direct access formatted file.
  - The UNFORMATTED specifier returns the value "NO" instead of "UNKNOWN" when it is not known whether unformatted I/O can be performed to the file.

    Note: Returning the value "NO" instead of "UNKNOWN" for this specifier violates the Fortran 95 standard and may be removed in the future.
- The OPEN statement:
  - If a file is opened with an unspecified STATUS keyword value, and is not named (no FILE specifier), the file is opened as a scratch file.

    For example:

    OPEN (UNIT = 4)
  - In contrast, when fpscomp nogeneral is in effect with an unspecified STATUS value with no FILE specifier, the FORT*n* environment variable and the FORT.n file name are used (where *n* is the unit number).
  - If the STATUS value was not specified and if the name of the file is "USER", the file is marked for deletion when it is closed.
  - It allows a file to be opened with the APPEND and READONLY characteristics.
  - If the default for the CARRIAGECONTROL specifier is assumed, it gives "LIST" carriage control to direct access formatted files instead of "NONE".
  - If the default for the CARRIAGECONTROL specifier is assumed and the device type is a terminal file, the file is given the default carriage control value of "FORTRAN" instead of "LIST".
  - It gives an opened file the additional default of write sharing.
  - It gives the file a default block size of 1024 instead of 8192.
  - If the default for the MODE and ACTION specifier is assumed and there was an error opening the file, try opening the file as read only, then write only.

Option	Description

	• If a file that is being re-opened has a different file type than the current existing file, an error is returned.
	• It gives direct access formatted files the same record type as Fortran PowerStation. This means accounting for the trailing Carriage Return/Line Feed pair (<CR><LF>) that is part of the record.
	• The STOP statement: It writes the Fortran PowerStation output string and/or returns the same exit condition values.
	• The WRITE statement:
	  • Writing to formatted direct files
	  When writing to a formatted file opened for direct access, records are written in the same record type format as Fortran PowerStation. This consists of adding the trailing Carriage Return/Line Feed pair <CR><LF>) that is part of the record.
	  It ignores the CARRIAGECONTROL specifier setting when writing to a formatted direct access file.
	  • Interpreting Fortran carriage control characters
	  When interpreting Fortran carriage control characters during formatted I/O, carriage control sequences are written that are the same as Fortran PowerStation. This is true for the "Space, 0, 1 and + " characters.
	  • Performing non-advancing I/O to the terminal
	  When performing non-advancing I/O to the terminal, output is written in the same format as Fortran PowerStation.
	  • Interpreting the backslash (\) and dollar ($) edit descriptors
	  When interpreting backslash and dollar edit descriptors during formatted I/O, sequences are written the same as Fortran PowerStation.
	  • Performing sequential writes
	  It allows sequential writes from an unformatted file opened for direct access.
	  Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be read with the sequential WRITE statement violates the Fortran 95 standard and may be removed in the future.
	Specifying `fpscomp general` sets `fpscomp ldio_spacing`.

`fpscomp ioformat`	Specifies that Fortran PowerStation semantic conventions and record formats should be used for list-directed formatted and unformatted I/O. The `fpscomp ioformat` option affects the following Fortran features:
	• The WRITE statement:
	  • For formatted list-directed WRITE statements, formatted internal list-directed WRITE statements, and formatted namelist WRITE statements, the output line, field width values, and the list-directed data type semantics are determined according to the following sample for real constants (N below):
	  For 1 <= N < 10**7, use F15.6 for single precision or F24.15 for double.
	  For N < 1 or N >= 10**7, use E15.6E2 for single precision or E24.15E3 for double.
	  See the Fortran PowerStation documentation for more detailed information about the other data types affected.

Option	Description

- For unformatted WRITE statements, the unformatted file semantics are dictated according to the Fortran PowerStation documentation; these semantics are different from the Intel Fortran file format. See the Fortran PowerStation documentation for more detailed information.

  The following table summarizes the default output formats for list-directed output with the intrinsic data types:

Data Type	Output Format with fpscomp noioformat	Output Format with fpscomp ioformat
BYTE	I5	I12
LOGICAL (all)	L2	L2
INTEGER(1)	I5	I12
INTEGER(2)	I7	I12
INTEGER(4)	I12	I12
INTEGER(8)	I22	I22
REAL(4)	1PG15.7E2	1PG16.6E2
REAL(8)	1PG24.15E3	1PG25.15E3
COMPLEX(4)	'(',1PG14.7E2, ', ', 1PG14.7E2, ') '	'(',1PG16.6E2, ', ', 1PG16.6E2, ') '
COMPLEX(8)	'(',1PG23.15E3, ', ', 1PG23.15E3, ') '	'(',1PG25.15E3, ', ', 1PG25.15E3, ') '
CHARACTER	Aw	Aw

- The READ statement:
  - For formatted list-directed READ statements, formatted internal list-directed READ statements, and formatted namelist READ statements, the field width values and the list-directed semantics are dictated according to the following sample for real constants (N below):

    For 1 <= N < 10**7, use F15.6 for single precision or F24.15 for double.

    For N < 1 or N >= 10**7, use E15.6E2 for single precision or E24.15E3 for double.

    See the Fortran PowerStation documentation for more detailed information about the other data types affected.
  - For unformatted READ statements, the unformatted file semantics are dictated according to the Fortran PowerStation documentation; these semantics are different from the Intel Fortran file format. See the Fortran PowerStation documentation for more detailed information.

Option	Description
`fpscomp nolibs`	Prevents the portability library from being passed to the linker.
`fpscomp ldio_spacing`	Specifies that at run time a blank should not be inserted after a numeric value before a character value (undelimited character string). This representation is used by Intel Fortran releases before Version 8.0 and by Fortran PowerStation. If you specify `fpscomp general`, it sets `fpscomp ldio_spacing`.
`fpscomp logicals`	Specifies that integers with a non-zero value are treated as true, integers with a zero value are treated as false. The literal constant .TRUE. has an integer value of 1, and the literal constant .FALSE. has an integer value of 0. This representation is used by Intel Fortran releases before Version 8.0 and by Fortran PowerStation. The default is `fpscomp nologicals`, which specifies that odd integer values (low bit one) are treated as true and even integer values (low bit zero) are treated as false. The literal constant .TRUE. has an integer value of -1, and the literal constant .FALSE. has an integer value of 0. This representation is used by Compaq* Visual Fortran. The internal representation of LOGICAL values is not specified by the Fortran standard. Programs which use integer values in LOGICAL contexts, or which pass LOGICAL values to procedures written in other languages, are non-portable and may not execute correctly. Intel recommends that you avoid coding practices that depend on the internal representation of LOGICAL values. The `fpscomp logicals` option affects the results of all logical expressions and affects the return value for the following Fortran features: • The INQUIRE statement specifiers OPENED, IOFOCUS, EXISTS, and NAMED • The EOF intrinsic function • The BTEST intrinsic function • The lexical intrinsic functions LLT, LLE, LGT, and LGE
`fpscomp all`	Specifies that all options should be used for compatibility with Fortran PowerStation. This is the same as specifying `fpscomp` with no keyword. Option `fpscomp all` enables full compatibility with Fortran PowerStation.

IDE Equivalent

Visual Studio: **Compatibility > Use Filenames from Command Line** (`/fpscomp:filesfromcmd`)

Compatibility > Use PowerStation I/O Format (`/fpscomp:ioformat`)

Compatibility > Use PowerStation Portability Library (`/fpscomp:nolibs`)

Compatibility > Use PowerStation List-Directed I/O Spacing (`/fpscomp:ldio_spacing`)

Compatibility > Use PowerStation Logical Values (`/fpscomp:logicals`)

Compatibility > Use Other PowerStation Run-time Behavior (`/fpscomp:general`)

Eclipse: None

Xcode: None

Alternate Options

None

gcc-name

Lets you specify the name of the gcc compiler that should be used to set up the link-time environment, including the location of standard libraries.

Syntax

Linux OS:

`-gcc-name=`*`name`*

macOS:

`None`

Windows OS:

`None`

Arguments

name	Is the name of the gcc compiler to use. It can include the path where the gcc compiler is located.

Default

OFF	The compiler uses the PATH setting to find the gcc compiler and resolve environment settings.

Description

This option lets you specify the name of the gcc compiler that should be used to set up the link-time environment, including the location of standard libraries. If you do not specify a path, the compiler will search the PATH settings for the compiler name you provide.

This option is helpful when you are referencing a non-standard gcc installation, or you have multiple gcc installations on your system. The compiler will match gcc version values to the gcc compiler you specify.

The C++ equivalent to option `-gcc-name` is `-gxx-name`.

IDE Equivalent

None

Alternate Options

None

Example

If the following option is specified, the compiler looks for the gcc compiler named `foobar` in the PATH setting:

```
-gcc-name=foobar
```

If the following option is specified, the compiler looks for the gcc compiler named `foobar` in the path specified:

```
-gcc-name=/a/b/foobar
```

See Also

`gxx-name` compiler option

gxx-name

Lets you specify the name of the g++ compiler that should be used to set up the link-time environment, including the location of standard libraries.

Syntax

Linux OS:

`-gxx-name=`*`name`*

macOS:

`None`

Windows OS:

`None`

Arguments

name	Is the name of the g++ compiler to use. It can include the path where the g++ compiler is located.

Default

OFF	The compiler uses the PATH setting to find the g++ compiler and resolve environment settings.

Description

This option lets you specify the name of the g++ compiler that should be used to set up the link-time environment, including the location of standard libraries. If you do not specify a path, the compiler will search the PATH settings for the compiler name you provide.

This option is helpful if you have multiple gcc++ installations on your system. The compiler will match gcc++ version values to the gcc++ compiler you specify.

The C equivalent to option `-gxx-name` is `-gcc-name`.

NOTE
When compiling a C++ file with `icc`, g++ is used to get the environment.

IDE Equivalent

None

Alternate Options

None

Example

If the following option is specified, the compiler looks for the g++ compiler named `foobar` in the PATH setting:

```
-gxx-name=foobar
```

If the following option is specified, the compiler looks for the g++ compiler named `foobar` in the path specified:

```
-gxx-name=/a/b/foobar
```

See Also

`gcc-name` compiler option

Qvc

*Specifies compatibility with Microsoft*Visual C++* or Microsoft Visual Studio*.*

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/Qvc14.2`

`/Qvc14.1`

Arguments

None

Default

varies — When the compiler is installed, it detects which version of Visual Studio is on your system. `Qvc` defaults to the form of the option that is compatible with that version. When multiple versions of Visual Studio are installed, the compiler installation lets you select which version you want to use. In this case, `Qvc` defaults to the version you choose.

Description

This option specifies compatibility with Microsoft* Visual C++ or Microsoft* Visual Studio.

Option	Description
`/Qvc14.2`	Specifies compatibility with Microsoft* Visual Studio 2019.
`/Qvc14.1`	Specifies compatibility with Microsoft* Visual Studio 2017.

IDE Equivalent

None

Alternate Options

None

vms

Causes the run-time system to behave like HP Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).*

Syntax

Linux OS and macOS:

`-vms`

`-novms`

Windows OS:

`/vms`

`/novms`

Arguments

None

Default

`novms` The run-time system follows default Intel® Fortran behavior.

Description

This option causes the run-time system to behave like HP* Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).

It affects the following language features:

- Certain defaults

  In the absence of other options, `vms` sets the defaults as `check format` and `check output_conversion`.
- Alignment

  Option `vms` does not affect the alignment of fields in records or items in common blocks. For compatibility with HP Fortran on OpenVMS systems, use `align norecords` to pack fields of records on the next byte boundary.
- Carriage control default

  If option `vms` and option `ccdefault default` are specified, carriage control defaults to FORTRAN if the file is formatted and the unit is connected to a terminal.
- INCLUDE qualifiers

  `/LIST` and `/NOLIST` are recognized at the end of the file name in an INCLUDE statement at compile time. If the file name in the INCLUDE statement does not specify the complete path, the path used is the current directory. Note that if `vms` is not specified, the path used is the directory where the file that contains the INCLUDE statement resides.
- Quotation mark character

  A quotation mark (") character is recognized as starting an octal constant ("0..7) instead of a character literal ("...").
- Deleted records in relative files

  When a record in a relative file is deleted, the first byte of that record is set to a known character (currently '@'). Attempts to read that record later result in ATTACCNON errors. The rest of the record (the whole record, if `vms` is not specified) is set to nulls for unformatted files and spaces for formatted files.
- ENDFILE records

  When an ENDFILE is performed on a sequential unit, an actual 1-byte record containing a Ctrl/Z is written to the file. If `vms` is not specified, an internal ENDFILE flag is set and the file is truncated. The `vms` option does not affect ENDFILE on relative files: these files are truncated.

- Implied logical unit numbers

  The `vms` option enables Intel® Fortran to recognize certain environment variables at run time for ACCEPT, PRINT, and TYPE statements and for READ and WRITE statements that do not specify a unit number (such as READ (*,1000)).
- Treatment of blanks in input

  The `vms` option causes the defaults for the keyword BLANK in OPEN statements to become 'NULL' for an explicit OPEN and 'ZERO' for an implicit OPEN of an external or internal file.
- OPEN statement effects

  Carriage control defaults to FORTRAN if the file is formatted, and the unit is connected to a terminal. Otherwise, carriage control defaults to LIST. The `vms` option affects the record length for direct access and relative organization files. The buffer size is increased by 1 to accommodate the deleted record character.
- Reading deleted records and ENDFILE records

  The run-time direct access READ routine checks the first byte of the retrieved record. If this byte is '@' or NULL ("\0"), then an ATTACCNON error is returned. The run-time sequential access READ routine checks to see if the record it just read is one byte long and contains a Ctrl/Z. If this is true, it returns EOF.

IDE Equivalent

Visual Studio: **Compatibility > Enable VMS Compatibility**

Eclipse: None

Xcode: **Compatibility > Enable VMS Compatibility**

Alternate Options

None

See Also

`align` compiler option

`ccdefault` compiler option

`check` compiler option

Linking or Linker Options

4Nportlib, 4Yportlib

Determines whether the compiler links to the library of portability routines.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/4Nportlib`

`/4Yportlib`

Arguments

None

Default

/4Yportlib	The library of portability routines is linked during compilation.

Description

Option /4Yportlib causes the compiler to link to the library of portability routines. This also includes Intel's functions for Microsoft* compatibility.

Option /4Nportlib prevents the compiler from linking to the library of portability routines.

IDE Equivalent

Visual Studio: **Libraries > Use Portlib Library**

Eclipse: None

Xcode: None

Alternate Options

None

Bdynamic

Enables dynamic linking of libraries at run time.

Syntax

Linux OS:

-Bdynamic

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	Limited dynamic linking occurs.

Description

This option enables dynamic linking of libraries at run time. Smaller executables are created than with static linking.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option -Bdynamic are linked dynamically until the end of the command line or until a -Bstatic option is encountered. The -Bstatic option enables static linking of libraries.

IDE Equivalent

None

Alternate Options

None

See Also

`Bstatic` compiler option

Bstatic

Enables static linking of a user's library.

Syntax

Linux OS:

`-Bstatic`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF Default static linking occurs.

Description

This option enables static linking of a user's library.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bstatic` are linked statically until the end of the command line or until a `-Bdynamic` option is encountered. The `-Bdynamic` option enables dynamic linking of libraries.

IDE Equivalent

None

Alternate Options

None

See Also

`Bdynamic` compiler option

Bsymbolic

Binds references to all global symbols in a program to the definitions within a user's shared library.

Syntax

Linux OS:

`-Bsymbolic`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	When a program is linked to a shared library, it can override the definition within the shared library.

Description

This option binds references to all global symbols in a program to the definitions within a user's shared library.

This option is only meaningful on Executable Linkage Format (ELF) platforms that support shared libraries.

Caution
This option can have unintended side-effects of disabling symbol preemption in the shared library.

IDE Equivalent

None

Alternate Options

None

See Also

`Bsymbolic-functions` compiler option

Bsymbolic-functions

Binds references to all global function symbols in a program to the definitions within a user's shared library.

Syntax

Linux OS:

`-Bsymbolic-functions`

macOS:

None

Windows OS:

None

Arguments

None

Default

OFF	When a program is linked to a shared library, it can override the definition within the shared library.

Description

This option binds references to all global function symbols in a program to the definitions within a user's shared library.

This option is only meaningful on Executable Linkage Format (ELF) platforms that support shared libraries.

> **Caution**
> This option can have unintended side-effects of disabling symbol preemption in the shared library.

IDE Equivalent

None

Alternate Options

None

See Also

`Bsymbolic` compiler option

cxxlib

Determines whether the compiler links using the C++ run-time libraries provided by gcc.

Syntax

Linux OS and macOS:

`-cxxlib`

`-no-cxxlib`

Windows OS:

`None`

Arguments

None

Default

`-no-cxxlib`	The compiler uses the default run-time libraries and does not link to any additional C++ run-time libraries.

Description

This option determines whether the compiler links to the standard C++ run-time library (libstdc++). It is useful for building mixed Fortran/C++ applications.

Option `-cxxlib=`*dir* can be used with option `-gcc-name=`*name* to specify the location *`dir`*`/bin/`*`name`*.

IDE Equivalent

None

Alternate Options

None

See Also

`gcc-name` compiler option

dbglibs

Tells the linker to search for unresolved references in a debug run-time library.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/dbglibs`

`/nodbglibs`

Arguments

None

Default

`/nodbglibs` The linker does not search for unresolved references in a debug run-time library.

Description

This option tells the linker to search for unresolved references in a debug run-time library.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

The following table shows which options to specify for a debug run-time library:

Type of Library	Options Required	Alternate Option
Debug single-threaded	`/libs:static` `/dbglibs`	`/MLd` (this is a deprecated option)

Debug multithreaded	`/libs:static`
	`/threads`
	`/dbglibs`
		`/MTd`

Multithreaded debug DLLs	`/libs:dll`
	`/threads`
	`/dbglibs`
		`/MDd`

Debug Fortran QuickWin multi-thread applications	`/libs:qwin`	None

Type of Library	Options Required	Alternate Option
	/dbglibs	
Debug Fortran standard graphics (QuickWin single-thread) applications	/libs:qwins /dbglibs	None

IDE Equivalent

None

Alternate Options

None

dll

Specifies that a program should be linked as a dynamic-link (DLL) library.

Syntax

Linux OS and macOS:

None

Windows OS:

/dll

Arguments

None

Default

OFF The program is not linked as a dynamic-link (DLL) library.

Description

This option specifies that a program should be linked as a dynamic-link (DLL) library instead of an executable (.exe) file. It overrides any previous specification of run-time routines to be used and enables the /libs:dll option.

If you use this option with the /libs:qwin or /libs:qwins option, the compiler issues a warning.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: /LD

dynamic-linker

Specifies a dynamic linker other than the default.

Syntax

Linux OS:

`-dynamic-linker file`

macOS:

`None`

Windows OS:

`None`

Arguments

file	Is the name of the dynamic linker to be used.

Default

OFF	The default dynamic linker is used.

Description

This option lets you specify a dynamic linker other than the default.

IDE Equivalent

None

Alternate Options

None

dynamiclib

Invokes the libtool command to generate dynamic libraries.

Syntax

Linux OS:

`None`

macOS:

`-dynamiclib`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option invokes the `libtool` command to generate dynamic libraries.

When passed this option, the compiler uses the `libtool` command to produce a dynamic library instead of an executable when linking.

To build static libraries, you should specify option `-staticlib` or `libtool -static <objects>`.

IDE Equivalent

None

Alternate Options

None

See Also

staticlib
compiler option

extlnk

Specifies file extensions to be passed directly to the linker.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/extlnk:`*`ext`*

Arguments

`ext`	Are the file extensions to be passed directly to the linker.

Default

OFF	Only the file extensions recognized by the compiler are passed to the linker.

Description

This option specifies file extensions (*ext*) to be passed directly to the linker. It is useful if your source file has a nonstandard extension.

You can specify one or more file extensions. A leading period before each extension is optional; for example, `/extlnk:myobj` and `/extlnk:.myobj` are equivalent.

IDE Equivalent

None

Alternate Options

None

F (Windows*)

Specifies the stack reserve amount for the program.

Syntax

Linux OS:

None

macOS:

None

Windows OS:

/Fn

Arguments

n	Is the stack reserve amount. It can be specified as a decimal integer or as a hexadecimal constant by using a C-style convention (for example, /F0x1000).

Default

OFF	The stack size default is chosen by the operating system.

Description

This option specifies the stack reserve amount for the program. The amount (*n*) is passed to the linker.

Note that the linker property pages have their own option to do this.

IDE Equivalent

None

Alternate Options

None

F (macOS*)

Adds a framework directory to the head of an include file search path.

Syntax

Linux OS:

None

macOS:

-Fdir

Windows OS:

None

Arguments

dir	Is the name for the framework directory.

Default

OFF	The compiler does not add a framework directory to the head of an include file search path.

Description

This option adds a framework directory to the head of an include file search path.

IDE Equivalent

None

Alternate Options

None

fuse-ld

Tells the compiler to use a different linker instead of the default linker (ld).

Syntax

Linux OS:

`-fuse-ld=`*`keyword`*

macOS:

`-fuse-ld=`*`keyword`*

Windows OS:

`None`

Arguments

`keyword`	Possible values are:	
	`bfd`	Tells the compiler to use the bfd linker.
	`gold`	Tells the compiler to use the gold linker.

Default

ld	The compiler uses the ld linker by default.

Description

This option tells the compiler to use a different linker instead of default linker (ld).

This option is provided for compatibility with gcc.

IDE Equivalent

None

Alternate Options

None

l

Tells the linker to search for a specified library when linking.

Syntax

Linux OS:

`-l`*string*

macOS:

`-l`*string*

Windows OS:

None

Arguments

string	Specifies the library (lib*string*) that the linker should search.

Default

OFF	The linker searches for standard libraries in standard directories.

Description

This option tells the linker to search for a specified library when linking.

When resolving references, the linker normally searches for libraries in several standard directories, in directories specified by the `L` option, then in the library specified by the `l` option.

The linker searches and processes libraries and object files in the order they are specified. So, you should specify this option following the last object file it applies to.

IDE Equivalent

None

Alternate Options

None

See Also

`L` compiler option

L

Tells the linker to search for libraries in a specified directory before searching the standard directories.

Syntax

Linux OS:

`-L`*dir*

macOS:

`-L`*dir*

Windows OS:

None

Arguments

dir Is the name of the directory to search for libraries.

Default

OFF The linker searches the standard directories for libraries.

Description

This option tells the linker to search for libraries in a specified directory before searching for them in the standard directories.

IDE Equivalent

None

Alternate Options

None

See Also

`l` compiler option

libs

Tells the compiler which type of run-time library to link to.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/libs[:keyword]`

Arguments

keyword Specifies the type of run-time library to link to. Possible values are:

`dll`	Specifies a multithreaded, dynamic-link (DLL) library.
`qwin`	Specifies the Fortran QuickWin library.
`qwins`	Specifies the Fortran Standard Graphics library.
`static`	Specifies a multi-threaded, static run-time library. This is the same as specifying `/libs` with no *keyword*. Note that some libraries do not have a static version, such as the OpenMP run-time libraries or the coarray run-time libraries.

Default

`/libs` The compiler links to a multi-threaded, static run-time library.

Description

This option tells the compiler which type of run-time library to link to.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

There are several types of libraries you can link to, depending on which compiler options you specify, as shown in the table below.

If you use the `/libs:dll` option and an unresolved reference is found in the DLL, it gets resolved when the program is executed, during program loading, reducing executable program size.

If you use the `/libs:qwin` or `/libs:qwins` option with the `/dll` option, the compiler issues a warning.

You cannot use the `/libs:qwin` option and options `/libs:dll /threads`.

The following table shows which options to specify for different run-time libraries:

Type of Library	Options Required	Alternate Option
Multithreaded static	`/libs` (or `/threads`)	`/MT`
Debug multithreaded static	`/libs` (or `/threads`) and `/dbglibs`	`/MTd`
Multithreaded DLLs	`/libs:dll`	`/MD`
Multithreaded debug DLLs	`/libs:dll` and `/dbglibs`	`/MDd`
Fortran QuickWin multi-doc applications	`/libs:qwin`	`/MW`
Fortran standard graphics (QuickWin single-doc) applications	`/libs:qwins`	`/MWs`
Debug Fortran QuickWin multi-doc applications	`/libs:qwin` `/dbglibs`	None
Debug Fortran standard graphics (QuickWin single-doc) applications	`/libs:qwins` `/dbglibs`	None

NOTE
This option adds directives to the compiled code, which the linker then reads without further input from the driver.

NOTE
Starting with the 13.0 release of the Intel compilers, the Intel® OpenMP* library is provided in DLL form only.

IDE Equivalent

Visual Studio: **Libraries > Runtime Library** (`/libs:{dll|qwin|qwins}`, `/threads`, `/dbglibs`)

Eclipse: None

Xcode: None

Alternate Options

`/libs`	Linux and macOS*: `-threads` Windows: `/threads`
`/libs:dll`	Linux and macOS*: None Windows: `/MD`
`/libs:qwin`	Linux and macOS*: None Windows: `/MW`
`/libs:qwins`	Linux and macOS*: None Windows: `/MWs`

See Also

`threads` compiler option

`dbglibs` compiler option

link

Passes user-specified options directly to the linker at compile time.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/link`

Arguments

None

Default

OFF No user-specified options are passed directly to the linker.

Description

This option passes user-specified options directly to the linker at compile time.

All options that appear following `/link` are passed directly to the linker.

IDE Equivalent

None

Alternate Options

None

See Also

`Xlinker` compiler option

map

Tells the linker to generate a link map file.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/map[:filename]`

`/nomap`

Arguments

`filename` Is the name for the link map file. It can be a file name or a directory name.

Default

`/nomap` No link map is generated.

Description

This option tells the linker to generate a link map file.

IDE Equivalent

Visual Studio: **Linker > Debugging > Generate Map File** (/MAP)

Eclipse: None

Xcode: None

Alternate Options

None

MD

Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/MD`

`/MDd`

Arguments

None

Default

OFF The linker searches for unresolved references in a multi-threaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, dynamic-link (DLL) run-time library. This is the same as specifying options `/libs:dll /threads /dbglibs`. You can also specify `/MDd`, where `d` indicates a debug version.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

IDE Equivalent

Visual Studio: **Libraries > Runtime Library**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

`libs` compiler option

`threads` compiler option

MDs

Tells the linker to search for unresolved references in a single-threaded, dynamic-link run-time library. This is a deprecated option. There is no replacement option.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/MDs`

`/MDsd`

Arguments

None

Default

OFF The linker searches for unresolved references in a single-threaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a single-threaded, dynamic-link (DLL) run-time library.

You can also specify `/MDsd`, where *d* indicates a debug version.

IDE Equivalent

None

Alternate Options

`/MDs`	Linux and macOS*: None Windows: `/libs:dll`

See Also

`libs` compiler option

MT

Tells the linker to search for unresolved references in a multithreaded, static run-time library.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/MT`

`/MTd`

Arguments

None

Default

`/MT /noreentrancy:threaded`	The linker searches for unresolved references in a multithreaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, static run-time library. This is the same as specifying options `/libs:static /threads /noreentrancy`. You can also specify `/MTd`, where `d` indicates a debug version.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

IDE Equivalent

Visual Studio: **Libraries > Runtime Library**

Eclipse: None

Xcode: None

Alternate Options

None

See Also

Qvc compiler option
libs compiler option
threads compiler option
reentrancy compiler option

nodefaultlibs

Prevents the compiler from using standard libraries when linking.

Syntax

Linux OS:

```
-nodefaultlibs
```

macOS:

```
-nodefaultlibs
```

Windows OS:

```
None
```

Arguments

None

Default

OFF	The standard libraries are linked.

Description

This option prevents the compiler from using standard libraries when linking.

IDE Equivalent

None

Alternate Options

None

See Also

nostdlib compiler option

nofor-main

Specifies that the main program is not written in Fortran.

Syntax

Linux OS and macOS:

```
-nofor-main
```

Windows OS:

```
None
```

Arguments

None

Default

OFF The compiler assumes the main program is written in Fortran.

Description

This option specifies that the main program is not written in Fortran. It is a link-time option that prevents the compiler from linking `for_main.o` into applications.

For example, if the main program is written in C and calls a Fortran subprogram, specify `-nofor-main` when compiling the program with the `ifort` command.

If you omit this option, the main program must be a Fortran program.

IDE Equivalent

None

Alternate Options

None

nostartfiles

Prevents the compiler from using standard startup files when linking.

Syntax

Linux OS:

`-nostartfiles`

macOS:

`-nostartfiles`

Windows OS:

`None`

Arguments

None

Default

OFF The compiler uses standard startup files when linking.

Description

This option prevents the compiler from using standard startup files when linking.

IDE Equivalent

None

Alternate Options

None

See Also

`nostdlib` compiler option

nostdlib

Prevents the compiler from using standard libraries and startup files when linking.

Syntax

Linux OS:

```
-nostdlib
```

macOS:

```
-nostdlib
```

Windows OS:

```
None
```

Arguments

None

Default

OFF The compiler uses standard startup files and standard libraries when linking.

Description

This option prevents the compiler from using standard libraries and startup files when linking.

IDE Equivalent

None

Alternate Options

None

See Also

`nodefaultlibs` compiler option

`nostartfiles` compiler option

pie

Determines whether the compiler generates position-independent code that will be linked into an executable.

Syntax

Linux OS:

```
-pie
```

```
-no-pie
```

macOS:

```
-pie
```

```
-no-pie
```

Windows OS:

None

Arguments

None

Default

varies On Linux* and on macOS* versions less than 10.7, the default is -no-pie. On macOS* 10.7 or greater, the default is -pie.

Description

This option determines whether the compiler generates position-independent code that will be linked into an executable. To enable generation of position-independent code that will be linked into an executable, specify `-pie`.

To disable generation of position-independent code that will be linked into an executable, specify `-no-pie`.

IDE Equivalent

None

Alternate Options

None

See Also

`fpic`
compiler option

pthread

Tells the compiler to use pthreads library for multithreading support.

Syntax

Linux OS:

`-pthread`

macOS:

`-pthread`

Windows OS:

None

Arguments

None

Default

OFF The compiler does not use pthreads library for multithreading support.

Description

Tells the compiler to use pthreads library for multithreading support.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-reentrancy threaded`

Windows: `/reentrancy:threaded`

shared

Tells the compiler to produce a dynamic shared object instead of an executable.

Syntax

Linux OS:

`-shared`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option tells the compiler to produce a dynamic shared object (DSO) instead of an executable. This includes linking in all libraries dynamically and passing `-shared` to the linker.

You must specify option `fpic` for the compilation of each object file you want to include in the shared library.

IDE Equivalent

None

Alternate Options

None

See Also

`dynamiclib` compiler option

`fpic` compiler option

`Xlinker` compiler option

shared-intel

Causes Intel-provided libraries to be linked in dynamically.

Syntax

Linux OS:

`-shared-intel`

macOS:

`-shared-intel`

Windows OS:

`None`

Arguments

None

Default

OFF Intel® libraries are linked in statically, with the exception of Intel's coarray runtime support library and Intel's OpenMP* runtime support library, which are linked in dynamically. To link the OpenMP* runtime support library statically, specify option `-qopenmp-link=static`.

Description

This option causes Intel-provided libraries to be linked in dynamically. It is the opposite of `-static-intel`.

This option is processed by the ifort (icc/icpc) command that initiates linking, adding library names explicitly to the link command.

If you specify option `-mcmodel=medium` or `-mcmodel=large`, it sets option `-shared-intel`.

NOTE
On macOS* systems, when you set "Intel Runtime Libraries" to "Dynamic", you must also set the DYLD_LIBRARY_PATH environment variable within Xcode* or an error will be displayed.

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **Runtime > Intel Runtime Libraries**

Alternate Options

None

See Also

`static-intel` compiler option
`qopenmp-link` compiler option

shared-libgcc

Links the GNU libgcc library dynamically.

Syntax

Linux OS:

`-shared-libgcc`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

`-shared-libgcc`	The compiler links the `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library dynamically. It is the opposite of option `static-libgcc`.

This option is processed by the ifort (icc/icpc) command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior of the `static` option, which causes all libraries to be linked statically.

IDE Equivalent

None

Alternate Options

None

See Also

static-libgcc compiler option

static

Prevents linking with shared libraries.

Syntax

Linux OS:

`-static`

macOS:

`None`

Windows OS:

`/static`

Arguments

None

Default

varies The compiler links with shared GNU libraries (Linux* systems) or shared Microsoft* libraries (Windows* systems) and it links with static Intel libraries, with the exception of the OpenMP* libraries and coarray library libicaf, which are linked in dynamically.

On Windows* systems, option /static is equivalent to option /MT.

Description

This option prevents linking with shared libraries. It causes the executable to link all libraries statically.

NOTE
This option does not cause static linking of libraries for which no static version is available, such as the OpenMP run-time libraries on Windows* or the coarray run-time libraries. These libraries can only be linked dynamically.

IDE Equivalent

None

Alternate Options

None

See Also

`MT` compiler option

`Qvc` compiler option

`static-intel` compiler option

static-intel

Causes Intel-provided libraries to be linked in statically.

Syntax

Linux OS:

`-static-intel`

macOS:

`-static-intel`

Windows OS:

`None`

Arguments

None

Default

ON Intel® libraries are linked in statically, with the exception of Intel's coarray runtime support library and Intel's OpenMP* runtime support library, which are linked in dynamically. To link the OpenMP* runtime support library statically, specify option `-qopenmp-link=static`.

Description

This option causes Intel-provided libraries to be linked in statically with certain exceptions (see the Default above). It is the opposite of `-shared-intel`.

This option is processed by the ifort command that initiates linking, adding library names explicitly to the link command.

If you specify option `-static-intel` while option `-mcmodel=medium` or `-mcmodel=large` is set, an error will be displayed.

If you specify option `-static-intel` and any of the Intel-provided libraries have no static version, a diagnostic will be displayed.

IDE Equivalent

Visual Studio: None

Eclipse: None

Xcode: **Runtime > Intel Runtime Libraries**

Alternate Options

None

See Also

`shared-intel` compiler option
`qopenmp-link` compiler option

static-libgcc

Links the GNU libgcc library statically.

Syntax

Linux OS:

`-static-libgcc`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF The compiler links the GNU `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library statically. It is the opposite of option `-shared-libgcc`.

This option is processed by the ifort command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior, which causes the library to be linked dynamically.

NOTE
If you want to use traceback, you must also link to the static version of the `libgcc` library. This library enables printing of backtrace information.

IDE Equivalent

None

Alternate Options

None

See Also

`shared-libgcc` compiler option
`static-libstdc++` compiler option

static-libstdc++

Links the GNU libstdc++ library statically.

Syntax

Linux OS:

`-static-libstdc++`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF The compiler links the GNU `libstdc++` library dynamically.

Description

This option links the GNU `libstdc++` library statically. This option is processed by the ifort (icc/icpc) command that initiates linking, adding library names explicitly to the link command.

This option is useful when you want to override the default behavior, which causes the library to be linked dynamically.

IDE Equivalent

None

Alternate Options

None

See Also

`static-libgcc` compiler option

staticlib

Invokes the libtool command to generate static libraries.

Syntax

Linux OS:

`None`

macOS:

`-staticlib`

Windows OS:

`None`

Arguments

None

Default

OFF The compiler produces an executable.

Description

This option invokes the `libtool` command to generate static libraries. This option is processed by the command that initiates linking, adding library names explicitly to the link command.

When passed this option, the compiler uses the `libtool` command to produce a static library instead of an executable when linking.

To build dynamic libraries, you should specify option `-dynamiclib` or `libtool -dynamic <objects>`.

IDE Equivalent

None

Alternate Options

None

See Also

dynamiclib
compiler option

T

Tells the linker to read link commands from a file.

Syntax

Linux OS:

`-T`*`filename`*

macOS:

`None`

Windows OS:

`None`

Arguments

filename Is the name of the file.

Default

OFF The linker does not read link commands from a file.

Description

This option tells the linker to read link commands from a file.

IDE Equivalent

None

Alternate Options

None

threads

Tells the linker to search for unresolved references in a multithreaded run-time library.

Syntax

Linux OS and macOS:

`-threads`

`-nothreads`

Windows OS:

`/threads`

`/nothreads`

Arguments

None

Default

`threads` The linker searches for unresolved references in a library that supports enabling thread-safe operation.

Description

This option tells the linker to search for unresolved references in a multithreaded run-time library.

This option sets option `reentrancy threaded`.

Windows systems: The following table shows which options to specify for a multithreaded run-time library.

Type of Library	Options Required	Alternate Option
Multithreaded	`/libs:static` `/threads`	`/MT`
Debug multithreaded	`/libs:static`	`/MTd`

Type of Library	Options Required	Alternate Option
	/threads	
	/dbglibs	
Multithreaded DLLs	/libs:dll	/MD
	/threads	
Multithreaded debug DLLs	/libs:dll	/MDd
	/threads	
	/dbglibs	

To ensure that a threadsafe and/or reentrant run-time library is linked and correctly initialized, option `threads` should also be used for the link step and for the compilation of the main routine.

NOTE
On Windows* systems, this option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker. On Linux* and macOS* systems, this option is processed by the `ifort` command that initiates linking, adding library names explicitly to the link command.

IDE Equivalent

None

Alternate Options

None

See Also

`reentrancy` compiler option

v

Specifies that driver tool commands should be displayed and executed.

Syntax

Linux OS:

`-v [filename]`

macOS:

`-v [filename]`

Windows OS:

`None`

Arguments

filename	Is the name of a source file to be compiled. A space must appear before the file name.

Default

OFF No tool commands are shown.

Description

This option specifies that driver tool commands should be displayed and executed.

If you use this option without specifying a source file name, the compiler displays only the version of the compiler.

If you want to display processing information (pass information and source file names), specify keyword `all` for the `watch` option.

IDE Equivalent

None

Alternate Options

Linux and macOS*: `-watch cmd`

Windows: `/watch:cmd`

See Also

`dryrun` compiler option
`watch` compiler option

Wa

Passes options to the assembler for processing.

Syntax

Linux OS:

`-Wa,option1[,option2,...]`

macOS:

`-Wa,option1[,option2,...]`

Windows OS:

`None`

Arguments

option	Is an assembler option. This option is not processed by the driver and is directly passed to the assembler.

Default

OFF No options are passed to the assembler.

Description

This option passes one or more options to the assembler for processing. If the assembler is not invoked, these options are ignored.

IDE Equivalent

None

Alternate Options

None

winapp

Tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/winapp`

Arguments

None

Default

OFF	No graphics or Fortran Windows application is created.

Description

This option tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.

This option is processed by the compiler, which adds directives to the compiled object file that are processed by the linker.

IDE Equivalent

Visual Studio: **Libraries > Use Common Windows Libraries**

Eclipse: None

Xcode: None

Alternate Options

Linux and macOS*: None

Windows: `/MG`

Wl

Passes options to the linker for processing.

Syntax

Linux OS:

`-Wl,`*`option1`*`[,`*`option2`*`,...]`

macOS:

`-Wl,`*`option1`*`[,`*`option2`*`,...]`

Windows OS:

`None`

Arguments

option	Is a linker option. This option is not processed by the driver and is directly passed to the linker.

Default

OFF	No options are passed to the linker.

Description

This option passes one or more options to the linker for processing. If the linker is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,link,`*options*.

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Wp

Passes options to the preprocessor.

Syntax

Linux OS:

`-Wp,`*option1*`[,`*option2*`,...]`

macOS:

`-Wp,`*option1*`[,`*option2*`,...]`

Windows OS:

`None`

Arguments

option	Is a preprocessor option. This option is not processed by the driver and is directly passed to the preprocessor.

Default

OFF	No options are passed to the preprocessor.

Description

This option passes one or more options to the preprocessor. If the preprocessor is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,`*fpp, options*.

IDE Equivalent

None

Alternate Options

None

See Also

`Qoption` compiler option

Xlinker

Passes a linker option directly to the linker.

Syntax

Linux OS:

`-Xlinker` *option*

macOS:

`-Xlinker` *option*

Windows OS:

`None`

Arguments

option	Is a linker option.

Default

OFF	No options are passed directly to the linker.

Description

This option passes a linker option directly to the linker. If `-Xlinker -shared` is specified, only `-shared` is passed to the linker and no special work is done to ensure proper linkage for generating a shared object. `-Xlinker` just takes whatever arguments are supplied and passes them directly to the linker.

If you want to pass compound options to the linker, for example "`-L $HOME/lib`", you must use the following method:

```
-Xlinker -L -Xlinker $HOME/lib
```

IDE Equivalent

None

Alternate Options

None

See Also

`shared` compiler option
`link` compiler option

Miscellaneous Options

bigobj

Increases the number of sections that an object file can contain.

Syntax

Linux OS:

`None`

macOS:

`None`

Windows OS:

`/bigobj`

Arguments

None

Default

OFF	An object file can hold up to 65,536 (2**16) addressable sections.

Description

This option increases the number of sections that an object file can contain. It increases the address capacity to 4,294,967,296(2**32).

This option may be helpful for .obj files that can hold more sections, such as machine generated code.

IDE Equivalent

None

Alternate Options

None

dryrun

Specifies that driver tool commands should be shown but not executed.

Syntax

Linux OS:

`-dryrun`

macOS:

`-dryrun`

Windows OS:

`None`

Arguments

None

Default

OFF No tool commands are shown, but they are executed.

Description

This option specifies that driver tool commands should be shown but not executed.

IDE Equivalent

None

None

Alternate Options

None

See Also

- v compiler option

dumpmachine

Displays the target machine and operating system configuration.

Syntax

Linux OS:

```
-dumpmachine
```

macOS:

```
-dumpmachine
```

Windows OS:

```
None
```

Arguments

None

Default

OFF The compiler does not display target machine or operating system information.

Description

This option displays the target machine and operating system configuration. No compilation is performed.

IDE Equivalent

None

Alternate Options

None

extfor

Specifies file extensions to be processed by the compiler as Fortran files.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/extfor:`*`ext`*

Arguments

ext	Are the file extensions to be processed as a Fortran file.

Default

OFF	Only the file extensions recognized by the compiler are processed as Fortran files.

Description

This option specifies file extensions (*ext*) to be processed by the compiler as Fortran files. It is useful if your source file has a nonstandard extension.

You can specify one or more file extensions. A leading period before each extension is optional; for example, `/extfor:myf95` and `/extfor:.myf95` are equivalent.

IDE Equivalent

None

Alternate Options

None

See Also

Tf compiler option

extfpp

Specifies file extensions to be recognized as a file to be preprocessed by the Fortran preprocessor.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/extfpp:`*`ext`*

Arguments

ext	Are the file extensions to be preprocessed by the Fortran preprocessor.

Default

OFF Only the file extensions recognized by the compiler are preprocessed by `fpp`.

Description

This option specifies file extensions (*ext*) to be recognized as a file to be preprocessed by the Fortran preprocessor (`fpp`). It is useful if your source file has a nonstandard extension.

You can specify one or more file extensions. A leading period before each extension is optional; for example, `/extfpp:myfpp` and `/extfpp:.myfpp` are equivalent.

IDE Equivalent

None

Alternate Options

None

global-hoist, Qglobal-hoist

Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.

Syntax

Linux OS:

`-global-hoist`

`-no-global-hoist`

macOS:

`-global-hoist`

`-no-global-hoist`

Windows OS:

`/Qglobal-hoist`

`/Qglobal-hoist-`

Arguments

None

Default

`-global-hoist` or `/Qglobal-hoist` Certain optimizations are enabled that can move memory loads.

Description

This option enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source. In most cases, these optimizations are safe and can improve performance.

The negative form of the option is useful for some applications, such as those that use shared or dynamically mapped memory, which can fail if a load is moved too early in the execution stream (for example, before the memory is mapped).

IDE Equivalent

None

Alternate Options

None

help

Displays all available compiler options or a category of compiler options.

Syntax

Linux OS:

`-help[category]`

macOS:

`-help[category]`

Windows OS:

`/help[category]`

Arguments

category Is a category or class of options to display. Possible values are:

`advanced`	Displays advanced optimization options that allow fine tuning of compilation or allow control over advanced features of the compiler.
`codegen`	Displays Code Generation options.
`compatibility`	Displays options affecting language compatibility.
`component`	Displays options for component control.
`data`	Displays options related to interpretation of data in programs or the storage of data.
`deprecated`	Displays options that have been deprecated.
`diagnostics`	Displays options that affect diagnostic messages displayed by the compiler.
`float`	Displays options that affect floating-point operations.
`help`	Displays all the available help categories.
`inline`	Displays options that affect inlining.
`ipo`	Displays Interprocedural Optimization (IPO) options
`language`	Displays options affecting the behavior of the compiler language features.

`link`	Displays linking or linker options.
`misc`	Displays miscellaneous options that do not fit within other categories.
`openmp`	Displays OpenMP and parallel processing options.
`opt`	Displays options that help you optimize code.
`output`	Displays options that provide control over compiler output.
`pgo`	Displays Profile Guided Optimization (PGO) options.
`preproc`	Displays options that affect preprocessing operations.
`reports`	Displays options for optimization reports.

Default

OFF No list is displayed unless this compiler option is specified.

Description

This option displays all available compiler options or a category of compiler options. If category is not specified, all available compiler options are displayed.

IDE Equivalent

None

Alternate Options

Linux and macOS*: None

Windows: `/?`

intel-freestanding

Lets you compile in the absence of a gcc environment.

Syntax

Linux OS:

`-intel-freestanding[=ver]`

macOS:

`None`

Windows OS:

`None`

Arguments

ver	Is a three-digit number that is used to determine the gcc version that the compiler should be compatible with for compilation. It also sets the corresponding GNUC macros.

The number will be normalized to reflect the gcc compiler version numbering scheme. For example, if you specify 493, it indicates the compiler should be compatible with gcc version 4.9.3.

Default

OFF The compiler uses default heuristics when choosing the gcc environment.

Description

This option lets you compile in the absence of a gcc environment. It disables any external compiler calls (such as calls to gcc) that the compiler driver normally performs by default.

This option also removes any default search locations for header and library files. So, for successful compilation and linking, you must provide these search locations.

This option does not affect ld, as, or fpp. They will be used for compilation as needed.

NOTE
This option does not imply option `-nostdinc -nostdlib`. If you want to assure a clean environment for compilation (including removal of Intel-specific header locations and libs), you should specify `-nostdinc` and/or `-nostdlib`.

NOTE
This option is supported for any Linux-target compiler, including a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

See Also

intel-freestanding-target-os compiler option
nostdlib compiler option
nostdinc compiler option, which is an alternate option for option X

intel-freestanding-target-os

Lets you specify the target operating system for compilation.

Syntax

Linux OS:

`-intel-freestanding-target-os=`*os*

macOS:

`None`

Windows OS:

`None`

Arguments

os	Is the target operating system for the Linux compiler. Currently, the only possible value is `linux`.

Default

OFF	The installed gcc determines the target operating system.

Description

This option lets you specify the target operating system for compilation. It sets option `-intel-freestanding`.

> **NOTE**
> This option is supported for any Linux-target compiler, including a Windows-host to Linux-target compiler.

IDE Equivalent

None

Alternate Options

None

See Also

intel-freestanding compiler option

libdir

Controls whether linker options for search libraries are included in object files generated by the compiler.

Syntax

Linux OS and macOS:

`None`

Windows OS:

`/libdir[:`*`keyword`*` ]`

`/nolibdir`

Arguments

keyword	Specifies the linker search options. Possible values are:
`none`	Prevents any linker search options from being included into the object file. This is the same as specifying `/nolibdir`.
`[no]automatic`	Determines whether linker search options for libraries automatically determined by the `ifort` command driver (default libraries) are included in the object file.
`[no]user`	Determines whether linker search options for libraries specified by the OBJCOMMENT source directives are included in the object file.

`all` Causes linker search options for the following libraries:

- Libraries automatically determined by the `ifort` command driver (default libraries)
- Libraries specified by the OBJCOMMENT directive to be included in the object file

This is the same as specifying `/libdir`.

Default

`/libdir:all` Linker search options for libraries automatically determined by the `ifort` command driver (default libraries) and libraries specified by the OBJCOMMENT directive are included in the object file.

Description

This option controls whether linker options for search libraries (`/DEFAULTLIB:library`) are included in object files generated by the compiler.

The linker option `/DEFAULTLIB:library` adds one library to the list of libraries that the linker searches when resolving references. A library specified with `/DEFAULTLIB:library` is searched after libraries specified on the command line and before default libraries named in .obj files.

IDE Equivalent

Visual Studio: **Libraries > Disable Default Library Search Rules** (`/libdir:[no]automatic`)

Libraries > Disable OBJCOMMENT Library Name in Object (`/libdir:[no]user`)

Eclipse: None

Xcode: None

Alternate Options

`/libdir:none` Linux and macOS*: None

Windows: `/Zl`

logo

Displays the compiler version information.

Syntax

Linux OS and macOS:

`-logo`

`-nologo`

Windows OS:

`/logo`

`/nologo`

Arguments

None

Default

Linux* and macOS*: `nologo`	The compiler version information is not displayed.
Windows* : `logo`	The compiler version information is displayed.

Description

This option displays the startup banner, which contains the following compiler information:

- The name of the compiler and its applicable architecture
- The major and minor version of the compiler, the update number, and the package number(for example, Version 11.1.0.047)
- The specific build and build date (for example, Build <builddate>)
- The copyright date of the software

This option can be placed anywhere on the command line.

IDE Equivalent

Visual Studio: **General > Suppress Startup Banner** (`/nologo`)

Eclipse: None

Xcode: **General > Show Startup Banner** (`-V`)

Alternate Options

Linux and macOS*: `-V`

Windows: None

multiple-processes, MP

Creates multiple processes that can be used to compile large numbers of source files at the same time.

Syntax

Linux OS:

`-multiple-processes[=n]`

macOS:

`-multiple-processes[=n]`

Windows OS:

`/MP[:n]`

Arguments

n	Is the maximum number of processes that the compiler should create.

Default

OFF	A single process is used to compile source files.

Description

This option creates multiple processes that can be used to compile large numbers of source files at the same time. It can improve performance by reducing the time it takes to compile source files on the command line.

This option causes the compiler to create one or more copies of itself, each in a separate process. These copies simultaneously compile the source files.

If *n* is not specified for this option, the default value is as follows:

- On Windows* systems, the value is based on the setting of the NUMBER_OF_PROCESSORS environment variable.
- On Linux* and macOS* systems, the value is 2.

This option applies to compilations, but not to linking or link-time code generation.

IDE Equivalent

Visual Studio: **General > Multi-processor Compilation**

Eclipse: None

Xcode: None

Alternate Options

None

print-sysroot

Prints the target sysroot directory that is used during compilation.

Syntax

Linux OS:

`-print-sysroot`

macOS:

`None`

Windows OS:

`None`

Arguments

None

Default

OFF	Nothing is printed.

Description

This option prints the target sysroot directory that is used during compilation.

This is the target sysroot directory that is specified in an environment file or in option `--sysroot`. This option is only effective if a target sysroot has been specified.

IDE Equivalent

None

Alternate Options

None

See Also

`sysroot` compiler option

save-temps, Qsave-temps

Tells the compiler to save intermediate files created during compilation.

Syntax

Linux OS:

`-save-temps`

`-no-save-temps`

macOS:

`-save-temps`

`-no-save-temps`

Windows OS:

`/Qsave-temps`

`/Qsave-temps-`

Arguments

None

Default

Linux* and macOS* systems: `-no-save-temps` Windows* systems: .obj files are saved	On Linux and macOS* systems, the compiler deletes intermediate files after compilation is completed. On Windows systems, the compiler saves only intermediate object files after compilation is completed.

Description

This option tells the compiler to save intermediate files created during compilation. The names of the files saved are based on the name of the source file; the files are saved in the current working directory.

If option `[Q]save-temps` is specified, the following occurs:

- The object .o file (Linux and macOS*) or .obj file (Windows) is saved.
- The assembler .s file (Linux and macOS*) or .asm file (Windows) is saved if you specified the `[Q]use-asm` option.
- The .i or .i90 file is saved if the fpp preprocessor is invoked.

If `-no-save-temps` is specified on Linux or macOS* systems, the following occurs:

- The .o file is put into `/tmp` and deleted after calling `ld`.
- The preprocessed file is not saved after it has been used by the compiler.

If `/Qsave-temps-` is specified on Windows systems, the following occurs:

- The .obj file is not saved after the linker step.

- The preprocessed file is not saved after it has been used by the compiler.

NOTE
This option only saves intermediate files that are normally created during compilation.

IDE Equivalent

None

Alternate Options

None

Example

If you compile program `my_foo.F` on a Linux or macOS* system and you specify option `-save-temps` and option `-use-asm`, the compilation will produce `files my_foo.o`, `my_foo.s`, and `my_foo.i`.

If you compile program `my_foo.fpp` on a Windows system and you specify option `/Qsave-temps` and option `/Quse-asm`, the compilation will produce `files my_foo.obj`, `my_foo.asm`, and `my_foo.i`.

sox

Tells the compiler to save the compilation options and version number in the executable file. It also lets you choose whether to include lists of certain routines.

Syntax

Linux OS:

`-sox[=keyword[,keyword]]`

`-no-sox`

macOS:

`None`

Windows OS:

`None`

Arguments

keyword Is the routine information to include. Possible values are:

`inline`	Includes a list of the routines that were inlined in each object.
`profile`	Includes a list of the routines that were compiled with the `-prof-use` option and for which the .dpi file had profile information, and an indication for each as to whether the profile information was USED (matched) or IGNORED (mismatched).

Default

`-no-sox` The compiler does not save these informational strings in the object file.

Description

This option tells the compiler to save the compilation options and version number in the executable file. It also lets you choose whether to include lists of certain routines. The information is embedded as a string in each object file or assembly output.

If you specify option `sox` with no *keyword*, the compiler saves the compiler options and version number used in the compilation of the objects that make up the executable.

When you specify this option, the size of the executable on disk is increased slightly. Each *keyword* you specify increases the size of the executable. When you link the object files into an executable file, the linker places each of the information strings into the header of the executable. It is then possible to use a tool, such as a strings utility, to determine what options were used to build the executable file.

IDE Equivalent

None

Alternate Options

None

Example

The following commands are equivalent:

```
-sox=profile -sox=inline
-sox=profile,inline
```

You can use the negative form of the option to disable and reset the option. For example:

```
-sox=profile -no-sox -sox=inline      ! This means -sox=inline
```

See Also

`prof-use, Qprof-use` compiler option

sysroot

Specifies the root directory where headers and libraries are located.

Syntax

Linux OS:

`--sysroot=`*`dir`*

macOS:

`None`

Windows OS:

`None`

Arguments

dir	Specifies the local directory that contains copies of target libraries in the corresponding subdirectories.

Default

Off	The compiler uses default settings to search for headers and libraries.

Description

This option specifies the root directory where headers and libraries are located.

For example, if the headers and libraries are normally located in `/usr/include` and `/usr/lib` respectively, `--sysroot=/mydir` will cause the compiler to search in `/mydir/usr/include` and `/mydir/usr/lib` for the headers and libraries.

IDE Equivalent

None

Alternate Options

None

See Also

`print-sysroot` compiler option

Tf

Tells the compiler to compile the file as a Fortran source file.

Syntax

Linux OS and macOS:

`-Tf` *`filename`*

Windows OS:

`/Tf` *`filename`*

Arguments

`filename` Is the name of the file.

Default

OFF Files that do not end in standard Fortran file extensions are not compiled as Fortran files.

Description

This option tells the compiler to compile the file as a Fortran source file.

This option is useful when you have a Fortran file with a nonstandard file extension (that is, not one of .F, .FOR, or .F90).

This option assumes the file specified uses fixed source form. If the file uses free source form, you must also specify option `free`.

IDE Equivalent

None

Alternate Options

None

See Also

`extfor` compiler option

`free` compiler option

watch

Tells the compiler to display certain information to the console output window.

Syntax

Linux OS:

`-watch[=keyword[, keyword...]]`

`-nowatch`

macOS:

`-watch[=keyword[, keyword...]]`

`-nowatch`

Windows OS:

`/watch[:keyword[, keyword...]]`

`/nowatch`

Arguments

keyword Determines what information is displayed. Possible values are:

`none`	Disables `cmd` and `source`.
`[no]cmd`	Determines whether driver tool commands are displayed and executed.
`[no]source`	Determines whether the name of the file being compiled is displayed.
`all`	Enables `cmd` and `source`.

Default

`nowatch` Pass information and source file names are not displayed to the console output window.

Description

Tells the compiler to display processing information (pass information and source file names) to the console output window.

Option **watch***keyword*	Description
`none`	Tells the compiler to not display pass information and source file names to the console output window. This is the same as specifying `nowatch`.
`cmd`	Tells the compiler to display and execute driver tool commands.
`source`	Tells the compiler to display the name of the file being compiled.
`all`	Tells the compiler to display pass information and source file names to the console output window. This is the same as specifying `watch` with no *keyword*. For heterogeneous compilation, the tool commands for the host and the offload compilations will be displayed.

IDE Equivalent

None

Alternate Options

`watch cmd` Linux and macOS*: `-v`

Windows: None

See Also

`v` compiler option

what

Tells the compiler to display its detailed version string.

Syntax

Linux OS and macOS:

`-what`

Windows OS:

`/what`

Arguments

None

Default

OFF The version strings are not displayed.

Description

This option tells the compiler to display its detailed version string.

IDE Equivalent

None

Alternate Options

None

Alternate Compiler Options

This topic lists alternate names for compiler options and show the primary option name. Some of the alternate option names are deprecated and may be removed in future releases.

For more information on compiler options, see the detailed descriptions of the individual, primary options.

Some of these options are deprecated. For more information, see Deprecated and Removed Options.

Alternate Linux* and macOS* Options	Primary Option Name
Code Generation:	
`-fp`	-fomit-frame-pointer

Alternate Linux* and macOS* Options	Primary Option Name
-mcpu	-mtune
Advanced Optimizations:	
-funroll-loops	-unroll
Profile Guided Optimization (PGO):	
-pg	-p
-qp	-p
OpenMP* and Parallel Processing Options:	
-fopenmp	-qopenmp
Floating-Point:	
-mieee-fp	-fltconsistency
Output, Debug, Precompiled Header (PCH):	
-fvar-tracking	-debug variable-locations
-fvar-tracking-assignments	-debug semantic-stepping
-V	-logo
Preprocessor:	
-cpp	-fpp
-DD	-d-lines
-nodefine	-noD
-nostdinc	-X
-P	-preprocess-only
Language:	
-72	-extend-source 72
-80	-extend-source 80
-132	-extend-source 132
-C	-check all
-CB	-check bounds
-common-args	-assume dummy_aliases
-CU	-check uninit
-FI	-fixed
-FR	-free
-fsyntax-only	-syntax-only
-mixed-str-len-arg	No equivalent on Linux* or macOS* systems. On Windows* systems, /iface:mixed_str_len_arg

Alternate Linux* and macOS* Options	Primary Option Name
-nbs	-assume nobcss
-std	-stand f03
-std90	-stand f90
-std95	-stand f95
-std03	-stand f03
-y	-syntax-only
-Zp	-align recnbyte
Data:	
-autodouble	-real-size 64
-automatic	-auto
-i2	-integer-size 16
-i4	-integer-size 32
-i8	-integer-size 64
-r8	-real-size 64
-r16	-real-size 128
Compiler Diagnostics:	
-e90	-warn stderrors
-e95	
-e03	
-error-limit	-diag-error-limit
-implicitnone	-warn declarations
-u	-warn declarations
-w	-warn none or -warn nogeneral
-W0	-warn none or -warn nogeneral
-W1	-warn general
Compatibility:	
-66	-f66
-onetrip	-f66
Linking or Linker:	
-i-dynamic	-shared-intel
-i-static	-static-intel

Alternate Windows* Options	Primary Option Name
Optimization:	

Alternate Windows* Options	Primary Option Name
/Ox	/O
OpenMP* and Parallel Processing Options:	
/openmp	/Qopenmp
Floating Point:	
/QIfist	/Qrcd
Output, Debug, Precompiled Header (PCH):	
/compile-only	/c
/Fe	/exe
/Fo	/object
/nolink	/c
/pdbfile	/Fd
/V	/bintext
Preprocessor:	
/define	/D
/include	/I
/nodefine	/noD
/noinclude	/X
/undefine	/U
Language:	
/4L72	-extend-source:72
/4L80	-extend-source:80
/4L132	-extend-source:132
/4Naltparam	/noaltparam
/4Yaltparam	/altparam
/4Nf	/fixed
/4Yf	/free
/4Ns	/stand:none
/4Ys	/stand:f90
/C	/check:all
/CB	/check:bounds
/CU	/check:uninit
/FI	/fixed

Alternate Windows* Options	Primary Option Name
/FR	/free
/Gm	/iface:cvf
/Gz	/iface:stdcall
/nbs	/assume:nobcss
/Qcommon-args	/assume:dummy_aliases
/RTCu	/check:uninit
/Zp	/align:recnbyte
/Zs	/syntax-only
Data:	
/4I2	/integer-size:16
/4I4	/integer-size:32
/4I8	/integer-size:64
/4Na	/noauto
/4Ya	/auto
/4R8	/real-size:64
/4R16	/real-size:128
/automatic	/auto
/Qauto	/auto
/Qautodouble	/real-size:64
/Zp	/align:recnbyte
Compiler Diagnostics:	
/4Nd	/warn:nodeclarations
/4Yd	/warn:declarations
/error-limit	/Qdiag-error-limit
/w	/warn:none or /warn:nogeneral
/W0	/warn:none or /warn:nogeneral
/W1	/warn:general
Compatibility:	
/Qonetrip	/f66
Linking or Linker:	
/LD	/dll
/MG	/winapp
/MW	/libs

Alternate Windows* Options	Primary Option Name
/MWs	/libs:qwins
Miscellaneous:	
/V	/logo
/Zl	/libdir:none

Floating-Point Operations

Understanding Floating-Point Operations

Programming Tradeoffs in Floating-Point Applications

In general, the programming objectives for floating-point applications fall into the following categories:

- **Accuracy:** The application produces results that are close to the correct result.
- **Reproducibility and portability:** The application produces consistent results across different runs, different sets of build options, different compilers, different platforms, and different architectures.
- **Performance:** The application produces fast, efficient code.

Based on the goal of an application, you will need to make tradeoffs among these objectives. For example, if you are developing a 3D graphics engine, performance may be the most important factor to consider, with reproducibility and accuracy as secondary concerns.

The Intel® Fortran Compiler provides several compiler options that allow you to tune your applications based on specific objectives. Broadly speaking, there are the floating-point specific options, such as the `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) option, and the fast-but-low-accuracy options, such as the `[Q]imf-max-error` option. The compiler optimizes and generates code differently when you specify these different compiler options. Select appropriate compiler options by carefully balancing your programming objectives and making tradeoffs among these objectives. Some of these options may influence the choice of math routines that are invoked.

Many routines in the *libirc*, *libm*, and *svml* library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.

Using Floating-Point Options

Take the following code as an example:

Example

```
REAL(4):: t0, t1, t2
 ...
t0=t1+t2+4.0+0.1
```

If you specify the `-fp-model extended` (Linux* and macOS*) or `/fp:extended` (Windows*) option in favor of accuracy, the compiler generates the following assembly code:

```
fld       DWORD PTR _t1
fadd      DWORD PTR _t2
fadd      DWORD PTR _Cnst4.0
fadd      DWORD PTR _Cnst0.1
fstp      DWORD PTR _t0
```

This code maximizes accuracy because it utilizes the highest mantissa precision available on the target platform. The code performance might suffer when managing the x87 stack, and it might yield results that cannot be reproduced on other platforms that do not have an equivalent extended precision type.

If you specify the `-fp-model source` (Linux* and macOS*) or `/fp:source` (Windows*) option in favor of reproducibility and portability, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
addss     xmm0, DWORD PTR _Cnst4.0
addss     xmm0, DWORD PTR _Cnst0.1
movss     DWORD PTR _t0, xmm0
```

This code maximizes portability by preserving the original order of the computation, and by using the IEEE single-precision type for all computations. It is not as accurate as the previous implementation, because the intermediate rounding error is greater compared to extended precision. It is not the highest performance implementation, because it does not take advantage of the opportunity to pre-compute `4.0 + 0.1`.

If you specify the `-fp-model fast` (Linux* and macOS*) or `/fp:fast` (Windows*) option in favor of performance, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _Cnst4.1
addss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
movss     DWORD PTR _t0, xmm0
```

This code maximizes performance using Intel® Streaming SIMD Extensions (Intel® SSE) instructions and pre-computing `4.0 + 0.1`. It is not as accurate as the first implementation, due to the greater intermediate rounding error. It does not provide reproducible results like the second implementation, because it must reorder the addition to pre-compute `4.0 + 0.1`. All compilers, on all platforms, at all optimization levels do not reorder the addition in the same way.

For many other applications, the considerations may be more complicated.

Using Fast-But-Low-Accuracy Options

The fast-but-low-accuracy options provide an easy way to control the accuracy of mathematical functions and utilize performance/accuracy tradeoffs offered by the Intel® Math Kernel Library (Intel® MKL). You can specify accuracy, via a command line interface, for all math functions or a selected set of math functions at the level more precise than low, medium, or high.

You specify the accuracy requirements as a set of function attributes that the compiler uses for selecting an appropriate function implementation in the math libraries. Examples using the attribute, `max-error`, are presented here. For example, use the following option to specify the relative error of two ULPs for all single, double, long double, and quad precision functions:

```
-fimf-max-error=2
```

To specify twelve bits of accuracy for a `sin` function, use:

```
-fimf-accuracy-bits=12:sin
```

To specify relative error of ten ULPs for a `sin` function, and four ULPs for other math functions called in the source file you are compiling, use:

```
-fimf-max-error=10:sin-fimf-max-error=4
```

On Windows systems, the Intel® Fortran Compiler defines the default value for the `max-error` attribute depending on the `/fp` option and `/Qfast-transcendentals` settings. In `/fp:fast` mode, or if fast but less accurate math functions are explicitly enabled by `/Qfast-transcendentals-`, then the Intel® Fortran Compiler sets a `max-error=4.0` for the call. Otherwise, it sets a `max-error=0.6`.

Dispatching of Math Routines

The Intel® Fortran Compiler optimizes calls to routines from the *libm* and *svml* libraries into direct CPU-specific calls, when the compilation configuration specifies the target CPU where the code is tuned, and if the set of instructions available for the code compilation is not narrower than the set of instructions available in the tuning target CPU.

For example:

- The code containing calls to the *EXP()* library function and compiled with `-mtune=corei7-avx` (specifies tuning target CPU that supports Intel® Advanced Vector Extensions (Intel® AVX)) and `-QxCORE-AVX2/-march=core-avx2` (specifies Intel® Advanced Vector Extensions 2 (Intel® AVX2) instructions set) call the *EXP()* routine that is optimized for processors with Intel® AVX support. This code provides the best performance for these processors.
- The same code, compiled with `-mtune=core-avx2` and `-QxAVX/-march=corei7-avx`, calls a library dispatch routine that picks the optimal CPU specific version of the *EXP()* routine in runtime. Dispatching cannot be avoided because the instruction set does not allow the use of Intel® AVX2. Dynamic dispatching provides the best performance with the Intel® AVX2 CPU.

The dispatching optimization applies to the *EXP()* routine, and to the other math routines with CPU specific implementations in the libraries. The dispatching optimization can be disabled using the `-fimf-force-dynamic-target` (or Qimf-force-dynamic-target) option. This option specifies a list of math routines that are improved with a dynamic dispatcher. (See the Intel® Fortran Compiler documentation for syntax examples.)

See Also

Using -fp-model(/fp) Options

`fimf-max-error, Qimf-max-error` compiler option

Floating-point Optimizations

Application performance is an important goal of the Intel® Fortran Compiler, even at default optimization levels. A number of optimizations involve transformations that might affect the floating-point behavior of the application, such as evaluation of constant expressions at compile time, hoisting invariant expressions out of loops, or changes in the order of evaluation of expressions. These optimizations usually help the compiler to produce the most efficient code possible. However, the optimizations might be contrary to the floating-point requirements of the application.

Some optimizations are not consistent with strict interpretation of the ANSI or ISO standards for Fortran. Such optimizations can cause differences in rounding and small variations in floating-point results that may be more or less accurate than the ANSI-conformant result.

The Intel® Fortran Compiler provides the `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) option, which allows you to control the optimizations performed when you build an application. The option allows you to specify the compiler rules for:

- **Value safety:** Whether the compiler may perform transformations that could affect the result. For example, in the SAFE mode, the compiler won't transform `x/x` to 1.0 because the value of `x` at runtime might be a zero or a NaN . The UNSAFE mode is the default.
- **Floating-point expression evaluation:** How the compiler should handle the rounding of intermediate expressions.
- **Floating-point contractions:** Whether the compiler should generate fused multiply-add (FMA) instructions on processors that support them. When enabled, the compiler may generate FMA instructions for combining multiply and add operations; when disabled, the compiler must generate separate multiply and add instructions with intermediate rounding.
- **Floating-point environment access:** Whether the compiler must account for the possibility that the program might access the floating-point environment, either by changing the default floating-point control settings or by reading the floating-point status flags. This is disabled by default. You can use the `-fp-model:strict` (Linux* and macOS*) `/fp:strict` (Windows*) option to enable it.

- **Precise floating-point exceptions:** Whether the compiler should account for the possibility that floating-point operations might produce an exception. This is disabled by default. You can use `-fp-model:strict` (Linux* and macOS*) or `/fp:strict` (Windows*); or `-fp-model:except` (Linux* and macOS*) or `/fp:except` (Windows*) to enable it.

The following table describes the impact of different keywords of the option on compiler rules and optimizations:

Keyword	Value Safety	Floating-Point Expression Evaluation	Floating-Point Contractions	Floating-Point Environment Access	Precise Floating-Point Exceptions
`precise` `source`	Safe	Source Source	Yes	No	No
`strict`	Safe	Source	No	Yes	Yes
`consistent`	Safe	Source	No	No	No
`fast=1` (default)	Unsafe	Unknown	Yes	No	No
`fast=2`	Very unsafe	Unknown	Yes	No	No
`except` `except-`	Unaffected Unaffected	Unaffected Unaffected	Unaffected Unaffected	Unaffected Unaffected	Yes No

NOTE
It is illegal to specify the `except` keyword in an unsafe safety mode.

Based on the objectives of an application, you can choose to use different sets of compiler options and keywords to enable or disable certain optimizations, so that you can get the desired result.

See Also
Using -fp-model (/fp) Option

Using the -fp-model (/fp) Option

The `-fp-model` (Linux* and macOS*) or `/fp` (Windows*) option allows you to control the optimizations on floating-point data. You can use this option to tune the performance, level of accuracy, or result consistency for floating-point applications across platforms and optimization levels.

For applications that do not require support for subnormal numbers, the `-fp-model` or `/fp` option can be combined with the `[Q]ftz` option to flush subnormal results to zero in order to obtain improved runtime performance on processors based on all Intel® architectures.

You can use keywords to specify the semantics to be used. The keywords specified for this option may influence the choice of math routines that are invoked. Many routines in the *libirc*, *libm*, and *libsvml* libraries are more highly optimized for Intel microprocessors than for non-Intel microprocessors. Possible values of the keywords are as follows:

Keyword	Description
`precise`	Enables value-safe optimizations on floating-point data and rounds intermediate results to source-defined precision.
`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.

Keyword	Description
`consistent`	Enables consistent, reproducible results for different optimization levels or between different processors of the same architecture. This setting is equivalent to the use of the following options: Windows*: `/fp:precise /Qfma- /Qimf-arch-consistency:true` Linux* and macOS*: `-fp-model precise -no-fma -fimf-arch-consistency=true`
`strict`	Enables `precise` and `except`, disables contractions, and enables the property that allows modification of the floating-point environment.
`source`	Enables value-safe optimizations on floating-point data and rounds intermediate results to source-defined precision (same as precise keyword).
`[no-]except` (Linux* and macOS*) or `except[-]` (Windows*)	Determines whether strict floating-point exception semantics are used.

The default value of the option is `-fp-model fast=1` or `/fp:fast=1`, which means that the compiler uses more aggressive optimizations on floating-point calculations.

NOTE
Using the default option keyword `-fp-model fast` or `/fp:fast`, you may get significant differences in your result depending on whether the compiler uses x87 or SSE/AVX instructions to implement floating-point operations. Results are more consistent when the other option keywords are used.

Several examples are provided to illustrate the usage of the keywords. These examples show:

- A small example of source code.

NOTE
The same source code is considered in all the included examples.

- The semantics that are used to interpret floating-point calculations in the source code.
- One or more possible ways the compiler may interpret the source code.

NOTE
There are several ways the compiler may interpret the code; we show just some of these possibilities.

-fp-model fast or /fp:fast

Example source code:

Example

```
REAL T0, T1, T2;
...
T0 = 4.0E + 0.1E + T1 + T2;
```

When this option is specified, the compiler applies the following semantics:

- Additions may be performed in any order.
- Intermediate expressions may use `single, double,` or `extended double` precision.

- The constant addition may be pre-computed, assuming the default rounding mode.

Using these semantics, some possible ways the compiler may interpret the original code are given below:

Example

```
REAL T0, T1, T2;
  ...
T0 = (T1 + T2) + 4.1E;
```

```
REAL T0, T1, T2;
  ...
T0 = (T1 + 4.1E) + T2;
```

See Also

`fp-model, fp` compiler option

Subnormal Numbers

A normalized number is a number for which both the exponent (including bias) and the most significant bit of the mantissa are non-zero. For such numbers, all the bits of the mantissa contribute to the precision of the representation.

The smallest normalized single-precision floating-point number greater than zero is about 1.1754943^{-38}. Smaller numbers are possible, but those numbers must be represented with a zero exponent and a mantissa whose leading bit(s) are zero, which leads to a loss of precision. These numbers are called subnormal numbers or subnormals(older specifications refer to these as denormal numbers).

Subnormal computations use hardware and/or operating system resources to handle denormals; these can cost hundreds of clock cycles. Subnormal computations take much longer to calculate than normal computations.

There are several ways to avoid subnormals and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger range.
- Flush subnormals to zero.

See Also

Reducing Impact of Subnormal Exceptions

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Institute of Electrical and Electronics Engineers, Inc*. (IEEE) web site for information about the current floating-point standards and recommendations

Floating-Point Environment

The floating-point environment is a collection of registers that control the behavior of the floating-point machine instructions and indicate the current floating-point status. The floating-point environment can include rounding mode controls, exception masks, flush-to-zero (`FTZ`) controls, exception status flags, and other floating-point related features.

For example, bit 15 of the `MXCSR` register enables the flush-to-zero mode, which controls the masked response to an single-instruction multiple-data (SIMD) floating-point underflow condition.

The floating-point environment affects most floating-point operations; therefore, correct configuration to meet your specific needs is important. For example, the exception mask bits define which exceptional conditions will be raised as exceptions by the processor. In general, the default floating-point environment is set by the operating system. You don't need to configure the floating-point environment unless the default floating-point environment does not suit your needs.

There are several methods available if you want to modify the default floating-point environment. For example, you can use inline assembly, compiler built-in functions, library functions, or command line options.

Changing the default floating-point environment affects runtime results only. This does not affect any calculations that are pre-computed at compile time.

See Also

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Setting the FTZ and DAZ Flags

In Intel® processors, the flush-to-zero (`FTZ`) and subnormals-are-zero (`DAZ`) flags in the `MXCSR` register are used to control floating-point calculations. Intel® Streaming SIMD Extensions (Intel® SSE) and Intel® Advanced Vector Extensions (Intel® AVX) instructions, including scalar and vector instructions, benefit from enabling the `FTZ` and `DAZ` flags. Floating-point computations using the Intel® SSE and Intel® AVX instructions are accelerated when the `FTZ` and `DAZ` flags are enabled. This improves the application's performance.

Use the `[Q]ftz` option to flush subnormal results to zero when the application is in the gradual underflow mode. This option may improve performance if the subnormal values are not critical to the application's behavior. The `[Q]ftz` option, when applied to the main program, sets the `FTZ` and the `DAZ` hardware flags. The negative forms of the `[Q]ftz` option (`-no-ftz` for Linux* and macOS*, and `/Qftz-` for Windows*) leave the flags as they are.

The following table describes how the compiler processes subnormal values based on the status of the `FTZ` and `DAZ` flags:

Flag	When set to ON, the compiler...	When set to OFF, the compiler...	Supported on
`FTZ`	...sets subnormal results from floating-point calculations to zero.	...does not change the subnormal results.	Intel® 64 and some IA-32 architectures
`DAZ`	...treats subnormal values used as input to floating-point instructions as zero.	...does not change the subnormal instruction inputs.	Intel® 64 and some IA-32 architectures

- `FTZ` and `DAZ` are not supported on all IA-32 architectures. The `FTZ` flag is supported only on IA-32 architectures that support Intel® SSE instructions.
- On systems based on the IA-32 and Intel® 64 architectures, `FTZ` only applies to Intel® SSE and Intel® AVX instructions. If the application generates subnormals using x87 instructions, `FTZ` does not apply.
- `DAZ` and `FTZ` flags are not compatible with the IEEE 754 standard, and should only be enabled when compliance to the IEEE standard is not required.

Options for `[Q]ftz` are performance options. Setting these options does not guarantee that all subnormals in a program are flushed to zero. They only cause subnormals generated at run-time to be flushed to zero.

On Intel® 64 and IA-32 systems, the compiler, by default, inserts code into the main routine to set the `FTZ` and `DAZ` flags. When the `[Q]ftz` option is used on IA-32 systems with the option `-msse2` or `/arch:sse2`, the compiler inserts code to that conditionally sets the `FTZ`/`DAZ` flags based on a run-time processor check. Using the negative form of `[Q]ftz` prevents the compiler from inserting any code that sets `FTZ` or `DAZ` flags.

When the `[Q]ftz` option is used in combination with an Intel® SSE-enabling option on systems based on the IA-32 architecture (for example, `-msse2` or `/arch:sse2`), the compiler inserts code in the main routine to set `FTZ` and `DAZ`. When the option `[Q]ftz` is used without an Intel® SSE-enabling option, the compiler inserts code that conditionally sets `FTZ` or `DAZ` based on a run-time processor check. The negative form of `[Q]ftz` prevents the compiler from inserting any code that might set `FTZ` or `DAZ`.

The `[Q]ftz` option only has an effect when the main program is being compiled. It sets the `FTZ`/`DAZ` mode for the process. The initial thread, and any subsequently created threads, operate in the `FTZ`/`DAZ` mode.

On systems based on Intel® 64 and IA-32 architectures, every optimization option `O` level, except `O0`, sets `[Q]ftz`.

If this option produces undesirable results of the numerical behavior of the program, turn the FTZ/DAZ mode off by using the negative form of [Q]ftz in the command line while still benefitting from the O3 optimizations.

Manually set the flags by calling the following Intel® Fortran intrinsic:

Example
`RESULT = FOR_SET_FPE (FOR_M_ABRUPT_UND)`

See Also

ftz, Qftz compiler option

Checking the Floating-point Stack State

On systems based on the IA-32 architecture, when an application calls a function that returns a floating-point value, the returned floating-point value is supposed to be on the top of the floating-point stack. If the return value is not used, the compiler must pop the value off of the floating-point stack in order to keep the floating-point stack in the correct state.

On systems based on Intel® 64 architecture, floating-point values are usually returned in the xmm0 register. The floating-point stack is used only when the return value is an internal 80-bit floating-point data type on Linux* and macOS* systems.

If the application calls a function without defining or incorrectly defining the function's prototype, the compiler cannot determine if the function must return a floating-point value. Consequently, the return value is not popped off the floating-point stack if it is not used. This can cause the floating-point stack to overflow.

The overflow of the stack results in two undesirable situations:

- A NaN value gets involved in the floating-point calculations
- The program results become unpredictable; the point where the program starts making errors can be arbitrarily far away from the point of the actual error.

For systems based on the IA-32 and Intel® 64 architectures, the [Q]fp-stack-check option checks whether a program makes a correct call to a function that should return a floating-point value. If an incorrect call is detected, the option places a code that marks the incorrect call in the program. The [Q]fp-stack-check option marks the incorrect call and makes it easy to find the error.

> **NOTE**
> The [Q]fp-stack-check option causes significant code generation after every function/subroutine call to ensure that the floating-point stack is maintained in the correct state. Therefore, using this option slows down the program being compiled. Use the option only as a debugging aid to find floating point stack underflow/overflow problems, which can be otherwise hard to find.

See Also

fp-stack-check, Qfp-stack-check compiler option

Tuning Performance

Overview: Tuning Performance

This section describes several programming guidelines that can help you improve the performance of floating-point applications:

- Avoid exceeding representable ranges during computation; handling these cases can have a performance impact.

- Use REAL variables in single precision format unless the extra precision obtained through `DOUBLE` or `REAL*8` is required because a larger precision formation will also increase memory size and bandwidth requirements. See Using Efficient Data Types section.
- Reduce the impact of subnormal exceptions for all supported architectures.
- Avoid mixed data type arithmetic expressions.

See Also

Avoiding Mixed Data Type Arithmetic Expressions
Reducing the Impact of Subnormal Exceptions
Using Efficient Data Types

Handling Floating-point Array Operations in a Loop Body

Following the guidelines below will help auto-vectorization of the loop.

- Statements within the loop body may contain float or double operations (typically on arrays). The following arithmetic operations are supported: addition, subtraction, multiplication, division, negation, square root, `MAX`, `MIN`, and mathematical functions such as `SIN` and `COS`.
- Writing to a single-precision scalar/array and a double scalar/array within the same loop decreases the chance of auto-vectorization due to the differences in the vector length (that is, the number of elements in the vector register) between float and double types. If auto-vectorization fails, try to avoid using mixed data types.

See Also

Programming Guidelines for Vectorization

Reducing the Impact of Subnormal Exceptions

Subnormal floating-point values are those that are too small to be represented in the normal manner; that is, the mantissa cannot be left-justified. Subnormal values require hardware or operating system interventions to handle the computation, so floating-point computations that result in subnormal values may have an adverse impact on performance.

There are several ways to handle subnormals to increase the performance of your application:

- Scale the values into the normalized range
- Use a higher precision data type with a larger range
- Flush subnormals to zero

For example, you can translate them to normalized numbers by multiplying them using a large scalar number, doing the remaining computations in the normal space, then scaling back down to the subnormal range. Consider using this method when the small subnormal values benefit the program design.

If you change the type declaration of a variable, you might also need to change associated library calls, unless these are generic; . Another strategy that might result in increased performance is to increase the amount of precision of intermediate values using the `-fp-model [double|extended]` option. However, this strategy might not eliminate all subnormal exceptions, so you must experiment with the performance of your application. You should verify that the gain in performance from eliminating subnormals is greater than the overhead of using a data type with higher precision and greater dynamic range.

In many cases, subnormal numbers can be treated safely as zero without adverse effects on program results. Depending on the target architecture, use flush-to-zero (`FTZ`) options.

IA-32 and Intel® 64 Architectures

These architectures take advantage of the `FTZ` (flush-to-zero) and `DAZ` (subnormals-are-zero) capabilities of Intel® Streaming SIMD Extensions (Intel® SSE) instructions.

By default, the Intel® Fortran Compiler inserts code into the main routine to enable `FTZ` and `DAZ` at optimization levels higher than `O0`. To enable `FTZ` and `DAZ` at `O0`, compile the source file containing PROGRAM using compiler option `[Q]ftz`. When the `[Q]ftz` option is used on IA-32-based systems with the option `-mia32` (Linux*) or `/arch:IA32` (Windows*), the compiler inserts code to conditionally enable `FTZ` and `DAZ` flags based on a run-time processor check. IA-32 is not available on macOS*.

NOTE
After using flush-to-zero, ensure that your program still gives correct results when treating subnormal values as zero.

See Also

Setting the FTZ and DAZ Flags

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Avoiding Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (REAL) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that `I` and `J` are both INTEGER variables, expressing a constant number (2.0) as an integer value (2) eliminates the need to convert the data. The following examples demonstrate inefficient and efficient code.

Examples

Inefficient Code Example

```
INTEGER I, J
  I = J / 2.0
```

Efficient Code Example

```
INTEGER I, J
  I = J / 2
```

Special Considerations for Auto-Vectorization of the Innermost Loops

Auto-vectorization of an innermost loop packs multiple data elements from consecutive loop iterations into a vector register, each of which is 128-bit (SSE) or 256 bit (AVX) in size.

Consider a loop that uses different sized data, for example, REAL and DOUBLE PRECISION. For REAL data, the compiler tries to pack data elements from four (SSE) or eight (AVX) consecutive iterations (32 bits x 4 = 128 bits, 32 bits x 8 = 256 bits). For DOUBLE PRECISION data, the compiler tries to pack data elements from two (SSE) or four (AVX) consecutive iterations (64 bits x 2 = 128 bits, 64 bits x 4 = 256 bits). Because of the mismatched number of iterations, the compiler sometimes fails to perform auto-vectorization of the loop, after trying to automatically remedy the situation.

If your attempt to auto-vectorize an innermost loop fails, it is a good practice to try using the same sized data. INTEGER and REAL are considered same sized data since both are 32-bit in size.

Examples

Example 1: Not auto-vectorizable code

```
DOUBLE PRECISION A(N), B(N)
REAL C(N), D(N)
DO I=1, N
   A(I)=D(I)
   C(I)=B(I)
ENDDO
```

Example 2: Auto-vectorizable after automatic distribution into two loops

```
DOUBLE PRECISION A(N), B(N)
REAL C(N), D(N)
DO I=1, N
   A(I)=B(I)
   C(I)=D(I)
ENDDO
```

Example 3: Auto-vectorizable as one loop

```
REAL A(N), B(N)
REAL C(N), D(N)
DO I=1, N
   A(I)=B(I)
   C(I)=D(I)
ENDDO
```

Using Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- Integer
- Single-precision real, expressed explicitly as `REAL`, `REAL (KIND=4)`, or `REAL*4`
- Double-precision real, expressed explicitly as `DOUBLE PRECISION`, `REAL (KIND=8)`, or `REAL*8`
- Extended-precision real, expressed explicitly as `REAL (KIND=16)` or `REAL*16`

NOTE
In an arithmetic expression, you should avoid mixing integer and floating-point data.

Libraries

Creating Static Libraries

Executables generated using static libraries are no different than executables generated from individual source or object files. Because static libraries are not required at runtime, you do not need to include them when you distribute your executable. Linking to a static library is generally faster than linking to individual object files.

When building objects for a static library from the `ifort` command line, include option `c` to suppress linking. Without this option, the linker will run and generate an error because the object is not a complete program.

Building a static library (Linux*)

1. Use the `c` option to generate object files from the source files:

```
ifort -c my_source1.f90 my_source2.f90 my_source3.f90
```

2. Use the Intel® `xiar` tool to create the library file from the object files:

```
xiar rc my_lib.a my_source1.o my_source2.o my_source3.o
```

3. Compile and link your project with your new library:

```
ifort main.f90 my_lib.a
```

If your library file and source files are in different directories, use the `-Ldir` option to indicate where your library is located:

```
ifort -L/for/libs main.f90 my_lib.a
```

Building a static library (macOS*)

1. Use the following command line to generate object files and create the library file:

```
ifort -o my_lib.a -staticlib mysource1.f90 mysource2.f90 mysource3.f90
```

2. Compile and link your project with your new library:

```
ifort main.f90 my_lib.a
```

If your library file and source files are in different directories, use the `-Ldir` option to indicate where your library is located:

```
ifort -L/for/libs main.f90 my_lib.a
```

Building a static library (Windows*)

To build a static library from the integrated development environment (IDE), select the **Fortran Static Library** project type.

To build a static library using the command line:

1. Use option `c` to generate object files from the source files:

```
ifort /c my_source1.f90 my_source2.f90
```

2. Use the Intel® `xilib` tool to create the library file from the object files:

```
xilib /out:my_lib.lib my_source1.obj my_source2.obj
```

3. Compile and link your project with your new library:

```
ifort main.f90 my_lib.lib
```

Creating Shared Libraries

This topic applies to Linux* and macOS*.

Shared libraries, also referred to as dynamic libraries, are linked differently than static libraries. At compile time, the linker insures that all the necessary symbols are either linked into the executable, or can be linked at runtime from the shared library. Executables compiled from shared libraries are smaller, but the shared libraries must be included with the executable to function correctly. When multiple programs use the same shared library, only one copy of the library is required in memory.

To create a shared library from a Fortran source file, process the files using the `ifort` command:

- You must specify the `-shared` option (Linux*) or the `-dynamiclib` option (macOS*) to create the `.so` or `.dylib` file. On Linux* and macOS* operating systems using either IA-32 architecture or Intel® 64 architecture, you must also specify option `-fpic` for the compilation of each object file you want to include in the shared library.
- You can specify the `-o`*output* option to name the output file.
- If you omit the `-c` option, you will create a shared library (`.so` file) directly from the command line in a single step.
- If you also omit the `-o`*output* option, the file name of the first Fortran file on the command line is used to create the file name of the `.so` file. You can specify additional options associated with shared library creation.
- If you specify the `-c` option, you will create an object file (`.o` file) that you can name with the `-o` option. To create a shared library, process the `.o` file with `ld` , specifying certain options associated with shared library creation.

Creating a Shared Library

There are several ways to create a shared library.

You can create a shared library file with a single `ifort` command:

```
ifort -shared -fpic octagon.f90 (Linux*)
ifort -dynamiclib octagon.f90 (macOS*)
```

The `-shared` or `-dynamiclib` option is required to create a shared library. The name of the source file is `octagon.f90`. You can specify multiple source files and object files.

Because the `-o` option was omitted, the name of the shared library file is `octagon.so` (Linux*) or `octagon.dylib` (macOS*).

You can use the `-static-intel` option to force the linker to use the static versions of the Intel-supplied libraries.

You can also create a shared library file with a combination of `ifort` and `ld` (Linux*) or `libtool` (macOS*) commands:

First, create the `.o` file, such as `octagon.o` in the following example:

```
ifort -c -fpic octagon.f90
```

The file `octagon.o` is then used as input to the `ld` (Linux*) or `libtool` (macOS*) command to create the shared library. The following example shows the command to create a shared library named `octagon.so` on a Linux* operating system:

```
ld -shared octagon.o \
     -lifport -lifcoremt -limf -lm -lcxa \
     -lpthread -lirc -lunwind -lc -lirc_s
```

Note the following:

- When you use `ld`, you need to list all Intel® Fortran libraries. It is easier and safer to use the `ifort` command. On macOS*, you would use `libtool`.
- The `-shared` option is required to create a shared library. On macOS*, use the `-dynamiclib` option, and also specify the following: `-arch_only i386`, `-noall_load`, `-weak_references_mismatches non-weak`.
- The name of the object file is `octagon.o`. You can specify multiple object (`.o`) files.
- The `-lifport` option and subsequent options are the standard list of libraries that the `ifort` command would have otherwise passed to `ld` or `libtool`. When you create a shared library, all symbols must be resolved.

It is recommended that you review the output of the `-dryrun` command to find the names of all the libraries used, to be able to specify them correctly.

If you are using the `ifort` command to link, you can use the `-Qoption` command to pass options to the `ld` linker. (You cannot use `-Qoption` on the `ld` command line.)

For more information on relevant compiler options, see the Compiler Options reference.

See also the `ld(1)` reference page.

Shared Library Restrictions

When creating a shared library with `ld`, be aware of the following restrictions:

- Shared libraries must not be linked with archive libraries.

  When creating a shared library, you can only depend on other shared libraries for resolving external references. If you need to reference a routine that currently resides in an archive library, put that routine in a separate shared library or include it in the shared library being created. You can specify multiple object (`.o`) files when creating a shared library.

  To put a routine in a separate shared library, obtain the source or object file for that routine, recompile if necessary, and create a separate shared library. You can specify an object file when recompiling with the `ifort` command or when creating the shared library with the `ld` command.

  To include a routine in the shared library being created, put the routine (source or object file) with other source files that make up the shared library and recompile if necessary.

  Now create the shared library, and specify the file containing that routine either during recompilation or when creating the shared library. You can specify an object file when recompiling with the `ifort` command or when creating the shared library with the `ld` or `libtool` command.
- When creating shared libraries, all symbols must be defined (resolved).

  Because all symbols must be defined to `ld` when you create a shared library, you must specify the shared libraries on the `ld` command line, including all standard Intel® Fortran libraries. The list of standard Intel® Fortran libraries can be specified by using the `-l`*`string`* option.

Installing Shared Libraries

Once the shared library is created, it must be installed for private or system-wide use before you run a program that refers to it:

- To install a *private* shared library (when you are testing, for example), set the environment variable `LD_LIBRARY_PATH`, as described in `ld(1)`. For macOS*, set the environment variable `DYLD_LIBRARY_PATH`.
- To install a *system-wide* shared library, place the shared library file in one of the standard directory paths used by `ld` or `libtool`.

Using Shared Libraries on macOS*

On macOS*, it is possible to store path information in shared libraries to perform library searches. The compiler installation changes the path to the installation directory, but you will need to modify these paths if you move the libraries elsewhere. For example, you may want to bundle redistributable Intel libraries with your application, which eliminates the dependency on libraries found on `DYLD_LIBRARY_PATH`.

If your compilations do not use `DYLD_LIBRARY_PATH` to find libraries, and you distribute executables that depend on shared libraries, then you will need to modify the Intel shared libraries (`../lib/*.dylib`) using the `install_name_tool` to set the correct path to the shared libraries. This also permits the end-user to launch the application by double-clicking on the executable. The code below will modify each library with the correct absolute path information:

```
for i in *.dylib
do
echo -change $i `pwd`/$i
done > changes
for i in *.dylib
do
install_name_tool `cat changes` $i
done
```

You can also use the `install_name_tool` command to set `@executable_path` to change path information for the libraries bundled in your application by changing the path appropriately.

Be sure to recompile your sources after modifying the libraries.

Calling Library Routines

The following tables show Intel® Fortran library routine groups and the USE statement required to include the interface definitions for the routines in that group:

Routines	USE statement
Portability	USE IFPORT
POSIX*	USE IFPOSIX
Miscellaneous Run-Time	USE IFCORE

The following are Windows only:

Routines	USE statement
Automation (AUTO) (systems using IA-32 architecture only)	USE IFAUTO
Component Object Model (COM) (systems using IA-32 architecture only)	USE IFCOM
Dialog (systems using IA-32 architecture only)	USE IFLOGM
Graphics	USE IFQWIN
National Language Support	USE IFNLS
QuickWin	USE IFQWIN
Serial port I/O (SPORT)(systems using IA-32 architecture only)	USE IFPORT

Language Reference section *Run-Time Library Routines* lists topics that provide an overview of the different groups of library routines as well as calling syntax for the routines. For example, add the following USE statement before any data declaration statements, such as IMPLICIT, NONE, or INTEGER:

```
USE IFPORT
```

If you want to minimize compile time for source files that use the Intel® Fortran library routines, add the ONLY keyword to the USE statement. For example:

```
USE IFPORT, only: getenv
```

Using the ONLY keyword limits the number of interfaces for that group of library routines.

To view the actual interface definitions, view the `.f90` file that corresponds to the `.mod` file. For example, if a routine requires a `USE IFCORE`, locate and use a text editor to view the file `ifcore.f90` in the standard INCLUDE directory.

You should avoid copying the actual interface definitions contained in the `ifport.f90` (or `ifcore.f90`, ...) into your program because future versions of Intel® Fortran might change these interface definitions.

Similarly, some of the library interface `.f90` files contain `USE` statements for a subgrouping of routines. However, if you specify a USE statement for such a subgroup, this module name may change in future version of the Intel® Fortran compiler. Although this will make compilation times faster, it might not be compatible with future versions of the compiler.

See Also

Run-Time Library Routines

Comparison of Intel® Visual Fortran and Windows* API Routines

This topic only applies to Windows*.

Intel® Visual Fortran provides Fortran language elements (such as intrinsic procedures and statements) that conform to the Fortran Standard. The Intel® Visual Fortran Compiler also provides language elements that are language extensions, including library routines.

The library routines provided by the Intel® Visual Fortran Compiler:

- Are intended to be called from the Fortran language. For example, character arguments are assumed to be Fortran character variables, not null-terminated C strings.
- May have `QQ` appended at the end of their names to differentiate them from equivalent Windows* routines.
- Are described in the Intel® Visual Fortran Compiler *Language Reference* online documentation, in the A to Z Reference. The routine description lists the appropriate USE statement needed to include the interface definitions, such as USE IFPORT.
- Call appropriate Windows* system API routines provided with the operating system.
- Are specific to Windows* systems, with one exception: most of the Intel® Fortran IFPORT Portability Library routines exist on most Linux* systems.

In contrast, the Windows* API routines provided by the Windows* operating system:

- Are intended to be called from the C language. For example, character arguments are assumed to be null-terminated C strings.
- Often have multiple words appended together in their names, such as `GetSystemTime`.
- Are described in the Windows* SDK online documentation. To look up a specific routine, visit the MSDN Library website. The calling format is listed near the top of the page, followed by a description of the routine's arguments. The QuickInfo at the bottom of the Windows* API routine documentation page lists the import library needed.
- Are also specific to Windows*.

Note that the Intel® Visual Fortran Compiler provides interface block definitions that simplify calling Windows* API routines from the Intel® Visual Fortran Compiler (such as allowing you to specify Fortran data arguments as being passed by reference). To obtain these interface definitions, add the USE IFWIN statement to your program.

There are many groups of Windows* routines (see the Platform SDK and other resources). These routines provide sophisticated window management, memory management, graphics support, threading, security, and networking.

You can access many Windows* API routines from any Fortran application, including Fortran Console and Fortran QuickWin applications. Only the Fortran Windows* application Understanding Project Types provides access to the full set of Win32 routines needed to create GUI applications.

Fortran Console applications are text-only applications. Fortran QuickWin applications allow you to build Windows*-style applications easily, but access only a small subset of the available Windows* API features. Fortran QuickWin applications also allow you to use graphics.

See Also

A to Z Reference
IFPORT Portability Library
Understanding Project Types

Specifying Consistent Library Types

This topic only applies to Windows*.

There are a number of Visual C++* run-time libraries that offer the same entry points but have different characteristics. The default Visual C++* library is `libc.lib`, which is single-threaded, non-debug, and static.

The Intel® Visual Fortran and Microsoft* Visual C++* libraries must be the same types. The incompatible types are listed below.

- Mixing single-threaded with multi-threaded versions of the libraries.
- Mixing static and dynamic-link versions of the libraries.
- Mixing debug with non-debug versions of the libraries.

The default Intel® Visual Fortran libraries depend on the project type:

Fortran Project Type	Default Libraries Used
Fortran Console	Static, single-threaded libraries `ifcore.lib` and `libc.lib`
Fortran Standard Graphics	Static, multithreaded libraries `ifcoremt.lib` and `libcmt.lib`
Fortran QuickWin	Static, multithreaded libraries `ifcoremt.lib` and `libcmt.lib`
Fortran Windows	Static, multithreaded libraries `ifcoremt.lib` and `libcmt.lib`
Fortran DLL	Dynamic-link libraries ifcoremd and msvcrt (and their import libraries)

Pure Fortran applications can have mismatched types of libraries. One common scenario is a Fortran QuickWin application that links with a Fortran Static library. Fortran QuickWin (and Fortran Standard Graphics) applications must use the static, multithreaded libraries, and by default, Fortran Static libraries are built using static, single-threaded libraries. Because this causes a conflict, the Fortran Static library and the QuickWin application must both be built using static, multithreaded libraries.

Similarly, different C/C++ applications link against different C libraries. If you mix the different types of applications without modifying the defaults, you can get conflicts. The debug version of a library has a letter *d* appended to its base file name:

- Static, multithreaded: `libcmt.lib` and `libcmtd.lib`
- Dynamic-link libraries: `msvcrt` and `msvcrtd` (`.lib` import library and `.dll`)

When using a **Debug** configuration, Visual C++* selects the debug libraries.

The Intel® Visual Fortran Compiler does not select debug libraries for any configuration, but provides settings that allow you to request their use. To specify different types of Fortran libraries in the IDE, select **Project** > **Properties**, then select the **Libraries category** on the **Fortran** tab:

- To specify static libraries, select the appropriate type of static (non-DLL) library under Run-Time Library (see `/libs:static`).
- To specify dynamic-link libraries, select the appropriate type of DLL library under Run-Time Library (see `/libs:dll`).

- To specify the debug libraries, select the appropriate type of Debug library under Run-Time Library (see `/[no]dbglibs`). If you specify debug libraries (`/dbglibs`) and also request DLL libraries (`/libs:dll`), be aware that this combination selects the debug versions of the Fortran DLLs. These Fortran DLL files have been linked against the C debug DLLs.
- When you specify QuickWin or Standard Graphics libraries under Run-Time Library, this selection implicitly requests multithreaded libraries.

> **NOTE** This option does not cause static linking of libraries for which no static version is available, such as the OpenMP run-time libraries on Windows* or the coarray run-time libraries. These libraries can only be linked dynamically.

See also Building Intel® Fortran C Mixed-Language Programs on Windows* Systems.

See Also

Building Intel® Fortran C Mixed-Language Programs on Windows* Systems

Redistributing Libraries When Deploying Applications

When you deploy your application to systems that do not have the Intel® Fortran Compiler installed, you need to redistribute certain Intel® libraries to which your application is linked. You can do so in one of the following ways:

- Statically link your application.

  An application built with statically-linked libraries eliminates the need to distribute runtime libraries with the application executable. By linking the application to the static libraries, you are not dependent on the Intel® Fortran or Intel® C/C++ dynamic shared libraries.
- Dynamically link your application.

  If you must build your application with dynamically linked (or shared) compiler libraries, you should address the following concerns:

  - You must build your application with shared or dynamic libraries that are redistributable.
  - Note the directory where the redistributables are installed and how the OS finds them.
  - You should determine which shared or dynamic libraries your application needs.

The information here is only introductory. See the Intel® Developer Zone articles at https://software.intel.com (search for "redistributable libraries") for everything you need to know to redistribute these libraries.

Redistribution of Application Binaries Built for Microsoft Windows**https://software.intel.com/en-us/articles/redistribution-of-application-binaries-built-for-microsoft-windows

*Redistribution of Application Binaries Built for Linux** (https://software.intel.com/en-us/articles/redistribution-of-application-binaries-built-for-linux

*Redistribution of Application Binaries Built for macOS**(https://software.intel.com/en-us/articles/redistribution-of-application-binaries-built-for-mac-os-x

The redistributable library installation package is available at: https://software.intel.com/en-us/articles/intelr-composer-redistributable-libraries-by-version

Storing Object Code in Static Libraries

Another way to organize source code used by several projects is to build a static library (for Windows*, `.lib` and for Linux* and macOS*, `.a`) containing the object files for the reused procedures. You can create a static library by doing the following:

- From the Microsoft* Visual Studio* integrated development environment (IDE), create and build a Fortran Static Library project type.

- From the command line, use the `xiar` command (on Linux* and macOS*) or the `xilib` command (Windows*).

After you have created a static library, you can use it as input to other types of Intel® Fortran Compiler projects.

See Also

Using Fortran Static Library Projects

Storing Routines in Shareable Libraries

You can organize the code in your application by storing the executable code for certain routines in a shareable library (`.dll` for Windows*, `.so` for Linux*, or `.dylib` for macOS*). You can then build your applications so that they call these routines from the shareable library.

When routines in a shareable library are called, the routines are loaded into memory at run-time as they are needed. This is most useful when several applications use a common group of routines. By storing these common routines in a shareable library, you reduce the size of each application that calls the library. In addition, you can update the routines in the library without having to rebuild any of the applications that call the library.

Using the Windows* API Routines

Including the Intel® Visual Fortran Interface Definitions for Windows* API Routines

This topic only applies to Windows*.

Intel® Fortran provides interface definitions for both the Intel Fortran library routines and most of the Windows* API routines in the standard include directory. Documentation on the Windows API can be found at msdn.microsoft.com.

To include the Windows API interface definitions, do one of the following:

1. Add the statement USE IFWIN to include all Windows API routine definitions.

   The USE IFWIN statement makes all parameters and interfaces for most Windows routines available to your Intel Fortran program. Any program or subprogram that uses the Windows features can include the statement USE IFWIN, which is needed in each subprogram that makes Windows API calls.

2. Add the USE statement before any declaration statements (such as IMPLICIT NONE or INTEGER). You can limit the type of parameters and interfaces for Windows applications to make compilation times faster. To do this, include only the subsets of the Windows API needed in multiple USE statements (see the file `...\INCLUDE\IFWIN.F90`).

   To locate the specific libraries for the routines being called, view the QuickInfo at the bottom of the routine page in the Windows API documentation at msdn.microsoft.com, which lists the import library name. For example, to call only GetSystemTime, you need the interface definitions provided in kernel32.mod (binary form). To do this, add the following USE statement:

```
USE KERNEL32
```

If you want to further minimize compile time, add the ONLY keyword to the USE statement. For example:

```
USE KERNEL32, only: GetSystemTime, GetLastError
```

Calling Windows API Routines

This topic describes general information about calling Windows API routines from Intel® Fortran applications. It contains the following information:

- Calling Windows API Routines Using the Intel® Visual Fortran Interface Definitions

- Understanding Data Types Differences

Calling Windows API Routines Using the Intel® Visual Fortran Interface Definitions

To call the appropriate Windows API routine after including the Intel® Visual Fortran interface definitions, follow these guidelines:

1. Examine the documentation for the appropriate routine in the Windows API (for example, GetSystemTime) to obtain the following information:
   - The number and order of arguments. You will need to declare and initialize each argument variable using the correct data type. In the case of the GetSystemTime routine, a structure (derived type) is passed by reference.

     If character arguments are present, add a null character as a terminator to the character variable before calling the Windows API routine.
   - Whether the routine returns a result (function) or not (subroutine). For example, because the GetSystemTime routine calling format starts with VOID, this routine should be called as a subroutine with a `CALL` statement.
2. If you are not sure about the data type of arguments or its function return value, you can examine the interface definitions in the appropriate .F90 file in `...\INCLUDE\`. For example, to view the interface definition for GetSystemTime:
   - In a text editor (such as Notepad), open the file kernel32.f90 from `...\INCLUDE\`.
   - Search (or Find) the routine name (such as GetSystemTime)
   - View the interface definition and compare it to the description in the Windows API documentation to see how the arguments are represented in Fortran. Take note of how arguments are passed; in some cases, such as when an arbitrary data type is passed by address, you must pass the address of a variable using the LOC intrinsic rather than the variable directly.
3. If one of the arguments is a structure, look up the definition in IFWINTY.F90 in `...\INCLUDE\`. For example, to view the data type definition for the T_SYSTEMTIME type used in GetSystemTime:
   - In a text editor (such as Notepad), open the file IFWINTY.F90 from `...\INCLUDE\`.
   - Search (or Find) the data type name (such as T_SYSTEMTIME).
   - View the data type definition and note the field names. In some cases, these will differ slightly from those listed in the Windows API documentation.
   - Define a variable name to use the derived-type definition in your program, such as:

```
TYPE (T_SYSTEMTIME) MYTIME
```

4. Many Windows API routines have an argument or return a value described as a "`handle`". This is generally an address-sized integer and must be declared using an appropriate `KIND` value, typically `HANDLE`, which automatically provides the correct value for 32- and 64-bit platforms. For example:

```
Integer(HANDLE) :: hwnd
```

Use the variable definition to call the Win32 routine. For example, the completed program follows:

```
! Getsystime.f90 file shows how to call a Windows API routine
! Since the only routine called is GetSystemTime, only include
! interface definitions from kernel32.mod instead of all modules
! included by ifwin.f90. Type definitions are defined in IFWINTY,
! which is used within KERNEL32.
PROGRAM Getsystime
USE KERNEL32
TYPE (T_SYSTEMTIME) MYTIME
CALL GetSystemTime(MYTIME)
WRITE (*,*) 'Current UTC time hour and minute:', Mytime.wHour, Mytime.Wminute
END PROGRAM
```

You can create a new Fortran Console (or QuickWin) application project, add the code shown above as a source file, build it, and view the result.

Understanding Data Type Differences

Module IFWINTY, which is used by IFWIN and the other Win32 API modules, defines a set of constants for INTEGER and REAL kinds that correspond to many of the type definitions provided in the Windows WINDOWS.H header file. Use these kind values in INTEGER and REAL declarations. The following table gives the correspondence of some of the more common Windows types:

Windows Data Type	Equivalent Fortran Data Type
BOOL, BOOLEAN	INTEGER(BOOL)
BYTE	INTEGER(BYTE)
CHAR, CCHAR, UCHAR	CHARACTER or INTEGER(UCHAR)
DWORD	INTEGER(DWORD)
ULONG	INTEGER(ULONG)
SHORT	INTEGER(SHORT)
LPHANDLE	INTEGER(LPHANDLE)
PLONG	INTEGER(PLONG)
DOUBLE	REAL(DOUBLE)

Use the kind constants instead of explicitly specifying the kind as a number, or assuming a default kind.

Note that the Windows BOOL type is not equivalent to Fortran LOGICAL and should not be used with Fortran LOGICAL operators and literal constants. Use the constants TRUE and FALSE, defined in IFWINTY, rather than the Fortran literals .TRUE. and .FALSE., and do not test BOOL values using LOGICAL expressions.

Additional notes about equivalent data types for arguments:

- If an argument is described in the Windows API documentation as a pointer, then the corresponding Fortran interface definition of that argument would have the REFERENCE property (see the ATTRIBUTES). Older interface definitions use the POINTER - Integer and pass the address of the argument, or the LOC intrinsic function.
- Pointer arguments on systems using IA-32 architecture are 32-bit (4 bytes) in length. Pointer arguments on systems using Intel® 64 architecture are 64 bits (8 bytes) in length.
- Be aware that Fortran character variables need to be null-terminated. You can do this by using the C string extension (see C Strings in Character Constants):

```
forstring = 'This is a null-terminated string.'C
```

  You can also concatenate a null using the C_NULL_CHAR constant from instrinsic module ISO_C_BINDING, or CHAR(0):

```
use, intrinsic :: ISO_C_BINDING
…
forstring = 'This is a null-terminated string'//C_NULL_CHAR
forstring2 = 'This is another null-terminated string'//CHAR(0)
```

The structures in WINDOWS.H have been converted to derived types in IFWINTY. Unions in structures are converted to union/maps within the derived type.

Names of components are generally unchanged. C bitfields do not translate directly to Fortran; collections of bitfields are declared as Fortran INTEGER types and individual bitfields are noted as comments in the source (IFWINTY.F90). To see how a particular Windows declaration was translated to Fortran, read the corresponding declaration in the appropriate .F90 source file in the Include folder.

Supplied Windows* API Modules

The Intel® Fortran Compiler provides the following Windows* API modules. These modules correspond to the Windows* import libraries of the same name.

ADVAPI32
COMCTL32
COMDLG32
GDI32
KERNEL32
LZ32
OLE32
OLEAUT32
PSAPI
SCRNSAVE
SHELL32
USER32
VERSION
WINMM
WINSPOOL
WS2_32
WSOCK32

Additionally, the `IFOPNGL` module declares OpenGL routines for Windows OS, with Fortran-specific names.

Math Libraries

The Intel® Fortran Compiler includes these math libraries:

Library name	Description
`libimf.a` (Linux* and macOS*)	Intel® Math Libraries, in addition to `libm.a`, the math library provided with gcc* Both of these libraries are linked in by default because certain math functions supported by the GNU* math library are not available in the Intel® Math Library. This linking arrangement allows the GNU* users to have all functions available when using `ifort`, with Intel optimized versions available when supported.

Library name	Description
	`libimf.a` is linked in before `libm.a`. If you link in `libm.a` first, it will change the versions of the math functions that are used. Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.
`libm.lib` (static library) and `libmmd.dll` (the DLL version) (Windows*)	Math Libraries provided by Intel. Many routines in the *libimf* library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

NOTE
It is strongly recommended to use the default rounding mode (round-to-nearest-even) when calling math library transcendental functions and compiling with default optimization or higher. Faster implementations (in terms of latency and/or throughput) of these functions are validated under the default round-to-nearest-even mode. Using other rounding modes may make results generated by these faster implementations less accurate, or set unexpected floating-point status flags. This behavior may be avoided by `-no-fast-transcendentals`, which disables calls to the faster implementations of math functions, or by `-fp-model strict`, which warns the compiler that it cannot assume default settings for the floating-point environment.

Many routines in the Intel® Math Library are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Intel® Math Kernel Library (Intel® MKL)	Math library of Fortran routines and functions that perform a wide variety of operations on vectors and matrices. The library also includes fast Fourier transform (fft) functions, as well as vector mathematical and vector statistical functions. For more information, see Using Intel® Performance Libraries with Microsoft Visual Studio* and the Intel® Math Kernel Library documentation.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Data and I/O

This section discusses the following:

- Fortran integer, logical, character, and Hollerith data representation
- Fortran input and output (I/O) topics, such as I/O devices, statements, and files, and OPEN and CLOSE statements

Data Representation Overview

Intel® Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte). For information on using nonnative big endian and VAX* floating-point formats, see Supported Native and Nonnative Numeric Formats.

The symbol :A in any figure specifies the address of the byte containing bit 0, which is the starting address of the represented data element.

The following table lists the intrinsic data types used by Intel® Fortran, the storage required, and valid ranges. For information on declaring Fortran intrinsic data types, see Type Declarations. For example, the declaration INTEGER(4) is the same as INTEGER(KIND=4) and INTEGER*4.

Fortran Data Types and Storage

Data Type	Storage	Description
BYTE	1 byte (8 bits)	A signed integer data type equivalent to INTEGER(1).
INTEGER	See INTEGER(2), INTEGER(4), and INTEGER(8)	A signed integer, either INTEGER(2), INTEGER(4), or INTEGER(8). The size is controlled by the `integer-size` compiler option.
INTEGER(1)	1 byte (8 bits)	A signed integer value from -128 to 127.
INTEGER(2)	2 bytes (16 bits)	A signed integer value from -32,768 to 32,767.
INTEGER(4)	4 bytes (32 bits)	A signed integer value from -2,147,483,648 to 2,147,483,647.
INTEGER(8)	8 bytes (64 bits)	A signed integer value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
REAL	See REAL(4), REAL(8), and REAL(16)	A real floating-point value, either REAL(4), REAL(8), or REAL(16). The size is controlled by the `real-size` compiler option.
DOUBLE PRECISION	See REAL(8) and REAL(16)	A double precision floating-point value, either REAL(8) or REAL(16). The size is controlled by the `double-size` compiler option.
REAL(4)	4 bytes (32 bits)	A single-precision real floating-point value in IEEE binary32 format ranging from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are subnormal.
REAL(8)	8 bytes (64 bits)	A double-precision real floating-point value in IEEE binary64 format ranging from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.94065645841246544D-324 are subnormal.
REAL(16)	16 bytes (128 bits)	An extended-precision real floating-point value in IEEE binary128 format ranging from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932.

Data Type	Storage	Description
COMPLEX	See COMPLEX(4), COMPLEX(8), and COMPLEX(16)	A complex floating-point value in a pair of real and imaginary parts that are either REAL(4), REAL(8), or REAL(16). The size is controlled by the `real-size` compiler option.
DOUBLE COMPLEX	See COMPLEX(8) and COMPLEX(16)	A double complex floating-point value in a pair of real and imaginary parts that are either REAL(8) or REAL(16). The size is controlled by the `double-size` compiler option.
COMPLEX(4)	8 bytes (64 bits)	A single-precision complex floating-point value in a pair of IEEE binary32 format parts: real and imaginary. The real and imaginary parts each range from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are subnormal.
COMPLEX(8)	16 bytes (128 bits)	A double-precision complex floating-point value in a pair of IEEE binary64 format parts: real and imaginary. The real and imaginary parts each range from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.9406564584124654D-324 are subnormal.
COMPLEX(16)	32 bytes (256 bits)	An extended-precision complex floating-point value in a pair of IEEE binary128 format parts: real and imaginary. The real and imaginary parts each range from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
LOGICAL	See LOGICAL(2), LOGICAL(4), and LOGICAL(8)	A logical value, either LOGICAL(2), LOGICAL(4), or LOGICAL(8). The size is controlled by the `integer-size` compiler option.
LOGICAL(1)	1 byte (8 bits)	A logical value of .TRUE. or .FALSE.
LOGICAL(2)	2 bytes (16 bits)	A logical value of .TRUE. or .FALSE.
LOGICAL(4)	4 bytes (32 bits)	A logical value of .TRUE. or .FALSE.
LOGICAL(8)	8 bytes (64 bits)	A logical value of .TRUE. or .FALSE.
CHARACTER	1 byte (8 bits) per character	Character data represented by character code convention. Declarations for Character Types can be in the form CHARACTER(LEN=*n*) or CHARACTER**n*, where *n* is the number of bytes or *n* is (*) to indicate passed-length format.
HOLLERITH	1 byte (8 bits) per Hollerith character	A Hollerith constant.

In addition, you can define Binary Constants.

Integer Data Representations

The Fortran numeric environment is flexible, which helps make Fortran a strong language for intensive numerical calculations. The Fortran standard purposely leaves the precision of numeric quantities and the method of rounding numeric results unspecified. This allows Fortran to operate efficiently for diverse applications on diverse systems.

The effect of math computations on integers is straightforward:

- INTEGER(KIND=1) Representation consists of a maximum positive integer (127), a minimum negative integer (-128), and all integers between them including zero.
- INTEGER(KIND=2) Representation consisst of a maximum positive integer (32,767), a minimum negative integer (-32,768) , and all integers between them including zero.
- INTEGER(KIND=4) Representation consists of a maximum positive integer (2,147,483,647), a minimum negative integer (-2,147,483,648), and all integers between them including zero.
- INTEGER(KIND=8) Representation consists of a maximum positive integer (9,223,372,036,854,775,807), a minimum negative integer (-9,223,372,036,854,775,808), and all integers between them including zero.

Operations on integers usually result in other integers within this range. Integer computations that produce values too large or too small to be represented in the desired KIND result in the loss of precision. One arithmetic rule to remember is that integer division results in truncation (for example, 8/3 evaluates to 2).

Integer data lengths can be 1, 2, 4, or 8 bytes in length.

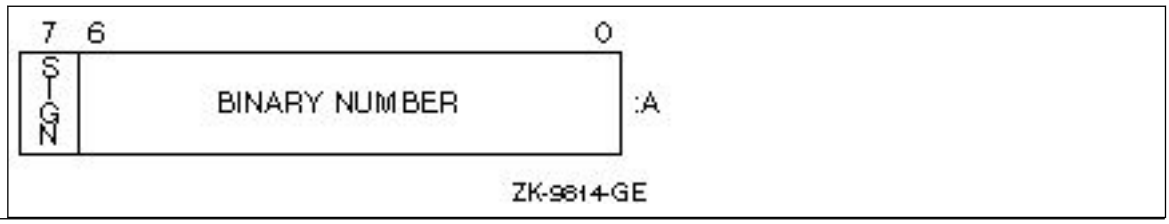
The default data size used for an INTEGER data declaration is INTEGER(4) (same as INTEGER(KIND=4)). However, you can specify a compiler option to override the default. Option `integer-size 16` can be used to specify INTEGER(2) and option `integer-size 64` can be used to specify INTEGER(8).

Integer data is signed with the sign bit being 0 (zero) for positive numbers and 1 for negative numbers.

INTEGER(KIND=1) Representation

INTEGER(1) values range from -128 to 127 and are stored in 1 byte, as shown below.

INTEGER(1) Data Representation



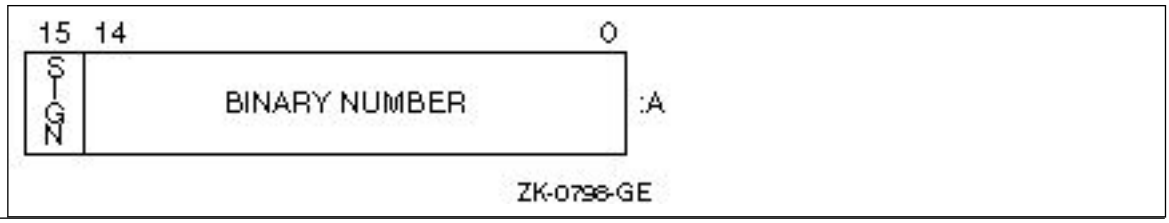
Integers are stored in a two's complement representation. For example:

```
+22 == 16 (hex)
 -7 == F9 (hex)
```

INTEGER(KIND=2) Representation

INTEGER(2) values range from -32,768 to 32,767 and are stored in 2 contiguous bytes, as shown below:

INTEGER(2) Data Representation



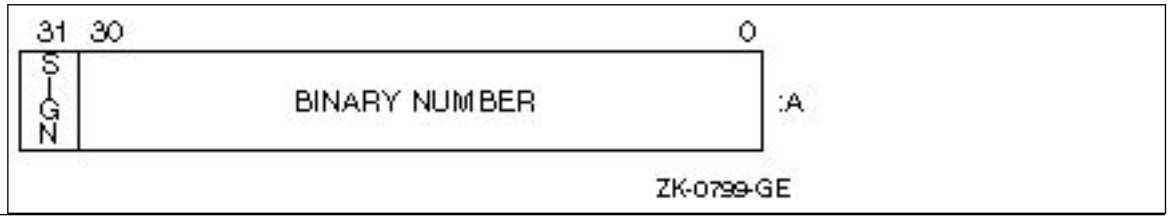
Integers are stored in a two's complement representation. For example:

```
+22 == 0016 (hex)
 -7 == FFF9 (hex)
```

INTEGER(KIND=4) Representation

INTEGER(4) values range from -2,147,483,648 to 2,147,483,647 and are stored in 4 contiguous bytes, as shown below.

INTEGER(4) Data Representation

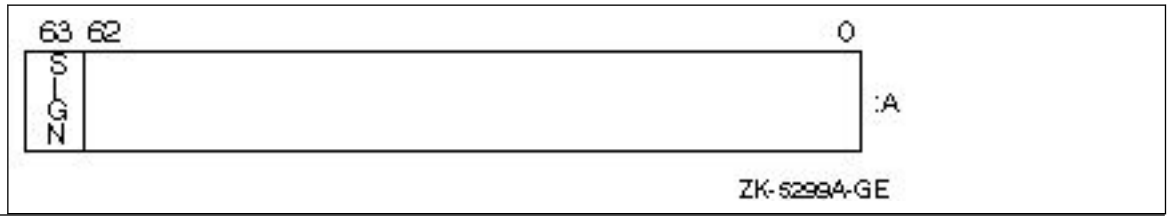


Integers are stored in a two's complement representation.

INTEGER(KIND=8) Representation

INTEGER(8) values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and are stored in 8 contiguous bytes, as shown below.

INTEGER(8) Data Representation



Integers are stored in a two's complement representation.

Logical Data Representations

Logical data can be 1, 2, 4, or 8 bytes in length.

The default data size used for a LOGICAL data declaration is LOGICAL(4) (same as LOGICAL(KIND=4)). However, you can specify a compiler option to override the default. Option `integer-size 16` can be used to specify LOGICAL(2) and option `integer-size 64` can be used to specify LOGICAL(8).

To improve performance on systems using Intel® 64 architecture, use LOGICAL(4) (or LOGICAL(8)) rather than LOGICAL(2) or LOGICAL(1). On systems using IA-32 architecture, use LOGICAL(4) rather than LOGICAL(8), LOGICAL(2), or LOGICAL(1).

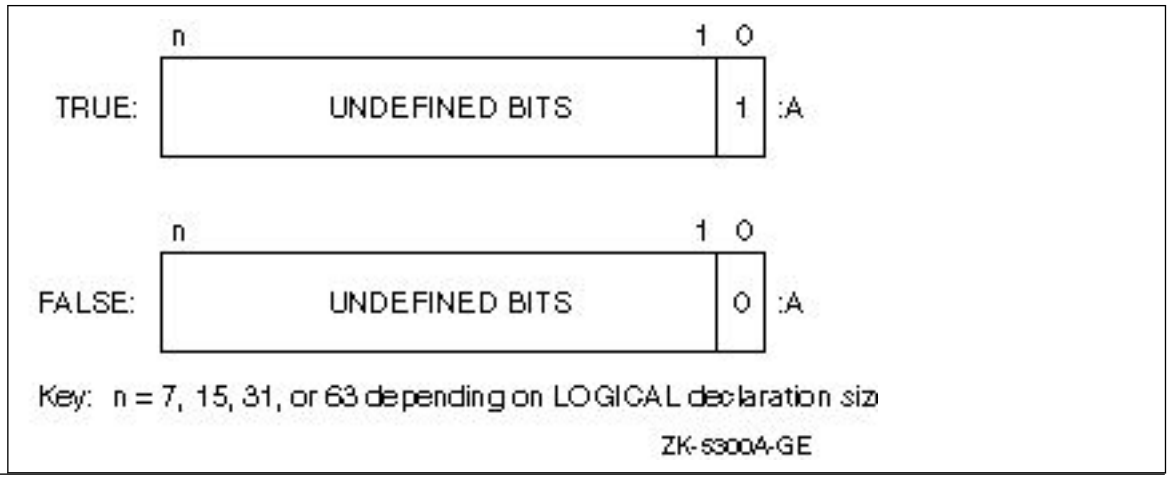
LOGICAL(KIND=1) values are stored in 1 byte. In addition to having logical values .TRUE. and .FALSE., LOGICAL(1) data can also have values in the range -128 to 127. Logical variables can also be interpreted as integer data.

In addition to LOGICAL(1), logical values can also be stored in 2 (LOGICAL(2)), 4 (LOGICAL(4)), or 8 (LOGICAL(8)) contiguous bytes, starting on an arbitrary byte boundary.

If the `fpscomp nological` compiler option is set (the default), the low-order bit determines whether the logical value is true or false. Specify `fpscomp logical` instead of `fpscomp nological` to interoperate with procedures written in C for Microsoft* Fortran PowerStation logical values, where 0 (zero) is false and non-zero values are true.

LOGICAL(1), LOGICAL(2), LOGICAL(4), and LOGICAL(8) data representations (when `fpscomp nological` is set) appear below.

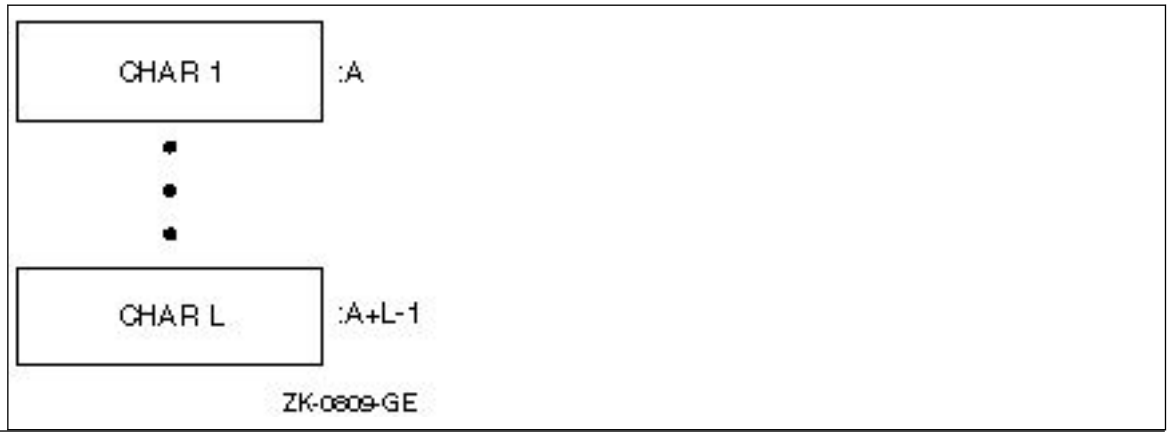
LOGICAL(1), LOGICAL(2), LOGICAL(4), and LOGICAL(8) Data Representations



Character Representation

A character string is a contiguous sequence of bytes in memory, as shown below.

CHARACTER Data Representation



A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes.

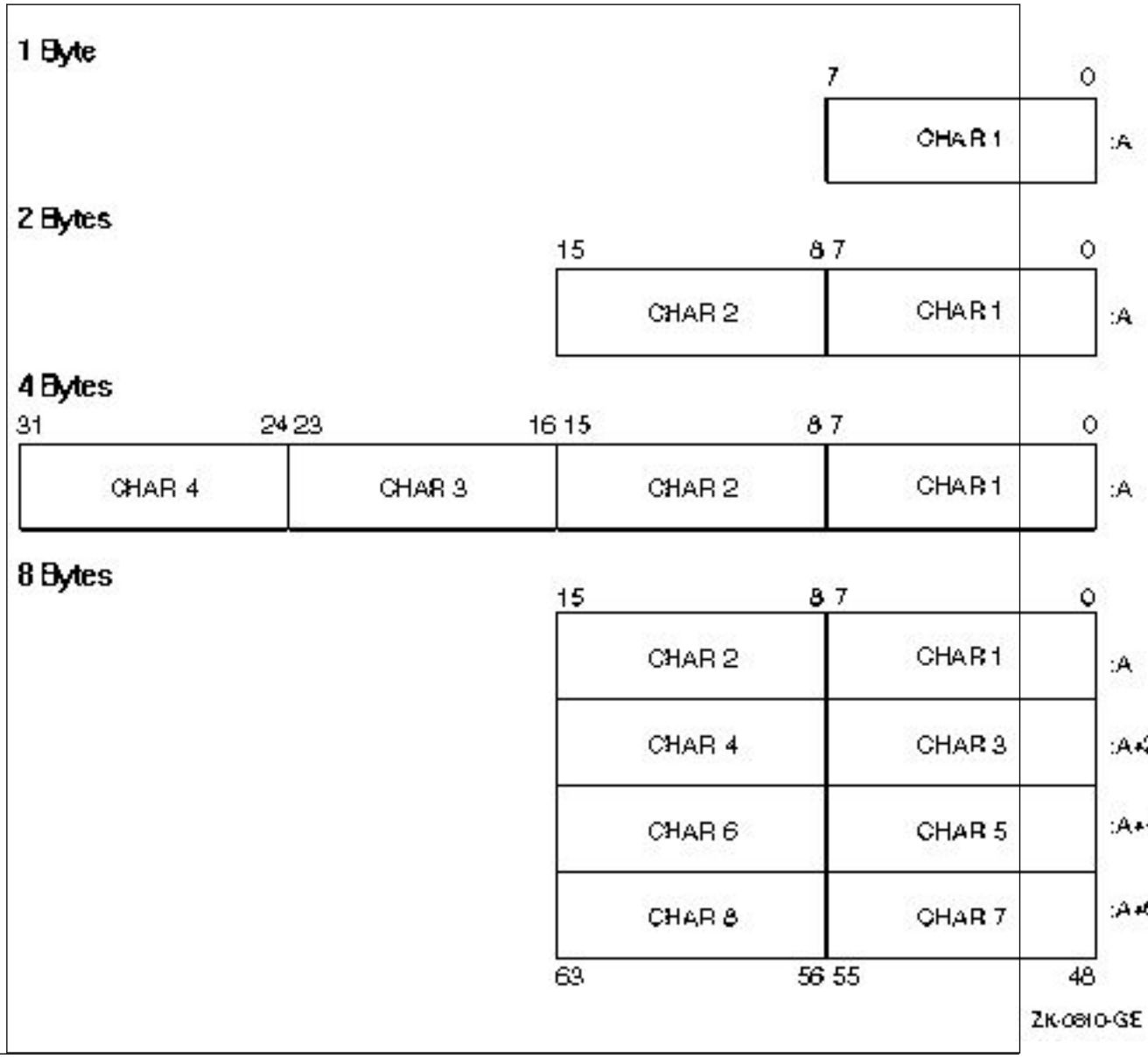
On Windows* systems, the length L of a string is in the range 1 through 2,147,483,647 (2**31-1) .

On Linux* systems, the length L of a string is in the range 1 through 2,147,483,647 (2**31-1) for systems based on IA-32 architecture and in the range 1 through 9,223,372,036,854,775,807 (2**63-1) for systems based on Intel® 64 architecture.

Hollerith Representation

Hollerith constants are stored internally, one character per byte, as shown below.

Hollerith Data Representation



Fortran I/O

In Fortran's I/O system, data is stored and transferred among files. All I/O data sources and destinations are considered files.

Devices such as the screen, keyboard, and printer are external files, as are data files stored on a device such as a disk.

Variables in memory can also act as a file on a disk and are typically used to convert ASCII representations of numbers to binary form. When variables are used in this way, they are called internal files.

For more information, see the individual topics in this section.

Logical Devices

Every file, internal or external, is associated with a logical device. You identify the logical device associated with a file by using Unit Specifier (UNIT=). The unit specifier for an internal file is the name of the character variable associated with it. The unit specifier for an external file is one of the following:

- A number you assign with the OPEN statement
- A number assigned by the Intel® Fortran run-time library with the OPEN specifier NEWUNIT=
- A number preconnected as a unit specifier to a device
- An asterisk (*)

The OPEN statement connects a unit number with an external file and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers. External unit specifiers that are preconnected to certain devices do not have to be opened. External units that you connect are disconnected when program execution terminates or when the unit is closed by a CLOSE statement.

A unit must not be connected to more than one file at a time, and a file must not be connected to more than one unit at a time. You can OPEN an already opened file but only to change some of the I/O options for the connection, not to connect an already opened file or unit to a different unit or file.

You must use a unit specifier for all I/O statements except in the following cases:

- ACCEPT, which always reads from standard input unless the FOR_ACCEPT environment variable is defined.
- INQUIRE by file, which specifies the filename rather than the unit with which the file is associated.
- PRINT, which always writes to standard output unless the FOR_PRINT environment variable is defined.
- READ statements that contain only an I/O list and format specifier, which read from standard input (UNIT=5) unless the FOR_READ environment variable is defined.
- WRITE statements that contain only an I/O list and format specifier, which write to standard output unless the FOR_WRITE environment variable is defined.
- TYPE, which always writes to standard output unless the FOR_TYPE environment variable is defined.

External Files

A unit specifier associated with an external file must be either an integer expression or an asterisk (*). The integer expression must be in the range 0 (zero) to a maximum value of 2,147,483,640. (The predefined parameters FOR_K_PRINT_UNITNO, FOR_K_TYPE_UNITNO, FOR_K_ACCEPT_UNITNO, and FOR_K_READ_UNITNO may not be in that range. For more information, see the Language Reference.)

The following example connects the external file `UNDAMP.DAT` to unit 10 and writes to it:

```
OPEN (UNIT = 10, FILE = 'UNDAMP.DAT')
WRITE (10, '(A18,\)') ' Undamped Motion:'
```

The asterisk (*) unit specifier specifies the keyboard when reading and the screen when writing. The following example uses the asterisk specifier to write to the screen:

```
WRITE (*, '(1X, A30,\)') ' Write this to the screen.'
```

Intel Fortran has four units preconnected to external files (devices), as shown in the following table.

External Unit Specifier	Environment Variable	Description
Asterisk (*)	None	Always represents the keyboard and screen (unless the appropriate environment variable is defined, such as FOR_READ).
0	FORT0	Initially represents the screen (unless FORT0 is explicitly defined)

5	FORT5	Initially represents the keyboard (unless FORT5 is explicitly defined)
6	FORT6	Initially represents the screen (unless FORT6 is explicitly defined)

The asterisk (*) specifier is the only unit specifier that cannot be reconnected to another file; attempting to close such a unit causes a compile-time error. Units 0, 5, and 6 can be connected to any file with the OPEN statement; if you close one of those units, the file is automatically reconnected to its preconnected device the next time an I/O statement attempts to use that unit.

Intel® Fortran does not support buffering to `stdin`, and does not buffer to `stdout` by default unless the `assume buffered_stdout` option is specified. All I/O to units * and 6 use line buffering by default. Therefore, C and Fortran output to `stdout` should work well as long as the C code is not performing buffering. If the C code is performing buffering, the C code will have to flush the buffers after each write. For more information on `stdout` and `stdin`, see Assigning Files to Logical Units.

You can change these preconnected files by doing one of the following:

- Using an OPEN statement to open unit 5, 6, or 0. When you explicitly OPEN a file for unit 5, 6, or 0, the OPEN statement keywords specify the file-related information to be used instead of the preconnected standard I/O file.
- Setting the appropriate environment variable (FORT*n*) to redirect I/O to an external file.

To redirect input or output from the standard preconnected files at run time, you can set the appropriate environment variable or use the appropriate shell redirection character in a pipe (such as > or <).

When you omit the file name in the OPEN statement or use an implicit OPEN, you can define the environment variable FORT*n* to specify the file name for a particular unit number *n*. An exception to this is when the `fpscomp filesfromcmd` compiler option is specified.

For example, if you want unit 6 to write to a file instead of standard output, set the environment variable FORT6 to the path and filename to be used before you run the program. If the appropriate environment variable is not defined, a default filename is used, in the form fort.*n* where *n* is the logical unit number.

The following example writes to the preconnected unit 6 (the screen), then reconnects unit 6 to an external file and writes to it, and finally reconnects unit 6 to the screen and writes to it:

```
    REAL a, b
!  Write to the screen (preconnected unit 6).
    WRITE(6, '('' This is unit 6'')')
!  Use the OPEN statement to connect unit 6
!  to an external file named 'COSINES'.
    OPEN (UNIT = 6, FILE = 'COSINES', STATUS = 'NEW')
    DO k = 1, 63, 1
         a = k/10
         b = COS (a)
!  Write to the file 'COSINES'.
          WRITE (6, 100) a, b
100        FORMAT (F3.1, F5.2)
    END DO
!  Close it.
    CLOSE (6)
!  Reconnect unit 6 to the screen, by writing to it.
    WRITE(6,' ('' Cosines completed'')')
    END
```

Internal Files

The unit specifier associated with an internal file is a scalar or array character variable:

- If the internal file is a scalar character variable, the file has only one record; its length is equal to that of the variable.
- If the internal file is an array character variable, the file has a record for each element in the array; each record's length is equal to one array element.

Follow these rules when using internal files:

- Use only formatted I/O, including I/O formatted with a format specification, namelist, and list-directed I/O. (List-directed and namelist I/O are treated as sequential formatted I/O.)
- If the character variable is an allocatable array or array part of an allocatable array, the array must be allocated before use as an internal file. If the character variable is a pointer, it must be associated with a target.
- Use only READ and WRITE statements. You cannot use file connection (OPEN, CLOSE), file positioning (REWIND, BACKSPACE), or file inquiry (INQUIRE) statements with internal files.

You can read and write internal files with FORMAT I/O statements, namelist I/O statements, or list-directed I/O statements exactly as you can external files. Before an I/O statement is executed, internal files are positioned at the beginning of the variable, before the first record.

With internal files, you can use the formatting capabilities of the I/O system to convert values between external character representations and Fortran internal memory representations. Reading from an internal file converts the ASCII representations into numeric, logical, or character representations, and writing to an internal file converts these representations into their ASCII representations.

This feature makes it possible to read a string of characters without knowing its exact format, examine the string, and interpret its contents. It also makes it possible, as in dialog boxes, for the user to enter a string and for your application to interpret it as a number.

If less than an entire record is written to an internal file, the rest of the record is filled with blanks.

In the following example, `str` and `fname` specify internal files:

```
   CHARACTER(10) str
   INTEGER n1, n2, n3
   CHARACTER(14) fname
   INTEGER   i
   str = " 1   2   3"
! List-directed READ sets n1 = 1, n2 = 2, n3 = 3.
   READ(str, *) n1, n2, n3
   i = 4
! Formatted WRITE sets fname = 'FM004.DAT'.
   WRITE (fname, 200) i
200 FORMAT ('FM', I3.3, '.DAT')
```

See Also

`assume` compiler option

Physical Devices (Windows*)

This topic only applies to Windows* operating systems.

I/O statements that do not refer to a specific file or I/O device read from standard input and write to standard output. Standard input is the keyboard, and standard output is the screen (console). To perform input and output on a physical device other than the keyboard or screen, you specify the device name as the filename to be read from or written to.

Some physical device names are determined by the host operating system; others are recognized by Intel Fortran. Extensions on most device names are ignored.

Filenames for Device I/O

Device	Description
CON	Console (standard output)
PRN	Printer
COM1	Serial port #1
COM2	Serial port #2
COM3	Serial port #3
COM4	Serial port #4
LPT1	Parallel Port #1
LPT2	Parallel Port #2
LPT3	Parallel Port #3
LPT4	Parallel Port #4
NUL	NULL device. Discards all output; contains no input
AUX	Serial port #1
LINE	Serial port #1
USER	Standard output
ERR	Standard error
CONOUT$	Standard output
CONIN$	Standard input

NOTE
If you use the LINE, USER, or ERR name with an extension, for example, LINE.TXT, Fortran will write to a file rather than to the device.

Examples of opening physical devices as units are:

```
OPEN (UNIT = 4, FILE = 'PRN')
OPEN (UNIT = 7, FILE = 'LPT2', ERR = 100)
```

Types of I/O Statements

The table below lists the Intel Fortran I/O statements:

Category and statement name	Description
File connection	
OPEN	Connects a unit number with an external file and specifies file connection characteristics.
CLOSE	Disconnects a unit number from an external file.
File inquiry	

Category and statement name	Description
DEFINE FILE	Specifies file characteristics for a direct access relative file and connects the unit number to the file, similar to an OPEN statement. Provided for compatibility with compilers older than FORTRAN-77.
INQUIRE	Returns information about a named file, a connection to a unit, or the length of an output item list.
Record position	
BACKSPACE	Moves the record position to the beginning of the previous record (sequential access only).
DELETE	Marks the record at the current record position in a relative file as deleted (direct access only).
ENDFILE	Writes an end-of-file marker after the current record (sequential access only).
FIND	Changes the record position in a direct access file. Provided for compatibility with compilers older than FORTRAN-77.
REWIND	Sets the record position to the beginning of the file (sequential access only).
Record input	
READ	Transfers data from an external file record or an internal file to internal storage.
ACCEPT	Reads input from stdin. Unlike READ, ACCEPT only provides formatted sequential input and does not specify a unit number.
Record output	
WRITE	Transfers data from internal storage to an external file record or to an internal file.
REWRITE	Transfers data from internal storage to an external file record at the current record position (direct access relative files only).
TYPE	Writes record output to stdout.
PRINT	Transfers data from internal storage to stdout. Unlike WRITE, PRINT only provides formatted sequential output and does not specify a unit number.
FLUSH	Flushes the contents of an external unit's buffer to its associated file.

In addition to the READ, WRITE, REWRITE, TYPE, and PRINT statements, other I/O record-related statements are limited to a specific file organization. For instance:

- The DELETE statement only applies to relative files. (Detecting deleted records is only available if the `vms` option was specified when the program was compiled.)
- The BACKSPACE statement only applies to sequential files open for sequential access.
- The REWIND statement only applies to sequential files open for sequential access and to direct access files.
- The ENDFILE statement only applies to certain types of sequential files open for sequential access and to direct access files.

The file-related statements (OPEN, INQUIRE, and CLOSE) apply to any relative or sequential file.

Forms of I/O Statements

Each type of record I/O statement can be coded in a variety of forms. The form you select depends on the nature of your data and how you want it treated. When opening a file, specify the form using the FORM specifier.

The following are the forms of I/O statements:

- *Formatted I/O statements* contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.
- *List-directed* and *namelist I/O statements* are similar to formatted statements in function. However, they use different mechanisms to control the translation of data: formatted I/O statements use explicit format specifiers, and list-directed and namelist I/O statements use data types.
- *Unformatted I/O statements* do not contain format specifiers and therefore do not translate the data being transferred (important when writing data that will be read later).

Formatted, list-directed, and namelist I/O forms require translation of data from internal (binary) form within a program to external (readable character) form in the records. Consider using unformatted I/O for the following reasons:

- Unformatted data avoids the translation process, so I/O tends to be faster.
- Unformatted data avoids the loss of precision in floating-point numbers when the output data will subsequently be used as input data.
- Unformatted data conserves file storage space (stored in binary form).

To write data to a file using formatted, list-directed, or namelist I/O statements, specify FORM= 'FORMATTED' when opening the file. To write data to a file using unformatted I/O statements, specify FORM= 'UNFORMATTED' when opening the file.

Data written using formatted, list-directed, or namelist I/O statements is referred to as *formatted data*. Data written using unformatted I/O statements is referred to as *unformatted data*.

When reading data from a file, you should use the same I/O statement form that was used to write the data to the file. For instance, if data was written to a file with a formatted I/O statement, you should read data from that file with a formatted I/O statement.

Although I/O statement form is usually the same for reading and writing data in a file, a program can read a file containing unformatted data (using unformatted input) and write it to a separate file containing formatted data (using formatted output). Similarly, a program can read a file containing formatted data and write it to a different file containing unformatted data.

You can access records in any sequential or relative file using sequential access. For relative files and certain (fixed-length) sequential files, you can also access records using direct access.

The table below shows categories for the main record I/O statements that can be used in Intel® Fortran programs.

File Type, Access, and I/O Form	Available Statements
External file, sequential access	
Formatted	READ, WRITE, PRINT, ACCEPT, TYPE, REWRITE
List-directed	READ, WRITE, PRINT, ACCEPT, TYPE
Namelist	READ, WRITE, PRINT, ACCEPT, TYPE
Unformatted	READ, WRITE, REWRITE
External file, direct access	
Formatted	READ, WRITE, REWRITE
Unformatted	READ, WRITE, REWRITE
External file, stream access	
Formatted	READ, WRITE
List-directed	READ, WRITE
Namelist	READ, WRITE

File Type, Access, and I/O Form	Available Statements
Unformatted	READ, WRITE
Internal file	
Formatted	READ, WRITE
List-directed	READ, WRITE
Unformatted	None

NOTE
You can use the REWRITE statement only for relative files, using direct access.

Assigning Files to Logical Units

Most I/O operations involve a disk file, keyboard, or screen display. Other devices can also be used:

- Sockets can be read from or written to if a USEROPEN routine (usually written in C) is used to open the socket.
- Pipes opened for read and write access block (wait until data is available) if you issue a READ to an empty pipe.
- Pipes opened for read-only access return EOF if you issue a READ to an empty pipe.

You can access the terminal screen or keyboard by using preconnected files listed in Logical Devices.

You can choose to assign files to logical units by using one of the following methods:

- Using default values, such as a preconnected unit
- Supplying a file name (and possibly a directory) in an OPEN statement
- Using environment variables

Using Default Values

In the following example, the PRINT statement is associated with a preconnected unit (stdout) by default.

```
PRINT *,100
```

The READ statement associates the logical unit 7 with the file `fort.7` (because the FILE specifier was omitted) by default:

```
OPEN (UNIT=7,STATUS='NEW')
READ (7,100)
```

Supplying a File Name in an OPEN Statement

The FILE specifier in an OPEN statement typically specifies only a file name (such as `filnam`) or contains both a directory and file name (such as `/usr/proj/filnam`).

For example:

```
OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
```

The DEFAULTFILE specifier in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).

Implied OPEN

Performing an implied OPEN means that the FILE and DEFAULTFILE specifier values are not specified and an environment variable is used, if present. Thus, if you used an implied OPEN, or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Using Environment Variables

You can use shell commands to set the appropriate environment variable to a value that indicates a directory (if needed) and a file name to associate a unit with an external file.

Intel Fortran recognizes environment variables for each logical I/O unit number in the form of FORT`n`, where `n` is the logical I/O unit number. If a file name is not specified in the OPEN statement and the corresponding FORT`n` environment variable is not set for that unit number, Intel Fortran generates a file name in the form fort.`n`, where `n` is the logical unit number.

Implied Intel Fortran Logical Unit Numbers

The ACCEPT, PRINT, and TYPE statements, and the use of an asterisk (*) in place of a unit number in READ and WRITE statements, do not include an explicit logical unit number.

Each of these Fortran statements uses an implicit internal logical unit number and environment variable. Each environment variable is in turn associated by default with one of the Fortran file names that are associated with standard I/O files. The table below shows these relationships:

Intel Fortran statement	Environment variable	Standard I/O file name
READ (*,f) iolist	FOR_READ	stdin
READ f,iolist	FOR_READ	stdin
ACCEPT f,iolist	FOR_ACCEPT	stdin
WRITE (*,f) iolist	FOR_PRINT	stdout
PRINT f,iolist	FOR_PRINT	stdout
TYPE f,iolist	FOR_TYPE	stdout
WRITE(0,f) iolist	FORT0	stderr
READ(5,f) iolist	FORT5	stdin
WRITE(6,f) iolist	FORT6	stdout

You can change the file associated with these Intel Fortran environment variables, as you would any other environment variable, by means of the environment variable assignment command. For example:

```
setenv FOR_READ /usr/users/smith/test.dat
```

After executing the preceding command, the environment variable for the READ statement using an asterisk refers to file `test.dat` in the specified directory.

NOTE
The association between the logical unit number and the physical file can occur at run-time. Instead of changing the logical unit numbers specified in the source program, you can change this association at run time to match the needs of the program and the available resources. For example, before running the program, a script file can set the appropriate environment variable or allow the terminal user to type a directory path, file name, or both.

File Organization

File organization refers to the way records are physically arranged on a storage device. This topic describes the two main types of file organization.

Related topics describe the following:

- *Record type* refers to whether records in a file are all the same length, are of varying length, or use other conventions to define where one record ends and another begins. For more information on record types, see Record Types.
- *Record access* refers to the method used to read records from or write records to a file, regardless of its organization. The way a file is organized does not necessarily imply the way in which the records within that file will be accessed. For more information on record access, see File Access and File Structure and Record Access.

Types of File Organization

Fortran supports two types of file organizations:

- Sequential
- Relative

The organization of a file is specified by means of the ORGANIZATION keyword in the OPEN statement.

The default file organization is always ORGANIZATION= 'SEQUENTIAL' for an OPEN statement.

You can store sequential files on magnetic tape or disk devices, and can use other peripheral devices, such as terminals, pipes, and line printers as sequential files.

You must store relative files on a disk device.

Sequential File Organization

A sequentially organized file consists of records arranged in the sequence in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, and so on). As a result, records can be added only at the end of the file. Attempting to add records at some place other than the end of the file will result in the file begin truncated at the end of the record just written.

Sequential files are usually read sequentially, starting with the first record in the file. Sequential files with a fixed-length record type that are stored on disk can also be accessed by relative record number (direct access).

Relative File Organization

Within a relative file are numbered positions, called *cells*. These cells are of fixed equal length and are consecutively numbered from 1 to *n*, where 1 is the first cell, and *n* is the last available cell in the file. Each cell either contains a single record or is empty. Records in a relative file are accessed according to cell number. A cell number is a record's relative record number; its location relative to the beginning of the file. By specifying relative record numbers, you can directly retrieve, add, or delete records regardless of their locations. You can only detect deleted records if you specify option `vms` when the program is compiled.

When creating a relative file, use the RECL value to determine the size of the fixed-length cells. Within the cells, you can store records of varying length, as long as their size does not exceed the cell size.

Internal Files and Scratch Files

Intel Fortran also supports internal files and scratch files.

Internal Files

When you use sequential access, you can use an internal file to reference character data in a buffer. The transfer occurs between internal storage spaces (unlike external files), such as between user variables and a character array.

An internal file consists of any of the following:

- Character variable
- Character-array element
- Character array
- Character substring
- Character array section without a vector subscript

Instead of specifying a unit number for the READ or WRITE statement, use an internal file specifier in the form of a character scalar memory reference or a character-array name reference.

An internal file is a designated internal storage space (variable buffer) of characters that is treated as a sequential file of fixed-length records. To perform internal I/O, use formatted and list-directed sequential READ and WRITE statements. You cannot use file-related statements such as OPEN and INQUIRE on an internal file (no unit number is used).

If an internal file is made up of a single character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the character variable, array element, or substring it contains. If an internal file is made up of a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined (a value has been assigned to the record).

Prior to each READ and WRITE statement, an internal file is always positioned at the beginning of the first record.

Scratch Files

Scratch files are created by specifying STATUS= 'SCRATCH' in an OPEN statement. By default, these temporary files are created by an OPEN statement and are deleted when closed.

File Access and File Structure

Fortran supports three methods of file access:

- Sequential
- Direct
- Stream

Fortran supports three kinds of file structure:

- Formatted
- Unformatted
- Binary

Sequential-access and direct-access files can have any of the three file structures. Stream-access files can have a file structure of formatted or unformatted.

Choosing a File Access and File Structure

Each kind of file has advantages and the best choice depends on the application you are developing:

- Formatted Files

You create a formatted file by opening it with the FORM='FORMATTED' option, or by omitting the FORM parameter when creating a sequential file. The records of a formatted file are stored as ASCII characters. Numbers that would otherwise be stored in binary form are converted to ASCII format. Each record ends with the ASCII carriage return (CR) and/or line feed (LF) characters.

If you need to view a data file's contents, use a formatted file. You can load a formatted file into a text editor and read its contents directly. The numbers will look like numbers and the strings will look like character strings (an unformatted or binary file looks like a set of hexadecimal characters).

- Unformatted Files

  You create an unformatted file by opening it with the FORM='UNFORMATTED' option, or by omitting the FORM parameter when creating a direct-access file. An unformatted file is a series of records composed of physical blocks. Each record contains a sequence of values stored in a representation that is close to that used in program memory. Little conversion is required during input/output.

  The lack of formatting makes these files quicker to access and more compact than files that store the same information in a formatted form. However, if the files contain numbers, you will not be able to read them with a text editor.
- Binary Files

  You create a binary file by specifying FORM='BINARY'. Binary files are similar to unformatted files, except binary files have no internal record format associated with them.
- Sequential-Access Files

  Data in sequential files must be accessed in order, one record after the other (unless you change your position in the file with the REWIND or BACKSPACE statements). Some methods of I/O are possible only with sequential files, including nonadvancing I/O, list-directed I/O, and namelist I/O. Internal files also must be sequential files. You must use sequential access for files associated with sequential devices.

  A sequential device is a physical storage device that does not allow explicit motion (other than reading or writing). The keyboard, screen, and printer are all sequential devices.
- Direct-Access Files

  Data in direct-access files can be read or written to in any order. Records are numbered sequentially, starting with record number 1. All records have the length specified by the RECL option in the OPEN statement. Data in direct-access files is accessed by specifying the record you want within the file. If you need random access I/O, use direct-access files. A common example of a random-access application is a database.
- Stream-Access Files

  Stream-access I/O is a method of accessing a file without reference to a record structure. With stream access, a file is seen as a continuous sequence of bytes and is addressed by a positive integer starting from 1.

  To enable stream access, specify ACCESS='STREAM' in the OPEN statement for the file. You can use the STREAM= specifier in the INQUIRE statement to determine if STREAM is listed in the set of possible access methods for the file. A value of YES, NO, or UNKNOWN is returned.

  A file enabled for stream access is positioned by file storage units (normally bytes) starting at position 1. To determine the current position, use the POS= specifier in an INQUIRE statement for the unit. You can indicate a required position in a read or write statement with a POS= specifier.

  Stream access can be either formatted or unformatted.

  When connected for formatted stream access, an external file has the following characteristics:

  - The first file storage unit in the file is at position 1. The relationship between positions of successive file storage units is processor dependent; not all positive integers need to correspond to valid positions.
  - Some file storage units may contain record markers; this imposes a record structure on the file in addition to its stream structure. If there is no record marker at the end of the file, the final record is incomplete. Writing an empty record with no record marker has no effect.

When connected for unformatted stream access, an external file has the following characteristics:

- The first file storage unit in the file is at position 1. The position of each subsequent file storage unit is one greater than that of the preceding file storage unit.
- If it is possible to position the file, the file storage units do not need to be read or written in order of their position. For example, you may be able to write the file storage unit at position 2, even though the file storage unit at position 1 has not been written.
- Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and a READ statement for this connection is allowed.
- You cannot use BACKSPACE in an unformatted stream.

File Records

Files may be composed of records. Each record is one entry in the file. A record can be a line from a terminal or a logical record on a magnetic tape or disk file. All records within one file are of the same type.

In Fortran, the number of bytes transferred to a record must be less than or equal to the record length. One record is transferred for each unformatted READ or WRITE statement. A formatted READ or WRITE statement can transfer more than one record using the slash (`/`) edit descriptor.

For binary files, a single READ or WRITE statement reads or writes as many records as needed to accommodate the number of bytes being transferred. On output, incomplete formatted records are padded with spaces. Incomplete unformatted and binary records are padded with undefined bytes (zeros).

Record Types

An I/O record is a collection of data items, called *fields*, which are logically related and are processed as a unit. The record type refers to the convention for storing fields in records.

The record type of the data within a file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

A number of record types are available, as shown in the following table. The table also lists the record overhead. This refers to bytes associated with each record that are used internally by the file system and are not available when a record is read or written. Knowing the record overhead helps when estimating the storage requirements for an application. Although the overhead bytes exist on the storage media, do not include them when specifying the record length with the RECL specifier in an OPEN statement.

Record Type	Available File Organizations and Portability Considerations	Record Overhead
Fixed-length	Relative or sequential file organizations.	None for sequential or for relative if the `vms` option is omitted or option `novms` is specified. One byte for relative if the `vms` option is specified.
Variable-length	Sequential file organization only. The variable-length record type is generally the most portable record type across multi-vendor platforms.	Eight bytes per record.
Segmented	Sequential file organization only and only for unformatted sequential access. The segmented record type is unique to Intel Fortran and should not be used for portability with programs written in languages other than Fortran or for places where Intel Fortran is not used. However, because the	Four bytes per record. One additional padding byte (space) is added if the specified record size is an odd number.

Record Type	Available File Organizations and Portability Considerations	Record Overhead
	segmented record type is unique to Intel Fortran products, formatted data in segmented files can be ported across Intel Fortran platforms.	
Stream (uses no record terminator)	Sequential file organization only.	None required.
Stream_CR (uses CR as record terminator)	Sequential file organization only.	One byte per record.
Stream_LF (uses LF as record terminator)	Sequential file organization only.	One byte per record.
Stream_CRLF (uses CR and LF as record terminator)	Sequential file organization only.	Two bytes per record.

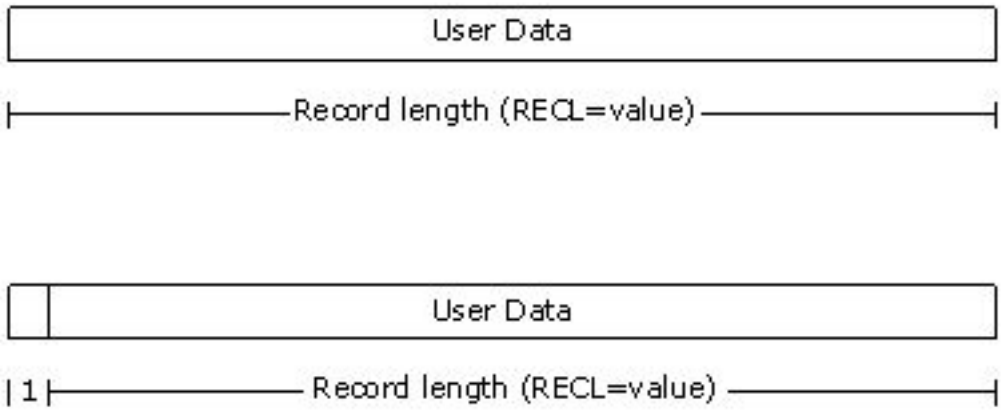
Fixed-Length Records

When you specify fixed-length records, you are specifying that all records in the file contain the same number of bytes. When you open a file that is to contain fixed-length records, you must specify the record size using the RECL keyword. A sequentially organized opened file for direct access must contain fixed-length records, to allow the record position in the file to be computed correctly.

For relative files, the layout and overhead of fixed-length records depends upon whether the program accessing the file was compiled using the `vms` option:

- For relative files where the `vms` option is omitted (the default), each record has no control information.
- For relative files where the `vms` option is specified, each record has one byte of control information at the beginning of the record.

The following figures show the record layout of fixed-length records. The first is for all sequential and relative files where the `vms` option is omitted. The second is for relative files where the `vms` option is specified.



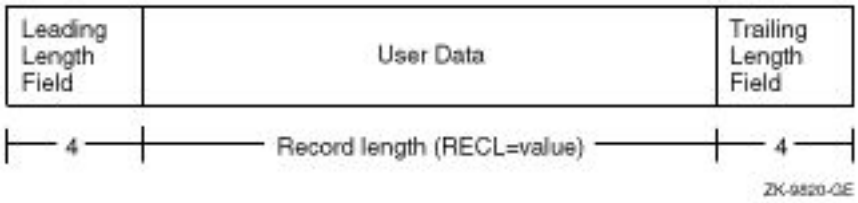
Variable-Length Records

Variable-length records can contain any number of bytes up to a specified maximum record length, and apply only to sequential files.

Variable-length records are prefixed and suffixed by 4 bytes of control information containing length fields. The trailing length field allows a BACKSPACE request to skip back over records efficiently. The 4-byte integer value stored in each length field indicates the number of data bytes (excluding overhead bytes) in that particular variable-length record.

The character count field of a variable-length record is available when you read the record by issuing a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes should be in an I/O list.

The following shows the record layout of variable-length records that are less than 2 gigabytes:



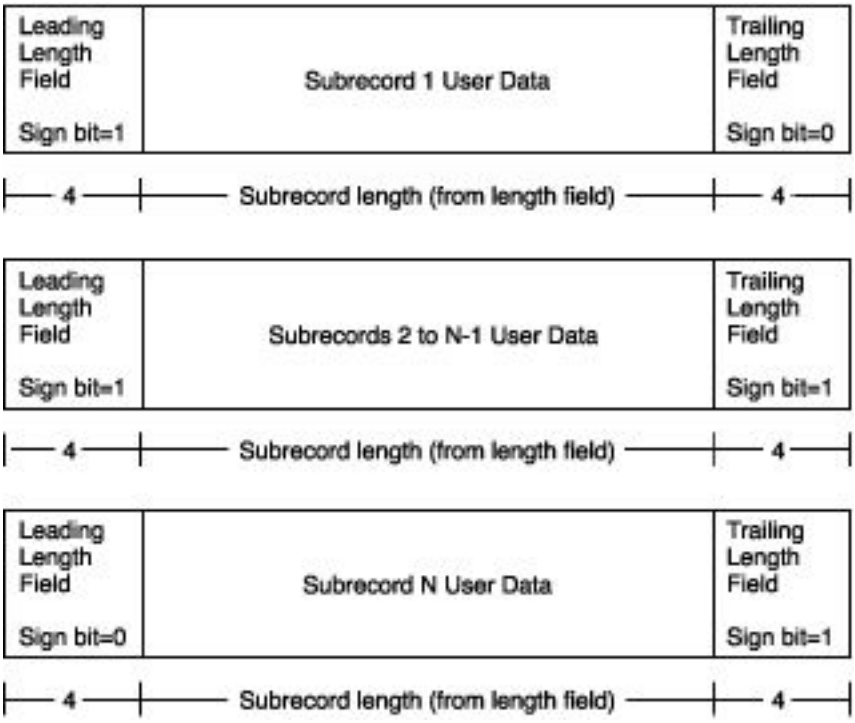
For a record length greater than 2,147,483,639 bytes, the record is divided into *subrecords*. The subrecord can be of any length from 1 to 2,147,483,639, inclusive.

The sign bit of the leading length field indicates whether the record is continued or not. The sign bit of the trailing length field indicates the presence of a preceding subrecord. The position of the sign bit is determined by the endian format of the file.

The following rules describe sign bit values:

- A subrecord that is continued has a leading length field with a sign bit value of 1.
- The last subrecord that makes up a record has a leading length field with a sign bit value of 0.
- A subrecord that has a preceding subrecord has a trailing length field with a sign bit value of 1.
- The first subrecord that makes up a record has a trailing length field with a sign bit value of 0.
- If the value of the sign bit is 1, the length of the record is stored in twos-complement notation.

The following shows the record layout of variable-length records that are greater than 2 gigabytes:



Files written with variable-length records by Intel Fortran programs usually cannot be accessed as text files. Instead, use the Stream_LF record format for text files with records of varying length.

Segmented Records

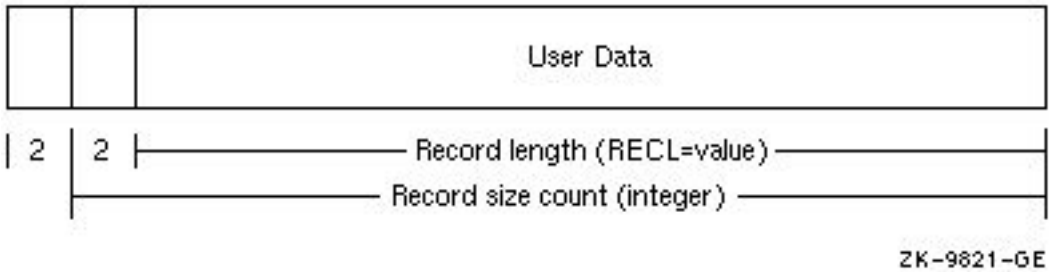
A segmented record is a single logical record consisting of one or more variable-length, unformatted records in a sequentially organized disk file. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

Segmented records are useful when you want to write exceptionally long records but cannot or do not wish to define one long variable-length record, perhaps because virtual memory limitations can prevent program execution. By using smaller, segmented records, you reduce the chance of problems caused by virtual memory limitations on systems on which the program may execute.

For disk files, the segmented record is a single logical record that consists of one or more segments. Each segment is a physical record. A segmented (logical) record can exceed the absolute maximum record length (2.14 billion bytes), but each segment (physical record) individually cannot exceed the maximum record length.

To access an unformatted sequential file that contains segmented records, specify FORM='UNFORMATTED' and RECORDTYPE='SEGMENTED' when you open the file.

The following shows that the layout of segmented records consists of 4 bytes of control information followed by the user data:



The control information consists of a 2-byte integer record length count (includes the 2 bytes used by the segment identifier), followed by a 2-byte integer segment identifier that identifies this segment as one of the following:

Identifier Value	Segment Identified
0	One of the segments between the first and last segments
1	First segment
2	Last segment
3	Only segment

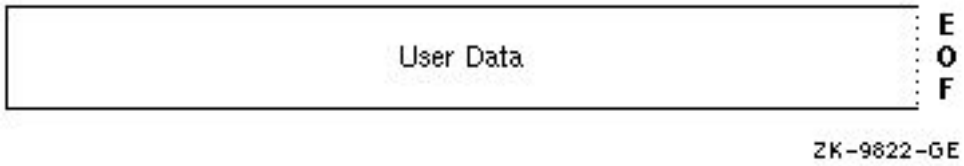
If the specified record length is an odd number, the user data will be padded with a single blank (one byte), but this extra byte is not added to the 2-byte integer record size count.

Avoid the segmented record type when the application requires that the same file be used for programs written in languages other than Intel Fortran and for non-Intel platforms.

Stream File Data

A stream file is not grouped into records and contains no control information. Stream files are used with CARRIAGECONTROL='NONE' and contain character or binary data that is read or written only to the extent of the variables specified on the input or output statement.

The following shows the layout of a stream file:



Stream_CR, Stream_LF and Stream_CRLF Records

Stream_CR, Stream_LF, and Stream_CRLF records are variable-length records whose length is indicated by explicit record terminators embedded in the data, not by a count. These terminators are automatically added when you write records to a stream-type file and are removed when you read records.

Each variety uses either a different 1-byte or 2-byte record terminator:

- Stream_CR files must not contain embedded carriage-return characters because Stream_CR files use only a carriage-return as the terminator.
- Stream_LF files must not contain embedded line-feed (new line) characters because Stream_LF files use only a line-feed (new line) as the terminator. This is the usual operating system text file record type on Linux* OS and macOS* systems.
- Stream_CRLF files must not contain embedded carriage returns or line-feed (new line) characters because Stream_CRLF files use a carriage return/line-feed (new line) pair as the terminator. This is the usual operating system text file record type on Windows* systems.

Guidelines for Choosing a Record Type

Before you choose a record type, consider whether your application will use formatted or unformatted data. If you are using formatted data, you can choose any record type except segmented. If you are using unformatted data, avoid the Stream, Stream_CR, Stream_LF and Stream_CRLF record types.

The segmented record type can only be used for unformatted sequential access with sequential files. You should not use segmented records for files that are read by programs written in languages other than Intel Fortran.

The Stream, Stream_CR, Stream_LF, Stream_CRLF and segmented record types can be used only with sequential files.

The default record type (RECORDTYPE) depends on the values for the ACCESS and FORM specifiers for the OPEN statement. (The RECORDTYPE= specifier is ignored for stream access.)

The record type of the file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

An I/O record is a collection of fields (data items) that are logically related and are usually processed as a unit.

Unless you specify nonadvancing I/O (ADVANCE specifier), each Intel Fortran I/O statement transfers at least one record.

Specifying the Line Terminator for Formatted Files

Use the FOR_FMT_TERMINATOR environment variable to specify the line terminator value used for Fortran formatted files with no explicit RECORDTYPE= specifier.

The FOR_FMT_TERMINATOR environment variable is processed once at the beginning of program execution. Whatever it specifies for specific units continues for the rest of the execution.

You can specify the numbers of the units to have a specific record terminator. The READ/WRITE statements that use these unit numbers will now use the specified record terminators. Other READ/WRITE statements will work in the usual way.

A RECORDTYPE=specifier on an OPEN statement overrides the value set by FOR_FMT_TERMINATOR. The FOR_FMT_TERMINATOR value is ignored for ACCESS='STREAM' files.

General Syntax for FOR_FMT_TERMINATOR

The syntax for this environment variable is as follows:

```
FOR_FMT_TERMINATOR=MODE[:ULIST][;MODE[:ULIST]]
```

where:

```
MODE=CR | LF | CRLF
ULIST = U | ULIST,U
U = decimal | decimal - decimal
```

- MODE specifies the record terminator to be used. The keyword `CR` means to terminate records with a carriage return. The keyword LF means to terminate records with a line feed; this is the default on Linux* OS and macOS* systems. The keyword CRLF means to terminate records with a carriage return/line feed pair; this is the default on Windows systems.
- Each list member "U" is a simple unit number or a number of units as a range. The number of list members is limited to 64.
- "decimal" is a non-negative decimal number less than 232.

No spaces are allowed inside the FOR_FMT_TERMINATOR value.

On Linux* systems, the following shows the command line for the variable setting in a bash-style shell:

```
Sh: export FOR_FMT_TERMINATOR=MODE:ULIST
```

NOTE
The environment variable value should be enclosed in quotes if the semicolon is present.

Example:

The following specifies that all input/output operations on unit numbers 10, 11, and 12 have records terminated with a carriage return/line feed pair:

```
FOR_FMT_TERMINATOR=CRLF:10-12
```

Record Length

Use the RECL specifier to specify the record length.

The units used for specifying record length depend on the form of the data:

- Formatted files (FORM= 'FORMATTED'): Specify the record length in bytes.
- Unformatted files (FORM= 'UNFORMATTED'): Specify the record length in 4-byte units, unless you specify the `assume byterecl` compiler option to request 1-byte units.

For all but variable-length sequential records on 64-bit addressable systems, the maximum record length is 2.147 billion bytes (2,147,483,647 minus the bytes for record overhead). For variable-length sequential records on 64-bit addressable systems, the theoretical maximum record length is about 17,000 gigabytes. When considering very large record sizes, also consider limiting factors such as system virtual memory.

NOTE
The RECL specifier is ignored for stream access.

Record Access

Record access refers to how records will be read from or written to a file, regardless of the file's organization. Record access is specified each time you open a file; it can be different each time. The type of record access permitted is determined by the combination of file organization and record type.

For example, you can fo the following:

- Add records to a sequential file with ORGANIZATION= 'SEQUENTIAL' and POSITION= 'APPEND' (or use ACCESS= 'APPEND').
- Add records sequentially by using multiple WRITE statements, close the file, and then open it again with ORGANIZATION= 'SEQUENTIAL' and ACCESS= 'SEQUENTIAL' (or ACCESS= 'DIRECT' if the sequential file has fixed-length records).

Sequential Access

Sequential access transfers records sequentially to or from files or I/O devices such as terminals. You can use sequential I/O with any type of supported file organization and record type.

If you select sequential access mode for files with sequential or relative organization, records are written to or read from the file starting at the beginning of the file and continuing through it, one record after another. A particular record can be retrieved only after all of the records preceding it have been read; new records can be written only at the end of the file.

Direct Access

Direct access transfers records selected by record number to and from either sequential files stored on disk with a fixed-length record type or relative organization files.

If you select direct access mode, you can determine the order in which records are read or written. Each READ or WRITE statement must include the relative record number, indicating the record to be read or written.

You can directly access a sequential disk file only if it contains fixed-length records. Because direct access uses cell numbers to find records, you can enter successive READ or WRITE statements requesting records that either precede or follow previously requested records. For example, the first of the following statements reads record 24; the second reads record 10:

```
READ (12,REC=24) I
READ (12,REC=10) J
```

Stream Access

Stream access transfers bytes from records sequentially until a record terminator is found or a specified number of bytes have been read or written. Formatted stream records are terminated with a new line character; unformatted stream data contains no record terminators. Bytes can be read from or written to a file by byte position, where the first byte of the file is position 1. For example:

```
OPEN (UNIT=12, ACCESS='STREAM')
READ (12) I, J, K           ! start at the first byte of the file
READ (12, POS=200) X, Y     ! then read staring at byte 200
READ (12) A, B              ! then read starting where the previous READ stopped
```

The POS= specifier on INQUIRE can be used to determine the current byte position in the file.

NOTE
The RECORDTYPE= specifier is ignored for stream access.

Limitations of Record Access by File Organization and Record Type

You can use sequential and direct access modes on sequential and relative files. However, direct access to a sequential organization file can only be done if the file resides on disk and contains fixed-length records.

The table below summarizes the types of access permitted for the various combinations of file organizations and record types.

Record Type	Sequential Access?	Direct Access?
Sequential file organization		
Fixed	Yes	Yes
Variable	Yes	No
Segmented	Yes	No
Stream	Yes	No
Stream_CR	Yes	No
Stream_LF	Yes	No
Stream_CRLF	Yes	No
Relative file organization		
Fixed	Yes	Yes

NOTE
Direct access and relative files require that the file resides on a disk device.

Record Transfer

I/O statements transfer all data as records. The amount of data that a record can contain depends on the following circumstances:

- With formatted I/O (except for fixed-length records), the number of items in the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.
- With namelist and list-directed output, the items listed in the NAMELIST statement or I/O statement list (in conjunction with the NAMELIST or list-directed formatting rules) determine the amount of data to be transferred.
- With unformatted I/O (except for fixed-length records), the I/O statement alone specifies the amount of data to be transferred.
- When you specify fixed-length records (RECORDTYPE= 'FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding).

Typically, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for a single I/O statement to transfer data from or to more than one record, depending on the form of I/O used.

Input Record Transfer

When using advancing I/O, if an input statement specifies fewer data fields (less data) than the record contains, the remaining fields are ignored.

If an input statement specifies more data fields than the record contains, one of the following occurs:

- For formatted input using advancing I/O, if the file was opened with PAD='YES', additional fields are read as spaces. If the file is opened with PAD='NO', an error occurs (the input statement should not specify more data fields than the record contains).
- For formatted input using nonadvancing I/O (ADVANCE='NO'), an end-of-record (EOR) condition is returned. If the file was opened with PAD='YES', additional fields are read as spaces.
- For list-directed input, another record is read.
- For NAMELIST input, another record is read.
- For unformatted input, an error occurs.

Output Record Transfer

If an output statement specifies fewer data fields than the record contains (less data than required to fill a record), the following occurs:

- With fixed-length records (RECORDTYPE= 'FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding) in the form of spaces (for a formatted record) or zeros (for an unformatted record).
- With other record types, the fields present are written and those omitted are not written (might result in a short record).

If the output statement specifies more data than the record can contain, an error occurs, as follows:

- With formatted or unformatted output using fixed-length records, if the items in the output statement and its associated format specifier result in a number of bytes that exceeds the maximum record length (RECL), an error occurs.
- With formatted or unformatted output not using fixed-length records, if the items in the output statement and its associated format specifier result in a number of bytes that exceeds the maximum record length (RECL), the Intel Fortran RTL attempts to increase the RECL value and write the longer record. To obtain the RECL value, use an INQUIRE statement.
- For list-directed output and namelist output, if the data specified exceeds the maximum record length (RECL), another record is written.

Specifying Default Pathnames and File Names

Intel® Fortran provides a number of ways of specifying all or part of a file specification (directory and file name). The following list uses the Linux* pathname `/usr/proj/testdata` as an example:

- The FILE specifier in an OPEN statement typically specifies only a file name (such as testdata) or contains both a directory and file name (such as `/usr/proj/testdata`).
- The DEFAULTFILE specifier in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).
- If you used an implied OPEN (described in Assigning Files to Logical Units) or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Examples of Applying Default Pathnames and File Names

For example, for an implied OPEN of unit number 3, Intel Fortran will check the environment variable FORT3. If the environment variable FORT3 is set, its value is used. If it is not set, the system supplies the file name `fort.3`.

In the following table, assume the current directory is `/usr/smith` and the I/O uses unit 1, as in the statement READ (1,100).

OPEN FILE value	OPEN DEFAULTFILE value	FORT1 environment variable value	Resulting pathname
not specified	not specified	not specified	/usr/smith/fort.1
not specified	not specified	test.dat	/usr/smith/test.dat
not specified	not checked	/usr/tmp/t.dat	/usr/tmp/t.dat
not specified	/tmp	not specified	/tmp/fort.1
not specified	/tmp	testdata	/tmp/testdata
not specified	/usr	lib/testdata	/usr/lib/testdata
file.dat	/usr/group	not checked	/usr/group/file.dat
/tmp/file.dat	not checked	not checked	/tmp/file.dat
file.dat	not specified	not checked	/usr/smith/file.dat

When the resulting file pathname begins with a tilde character (~), C-shell-style pathname substitution is used (regardless of what shell is being used), such as a top-level directory (below the root). For additional information on tilde pathname substitution, see csh(1).

Rules for Applying Default Pathnames and File Names

Intel Fortran determines the file name and the directory path based on certain rules. It determines a file name string as follows:

- If the FILE specifier is present, its value is used.
- If the FILE specifier is not present, Intel Fortran examines the corresponding environment variable. If the corresponding environment variable is set, that value is used. If the corresponding environment variable is not set, a file name in the form fort.*n* is used.

Once Intel Fortran determines the resulting file name string, it determines the directory (which optionally precedes the file name) as follows:

- If the resulting file name string contains an absolute pathname, it is used and the DEFAULTFILE specifier, environment variable, and current directory values are ignored.
- If the resulting file name string does not contain an absolute pathname, Intel Fortran examines the DEFAULTFILE specifier and current directory value: If the corresponding environment variable is set and specifies an absolute pathname, that value is used. Otherwise, the DEFAULTFILE specifier value, if present, is used. If the DEFAULTFILE specifier is not present, Intel Fortran uses the current directory as an absolute pathname.

Opening Files: OPEN Statement

To open a file, you can use a preconnected file (as described in Logical Devices) or can open a file with an OPEN statement. The OPEN statement lets you specify the file connection characteristics and other information.

OPEN Statement Specifiers

The OPEN statement connects a unit number with an external file and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers. Once you open a file, you should close it before opening it again unless it is a preconnected file.

If you open a unit number that was opened previously (without being closed), one of the following occurs:

- If you specify a file specification that does not match the one specified for the original open, the Intel Fortran run-time system closes the original file and then opens the second file. This resets the current record position for the second file.
- If you specify a file specification that matches the one specified for the original open, the file is reconnected without the internal equivalent of the CLOSE and OPEN. This lets you change one or more OPEN statement run-time specifiers while maintaining the record position context.

You can use the INQUIRE Statement to obtain information about whether or not a file is opened by your program.

Especially when creating a new file using the OPEN statement, examine the defaults (see the description of the OPEN statement) or explicitly specify file attributes with the appropriate OPEN statement specifiers.

Specifiers for File and Unit Information

These specifiers identify file and unit information:

- UNIT specifies the logical unit number.
- NEWUNIT specifies that the Intel® Fortran run-time library should select an unused logical unit number.
- FILE (or NAME) and DEFAULTFILE specify the directory and/or file name of an external file.
- STATUS or TYPE indicates whether to create a new file, overwrite an existing file, open an existing file, or use a scratch file.
- STATUS or DISPOSE specifies the file existence status after CLOSE.

Specifiers for File and Record Characteristics

These specifiers identify file and record characteristics:

- ORGANIZATION indicates the file organization (sequential or relative).
- RECORDTYPE indicates which record type to use.
- FORM indicates whether records are formatted or unformatted.
- CARRIAGECONTROL indicates the terminal control type.
- RECL or RECORDSIZE specifies the record size.

Specifier for Special File Open Routine

USEROPEN names the routine that will open the file to establish special context that changes the effect of subsequent Intel Fortran I/O statements.

Specifiers for File Access, Processing, and Position

These specifiers identify file access, processing, and position:

- ACCESS indicates the access mode (direct, sequential, or stream).
- SHARED sets file locking for shared access. Indicates that other users can access the same file.
- NOSHARED sets file locking for exclusive access. Indicates that other users who use file locking mechanisms cannot access the same file.
- SHARE specifies shared or exclusive access; for example, SHARE='DENYNONE' or SHARE='DENYRW'.
- POSITION indicates whether to position the file at the beginning of file, before the end-of-file record, or leave it as is (unchanged).
- ACTION or READONLY indicates whether statements will be used to only read records, only write records, or both read and write records.
- MAXREC specifies the maximum record number for direct access.
- ASSOCIATEVARIABLE specifies the variable containing the next record number for direct access.
- ASYNCHRONOUS specifies whether input/output should be performed asynchronously.

Specifiers for Record Transfer Characteristics

These specifiers identify record transfer characteristics:

- BLANK indicates whether to ignore blanks in numeric fields.
- DELIM specifies the delimiter character for character constants in list-directed or namelist output.
- PAD, when reading formatted records, indicates whether padding characters should be added if the item list and format specification require more data than the record contains.
- BUFFERED indicates whether buffered or non-buffered I/O should be used.
- BLOCKSIZE specifies the physical I/O buffer or transfer size.
- BUFFERCOUNT specifies the number of physical I/O buffers.
- CONVERT specifies the format of unformatted numeric data.

Specifiers for Error-Handling Capabilities

These specifiers are used for error handling:

- ERR specifies a label to branch to if an error occurs.
- IOSTAT specifies the integer variable to receive the error (IOSTAT) number if an error occurs.

Specifier for File Close Action

DISPOSE identifies the action to take when the file is closed.

Specifying File Locations in an OPEN Statement

You can use the FILE and DEFAULTFILE specifiers of the OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. (The Language Reference Manual describes the OPEN statement in greater detail.)

For example:

```
OPEN (UNIT=4, FILE='/usr/users/smith/test.dat', STATUS='OLD')
```

The file `test.dat` in directory `/usr/users/smith` is opened on logical unit 4. No defaults are applied, because both the directory and file name were specified. The value of the FILE specifier can be a character constant, variable, or expression.

In the following interactive example, the user supplies the file name and the DEFAULTFILE specifier supplies the default values for the full pathname string. The file to be opened is located in `/usr/users/smith` and is concatenated with the file name typed by the user into the variable DOC:

```
CHARACTER(LEN=9) DOC
WRITE (6,*)  'Type file name '
READ (5,*) DOC
OPEN (UNIT=2, FILE=DOC, DEFAULTFILE='/usr/users/smith',STATUS='OLD')
```

A slash (backslash on Windows systems) is appended to the end of the default file string if it does not have one.

Obtaining File Information: INQUIRE Statement

The INQUIRE statement returns information about a file and has the following forms:

- Inquiry by unit
- Inquiry by file name
- Inquiry by output item list
- Inquiry by directory

Inquiry by Unit

An inquiry by unit is usually done for an opened (connected) file. An inquiry by unit causes the Intel® Fortran RTL to check whether the specified unit is connected or not. One of the following occurs, depending on whether the unit is connected or not:

If the unit is connected:

- The EXIST and OPENED specifier variables indicate a true value.
- The pathname and file name are returned in the NAME specifier variable (if the file is named).
- Other information requested on the previously connected file is returned.
- Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement.
- The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

If the unit is not connected:

- The OPENED specifier indicates a false value.
- The unit NUMBER specifier variable is returned as a value of -1.
- Any other information returned will be undefined or default values for the various specifiers.

For example, the following INQUIRE statement shows whether unit 3 has a file connected (OPENED specifier) in logical variable I_OPENED, the name (case-sensitive) in character variable I_NAME, and whether the file is opened for READ, WRITE, or READWRITE access in character variable I_ACTION:

```
INQUIRE (3, OPENED=I_OPENED, NAME=I_NAME, ACTION=I_ACTION)
```

Inquiry by File Name

An inquiry by name causes the Intel Fortran RTL to scan its list of open files for a matching file name. One of the following occurs, depending on whether a match occurs or not:

If a match occurs:

- The EXIST and OPENED specifier variables indicate a true value.
- The pathname and file name are returned in the NAME specifier variable.
- The UNIT number is returned in the NUMBER specifier variable.
- Other information requested on the previously connected file is returned.
- Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement.

- The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

If no match occurs:

- The OPENED specifier variable indicates a false value.
- The unit NUMBER specifier variable is returned as a value of -1.
- The EXIST specifier variable indicates (true or false) whether the named file exists on the device or not.
- If the file does exist, the NAME specifier variable contains the pathname and file name.
- Any other information returned will be default values for the various specifiers, based on any information specified when calling INQUIRE.

The following INQUIRE statement returns whether the file named log_file is connected in logical variable I_OPEN, whether the file exists in logical variable I_EXIST, and the unit number in integer variable I_NUMBER:

```
INQUIRE (FILE='log_file', OPENED=I_OPEN, EXIST=I_EXIST, NUMBER=I_NUMBER)
```

Inquiry by Output Item List

Unlike inquiry by unit or inquiry by name, inquiry by output item list does not attempt to access any external file. It returns the length of a record for a list of variables that would be used for unformatted WRITE, READ, and REWRITE statements. The following INQUIRE statement returns the maximum record length of the variable list in variable I_RECLENGTH. This variable is then used to specify the RECL value in the OPEN statement:

```
INQUIRE (IOLENGTH=I_RECLENGTH) A, B, H
OPEN (FILE='test.dat', FORM='UNFORMATTED', RECL=I_RECLENGTH, UNIT=9)
```

For an unformatted file, the IOLENGTH value is returned using 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

Inquiry by Directory

An inquiry by directory verifies that a directory exists.

If the directory exists:

- The EXIST specifier variable indicates a true value.
- The full directory pathname is returned in the DIRSPEC specifier variable.

If the directory does not exist:

- The EXIST specified variable indicates a false value.
- The value of the DIRSPEC specifier variable is unchanged.

For example, the following INQUIRE statement returns the full directory pathname:

```
LOGICAL        ::L_EXISTS
CHARACTER (255)::C_DIRSPEC
INQUIRE (DIRECTORY=".", DIRSPEC=C_DIRSPEC, EXIST=L_EXISTS)
```

The following INQUIRE statement verifies that a directory does not exist:

```
INQUIRE (DIRECTORY="I-DO-NOT-EXIST", EXIST=L_EXISTS)
```

Closing Files: CLOSE Statement

Usually, any external file opened should be closed by the same program before it completes. The CLOSE statement flushes any output buffers and disconnects the unit and its external file. You must specify the unit number (UNIT specifier) to be closed.

You can also specify:

- Whether the file should be deleted or kept (STATUS specifier)
- Error handling information (ERR and IOSTAT specifiers)

To delete a file when closing it:

- In the OPEN statement, specify the ACTION keyword (such as ACTION='READ'). Avoid using the READONLY keyword, because a file opened using the READONLY keyword cannot be deleted when it is closed.
- In the CLOSE statement, specify the keyword STATUS='DELETE'.

If you opened an external file and did an inquire by unit, but do not like the default value for the ACCESS specifier, you can close the file and then reopen it, explicitly specifying the ACCESS desired.

Typically, it is not necessary to close preconnected units. Internal files are neither opened nor closed.

Record I/O Statement Specifiers

After you open a file or use a preconnected file, you can use the following statements:

- READ, WRITE, ACCEPT, and PRINT to perform record I/O.
- BACKSPACE, ENDFILE, and REWIND to set record position within the file.
- DELETE, REWRITE, TYPE, and FIND to perform various operations.

The record I/O statement must use the appropriate record I/O form (formatted, list-directed, namelist, or unformatted).

You can use the following specifiers with the READ and WRITE record I/O statements:

- UNIT specifies the unit number to or from which input or output will occur.
- END specifies a label to branch to if end-of-file occurs; only applies to input statements on sequential files.
- ERR specifies a label to branch to if an error occurs.
- IOSTAT specifies an integer variable to contain the error number if an error occurs.
- FMT specifies a label of a FORMAT statement or character data specifying a FORMAT.
- NML specifies the name of a NAMELIST.
- REC specifies a record number for direct access.

When using nonadvancing I/O, use the ADVANCE, EOR, and SIZE specifiers.

When using the REWRITE statement, you can use the UNIT, FMT, ERR, and IOSTAT specifiers.

File Sharing (Linux* and macOS*)

This topic only applies to Linux* and macOS* operating systems.

Depending on the value specified by the ACTION (or READONLY) specifier in the OPEN statement, the file will be opened by your program for reading, writing, or both reading and writing records. This simply checks that the program itself executes the type of statements intended.

File locking mechanisms allow users to enable or restrict access to a particular file when that file is being accessed by another process.

Intel® Fortran file locking features provide three file access modes:

- Implicit Shared mode, which occurs when no mode is specified. This is also called No Locking.
- Explicit Shared mode, when all cooperating processes have access to a file. This mode is set in the OPEN statement by the SHARED specifier or the SHARE='DENYNONE' specifier.
- Exclusive mode, when only one process has access to a file. This mode is set in the OPEN statement by the NOSHARED specifier or the SHARE='DENYRW' specifier.

The file locking mechanism looks for explicit setting of the corresponding specifier in the OPEN statement. Otherwise, the Fortran run time does not perform any setting or checking for file locking and the process can access the file regardless of the fact that other processes have already opened or locked the file.

Example 1: Implicit Shared Mode (No Locking)

Process 1 opens the file without a specifier, resulting in no locking.

Process 2 now tries to open the file:

- It gains access regardless of the mode it is using.

Example 2: Explicit Shared Mode

Process 1 opens the file with Explicit Shared mode.

Process 2 now tries to open the file:

- If process 2 opens the file with Explicit Shared mode or Implicit Shared (No Locking) mode, it gets access to the file.
- If process 2 opens the file with Exclusive mode, it receives an error.

Example 3: Exclusive Mode

Process 1 opens the file with Exclusive mode.

Process 2 now tries to open the file:

- If process 2 opens the file with Implicit Shared (No Locking) mode, it gets access to the file.
- If process 2 opens the file with Explicit Shared or Exclusive mode, it receives an error.

The Fortran runtime does not coordinate file entry updates during cooperative access. The user needs to coordinate access times among cooperating processes to handle the possibility of simultaneous WRITE and REWRITE statements on the same record positions.

Specifying the Initial Record Position

When you open a disk file, you can use the OPEN statement POSITION specifier to request one of the following initial record positions within the file:

- The initial position before the first record (POSITION='REWIND')

  A sequential access READ or WRITE statement will read or write the first record in the file.
- A point beyond the last record in the file (POSITION='APPEND'), just before the end-of-file record, if one exists

  For a new file, this is the initial position before the first record (same as 'REWIND'). You might specify 'APPEND' before you write records to an existing sequential file using sequential access.
- The current position (POSITION='ASIS')

  This is usually used only to maintain the current record position when reconnecting a file. The second OPEN specifies the same unit number and specifies the same file name (or omits it), which leaves the file open, retaining the current record position. However, if the second OPEN specifies a different file name for the same unit number, the current file will be closed and the different file will be opened.

The following I/O statements allow you to change the current record position:

- REWIND sets the record position to the initial position before the first record. A sequential access READ or WRITE statement would read or write the first record in the file.
- BACKSPACE sets the record position to the previous record in a file. Using sequential access, if you wrote record 5, issued a BACKSPACE to that unit, and then read from that unit, you would read record 5.
- ENDFILE writes an end-of-file marker. This is typically done after writing records using sequential access just before you close the file.

Unless you use nonadvancing I/O, reading and writing records usually advances the current record position by one record. More than one record might be transferred using a single record I/O statement.

Advancing and Nonadvancing Record I/O

After you open a file, if you omit the ADVANCE specifier (or specify ADVANCE= 'YES') in READ and WRITE statements, advancing I/O (normal Fortran I/O) will be used for record access. When using advancing I/O:

- Record I/O statements transfer one entire record (or multiple records).
- Record I/O statements advance the current record position to a position before the next record.

You can request nonadvancing I/O for the file by specifying the ADVANCE= 'NO' specifier in a READ and WRITE statement. You can use nonadvancing I/O only for sequential access to external files using formatted I/O (not list-directed or namelist).

When you use nonadvancing I/O, the current record position does not change, and part of the record might be transferred, unlike advancing I/O where one or more entire records are always transferred.

You can alternate between advancing and nonadvancing I/O by specifying different values for the ADVANCE specifier ('YES' and 'NO') in the READ and WRITE record I/O statements.

When reading records with either advancing or nonadvancing I/O, you can use the END specifier to branch to a specified label when the end of the file is read.

Because nonadvancing I/O might not read an entire record, it also supports an EOR specifier to branch to a specified label when the end of the record is read. If you omit the EOR and the IOSTAT specifiers when using nonadvancing I/O, an error results when the end-of-record is read.

When using nonadvancing input, you can use the SIZE specifier to return the number of characters read. For example, in the following READ statement, SIZE=X (where variable X is an integer) returns the number of characters read in X and an end-of-record condition causes a branch to label 700:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

User-Supplied OPEN Procedures: USEROPEN Specifier

You can use the USEROPEN specifier in an OPEN statement to pass control to a routine that directly opens a file. The called routine can use system calls or library routines to open the file and may establish special context that changes the effect of subsequent I/O statements.

The Intel® Fortran Run-Time Library (RTL) I/O support routines call the USEROPEN function in place of the system calls usually used when the file is first opened for I/O. The USEROPEN specifier in an OPEN statement specifies the name of a function to receive control.

The called function must open a file (or pipe) and return the file descriptor of the file (or pipe) it has opened when control is returned to the RTL. The called function may specify different options when it opens the file than a normal OPEN statement would. It may specify a different file.

You can obtain the file descriptor from the Intel® Fortran RTL for a specific unit number by using the PXFFILENO routine.

Although the called function can be written in other languages (such as Fortran), C is usually the best choice for making system calls, such as open or create.

NOTE
If your application requires that you use C to perform the file open and close, as well as all record operations, call the appropriate C procedure from the Intel® Fortran program without using the Fortran OPEN statement.

NOTE
If a filename has been specified in the OPEN statement that included the USEROPEN specifier, any subsequent CLOSE statement specifying STATUS=DELETE (or DISPOSE=DELETE) only acts on the filename specified in the OPEN statement. If you specified a different filename in the function named in USEROPEN, the CLOSE statement will have no effect on that filename.

Syntax and Behavior of the USEROPEN Specifier

The USEROPEN specifier for the OPEN statement has the form:

USEROPEN = *function-name*

The *function-name* represents the name of an external function. The external function can be written in Fortran, C, or other languages.

The return value is the *file descriptor*. If an error occurs in the function, it should return -1.

In the calling program, the function must be declared in an EXTERNAL statement. For example, the following Intel® Fortran code can be used to call the USEROPEN procedure UOPEN (known to the linker as `uopen_`):

```
EXTERNAL  UOPEN
INTEGER   UOPEN
.
.
.
OPEN (UNIT=10, FILE='/usr/test/data', STATUS='NEW', USEROPEN=UOPEN)
```

During the execution of the OPEN statement, the external procedure called `uopen_` receives control. The function opens the file, may perform other operations, and subsequently returns control (with the file descriptor) to the RTL. You can use other system calls or library routines within the USEROPEN function.

In most cases, the USEROPEN function modifies the open flags argument passed by the Intel® Fortran RTL or uses a new value before the open (or create) system call. After the function opens the file, it must return control to the RTL.

On Linux and macOS*, the *file descriptor* is a 4-byte integer on both 32-bit and 64-bit systems. On Windows, the *file descriptor* is a 4-byte integer on 32-bit systems and an 8-byte integer on 64-bit systems:

- If the USEROPEN function is written in C, declare it as a C function.
- If the USEROPEN function is written in Fortran, declare it as a FUNCTION, perhaps with an interface block.

The called function must return the *file descriptor* to the RTL, or -1 if the function detects an error .

The following shows the available arguments and definitions for Linux* and macOS*, and then for Windows*:

Linux* and macOS* Arguments and Definitions:

```
int   uopen_ (          (1)
char  *file_name,       (2)
int   *open_flags,      (3)
int   *create_mode,     (4)
int   *lun,             (5)
int   file_length);     (6)
```

On Linux* and macOS* systems, the function definition and the arguments passed from the Intel® Fortran RTL are as follows:

1. The function must be declared as a 4-byte integer (int).
2. Indicates the pathname to be opened; the pathname includes the file name.
3. Indicates the open flags. The open flags are described in the header file `/usr/include/sys/file.h` or open(2).

4. Indicates the create mode, which is the protection needed when creating a Linux* OS-style file. The create modes are described in open(2).
5. Indicates the logical unit number.
6. Indicates the pathname length (hidden character length argument of the pathname).

Argument Notes for Linux* and macOS*:

The open system call (see open(2)) requires the passed pathname, the open flags (which define the type access needed, whether the file exists, and so on), and the create mode. The logical unit number specified in the OPEN statement is passed in case the USEROPEN function needs it. The hidden character length of the pathname is also passed.

Windows* Arguments and Definitions:

```
int uopen_ (                              (1)
char   *filename,                         (2)
int    *desired_access,                   (3)
int    *share_mode,                       (4)
int     a_null,          /* always 0 */   (5)
int    *flags_attr,                       (6)
int     b_null,          /* always 0 */   (7)
int    *unit,                             (8)
int    *flen);                            (9)
```

On Windows* systems, the function definition and the arguments passed from the Intel® Fortran RTL are as follows:

1. The function must be declared as a 4-byte integer (int) on 32-bit systems and an 8-byte integer (long long int) on 64-bit systems.
2. Indicates the pathname to be opened; the pathname includes the file name.
3. Indicates the mode of access. It can be set to read, write, or read/write.
4. Indicates the file protection mode.
5. This is a NULL that is passed as a literal zero by value.
6. This sets flags that specify file modes and several kinds of file features (such as whether to use sequential access or random access, whether to delete on close, etc.)
7. This is a NULL that is passed as a literal zero by value.
8. Indicates the logical unit number.
9. Indicates the pathname length (the hidden character length argument of the pathname).

Argument Notes for Windows*:

The argument list for a USEROPEN routine on Windows is very similar to the argument list for the Microsoft* Windows function CreateFile. This lets you easily write a USEROPEN routine and pass the input arguments to a call to CreateFile. The CreateFile system call requires the filename, the desired_access, the shared_mode, and the flags_attr. These arguments have been set to reflect the file semantics requested in the OPEN statement. The logical unit number specified in the OPEN statement is passed in case the USEROPEN function needs it. The hidden character length of the pathname is also passed.

On 32-bit Windows*, a Fortran USEROPEN function must use the default "C, Reference" calling convention. If you have used the `iface` compiler option to change the default calling convention to "stdcall" or "cvf", you will need to add a !DIR$ ATTRIBUTES DEFAULT directive in the function source to have it use the correct calling convention.

Restrictions of Called USEROPEN Functions

The Intel® Fortran RTL uses exactly one file descriptor per logical unit, which must be returned by the called function. Because of this, only certain system calls or library routines can be used to open the file.

On Linux* systems, system calls and library routines which do not return a file descriptor include `mknod` (see `mknod(2)`) and `fopen` (see `fopen(3)`). For example, the `fopen` routine returns a file pointer instead of a file descriptor.

The following Intel® Fortran code calls the USEROPEN function named UOPEN:

```
EXTERNAL  UOPEN
INTEGER   UOPEN
.
.
.
OPEN (UNIT=1,FILE='ex1.dat',STATUS='NEW',USEROPEN=UOPEN,
ERR=9,IOSTAT=errnum)
```

If UOPEN is a Fortran function, its name is decorated appropriately for Fortran.

Likewise, if UOPEN is a C function, its name is decorated appropriately for C, as long as the following line is included in the above code:

```
!DIR$ ATTRIBUTES C::UOPEN
```

Compiling and Linking the C and Intel® Fortran Programs

Use the `icc` or `icl` command to compile the called `uopen` C function `uopen.c`, and the `ifort` command to compile the Intel® Fortran calling program `ex1.f`. The same `ifort` command also links both object files by using the appropriate libraries:

```
icc -c uopen.c (Linux* OS)
icl -c uopen.c (Windows* OS)
ifort ex1.f uopen.o
```

Examples

The following shows an example on Linux and macOS* systems and an example on Windows systems.

Example on Linux and macOS* systems:

```
      PROGRAM UserOpenMain
      IMPLICIT NONE

      EXTERNAL      UOPEN
      INTEGER(4)    UOPEN

      CHARACTER(10) :: FileName="UOPEN.DAT"
      INTEGER       :: IOS
      CHARACTER(255):: InqFullName
      CHARACTER(100):: InqFileName
      INTEGER       :: InqLun
      CHARACTER(30) :: WriteOutBuffer="Write_One_Record_to_the_File. "
      CHARACTER(30) :: ReadInBuffer  ="??????????????????????????????"

110   FORMAT( X,"FortranMain: ",A," Created (iostat=",I0,")")
115   FORMAT( X,"FortranMain: ",A,": Creation Failed (iostat=",I0,")")
120   FORMAT( X,"FortranMain: ",A,": ERROR: INQUIRE Returned Wrong FileName")
130   FORMAT( X,"FortranMain: ",A,": ERROR: ReadIn and WriteOut Buffers Do Not Match")

      WRITE(*,'(X,"FortranMain: Test the USEROPEN Facility of Open")')

      OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='REPLACE',USEROPEN=UOPEN, &
           IOSTAT=ios, ACTION='READWRITE')
```

```
!     When the OPEN statement is executed, the uopen_ function receives control.
!     The uopen_ function opens the file by calling open(), and subsequently
!     returns control with the handle returned by open().

      IF (IOS .EQ. 0)  THEN
         WRITE(*,110) TRIM(FileName), IOS
         INQUIRE(10, NAME=InqFullName)
         CALL ParseForFileName(InqFullName,InqFileName)
         IF (InqFileName .NE. FileName) THEN
             WRITE(*,120) TRIM(FileName)
         END IF
      ELSE
         WRITE(*,115) TRIM(FileName), IOS
         GOTO 9999
      END IF

      WRITE(10,*) WriteOutBuffer
      REWIND(10)
      READ(10,*) ReadInBuffer
      IF (ReadinBuffer .NE. WriteOutbuffer) THEN
         WRITE(*,130) TRIM(FileName)
      END IF

      CLOSE(10)
      WRITE(*,'(X,"FortranMain: Test of USEROPEN Completed")')

9999  CONTINUE
      END

!-----------------------------------------------------------------
! SUBROUTINE: ParseForFileName
!             Takes a full pathname and returns the filename
!             with its extension.
!-----------------------------------------------------------------
      SUBROUTINE ParseForFileName(FullName,FileName)

      CHARACTER(255):: FullName
      CHARACTER(255):: FileName
      INTEGER       :: P

      P = INDEX(FullName,'/',.TRUE.)
      FileName = FullName(P+1:)

      END

//
// File: UserOpen_Sub.c
//
// This routine opens a file using data passed from the Intel(c) Fortran OPEN statement.
//

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
```

```
#include <errno.h>
int uopen_ ( char *file_name,   /* access read: name of the file to open (null terminated) */
             int  *open_flags,  /* access read: READ/WRITE, see file.h or open(2) */
             int  *create_mode, /* access read: set if the file is to be created */
             int  *unit_num,    /* access read: logical unit number to be opened */
             int  filenam_len ) /* access read: number of characters in file_name */
{
    /*
    ** The returned value is the following:
    **    value != -1 is a valid file descriptor
    **    value == -1 is returned on an error
    */
    int return_value;

    printf(" %s: Opening FILENAME = %s\n", __FILE__, file_name);
    printf(" %s: open_flags = 0x%8.8x\n", __FILE__, *open_flags);
    if ( *open_flags & O_CREAT ) {
        printf(" %s: the file is being created, create_mode = 0x%8.8x\n", __FILE__,
*create_mode);
    }

    printf(" %s: open() ", __FILE__);
    return_value = open(file_name, *open_flags, *create_mode);
    if (return_value != 0) {
        printf("FAILED.\n");
        return_value = -1;
    } else {
        printf("SUCCEEDED.\n");
    }

    return (return_value);
} /* end of uopen_() */
```

Example on Windows systems:

In the calling Fortran program, the function named in USEROPEN must first be declared in an EXTERNAL statement. For example, the following Fortran code might be used to call the USEROPEN procedure UOPEN:

```
 IMPLICIT INTEGER (A-Z)
 EXTERNAL UOPEN
 INTEGER(INT_PTR_KIND()) UOPEN
 ...
 OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='NEW',USEROPEN=UOPEN)
```

When the OPEN statement is executed, the UOPEN function receives control. The function opens the file by calling CreateFile( ), performs whatever operations were specified, and subsequently returns control (with the *handle* returned by CreateFile( )) to the calling Fortran program.

Here is what the UOPEN function might look like:

```
       INTEGER(INT_PTR_KIND()) FUNCTION UOPEN( FILENAME,      &
                               DESIRED_ACCESS, &
                               SHARE_MODE,     &
                               A_NULL,         &
                               CREATE_DISP,    &
                               FLAGS_ATTR,     &
                               B_NULL,         &
                               UNIT,           &
                               FLEN )
```

```
       !DIR$ ATTRIBUTES C, ALIAS:'_UOPEN' :: UOPEN
       !DIR$ ATTRIBUTES REFERENCE :: FILENAME
       !DIR$ ATTRIBUTES REFERENCE :: DESIRED_ACCESS
       !DIR$ ATTRIBUTES REFERENCE :: SHARE_MODE
       !DIR$ ATTRIBUTES REFERENCE :: CREATE_DISP
       !DIR$ ATTRIBUTES REFERENCE :: FLAGS_ATTR
       !DIR$ ATTRIBUTES REFERENCE :: UNIT

       USE IFWIN
       IMPLICIT INTEGER (A-Z)
       CHARACTER*(FLEN) FILENAME
       TYPE(T_SECURITY_ATTRIBUTES), POINTER :: NULL_SEC_ATTR

! Set the FILE_FLAG_WRITE_THROUGH bit in the flag attributes to CreateFile( )
! (for whatever reason)
       FLAGS_ATTR = FLAGS_ATTR + FILE_FLAG_WRITE_THROUGH

! Do the CreateFile( ) call and return the status to the Fortran rtl
       STS = CreateFile( FILENAME,            &
                         DESIRED_ACCESS,      &
                         SHARE_MODE,          &
                         NULL_SEC_ATTR,       &
                         CREATE_DISP,         &
                         FLAGS_ATTR,          &
                         0 )

       UOPEN = STS
       RETURN
       END
```

The UOPEN function is declared to use the cdecl calling convention, so it matches the Fortran rtl declaration of a useropen routine.

The following function definition and arguments are passed from the Intel Fortran Run-time Library to the function named in USEROPEN:

```
       INTEGER(INT_PTR_KIND()) FUNCTION UOPEN( FILENAME,       &
                               DESIRED_ACCESS, &
                               SHARE_MODE,     &
                               A_NULL,         &
                               CREATE_DISP,    &
                               FLAGS_ATTR,     &
                               B_NULL,         &
                               UNIT,           &
                               FLEN )
       !DIR$ ATTRIBUTES C, ALIAS:'_UOPEN' :: UOPEN
       !DIR$ ATTRIBUTES REFERENCE :: DESIRED_ACCESS
       !DIR$ ATTRIBUTES REFERENCE :: SHARE_MODE
       !DIR$ ATTRIBUTES REFERENCE :: CREATE_DISP
       !DIR$ ATTRIBUTES REFERENCE :: FLAGS_ATTR
       !DIR$ ATTRIBUTES REFERENCE :: UNIT
```

The first 7 arguments correspond to the CreateFile( ) api arguments. The value of these arguments is set according the caller's OPEN( ) arguments:

FILENAME	Is the address of a null terminated character string that is the name of the file.

DESIRED_ACCESS	Is the desired access (read-write) mode passed by reference.
SHARE_MODE	Is the file sharing mode passed by reference.
A_NULL	Is always null. The Fortran runtime library always passes a NULL for the pointer to a SECURITY_ATTRIBUTES structure in its CreateFile( ) call.
CREATE_DISP	Is the creation disposition specifying what action to take on files that exist, and what action to take on files that do not exist. It is passed by reference.
FLAGS_ATTR	Specifies the file attributes and flags for the file. It is passed by reference.
B_NULL	Is always null. The Fortran runtime library always passes a NULL for the handle to a template file in it's CreateFile( ) call.

The last 2 arguments are the Fortran unit number and length of the file name:

UNIT	Is the Fortran unit number on which this OPEN is being done. It is passed by reference.
FLEN	Is the length of the file name, not counting the terminating null, and passed by value.

See Also

OPEN: USEROPEN Specifier
PXFFILENO

Microsoft Fortran PowerStation Compatible Files (Windows*)

This topic only applies to Windows*.

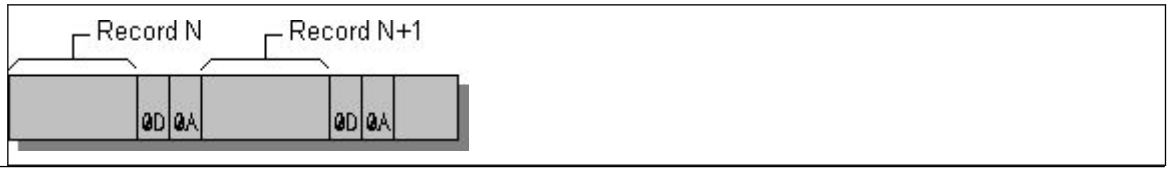
When using the `fpscomp` option for Microsoft* Fortran PowerStation compatibility, the following types of files are possible:

- Formatted Sequential
- Formatted Direct
- Unformatted Sequential
- Unformatted Direct
- Binary Sequential
- Binary Direct

Formatted Sequential Files

A formatted sequential file is a series of formatted records written sequentially and read in the order in which they appear in the file. Records can vary in length and can be empty. They are separated by carriage return (0D) and line feed (0A) characters as shown in the following figure.

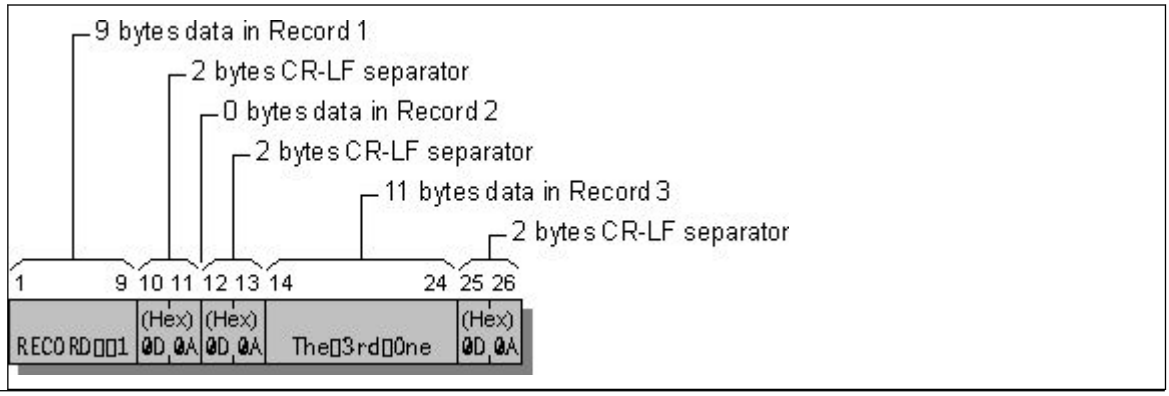
Formatted Records in a Formatted Sequential File



An example of a program writing three records to a formatted sequential file is given below. The resulting file is shown in the following figure.

```
  OPEN (3, FILE='FSEQ')
! FSEQ is a formatted sequential file by default.
  WRITE (3, '(A, I3)') 'RECORD', 1
  WRITE (3, '()')
  WRITE (3, '(A11)') 'The 3rd One'
  CLOSE (3)
  END
```

Formatted Sequential File



Formatted Direct Files

In a formatted direct file, all of the records are the same length and can be written or read in any order. The record size is specified with the RECL option in an OPEN statement and should be equal to or greater than the number of bytes in the longest record.

The carriage return (CR) and line feed (LF) characters are record separators and are not included in the RECL value. Once a direct-access record has been written, you cannot delete it, but you can rewrite it.

During output to a formatted direct file, if data does not completely fill a record, the compiler pads the remaining portion of the record with blank spaces. The blanks ensure that the file contains only completely filled records, all of the same length. During input, the compiler by default also adds filler bytes (blanks) to the input record if the input list and format require more data than the record contains.

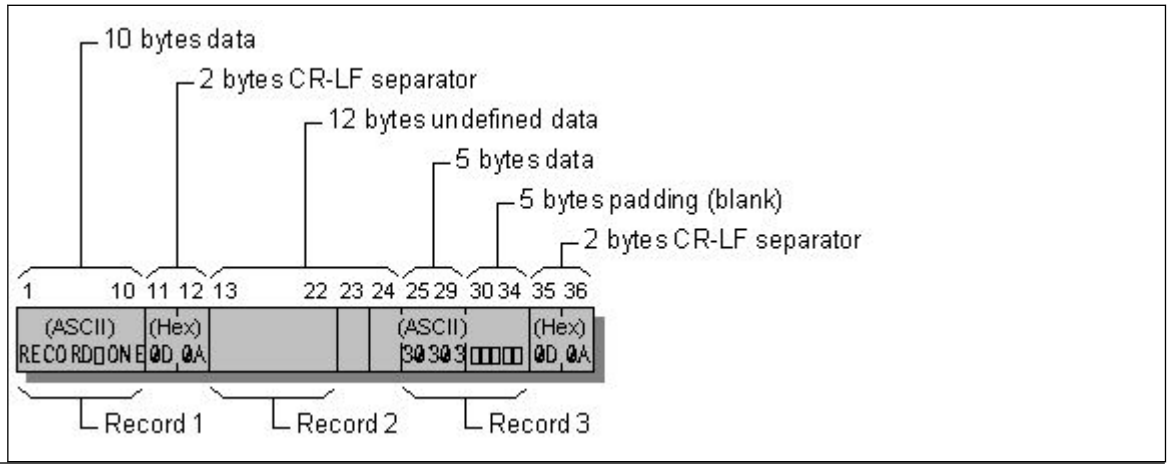
You can override the default blank padding on input by setting PAD='NO' in the OPEN statement for the file. If PAD='NO', the input record must contain the amount of data indicated by the input list and format specification. Otherwise, an error occurs. PAD='NO' has no effect on output.

An example of a program writing two records, record one and record three, to a formatted direct file is given below. The result is shown in the following figure.

```
OPEN (3,FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT',RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
```

```
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

Formatted Direct File



Unformatted Sequential Files

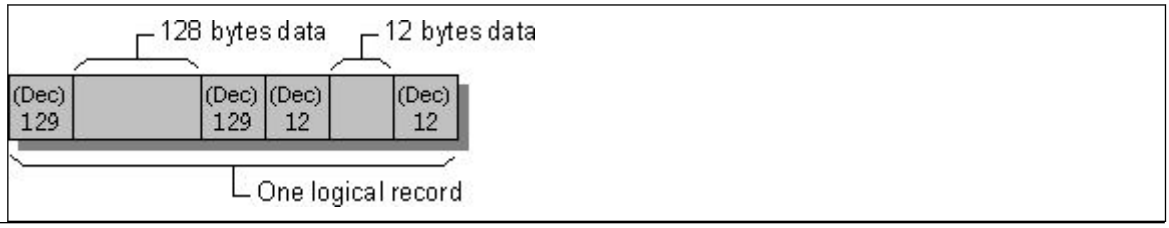
Unformatted sequential files are organized slightly differently on different platforms. This section describes unformatted sequential files created by Intel® Fortran when the `fpscomp` option is specified.

The records in an unformatted sequential file can vary in length. Unformatted sequential files are organized in chunks of 130 bytes or less called *physical blocks*. Each physical block consists of the data you send to the file (up to 128 bytes) plus two 1-byte "length bytes" inserted by the compiler. The length bytes indicate where each record begins and ends.

A *logical record* refers to an unformatted record that contains one or more physical blocks. (See the following figure.) Logical records can be as big as you want; the compiler will use as many physical blocks as necessary.

When you create a logical record consisting of more than one physical block, the compiler sets the length byte to 129 to indicate that the data in the current physical block continues on into the next physical block. For example, if you write 140 bytes of data, the logical record has the structure shown in the following figure.

Logical Record in Unformatted Sequential File



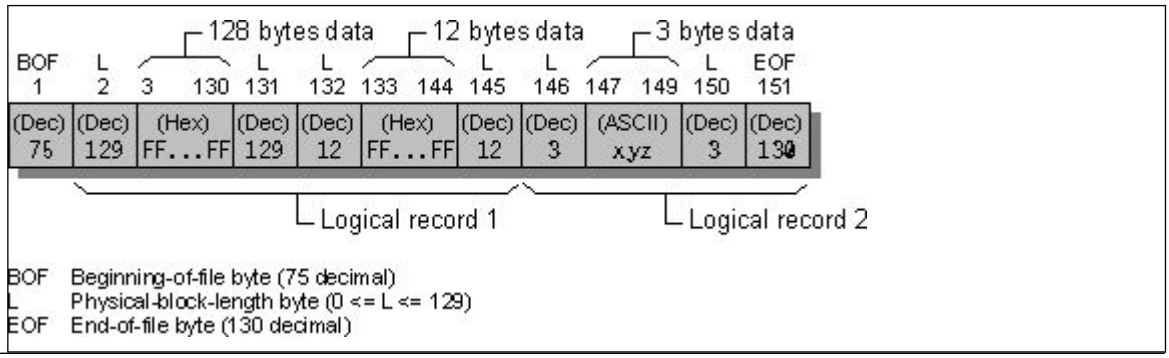
The first and last bytes in an unformatted sequential file are reserved; the first contains a value of 75, and the last holds a value of 130. Fortran uses these bytes for error checking and end-of-file references.

The following program creates the unformatted sequential file shown in the following figure:

```
!   Note: The file is sequential by default
!         -1 is FF FF FF FF hexadecimal.
```

```
!
  CHARACTER xyz(3)
  INTEGER(4) idata(35)
  DATA      idata /35 * -1/, xyz /'x', 'y', 'z'/
!
! Open the file and write out a 140-byte record:
! 128 bytes (block) + 12 bytes = 140 for IDATA, then 3 bytes for XYZ.
  OPEN (3, FILE='UFSEQ',FORM='UNFORMATTED')
  WRITE (3) idata
  WRITE (3) xyz
  CLOSE (3)
  END
```

Unformatted Sequential File



Unformatted Direct Files

An unformatted direct file is a series of unformatted records. You can write or read the records in any order you choose. All records have the same length, given by the RECL specifier in an OPEN statement. No delimiting bytes separate records or otherwise indicate record structure.

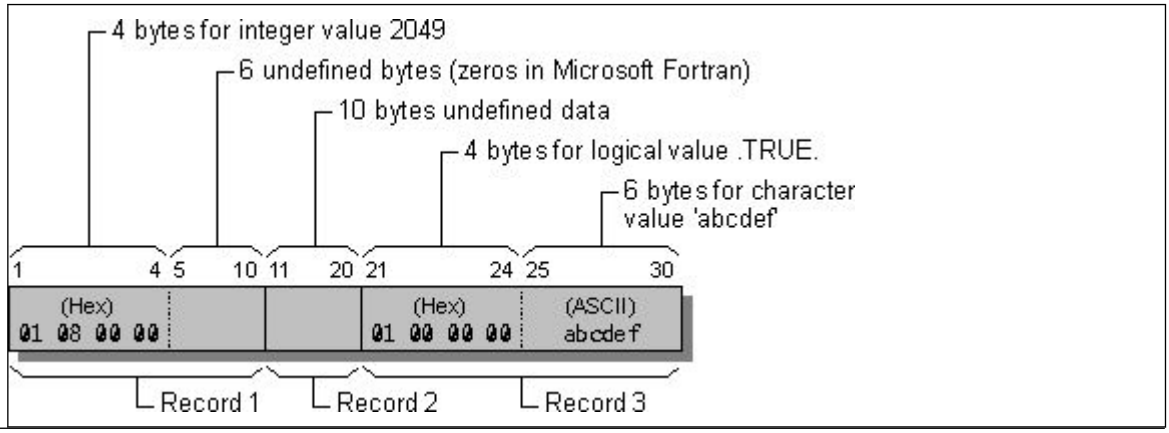
You can write a partial record to an unformatted direct file. Intel Fortran pads these records to the fixed record length with ASCII NULL characters. Unwritten records in the file contain undefined data.

The following program creates the sample unformatted direct file shown in the following figure:

```
OPEN (3, FILE='UFDIR', RECL=10,&
  & FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
WRITE (3, REC=3) .TRUE., 'abcdef'
```

```
  WRITE (3, REC=1) 2049
  CLOSE (3)
  END
```

Unformatted Direct File



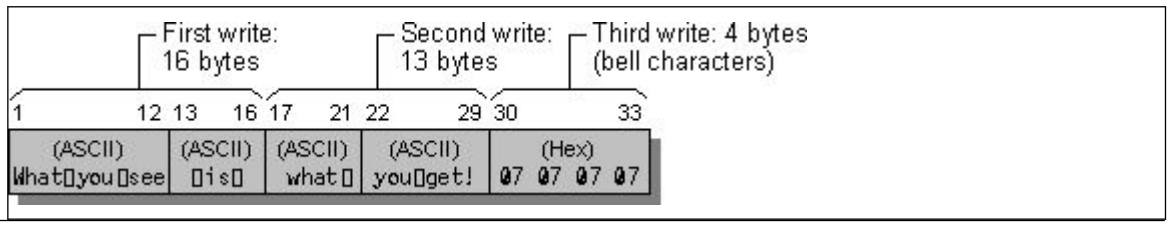
Binary Sequential Files

A binary sequential file is a series of values written and read in the same order and stored as binary numbers. No record boundaries exist, and no special bytes indicate file structure. Data is read and written without changes in form or length. For any I/O data item, the sequence of bytes in memory is the sequence of bytes in the file.

The next program creates the binary sequential file shown in the following figure:

```
!  NOTE: 07 is the bell character
!     Sequential is assumed by default.
!
  INTEGER(1) bells(4)
  CHARACTER(4) wys(3)
  CHARACTER(4) cvar
  DATA bells /4*7/
  DATA cvar /' is '/,wys /'What',' you',' see'/
  OPEN (3, FILE='BSEQ',FORM='BINARY')
  WRITE (3) wys, cvar
  WRITE (3) 'what ', 'you get!'
  WRITE (3) bells
  CLOSE (3)
  END
```

Binary Sequential File



Binary Direct Files

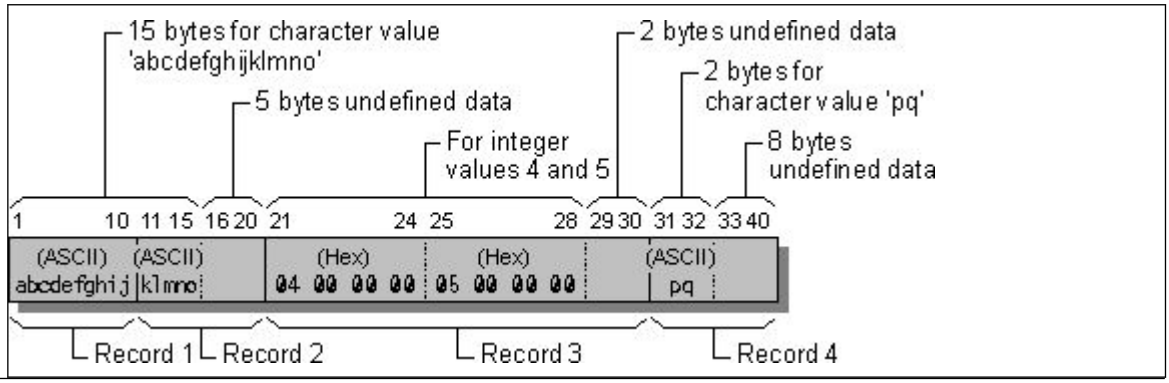
A binary direct file stores records as a series of binary numbers, accessible in any order. Each record in the file has the same length, as specified by the RECL argument to the OPEN statement. You can write partial records to binary direct files; any unused portion of the record will contain undefined data.

A single read or write operation can transfer more data than a record contains by continuing the operation into the next records of the file. Performing such an operation on an unformatted direct file would cause an error. Valid I/O operations for unformatted direct files produce identical results when they are performed on binary direct files, provided the operations do not depend on zero padding in partial records.

The following program creates the binary direct file shown in the following figure:

```
OPEN (3, FILE='BDIR',RECL=10,FORM='BINARY',ACCESS='DIRECT')
WRITE (3, REC=1) 'abcdefghijklmno'
WRITE (3) 4,5
WRITE (3, REC=4) 'pq'
CLOSE (3)  END
```

Binary Direct File



Using Asynchronous I/O

For external files, you can specify that I/O should be asynchronous. By doing this, you allow other statements to execute while an I/O statement is executing.

NOTE
In order to execute a program that uses asynchronous I/O on Linux* or macOS* systems, you must explicitly include one of the following compiler command line options when you compile and link your program:

- `-threads`
- `-reentrancy threaded`
- `-openmp`

On Windows* systems, no extra options are needed to execute a program that uses asynchronous I/O.

Using the ASYNCHRONOUS Specifier

Asynchronous I/O is supported for all READ and WRITE operations to external files. However, if you specify asynchronous I/O, you cannot use variable format expressions in formatted I/O operations.

To allow asynchronous I/O for a file, first specify ASYNCHRONOUS='YES' in its OPEN statement, then do the same for each READ or WRITE statement that you want to execute in this manner.

Execution of an asynchronous I/O statement initiates a "pending" I/O operation, which can be terminated in the following ways:

- by an explicit WAIT (initno) statement, which performs a wait operation for the specified pending asynchronous data transfer operation
- by a CLOSE statement for the file
- by a file-positioning statement such as REWIND or BACKSPACE
- by an INQUIRE statement for the file

Use the WAIT statement to ensure that the objects used in the asynchronous data transfer statements are not prematurely deallocated. (This is especially important for local stack objects and allocatable objects which may be deallocated before completion of the pending operation.) If you do not specify the wait operation, the program may terminate with an Access violation error. The following example shows use of the WAIT statement:

```
module mod
   real, allocatable :: X(:)
end module mod
subroutine sbr()
use mod
integer :: Y(500)
   !X and Y initialization
   allocate (X(500))
   call foo1(X, Y)
   !asynchronous writing
   open(1, asynchronous='yes')
   write(1, asynchronous='yes') X, Y
   !some computation
   call foo2()
   !wait operation
   wait(1)
   !X deallocation
   deallocate(X)
   !stack allocated object Y will be deallocated when the routine returns
end subroutine sbr
```

You can use the INQUIRE statement with the keyword of ASYNCHRONOUS (ASYNCHRONOUS=specifier) to determine whether asynchronous I/O is allowed. If it is allowed, a value of YES is returned.

Additionally, you can use the INQUIRE statement with the keyword of PENDING (PENDING= specifier) to determine whether previously pending asynchronous data transfers are complete.

If an ID= specifier appears and the specified data transfer operation is complete, the variable specified by PENDING is assigned the value False and the INQUIRE statement performs a wait operation for the specified data transfer.

If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit are complete, the variable specified by PENDING is assigned the value False and the INQUIRE statement performs wait operations for all previously pending data transfers for the specified unit.

Otherwise, the variable specified by PENDING is assigned the value True and no wait operations are performed. Previously pending data transfers remain pending.

Using the ASYNCHRONOUS Attribute

A data attribute called ASYNCHRONOUS specifies that a variable may be subject to asynchronous input/output. Assigning this attribute to a variable allows certain optimizations to occur.

See Also

Asynchronous Specifier (ASYNCHRONOUS=)
OPEN/ASYNCHRONOUS Specifier

INQUIRE/ASYNCHRONOUS Specifier
ASYNCHRONOUS Statement and Attributes

Mixed Language Programming

Programming with Mixed Languages Overview

Mixed-language programming is the process of building programs in which the source code is written in two or more languages. It allows you to:

- Call existing code that is written in another language
- Use procedures that may be difficult to implement in a particular language

In mixed-language programming, a routine written in one language calls a function, procedure, or subroutine written in another language. For example, a Fortran program may need to call an existing shared library or system procedure written in another language.

Although mixed language programming is possible between Intel® Fortran and other languages, the primary focus of this section is programming using Intel® Fortran and C. Mixed language programming between these languages is relatively straightforward for these reasons:

- Fortran implements functions, subroutines, and procedures in approximately the same way as C.
- Fortran provides many standard features to improve interoperability with C. An entity is considered to be interoperable if equivalent declarations are possible in both languages. Interoperability is provided for variables, derived types, and procedures. For more information, see Standard Fortran and C Interoperability.

Standard Fortran and C Interoperability

The Intel® Fortran Compiler supports the Fortran standardized mechanism for allowing Fortran code to reliably communicate (or *interoperate*) with C code.

This mechanism includes a group of features for C interoperability, enabling mixed-language programming in a more portable manner.

The Fortran standard discusses interoperability in terms of a "companion C processor." Each Fortran implementation is free to choose which C is its companion. Although the standard explicitly specifies a companion C (not C++) processor, you can use C++, as long as you use features that are compatible with C when interoperating with Fortran.

For Intel® Fortran, the supported C companion is the Intel® C++ Compiler or the Microsoft* Visual C++* Compiler on Windows*, and the Intel® C++ Compiler or gcc* on Linux* and macOS*.

The core principle of interoperability is that something should work the same way in Fortran as it does in C. In terms of interoperability, the following applies:

- Fortran provides a way to reference procedures that are defined by the C programming language or procedures that are described by C prototypes, even if they are not actually defined by means of C.
- Conversely, a procedure defined by a Fortran subprogram can be referenced by a function defined by means of C.
- In addition, you can define global variables that are associated with C variables whose names have external linkage.
- You can also declare Fortran variables, data structures and enumerations that correspond to similar declarations in C.

The following sections describe interoperability requirements for types, variables, procedures, and global data.

Example of Fortran Calling C

The following example calls a C function.

C Function Prototype Example

```
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts);
```

Fortran Module Example

```
module ftn_C_2
   interface
     integer (C_INT) function C_Library_Function &
     (sendbuf, sendcount, recvcounts) &
        BIND(C, name='C_Library_Function')
        use, intrinsic :: ISO_C_BINDING
        implicit none
        type (C_PTR), value :: sendbuf
        integer (C_INT), value :: sendcount
        type (C_PTR), value :: recvcounts
     end function C_Library_Function
   end interface
end module ftn_C_2
```

Fortran Calling Sequence Example

```
use, intrinsic :: ISO_C_BINDING, only: C_INT, C_FLOAT, C_LOC
use ftn_C_2
...
real (C_FLOAT), target :: send(100)
integer (C_INT) :: sendcount
integer (C_INT), ALLOCATABLE, target :: recvcounts(100)
...
ALLOCATE( recvcounts(100) )
...
call C_Library_Function(C_LOC(send), sendcount, &
C_LOC(recvcounts))
...
```

Example of C Calling Fortran

The following example calls a Fortran subroutine called `Simulation`. This subroutine corresponds to the C void function `simulation`.

Fortran Code Example

```
subroutine Simulation(alpha, beta, gamma, delta, arrays) BIND(C)
   use, intrinsic :: ISO_C_BINDING
   implicit none
   integer (C_LONG), value :: alpha
   real (C_DOUBLE), intent(inout) :: beta
   integer (C_LONG), intent(out) :: gamma
   real (C_DOUBLE),dimension(*),intent(in) :: delta
   type, BIND(C) :: pass
      integer (C_INT) :: lenc, lenf
      type (C_PTR) :: c, f
   end type pass
```

Fortran Code Example

```
   type (pass), intent(inout) :: arrays
   real (C_FLOAT), ALLOCATABLE, target, save :: eta(:)
   real (C_FLOAT), pointer :: c_array(:)
   ...
   ! Associate c_array with an array allocated in C
   call C_F_POINTER (arrays%c, c_array, (/arrays%lenc/) )
   ...
   ! Allocate an array and make it available in C
   arrays%lenf = 100
   ALLOCATE (eta(arrays%lenf))
   arrays%f = c_loc(eta)
   ...
end subroutine Simulation
```

C Struct declaration Example

```
struct pass {int lenc, lenf; float *c, *f;};
```

C Function Prototype Example

```
void simulation(long alpha, double *beta, long *gamma, double delta[], struct pass *arrays);
```

C Calling sequence Example

```
simulation(alpha, &beta, &gamma, delta, &arrays);
```

Using Standard Fortran Interoperability Syntax for Existing Fortran Extensions

Before the introduction of the Standard Fortran interoperability features, extensions provided by Intel® Fortran were used to facilitate programming using code written in Fortran and C. The Fortran 2003 standard defined new language features for C interoperability; in many cases, these features can be used instead of existing Intel® Fortran extensions. The following table lists the legacy Intel® Fortran extensions and shows their Standard Fortran equivalents.

NOTE
Existing code using legacy extensions will continue to work; however, if you are writing new code, you should use the standard syntax.

Legacy Extension	Standard Fortran Interoperability Equivalent
ATTRIBUTES ALIAS	Use BIND(C,NAME="alias-name"). The BIND(C) syntax also implies ATTRIBUTES DECORATE; the compiler applies whatever name decoration (leading or trailing underscores) that C

Legacy Extension	Standard Fortran Interoperability Equivalent
	would use for the name. If the NAME= keyword is omitted, Intel® Fortran will use the Fortran name converted to lowercase on all platforms. If the procedure has no arguments, you must indicate that with () when adding BIND. All arguments to a routine specified as "interoperable" (with BIND(C)) must themselves be interoperable. You can also specify BIND(C) on module variables and COMMON blocks.
ATTRIBUTES C	Use BIND(C) (see ALIAS above), which has a similar effect to ATTRIBUTES C except that it does not change the argument passing mechanism to be by-value. Use the Fortran standard VALUE attribute if you need to pass by value. BIND(C) also specifies that small records are passed and returned as function value results the same way C would, which may be different from the Intel® Fortran default. Like ATTRIBUTES C, BIND(C) lowercases the external name and adds any necessary name decoration.
ATTRIBUTES DECORATE	Typically used with ATTRIBUTES ALIAS. See ATTRIBUTES ALIAS above.
ATTRIBUTES DEFAULT	Not needed when using BIND(C); the compiler always uses the semantics of the C compiler regardless of the setting of command-line options such as `iface`.
ATTRIBUTES EXTERN	Use BIND(C) with a module variable.
ATTRIBUTES REFERENCE	Not needed when using BIND(C). Typically used with character arguments, or to override the implicit pass-by-value of ATTRIBUTES C.
ATTRIBUTES STDCALL	No equivalent. The use of STDCALL with BIND(C) is supported in Intel® Fortran and, for Windows* on IA-32 architecture only, it changes the external procedure name and stack conventions. For more information, see ATTRIBUTES C and STDCALL.
ATTRIBUTES VALUE	Use the Fortran-standard VALUE attribute. This has an additional effect when used with a Fortran procedure; the dummy argument is received by value and then copied to a temporary

Legacy Extension	Standard Fortran Interoperability Equivalent
	variable that can be modified within the procedure. Once the procedure exits, the temporary value is discarded.
ATTRIBUTES VARYING	No equivalent.
%LOC function	Use the C_LOC function from intrinsic module ISO_C_BINDING, which may be an appropriate substitute for variables. C_FUNLOC is the corresponding function for procedures.
%VAL function	Declare the argument with the VALUE attribute.
%REF function	No equivalent; this is the default when BIND(C) is used.

Standard Tools for Interoperability

ISO_C_BINDING

The intrinsic module provides a set of named constants and procedures that can assist you in programming with mixed languages.

Using Intrinsic Module Example

```
USE, INTRINSIC::ISO_C_BINDING
```

There are two groups of named constants included in this intrinsic module -- those that hold kind type parameter values for intrinsic types and those that provide a Fortran equivalent to some of the C special characters, including:

Named constant	C definition	Value
C_NULL_CHAR	null character	'\0'
C_ALERT	alert	'\a'
C_BACKSPACE	backspace	'\b'
C_FORMFEED	form feed	'\f'
C_NEW_LINE	new line	'\n'
C_CARRIAGE_RETURN	carriage return	'\r'
C_HORIZONTAL_TAB	horizontal tab	'\t'
C_VERTICAL_TAB	vertical tab	'\v'

The procedures included in ISO_C_BINDING are all generic, not specific. With the exception of C_F_POINTER and C_F_PROCPOINTER, all of the procedures are pure. They include:

- C_ASSOCIATED
- C_F_POINTER
- C_F_PROCPOINTER
- C_FUNLOC
- C_LOC

- C_SIZEOF

Additionally, ISO_C_BINDING includes the following derived types to interoperate with C pointers:

- C_PTR
- C_FUNPTR
- C_NULL_PTR
- C_NULL_FUNPTR

BIND(C)

Fortran 2003 introduced the *language-binding-spec* attribute, using the keyword BIND. The syntax is:

BIND(C [, NAME=*scalar-default-char-constant-expr*])

C is the only language name you can specify. Generally, an entity with the BIND(C) attribute behaves as if it were the corresponding entity in the companion C processor.

The optional NAME= specifier allows you to use a different external name than the one the standard prescribes, which is what the companion C processor would do with the Fortran name forced to lowercase. If the C name was mixed-case, specify the mixed-case name here without any name decoration, such as leading underscores. The compiler applies whatever decoration the C processor would with the same name. So NAME= is not a strict replacement for the ALIAS attribute extension, it is more like ALIAS combined with DECORATE.

BIND(C) is not equivalent to ATTRIBUTES C. Although these both downcase the external name, BIND(C) does not imply pass-by-value and has other effects that are different from ATTRIBUTES C.

You can specify ATTRIBUTES STDCALL for an interoperable procedure (a procedure whose declaration includes the BIND(C) language binding attribute). This combination has the following effects for Windows* applications targeting IA-32 architecture:

- The calling mechanism is changed to STDCALL, which affects how the stack is cleaned up on procedure exit.
- The external name from the BIND attribute is suffixed with `@n`, where n is the number of bytes to be removed from the stack on return.

No other effects from STDCALL, such as pass-by-value, are provided. The Fortran standard VALUE attribute (not ATTRIBUTES VALUE) may be used if desired. For all other platforms, specifying STDCALL with BIND(C) has no effect. For more information, see ATTRIBUTES C and STDCALL.

BIND(C) on procedures

BIND(C) is often used on procedures. It can be specified in INTERFACE blocks to specify the interface of an external interoperable procedure, and on any Fortran procedures that you want to make interoperable. For SUBROUTINEs and FUNCTIONs, you need to specify BIND(C) after the argument list. For functions, you can specify BIND(C) before or after the RESULT clause. For example:

```
SUBROUTINE INTEROP_SUB (ARG) BIND(C,NAME="InteropSub")

FUNCTION INTEROP_FUN (ARG) BIND(C,NAME="InteropFun") RESULT (IAMFUN)
```

The above includes an external name, which is not required. You can also specify BIND(C) in a PROCEDURE declaration, in which case it appears in the normal list of attributes before the :: separator.

Specifying BIND(C) for procedures results in the following:

- The external name is what the C compiler would use, with the Fortran name lowercased (unless NAME= is specified).
- Arguments are passed and received by reference (unless the VALUE attribute is specified for the argument).
- Only interoperable arguments are allowed.
- No hidden arguments are allowed.
- Function type must be interoperable and function values are returned exactly as the C compiler would (this mainly affects small derived types).

The standard allows for passing character strings to interoperable procedures. You can pass a character argument of default kind (kind C_CHAR is the default in Intel® Fortran); the corresponding dummy argument is an explicit-shape array of single characters. Because no length is passed, you will need to provide the length another way (usually by appending a NUL character on the end.)

The standard does not provide an easy way to write a Fortran interoperable routine that accepts a string from C. You must declare the dummy argument as an array of single characters. You can use the combination of the C_LOC and C_F_POINTER procedures, both defined in intrinsic module ISO_C_BINDING, to "cast" a character array argument to a Fortran pointer to a character string. An example follows:

```
subroutine Interop_sub(arg) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
CHARACTER, DIMENSION(*) :: ARG
CHARACTER(1000), POINTER :: LCL_ARG ! Local pointer to argument
CALL C_F_POINTER(C_LOC(ARG), LCL_ARG)
```

In the above, `ARG` comes in as an assumed-size array of single characters, with no length. The C_F_POINTER subroutine converts a C pointer to a Fortran pointer. C_LOC is used to get a C pointer (of type C_PTR) of the argument and convert it to the Fortran pointer `LCL_ARG`. The result is that you can use `LCL_ARG` as a character string.

In the example, `LCL_ARG` has an explicit length. There is no way to make this pointer the correct length of the incoming string. Use INDEX to find the location of the NUL character, or some other method, to obtain the length and then use that in subsequent references.

See Also

BIND

Interoperating with arguments using C descriptors

These are facilities for interoperable procedure interfaces that specify dummy arguments that are assumed-shape arrays, have assumed character length, or have the ALLOCATABLE, POINTER, or OPTIONAL attributes.

Assumed-rank

An assumed-rank object is a dummy variable whose rank is assumed from its actual argument. This facilitates interoperability with C functions that can accept arguments of arbitrary rank. The intrinsic function, RANK, can be used to obtain the rank of an assumed-rank variable.

A procedure must have an explicit interface if it has a dummy argument that is assumed-rank.

The SHAPE, SIZE, and UBOUND intrinsic functions are defined for an assumed-rank array that is associated with an assumed-size array.

An assumed-rank dummy argument may correspond to an actual argument of any rank. If the actual argument has rank zero, the dummy argument has rank zero; the shape is a zero-sized array and the LBOUND and UBOUND intrinsic functions, with no DIM argument, return zero-sized arrays. If the actual argument has rank greater than zero, the rank and extents of the dummy argument are assumed from the actual argument, including no final extent for an assumed-size array. If the actual argument is an array and the dummy argument has the ALLOCATABLE or POINTER attribute, the bounds of the dummy argument are assumed from the actual argument.

Assumed-type

An assumed-type object is a dummy variable declared as `TYPE(*)`. This simplifies interoperability with C formal parameters of type `(void *)`.

An explicit interface is required for a procedure that has a dummy argument that is assumed-type because an assumed-type dummy argument is polymorphic.

An assumed-type dummy argument must not correspond to an actual argument that is of derived type with type parameters, type-bound procedures, or final subroutines.

CFI_cdesc

A C descriptor is a C structure of type `CFI_cdesc` that is defined in the file `ISO_Fortran_binding.h`.

Restrictions for BIND(C)

If BIND(C) is specified for a procedure, each dummy argument must be an interoperable procedure or a variable that is interoperable, assumed shape, assumed rank, assumed type, of assumed character length, or has the ALLOCATABLE or POINTER attribute. If BIND(C)is specified for a function, the function result must be an interoperable scalar variable.

A dummy argument of a procedure that is BIND(C) must not have both the OPTIONAL and VALUE attributes.

A variable that is a dummy argument of a procedure that is BIND(C) must be of interoperable type or assumed type.

A coarray shall not be a dummy argument of a BIND(C) procedure.

The ALLOCATABLE or POINTER attribute must not be specified for a default-initialized dummy argument of a BIND(C) procedure.

Further Requirements

Variables with the ASYNCHRONOUS attribute can be used for asynchronous communications between Fortran and C procedures.

When a Fortran procedure that has an INTENT (OUT) allocatable dummy argument is invoked by a C function, and the actual argument in the C function is the address of a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure. When a C function is invoked from a Fortran procedure via an interface with an INTENT (OUT) allocatable dummy argument, and the actual argument in the reference to the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

ISO_Fortran_binding.h

The types, macros, and functions declared in `ISO_Fortran_binding.h` can be used by a C function to interpret C descriptors and allocate and deallocate objects represented by C descriptors. These provide a means to specify a C prototype that interoperates with a Fortran interface that has an allocatable, assumed character length, assumed-rank, assumed-shape, or data pointer dummy argument.

The `ISO_Fortran_binding.h` is a C header file that contains these definitions:

- The C structures `CFI_cdesc_t` and `CFI_dim_t`
- The typedef definitions for `CFI_attribute_t`, `CFI_index_t`, `CFI_rank_t`, and `CFI_type_t`
- The macro `CFI_CDESC_T` and macro definitions that expand to integer constants with various useful values
- The C function prototypes or macro definitions for `CFI_address`, `CFI_allocate`, `CFI_deallocate`, `CFI_establish`, `CFI_is_contiguous`, `CFI_section`, `CFI_select_part`, and `CFI_setpointer`. Some of these functions return an error indicator; this is an integer value that indicates whether an error condition was detected. The value zero `CFI_SUCCESS` indicates that no error condition was detected, and a nonzero value indicates which error condition was detected. The "Macros and typedefs in ISO_Fortran_binding.h" section lists the standard error conditions and the macro names for their corresponding error codes.

  In function arguments representing subscripts, bounds, extents, or strides, the ordering of the elements is the same as the ordering of the elements of the `dim` member of a C descriptor.

Restrictions on C functions interoperating with Fortran procedures

Any C function inter-operating with Fortran procedures must meet these restrictions:

- A C descriptor shall not be initialized, updated, or copied other than by calling the functions in `ISO_Fortran_binding.h`.
- If the address of a C descriptor is a formal parameter that corresponds to a Fortran actual argument or is a C actual argument that corresponds to a Fortran dummy argument,
  - the C descriptor shall not be modified if either the corresponding dummy argument in the Fortran interface has the INTENT(IN) attribute or the C descriptor is for a nonallocatable nonpointer object, and
  - the `base_addr` member of the C descriptor shall not be accessed before it is given a value if the corresponding dummy argument in the Fortran interface has the POINTER and INTENT(OUT) attributes.

  In this context, modification refers to any change to the location or contents of the C descriptor, including establishing and updating. The intent of these restrictions is that the C descriptors will remain intact at all times that they are accessible to an active Fortran procedure, so that the Fortran code is not required to copy them.
- Within a C function, an allocatable object shall be allocated or deallocated only by execution of the `CFI_allocate` and `CFI_deallocate` functions. A Fortran pointer can become associated with a target by execution of the CFI allocate function.
- Calling `CFI_allocate` or `CFI_deallocate` for a C descriptor changes the allocation status of the Fortran variable it describes and causes the allocation status of any associated allocatable variable to change accordingly.
- If the address of an object is the value of a formal parameter that corresponds to a nonpointer dummy argument in a BIND(C) interface, then
  - if the dummy argument has the INTENT (IN) attribute, the object must not be defined or become undefined, and
  - if the dummy argument has the INTENT (OUT) attribute, the object must not be referenced before it is defined.
- When a Fortran object is deallocated, or execution of its host function is completed, or its association status becomes undefined, all C descriptors and C pointers to any part of it become undefined, and any further use of them is undefined behavior.
- A C descriptor whose address is a formal parameter that corresponds to a Fortran dummy argument becomes undefined on return from a call to the function from Fortran. If the dummy argument does not have either the TARGET or ASYNCHRONOUS attribute, all C pointers to any part of the object described by the C descriptor become undefined on return from the call, and any further use of them is undefined behavior.
- When a Fortran object is deallocated, or execution of its host function is completed, or its association status becomes undefined, all C descriptors and C pointers to any part of it become undefined, and any further use of them is undefined behavior.
- If the address of a C descriptor is passed as an actual argument to a Fortran procedure, the C lifetime of the C descriptor shall not end before the return from the procedure call.
- If an object is passed to a Fortran procedure as a nonallocatable, nonpointer dummy argument, its lifetime shall not end before the return from the procedure call.
- If the lifetime of a C descriptor for an allocatable object that was established by C ends before the program exits, the object shall be unallocated at that time.
- If a Fortran pointer becomes associated with a C object, the association status of the Fortran pointer becomes undefined when the lifetime of the C object ends.

See Also

C Structures, Typedefs, and Macros for Interoperability

C Structures, Typedefs, and Macros for Interoperability

The C structures CFI_dim_t and CFI_cdesc_t

CFI_dim_t is a typedef name for a C structure. It is used to represent lower bound, extent, and memory stride information for one dimension of an array. CFI_dim_t contains at least the following members in any order:

- CFI_index_t lower_bound – The value of the lower bound for the dimension being described.
- CFI_index_t extent – The value is the number of elements in the dimension being described or -1 for the last dimension of an assumed-size array.
- CFI_index_t sm – The value is the memory stride for a dimension; this is the difference in bytes between the addresses of successive elements in the dimension being described.

CFI_cdesc_t is a typedef name for a C structure, which contains a flexible array member. The first three members of the structure are base_addr, elem_len, and version in that order. The final member is dim. All other members must be between version and dim, in any order.

- void * base_addr – If the object Interoperating with arguments using C descriptors is an unallocated allocatable variable or a pointer that is disassociated, the value is a null pointer. If the object has zero size, the value is not a null pointer but is processor-11 dependent. Otherwise, the value is the base address of the object being described. The base address of a scalar is its C address. The base address of an array is the C address of the first element in Fortran array element order.
- size_t elem_len – If the object is a scalar, the value is the storage size in bytes of the object; otherwise, the value is the storage size in bytes of an element of the object.
- int version – The value is the value of CFI_VERSION in the source file ISO_Fortran_binding.h that defined the format and meaning of this C descriptor when the descriptor was established.
- CFI_rank_t rank – The value is the number of dimensions of the Fortran object being described; if the object is a scalar, the value is zero.
- CFI_type_t type – The value is the specifier for the type of the object. Each interoperable intrinsic C type has a specifier. Specifiers are also provided to indicate that the type of the object is an interoperable structure, or is unknown. The macros listed in the table below provide values that correspond to each type code specifier.

Macros for Type Codes

Macro name	C Type
CFI_type_signed_char	signed char
CFI_type_short	short int
CFI_type_int	int
CFI_type_long	long int
CFI_type_long_long	long long int
CFI_type_size_t	size_t
CFI_type_int8_t	int8_t
CFI_type_int16_t	int16_t
CFI_type_int32_t	int32_t
CFI_type_int64_t	int64_t
CFI_type_int_least8_t	int_least8_t
CFI_type_int_least16_t	int_least16_t
CFI_type_int_least32_t	int_least32_t
CFI_type_int_least64_t	int_least64_t
CFI_type_int_fast8_t	int_fast8_t

Macro name	C Type
CFI_type_int_fast16_t	int_fast16_t
CFI_type_int_fast32_t	int_fast32_t
CFI_type_int_fast64_t	int_fast64_t
CFI_type_intmax_t	intmax_t
CFI_type_intptr_t	intptr_t
CFI_type_ptrdiff_t	ptrdiff_t
CFI_type_float	float
CFI_type_double	double
CFI_type_long_double	long double
CFI_type_float_Complex	float _Complex
CFI_type_double_Complex	double _Complex
CFI_type_long_double_Complex	long double _Complex
CFI_type_Bool	_Bool
CFI_type_char	char
CFI_type_cptr	void *
CFI_type_struct	interoperable C structure
CFI_type_other	not otherwise specified

  - The value for `CFI_type_other` is negative and distinct from all other type specifiers.
  - `CFI_type_struct` specifies a C structure that is interoperable with a Fortran derived type; its value is positive and distinct from all other type specifiers.
  - If a C type is not interoperable with a Fortran type and kind supported by the Fortran processor, its macro evaluates to a negative value.
  - Otherwise, the value for an intrinsic type is positive.
- `CFI_attribute_t attribute` – The value is the value of an attribute code that indicates whether the object described is allocatable, a data pointer, or a nonallocatable, nonpointer data object. The values are nonnegative and distinct.

  The macros listed in the table below provide values that correspond to each attribute code.

  Macros for Attribute Codes

Macro	Attribute code indicated
`CFI_attribute_pointer`	data pointer object
`CFI_attribute_allocatable`	allocatable object
`CFI_attribute_other`	nonallocatable, nonpointer object

  - `CFI_attribute_pointer` specifies a data object with the Fortran POINTER attribute.
  - `CFI_attribute_allocatable` specifies an object with the Fortran ALLOCATABLE attribute.
  - `CFI_attribute_other` specifies a nonallocatable nonpointer object.
- `CFI_dim_t dim` – The number of elements in the dim array is equal to the rank of the object. Each element of the array contains the lower bound, extent, and memory stride information for the corresponding dimension of the Fortran object.

For a C descriptor of an array pointer or allocatable array, the value of the lower_bound member of each element of the dim member of the descriptor is determined by argument association, allocation, or pointer association. For a C descriptor of a nonallocatable nonpointer object, the value of the lower_bound member of each element of the dim member of the descriptor is zero.

In a C descriptor of an assumed-size array, the extent member of the last element of the dim member has the value -1. The value of elem_len for a Fortran CHARACTER object is equal to the character length times the number of bytes of a single character of that kind. If the kind is C_CHAR, this value will be equal to the character length.

Macros and typedefs in ISO_Fortran_binding.h

Except for CFI_CDESC_T, each macro defined in ISO_Fortran_binding.h expands to an integer constant expression that is either a single token or a parenthesized expression that is suitable for use in #if preprocessing directives.

CFI_CDESC_T is a function-like macro that takes one argument, which is the rank of the C descriptor to create, and evaluates to an unqualified type of suitable size and alignment for defining a variable to use as a C descriptor of that rank. The argument shall be an integer constant expression with a value that is greater than or equal to zero and less than or equal to CFI_MAX_RANK. A pointer to a variable declared using CFI_CDESC_T can be cast to CFI_cdesc_t *. A variable declared using CFI_CDESC_T must not have an initializer.

The CFI_CDESC_T macro provides the memory for a C descriptor. The address of an entity declared using the macro is not usable as an actual argument corresponding to a formal parameter of type CFI_cdesc_t * without an explicit cast.

CFI_index_t is a typedef name for a standard signed integer type capable of representing the result of subtracting two pointers.

The CFI_MAX_RANK macro has a value equal to the largest rank supported, i.e., 31. CFI_rank_t is a typedef name for a standard integer type capable of representing the largest supported rank.

The CFI_VERSION macro has a processor-dependent value that encodes the version of the ISO_Fortran_binding.h source file containing this macro. This value is increased if a new version of the source file is incompatible with the previous version.

CFI_attribute_t is a typedef name for a standard integer type capable of representing the values of the attribute codes.

CFI_type_t is a typedef name for a standard integer type capable of representing the values for the supported type specifiers.

The macros in the table below are for use as error codes. The macro CFI_SUCCESS is the integer constant 0. The value of each macro other than CFI_SUCCESS is nonzero and is different from the values of the other error code macros.

Macros for Error Codes

Macro name	Error condition
CFI_SUCCESS	No error detected.
CFI_ERROR_BASE_ADDR_NULL	The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.
CFI_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.
CFI_INVALID_ELEM_LEN	The value supplied for the element length member of a C descriptor is not valid.
CFI_INVALID_RANK	The value supplied for the rank member of a C descriptor is not valid.
CFI_INVALID_TYPE	The value supplied for the type member of a C descriptor is not valid.
CFI_INVALID_ATTRIBUTE	The value supplied for the attribute member of a C descriptor is not valid.

Macro name	Error condition
CFI_INVALID_EXTENT	The value supplied for the extent member of a CFI_dim_t structure is not valid.
CFI_INVALID_DESCRIPTOR	A C descriptor is invalid in some way.
CFI_ERROR_MEM_ALLOCATION	Memory allocation failed.
CFI_ERROR_OUT_OF_BOUNDS	A reference is out of bounds.

See Also

Interoperating with arguments using C descriptors

Data Types

Fortran and C support many of the same data types, but there are no direct correlations.

One difference is that Fortran has the concept of kinds. C treats these kinds as distinct types. For example, the Fortran INTEGER. C has integer types that range from `short int` to `long long int`, and has specialty types such as `intptr_t`. Fortran may not have a corresponding kind. For each interoperable C `integer` type, the ISO_C_BINDING declares a named constant (PARAMETER) that gives the kind number for the implementation's equivalent INTEGER kind.

Consider the simple C `int` type. This corresponds to INTEGER(C_INT), where C_INT is defined in the ISO_C_BINDING. In Intel® Fortran, the value is always four, as a C `int` corresponds with Fortran INTEGER(4). Other Fortran implementations may use different kind numbers. Using the named constant ensures portability.

Now consider the C `intptr_t` type. This integer is large enough to hold a pointer (address). In Intel® Fortran, this corresponds to INTEGER(C_INTPTR_T). It is INTEGER(4) when building a 32-bit application, and INTEGER(8) when building a 64-bit application.

Fortran has no unsigned integer types, so there are no constants for C unsigned types. These types are *not* interoperable.

If there is a kind of C type that is not supported by the Fortran implementation, the named constant for that type is defined as -1 and generates a compile-time error if used.

There are constants defined for REAL, COMPLEX, LOGICAL, and CHARACTER. For REAL, the standard offers the possibility of a C `long double` type. This is implemented in different ways by various C compilers on various platforms supported by Intel® Fortran.

- **gcc* on 32-bit Linux*:** The `long double` is an 80-bit floating type, as supported by the X87 instruction set. It is not supported by Intel® Fortran, so the C_LONG_DOUBLE is -1.
- **gcc on macOS*:** The C_LONG_DOUBLE is defined as 16.
- **Windows*:** The `long double` is treated the same as a `double`, so the C_LONG_DOUBLE is defined as 8.
- **64-bit Linux*:** The C_LONG_DOUBLE is defined as 16.

NOTE Use the constants for kind values and the corresponding types in C to ensure matching.

NOTE In Intel® C/C++ on Linux*, a `long double` can be forced to be a 128-bit IEEE format floating-point data type using the `-mlong-double-128` command line option. On all platforms, Intel® C/C++ supports `__float128` in the source code. On Windows*, Intel® C/C++ supports the `_Quad` type in the source code by using the command line option, `/Qoption,cpp,--extended_float_types`.

LOGICAL and CHARACTER need special treatment for interoperability. The Fortran standard states that LOGICAL corresponds to the (ISO_C_BINDING) C `_Bool` type, and defines a single kind value of C_BOOL, which is 1 in Intel® Fortran. By default, Intel® Fortran, tests LOGICALs for true/false differently than C does. Where C uses zero for false and not-zero for true, Intel® Fortran defaults to using -1 (all bits set) as true and

zero as false. If you are going to use LOGICAL types to interoperate with C, specify the option `fpscomp[:]logicals` to change the interpretation to be C-like. This is included when using the option `standard-semantics`, which is recommended for using Fortran 2003 (or later) features. C does not have character strings. It has arrays of single characters, and this is how strings in Fortran must be represented. There is a kind value defined as C_CHAR, which corresponds to the C `char` type. Only character variables with a length of one are interoperable. See Procedures for more information.

Derived types can also be interoperable. For additional information and restrictions, see Derived Types.

See Also

Procedures
Derived Types

Scalar Types

The commonly used types are included in the following table. The following applies:

- Integer types in Fortran are always signed. In C, integer types may be specified as signed or unsigned, but are signed by default.
- The values of C_LONG, C_SIZE_T, C_LONG_DOUBLE, and C_LONG_DOUBLE_COMPLEX are different on different platforms.

Named constant from ISO_C_BINDING (kind type parameter if value is positive)	C type	Equivalent Fortran type
C_SHORT	short int	INTEGER(KIND=2)
C_INT	int	INTEGER(KIND=4)
C_LONG	long int	INTEGER (KIND=4 or 8)
C_LONG_LONG	long long int	INTEGER(KIND=8)
C_SIGNED_CHAR	signed char unsigned char	INTEGER(KIND=1)
C_SIZE_T	size_t	INTEGER(KIND=4 or 8)
C_INT8_T	int8_t	INTEGER(KIND=1)
C_INT16_T	int16_t	INTEGER(KIND=2)
C_INT32_T	int32_t	INTEGER(KIND=4)
C_INT64_T	int64_t	INTEGER(KIND=8)
C_FLOAT	float	REAL(KIND=4)
C_DOUBLE	double	REAL(KIND=8)
C_LONG_DOUBLE	long double	REAL(KIND=8 or 16)
C_FLOAT_COMPLEX	float _Complex	COMPLEX(KIND=4)
C_DOUBLE_COMPLEX	double _Complex	COMPLEX(KIND=8)
C_LONG_DOUBLE_COMPLEX	long double _Complex	COMPLEX(KIND=8 or 16)
C_BOOL	_Bool	LOGICAL(KIND=1)
C_CHAR	char	CHARACTER(LEN=1)

While there are named constants for all possible C types, every type is not necessarily supported on every processor. Lack of support is indicated by a negative value for the constant in the module.

For a character type to be interoperable, you must either omit the length type parameter or specify it using a constant expression whose value is one.

Characters

The C language does not have character strings. Instead, it has arrays of single characters, so this is how you must represent a character string in Fortran.

There is a kind value defined, C_CHAR, corresponding to the C `char` type. However, only character variables with a length of one (1) are interoperable.

The following shows a Fortran program passing a string to a C routine and the C routine calling a Fortran routine with a new string.

Fortran Program Example

```
program demo_character_interop
    use, intrinsic :: iso_c_binding
    implicit none

    interface
    subroutine c_append (string) bind(C)
    character(len=1), dimension(*), intent(in) :: string
    end subroutine c_append
    end interface

    ! Call C routine passing an ordinary character value
    ! The language allows this to match an array of
    ! single characters of default kind
    call c_append('Intel Fortran'//C_NULL_CHAR)
end program demo_character_interop

subroutine fort_print (string) bind(C)
    use, intrinsic :: iso_c_binding
    implicit none

    ! Must declare argument as an array of single characters
    ! in order to be "interoperable"
    character(len=1), dimension(100), intent(in) :: string
    integer s_len ! Length of string
    character(100), pointer :: string1
    character(100) :: string2

    ! One way to convert the array to a character variable
    call C_F_POINTER(C_LOC(string),string1)
    s_len = INDEX(string1,C_NULL_CHAR) - 1
    print *, string1(1:s_len)

    ! Another way to convert
    string2 = TRANSFER(string,string2) ! May move garbage if source length < 100
    s_len = INDEX(string2,C_NULL_CHAR) - 1
    print *, string2(1:s_len)

    end subroutine fort_print
```

C Routine Example

```
C module (c_append.c):
#include <string.h>
extern void fort_print(char * string); /* Fortran routine */
void c_append (char * string) {
    char mystring[100];
    strcpy(mystring, string);
    strcat(mystring, " interoperates with C");
    /* Call Fortran routine passing new string */
    fort_print (mystring);
    return;
}
```

Pointers

For interoperating with C pointers, the module ISO_C_BINDING contains the derived types C_PTR and C_FUNPTR, which are interoperable with C object and function type pointers, respectively.

These types, as well as certain procedures in the module, provide the mechanism for passing dynamic arrays between the two languages. Because its elements do not need to be contiguous in memory, a Fortran pointer target or assumed-shape array cannot be passed to C. However, you can pass an allocated allocatable array to C, and you can associate an array allocated in C with a Fortran pointer. Additionally, as shown in the following, you can convert a pointer in C format to one in Fortran format.

Fortran Program Example

```
program demo_c_f_pointer
       use, intrinsic :: iso_c_binding
       implicit none

       interface
         function make_array(n_elements) bind(C)
           import ! Make iso_c_binding visible here
           type(C_PTR) :: make_array
           integer(C_INT), value, intent(IN) :: n_elements
         end function make_array
       end interface

       type(C_PTR) :: cptr_to_array
       integer(C_INT), pointer :: array(:) => NULL()
       integer, parameter :: n_elements = 3 ! Number of elements

       ! Call C function to create and populate an array
       cptr_to_array = make_array(n_elements)
       ! Convert to Fortran pointer to array of n_elements elements
       call C_F_POINTER (cptr_to_array, array, [n_elements])
       ! Print value
       print *, array

end program demo_c_f_pointer
```

C Module Example

```
#include <stdlib.h>
int  *make_array(int n_elements) {
          int *parray;
```

C Module Example

```
          int i;
   parray = (int*) malloc(n_elements * sizeof(int));
   for (i = 0; i < n_elements; i++) {
       parray[i] = i+1;
   }
   return parray;
}
```

Derived Types

For a derived type to be interoperable with C, you must specify the BIND attribute, as shown in the following:

Specifying BIND Example

```
type, BIND(C) :: MyType
```

Additionally, as shown in the examples that follow, each component must have an interoperable type and interoperable type parameters, must not be a pointer, and must not be allocatable. This allows Fortran and C types to correspond.

Interoperable Type and Type Parameter Example

```
typedef struct {
  int m, n;
  float r;
} MyCtype
```

The above is interoperable with the following:

```
use, intrinsic :: ISO_C_BINDING
type, BIND(C) :: MyFtype
integer(C_INT) :: i, j
real(C_FLOAT) :: s
end type MyFtype
```

The following restrictions apply to a derived type specified with the BIND attribute:

- It cannot have the SEQUENCE attribute
- It cannot be an extended type
- It cannot have type-bound procedures

Variables

A scalar Fortran variable is interoperable if its type and type parameters are interoperable and it is not a pointer.

An array Fortran variable is interoperable if its type and type parameters are interoperable and it has an explicit shape or assumed size. It interoperates with a C array of the same type, type parameters, and shape, but with subscripts reversed.

For example, a Fortran array declared as `integer :: a(18, 3:7, *)` is interoperable with a C array declared as `int b[][5][18]`.

Scalar variables are interoperable only if their type parameters (kind and length) are interoperable (see above), they are not a coarray, do not have the POINTER or ALLOCATABLE attribute, and if character length is not assumed nor defined by a non-constant expression.

Arrays are interoperable if the base type meets the scalar variable requirements above, if they are explicit shape or assumed-size, and are not zero-sized. Furthermore, assumed-size arrays are interoperable only with C arrays that have no size specified. An allocatable array, coarray, or array pointer is not interoperable.

Global Data

Global Data Overview

A module variable or a common block can interoperate with a C global variable if the Fortran entity uses the BIND attribute and the members of that entity are also interoperable. For example, consider the entities `c_extern`, `c2`, `com` and `single` in the following module:

Interoperability Example

```
module LINK_TO_C_VARS
   use, intrinsic :: ISO_C_BINDING
   integer(C_INT), BIND(C) :: c_extern
   integer(C_LONG) :: c2
   BIND(C, name='myVariable') :: c2
   common /com/ r,s
   real(C_FLOAT) :: r,s,t
   BIND(C) :: /com/, /single/
   common /single/ t
end module LINK_TO_C_VARS
```

These can interoperate with the following C external variables

```
int c_extern;
long myVariable;
struct {float r, s;} com;
float single;
```

Accessing Global Parameters Example

```
    MODULE Examp
      integer, BIND(C)::idata(20)
      real::rdata(10)
    END MODULE
```

In Fortran, a variable can access a global parameter by using the standard BIND C attribute.

In the above, two external variables are created (the latter with proper case and decoration): *idata* and *foo_mp_rdata*.

Use the BIND attribute to resolve name discrepancies with C.

Global Variable Statement Example

```
  int idata[20]; // declared as global (outside of any function)
```

Fortran can declare the variable global (COMMON) and other languages can reference it as external

```
 ! Fortran declaring PI global
  real pi_r
  COMMON /pi/ pi_r ! Common Block and variable names
  BIND(C)::/pi/
```

Use of BIND(C) above means that the name will be appropriately decorated for the target and made lowercase.

In C, the variable is referenced as an external with the statement

```
//C code with external reference to pi
extern float pi;
```

Note that the global name C references is the name of the Fortran common block (`pi`), not the name of a variable within a common block (`pi_r`). Therefore, you cannot use blank common (unnamed) to make data accessible between C and Fortran.

See Also

BIND C **Statement and Attribute:** Specifies that an object is interoperable with C and has external linkage.

COMMON

To reference C structures from Fortran common blocks and vice versa, you must take into account how common blocks and structures differ in their methods of storing member variables in memory. Fortran places common block variables into memory in order as close together as possible, with the following rules:

- A single BYTE, INTEGER(1), LOGICAL(1), or CHARACTER variable in common block list begins immediately following the previous variable or array in memory.
- All other types of single variables begin at the next even address immediately following the previous variable or array in memory.
- All arrays of variables begin on the next even address immediately following the previous variable or array in memory, except for CHARACTER arrays which always follow immediately after the previous variable or array.
- All common blocks begin on a four-byte aligned address.

Because of these rules, you must consider the alignment of C structure elements with Fortran common block elements. Specifically, you should ensure interoperability either by making all variables exactly equivalent types and kinds in both languages (using only 4-byte and 8-byte data types in both languages simplifies this) or by using the C pack pragmas in the C code around the C structure. This makes C data packing compatible with Fortran data packing.

As an example, suppose your Fortran code has a common block named `Really`, as shown:

Fortran Code Example

```
USE, INTRINSIC::ISO_C_BINDING
  REAL (C_float)::x,y,z(6)
  REAL (C_double)::ydbl
  COMMON, BIND(C) / Really /x, y, z(6), ydbl
```

You can access this data structure from your C code with the following external data structures.

C Code Example

```
#pragma pack(2)
extern struct {
 float x, y, z[6];
 double ydbl;
} Really;
#pragma pack()
```

To restore the original packing, you must add `#pragma pack()` at the end of the C structure.

You can also access C structures from Fortran by creating common blocks that correspond to those structures. This is the reverse case from that shown above. However, the implementation is the same; after common blocks and structures have been defined and given a common address (name), and, assuming the alignment in memory has been accounted for, both languages share the same memory locations for the variables.

Once you have accounted for alignment and padding, you can give C access to an entire common block or set of common blocks. Alternatively, you can pass individual members of a Fortran common block in an argument list, just as you can any other data item.

Procedures

For a Fortran procedure to be interoperable with C, it must have an explicit interface and be declared with the `BIND` attribute, as shown in the following:

BIND Interface Example

```
function Func(i, j, k, l, m) BIND(C)
```

In the case of a function, the result must be scalar and interoperable.

A procedure has an associated binding label, which is global in scope. This label is the name recognized by the C processor and is, by default, the lower-case version of the Fortran name (plus any needed leading or trailing underscores). For example, the above function has the binding label `func`. You can specify an alternative binding label as follows:

Alternate Binding Label Example

```
function Func(i, j, k, l, m) BIND(C, name='myC_Func')
```

All dummy arguments must be interoperable. Furthermore, you must ensure that either the Fortran routine uses the `VALUE` attribute for scalar dummy arguments, or that the C routine receives these scalar arguments as pointers to the scalar values. Consider the following call to this C function:

Call to C Function Example

```
int c_func(int x, int *y);
```

As shown here, the interface for the Fortran call to `c_func` must have `x` passed with the `VALUE` attribute. `y` should not have the `VALUE` attribute, since it is received as a pointer:

Fortran Call Example

```
interface
   integer (C_INT) function C_Func(x, y) BIND(C)
     use, intrinsic :: ISO_C_BINDING
     implicit none
     integer (C_INT), value :: x
     integer (C_INT) :: y
   end function C_Func
end interface
```

Alternatively, the declaration for `y` can be specified as a `C_PTR` passed by value:

Passing Pointer by Value Example

```
type (C_PTR), value :: y
```

Platform Specifics

Summary of Mixed-Language Issues

There are important differences in Fortran and C/C++ mixed-language programming; for instance, argument passing, naming conventions, and other interface issues must be thoughtfully and consistently reconciled between any two languages to prevent program failure and indeterminate results. However, the advantages of mixed-language programming often make the extra effort worthwhile.

A summary of major Fortran and C/C++ mixed-language issues follows:

- Generally, Fortran/C programs are mixed to allow one language to use existing code written in the other. Either Fortran or C can call the other, so the main routine can be in either language. On Linux* and macOS* systems, if Fortran is not the main routine, the `-nofor-main` compiler option must be specified on the `ifort` command line that links the application.
- When the main program is written in Fortran, the Fortran compiler automatically creates any code needed to initialize the Fortran Run-time Library (RTL). The RTL provides the Fortran environment for input/output and exception handling. When the main program is written in C/C++, the C main program needs to call for_rtl_init_ to initialize the Fortran RTL and for_rtl_finish_ at the end of the C main program to shut down the Fortran RTL gracefully. With the Fortran RTL initialized, Fortran I/O and error handling will work correctly even when C/C++ routines are called.
- For mixed-language applications, the Intel® Fortran main program can call subprograms written in C/C++ if the appropriate calling conventions are used.

  To use the same Microsoft* visual development environment for multiple languages, you must have the same version of the visual development environment for your languages.
- On Linux* and macOS* systems, Fortran adds an underscore to the end of external names; C does not.
- Fortran changes the case of external names to lowercase on Linux* and macOS* and to uppercase on Windows*; C leaves them in their original case.
- By default, Fortran passes data by reference; C by value. In both languages, the other method can be specified.

  NOTE
  It is possible to override some default Fortran behavior by using BIND(C) specifier. This is the preferred method. You can also override default Fortran behavior by using ATTRIBUTES and ALIAS.

- Fortran subroutines are equivalent to C void routines.
- Fortran requires that the length of strings be passed; C is able to calculate the length based on the presence of a trailing null. Therefore, if Fortran is passing a string to a C routine, that string needs to be terminated by a null; for example:

```
"mystring"c or  StringVar // CHAR(0)
```

- For COMPLEX, REAL*16, CHARACTER, derived types and ALLOCATABLE or array data types, Fortran adds a hidden first argument to contain function return values.
- On Linux*, the `-fexceptions` option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.
- For more information on debugging mixed language programs on Windows*, see Debugging Mixed-Language Programs.

Calling Subprograms from the Main Program (Windows*)

In mixed-language applications, an Intel® Visual Fortran main program can call subprograms written in a variety of languages. Conversely, an Intel® Visual Fortran subprogram (including those contained within a DLL or static library) can be called from a main program written in another language.

Microsoft Visual Studio* projects support a single language; therefore, code for each language must exist in its own project.

This topic summarizes mixed language compatibility with Intel® Visual Fortran for both managed code and unmanaged code. Managed code is architecture-independent code that runs under the control of the Microsoft* .NET Common Language Runtime Environment; unmanaged code is native, architecture-specific code.

Mixed-language applications can supply programs in a variety of formats:

Format	Created by	Callable by:
Compiled objects (.OBJ) and static libraries (.LIB) **NOTE** Objects and libraries must be link-compatible and not have conflicting names in their language support libraries	Intel® Visual Fortran, Intel® C++, Microsoft Visual C++* (unmanaged)	Intel® Visual Fortran, Intel® C++, Microsoft Visual C++* (unmanaged)
Dynamic Link Library (.DLL)	Intel® Visual Fortran, Intel® C++, Microsoft Visual C++* (unmanaged), Microsoft Visual Basic* (unmanaged), many more	Intel® Visual Fortran, Intel® C++, Microsoft Visual C++* (both managed and unmanaged), Microsoft Visual Basic* (managed and unmanaged), many others
.NET managed code assembly	Microsoft Visual C++ .NET, Microsoft Visual Basic .NET, other .NET languages	Intel® Visual Fortran (with interface generated by Fortran Module Wizard), .NET languages

Passing Arguments in Mixed-Language Programming

By default, Fortran programs pass arguments by reference; that is, they pass a pointer to each actual argument rather than the value of the argument. C programs typically pass arguments by value. Consider the following:

- When a Fortran program calls a C function, the C function's formal arguments must be declared as pointers to the appropriate data type.
- When a C program calls a Fortran subprogram, each actual argument must be specified explicitly as a pointer.

You can pass data between Fortran and C/C++ using argument lists just as you can within each language (for example, the argument list `a`, `b` and `c` in `CALL MYSUB(a,b,c)`). There are two ways to pass individual arguments:

- *By value*, which passes the argument's value.
- *By reference*, which passes the address of the arguments. On systems based on IA-32 architecture, Fortran, C, and C++ use 4-byte addresses. On systems based on Intel® 64 architecture, these languages use 8-byte addresses.

You need to make sure that for every call, the calling program and the called routine agree on how each argument is passed. Otherwise, the called routine receives bad data.

The Fortran technique for passing arguments changes depending on the calling convention specified. By default, Fortran passes all data by reference (except the hidden length argument of strings, which is passed by value).

On Windows* systems using IA-32 architecture only, you can alter the default calling convention. You can use either the `/iface:stdcall` option (`stdcall`) or the `/iface:cvf` option (Compaq* and Powerstation compatibility) to change the default calling convention, or the `VALUE` or `C` attributes in an explicit interface using the ATTRIBUTES directive. For more information on the ATTRIBUTES directive, see the Intel® Fortran Language Reference.

Both options cause the routine compiled and routines that it calls to have a @<*n*> appended to the external symbol name, where n is the number of bytes of all parameters. Both options assume that any routine called from a Fortran routine compiled this way will do its own stack cleanup, "callee pops." `/iface:cvf` also changes the way that `CHARACTER` variables are passed. With `/iface:cvf`, `CHARACTER` variables are passed as address/length pairs (that is, `/iface:mixed_str_len_arg`).

See Also

Handling Fortran Array Pointers and Allocatable Arrays

Stack Considerations in Calling Conventions (Windows*)

In the C calling convention, which is the default, the calling routine always adjusts the stack immediately after the called routine returns control. This produces slightly larger object code because the code that restores the stack must exist at every point a procedure is called.

> **NOTE**
> For 32-bit applications, use of the STDCALL calling convention means that the called procedure controls the stack. The code to restore the stack resides in the called procedure, so the code needs to appear only once. For 64-bit applications, the stack adjustment and @n name suffix features of STDCALL are not used.

The C calling convention makes calling with a variable number of arguments possible. In the C calling convention, the caller cleans up the stack, so it is possible to write a routine with a variable number of arguments. Therefore, the return point has the same address relative to the frame pointer, regardless of how many arguments are actually passed. Because of this, when the calling routine controls the stack, it knows how many arguments it passed, how big they are and where they reside in the stack. It can skip passing an argument and still keep track.

You can call routines with a variable number of arguments by including the ATTRIBUTES C option in your interface to a routine.

Naming Conventions

Naming conventions are as follows:

- A leading (prefix) underscore for Windows* operating systems based on IA-32 architecture; no underscores for Windows* operating systems based on other architectures.
- A trailing (postfix) underscore for all Linux* operating systems
- Leading and trailing underscores for all macOS* operating systems

C/C++ Naming Conventions

By default, the Fortran compiler converts function and subprogram names to lower case for Linux* and macOS* and upper case for Windows*. The C compiler never performs case conversion. A C procedure called from a Fortran program must, therefore, be named using the appropriate case.

C++ uses the same calling convention and argument-passing techniques as C, but naming conventions differ because of C++ decoration of external symbols. When the C++ code resides in a `.cpp` file, C++ name decoration semantics are applied to external names, often resulting in linker errors. The extern "C" syntax makes it possible for a C++ module to share data and routines with other languages by causing C++ to drop name decoration.

The following example declares `prn` as an external function using the C naming convention. This declaration appears in C++ source code:

```
extern "C" { void prn(); }
```

To call functions written in Fortran, declare the function as you would in C and use a "C" linkage specification. For example, to call the Fortran function FACT from C++, declare it as follows:

```
extern "C" { int fact( int* n ); }
```

The extern "C" syntax can be used to adjust a call from C++ to other languages, or to change the naming convention of C++ routines called from other languages. However, extern "C" can only be used from within C ++. If the C++ code does not use extern "C" and cannot be changed, you can call C++ routines only by determining the name decoration and generating it from the other language. Such an approach should only be used as a last resort, because the decoration scheme is not guaranteed to remain the same between versions.

When using extern "C" in a C++ program, keep in mind the following restrictions:

- You cannot declare a member function with extern "C".
- You can specify extern "C" for only one instance of an overloaded function; all other instances of an overloaded function have C++ linkage.

Compiling and Linking Intel® Fortran/C Programs

Your application can contain both C and Fortran source files. If your main program is a Fortran source file (`myprog.for`) that calls a routine written in C (`cfunc.c`), you can use the following sequence of commands to build your application.

Linux* and macOS*:

```
icc -c cfunc.c
ifort -o myprog myprog.for cfunc.o
```

Windows*:

```
icl /c cfunc.c
ifort myprog.for cfunc.obj
 /link /out:myprog.exe
```

The `icc` or `icl` command for Intel® C++ or the `cl` command (for Microsoft Visual C++*) compiles `cfunc.c`. The `-c` or `/c` option specifies that the linker is not called. This command creates `cfunc.o` (Linux* and macOS*) or `cfunc.obj` (Windows*).

The ifort command compiles `myprog.for` and links `cfunc.o` (Linux* and macOS*) or `cfunc.obj` (Windows*) with the object file created from `myprog.for` to create the executable.

Additionally, on Linux* and macOS* systems, you may need to specify one or more of the following options:

- Use the `-cxxlib` compiler option to tell the compiler to link using the C++ run-time libraries. By default, C++ libraries are not linked with Fortran applications.
- Use the `-fexceptions` compiler option to enable C++ exception handling table generation so C++ programs can handle C++ exceptions when there are calls to Fortran routines on the call stack. This option causes additional information to be added to the object file that is required during C++ exception handling. By default, mixed Fortran/C++ applications abort in the Fortran code if a C++ exception is thrown.

- Use the `-nofor_main` compiler option if your C/C++ program contains the `main()` entry point and is calling an Intel® Fortran subprogram, as shown in the following:

```
icc -c  cmain.c
ifort -nofor_main cmain.o fsub.f90
```

For more information about compiling and linking Intel® Fortran and C++ programs on Windows* operating systems, and the libraries used, see Specifying Consistent Library Types.

Building Intel® Fortran/C Mixed-Language Programs (Windows*)

When you understand and reconcile the calling, naming and argument passing conventions between Fortran and C, you are ready to build an application.

If you are using Microsoft Visual C++* or Intel® C++, you can edit, compile and debug your code within the Microsoft integrated development environment. If you are using another C compiler, you can edit your code within the integrated development environment. However, you must compile your code outside the integrated development environment and either build the Fortran/C program on the command line or add the compiled C `.OBJ` file to your Fortran project in the Microsoft IDE.

As an example of building from the command line, if you have a main C program `CMAIN.C` that calls Fortran subroutines contained in `FORSUBS.F90`, you can create the `CMAIN` application with the following commands:

```
icl /c cmain.c
ifort cmain.obj forsubs.f90
```

Intel® Visual Fortran accepts an object file for the main program written in C and compiled by the C compiler. The compiler compiles the .F90 file and then has the linker create an executable file under the name `CMAIN.EXE` using the two object files.

Either compiler (C or Fortran) can do the linking, regardless of which language the main program is written in; however, if you use the Fortran compiler first, you must include `LIBIFCORE.LIB` and `IFCONSOL.LIB` with the C compiler, and you may experience some difficulty with the version of the runtime library used by the C compiler. For these reasons, you may prefer to use the C compiler first or specify your project settings for both Fortran and C so there is agreement on the C library to link against, making sure that your application links against only one copy of the C library.

If you are using the IDE to build your application, Fortran uses Fortran and C libraries depending on the information specified in the Fortran folder in **Project > Properties** (**Project Settings** dialog box). You can also specify linker settings with the Linker folder in the **Project Settings** dialog box.

In the Fortran folder, within the **Libraries** property page, the **RunTime Library** category determines the libraries selected.

Runtime Library	Fortran Link Library Used	C Link Library Used
Debug Single-Threaded	libifcore.lib	libcmtd.lib
Multithreaded	libifcoremt.lib	libcmt.lib
Debug Multithreaded	libifcoremt.lib	libcmtd.lib
Debug Single-Threaded DLL	libifcorertd.lib (libifcorertd.dll)	msvcrt.lib (msvcrt.dll)
Multithreaded DLL	libifcoremd.lib (libifcoremd.dll)	msvcrt.lib (msvcrt.dll)
Debug Multithreaded DLL	libifcoremdd.lib (libifcoremdd.dll)	msvcrt.lib (msvcrt.dll)

For example, select Debug Multi-threaded DLL in the Run-time Library list to specify that DLL (`/libs:DLL`), multi-threaded (`/threads`), and debug (`/dbglibs`) libraries should be linked against, namely Fortran import library `libifcoremdd.lib` and its DLL library `libifcoremdd.dll` and C/C++ import library `msvcrtd.lib` and its DLL library `msvcrtd.dll`.

A mixed language solution containing a Fortran library project should have **Disable Default Library Search Rules** set to No in the IDE. To check this setting, choose **Project > Properties** and then choose the Libraries category. If you change the **Disable Default Library Search Rules** setting to Yes, you will need to explicitly add the needed runtime libraries to the non-Fortran project. If you are adding libraries explicitly, make sure you add `IFCONSOL.LIB` to the libraries of the non-Fortran project. This library is needed to perform almost any kind of I/O with Intel® Visual Fortran.

When you have a C++ main program and a Fortran library subproject, you need to manually add the library path to the Intel® Visual Fortran LIB folder. You only need to do this once per user. To add the path, choose **Tools > Options > Projects and Solutions > VC++ Directories**. Use the **Show directories for:** dropdown item to select Library files. Add the path to the displayed list.

The way Microsoft Visual C++* chooses libraries is also based upon the **Project > Properties** item, but within the **C/C++** tab. In the **Code Generation** category, the Run-time library item lists the following C libraries:

Menu Item Selected	CL Option or Project Type Enabled	Default Library Specified in Object File
Multithreaded	/MTd	libcmt.lib
Multithreaded DLL	/MD	msvcrt.lib (msvcrnn.dll)
Debug Multithreaded	/MTd	libcmtd.lib
Debug Multithreaded DLL	/MDd	msvcrtd.lib (msvcrtd.dll)

You must take care to choose the same type of run-time libraries in both your Fortran and C project. For example, if you select Multithreaded DLL in your Fortran project, you must select Multithreaded DLL in your C project. Otherwise, when your build your mixed Fortran/C application, you will receive errors from the Linker regarding undefined and/or duplicate symbols.

If you are using Microsoft Visual C++* or Intel® C++ , the Microsoft integrated development environment can build mixed Fortran/C applications transparently, with no special directives or steps on your part. You can edit and browse your C and Fortran programs with appropriate syntax coloring for the language. You need to place your Fortran source files into a Fortran project and your C/C++ files into a Visual C++* project.

When you debug a mixed Visual C++/Fortran application, the debugger will adjust to the code type as it steps through: the C or Fortran expression evaluator will be selected automatically based on the code being debugged, and the stack window will show Fortran data types for Fortran procedures and C data types for C procedures.

Implementation Specifics

Fortran Module Naming Conventions

Fortran module entities (data and procedures) have external names that differ from other external entities. Module names use the convention:

```
modulename_mp_entity_(Linux* OS and macOS*)
_MODULENAME_mp_ENTITY(Windows* OS)
```

modulename is the name of the module. For Windows* operating systems, the name is uppercase by default.

entity is the name of the module procedure or module data contained within *MODULENAME*. For Windows* operating systems, *ENTITY* is uppercase by default.

*_mp_* is the separator between the module and entity names and is always lowercase (except when the `assume std_mod_proc_name` compiler option or `standard-semantics` compiler option is used).

For example:

```
   MODULE mymod
     INTEGER a
  CONTAINS
     SUBROUTINE b (j)
        INTEGER j
     END SUBROUTINE
  END MODULE
```

This results in the following symbols being defined in the compiled object file on Linux* operating systems. (On macOS* operating systems, the symbols would begin with an underscore.)

```
mymod_mp_a_
mymod_mp_b_
```

The following symbols are defined in the compiled object file on Windows* operating systems based on IA-32 architecture:

```
_MYMOD_mp_A
_MYMOD_mp_B
```

On Windows* operating systems based on Intel® 64 architecture, there is no beginning underscore.

Handling Fortran Array Pointers and Allocatable Arrays

When a Standard Fortran array pointer or array is passed to another language, either its descriptor or its base address can be passed.

The following affects how Standard Fortran array pointers and arrays are passed:

- The ATTRIBUTES properties in effect
- The INTERFACE, if any, of the procedure they are passed to

If the INTERFACE declares the array pointer or array with deferred shape (for example, `ARRAY(:)`), its descriptor is passed. If the INTERFACE declares the array pointer or array with fixed shape, or if there is no interface, the array pointer or array is passed by base address as a contiguous array, which is like passing the first element of an array for contiguous array slices.

Use of the REFERENCE attribute on an array has no effect. Additionally, the VALUE attribute cannot be used with array arguments.

When you pass a Fortran array pointer or an array by descriptor to a non-Fortran routine, that routine needs to know how to interpret the descriptor. Part of the descriptor is a pointer to address space, such as a C pointer, and part of it is a description of the pointer or array properties, such as its rank, stride, and bounds.

For information about the Intel Fortran array descriptor format, see Handling Fortran Array Descriptors.

Standard Fortran array pointers and assumed-shape arrays are passed by passing the address of the array descriptor.

Standard Fortran pointers that point to scalar data contain the address of the data and are not passed by descriptor.

Handling Fortran Array Descriptors

For cases where Standard Fortran needs to keep track of more than a pointer memory address, the Intel® Fortran Compiler uses an *array descriptor*, which stores the details of how an array is organized.

When using an explicit interface (by association or procedure interface block), the compiler generates a descriptor for the following types of array arguments:

- Fortran array pointers
- Assumed-shape arrays
- Allocatable arrays
- Coarrays

- Class objects

Certain types of arguments do not use a descriptor, even when an appropriate explicit interface is provided. For example, explicit-shape and assumed-size arrays do not use a descriptor. In contrast, array pointers and allocatable arrays use descriptors regardless of whether they are used as arguments.

When calling between Intel® Fortran and C/C++, use an *implicit* interface, which allows the array argument to be passed *without* an Intel® Fortran descriptor. However, for cases where the called routine needs the information in the Intel® Fortran descriptor, declare the routine with an *explicit* interface and specify the dummy array as either an assumed-shape array or with the pointer attribute.

You can associate a Fortran array pointer with any piece of memory, organized in any way desired (as long as it is "rectangular" in terms of array bounds). You can also pass Fortran array pointers to the C language, and have it correctly interpret the descriptor to obtain the information it needs.

The downside to using array descriptors is that it can increase the opportunity for errors. Additionally, the corresponding code is not portable. Specifically:

- If the descriptor is not defined correctly, the program may access the wrong memory address, possibly causing a General Protection Fault.
- Array descriptor formats are specific to each Fortran compiler. Code that uses array descriptors is not portable to other compilers or platforms.
- The array descriptor format may change in the future.
- If the descriptor was built by the compiler, it cannot be modified by the user. Changing fields of existing descriptors is illegal.

The components of the current Intel® Fortran array descriptor on systems using IA-32 architecture are as follows:

- Bytes 0 to 3 contain the base address.
- Bytes 4 to 7 contain the size of a single element of the array.
- Bytes 8 to 11 are reserved and should not be explicitly set.
- Bytes 12 to 15 contain a set of flags used to store information about the array. This includes:
  - bit 1 (0x01): array is defined -- set if the array has been defined (storage allocated).
  - bit 2 (0x02): no deallocation allowed -- set if the array pointed to cannot be deallocated (that is, it is an explicit array target).
  - bit 3 (0x04): array is contiguous -- set if the array pointed to is a contiguous whole array or slice.
  - bit 8 (0x80): set if the array pointed to is ALLOCATABLE.
  - Other bits are reserved.
- Bytes 16 to 19 contain the number of dimensions (rank) of the array.
- Bytes 20 to 23 are reserved and should not be explicitly set.
- The remaining bytes contain information about each dimension (up to 31). Each dimension is described by a set of three 4-byte entities:
  - The number of elements (extent).
  - The distance between the starting address of two successive elements in this dimension, in bytes.
  - The lower bound.

An array of rank one requires three additional 4-byte entities for a total of nine 4-byte entities (6 + 3*1) and ends at byte 35. An array of rank seven is described in a total of twenty-seven 4-byte entities (6 + 3*7) and ends at byte 107.

Consider the Following Declaration

```
  integer,target :: a(10,10)
 integer,pointer :: p(:,:)
 p => a(9:1:-2,1:9:3)
 call f(p)
...
```

The descriptor for actual argument *p* would contain the following values:

- Bytes 0 to 3 contain the base address (assigned at run-time).
- Bytes 4 to 7 are set to 4 (size of a single element).
- Bytes 8 to 11 are reserved.

- Bytes 12 to 15 contain 3 (array is defined and deallocation is not allowed).
- Bytes 16 to 19 contain 2 (rank).
- Bytes 20 to 23 are reserved.
- Bytes 24 to 35 contain information for the first dimension, as follows:
  - 5 (extent).
  - -8 (distance between elements).
  - 9 (the lower bound).
- For the second dimension, bytes 36 to 47 contain:
  - 3 (extent).
  - 120 (distance between elements).
  - 1 (the lower bound).
- Byte 47 is the last byte for this example.

NOTE
The format for the descriptor on systems using Intel® 64 architecture is identical to that on systems using IA-32 architecture, except that all fields are 8-bytes long, instead of 4-bytes.

Returning Character Data Types

A discrepancy occurs when working with C and Fortran character strings: C strings are null-terminated while Fortran strings have known lengths. A C routine that returns a character string will be returning a pointer to a null-terminated string. A Fortran routine will not have any knowledge of the string's length.

If a Fortran routine is returning a string to a C program, the Fortran program must null-terminate the string.

Example of Returning Character Types from C to Fortran

The following shows the Fortran code that declares interfaces to a C routine and calls the C routine. In the example, the pointer returned by the C routine can be passed along to another C routine. However, it is not usable by Fortran as-is since Fortran has no knowledge of the string's length.

Fortran Code Example

```
! This program declares a C program that returns a char*
!  calls it, and has to pass that return value along to
!  a different c program because type(c_ptr) is limited
!  in a pure Fortran program.

program FCallsCReturnsCptr

use, intrinsic :: iso_c_binding

! declare the interfaces to the C routines

interface
   type(C_PTR)function my_croutine1 ( input) bind(c)
     import
     integer(c_int), value :: input
   end function my_croutine1

   subroutine my_cprint( string ) bind(c)
     import c_ptr
     type(C_PTR), value :: string
   end subroutine my_cprint
end interface
```

Fortran Code Example

```
   call my_cprint(my_croutine1(42))

end program
```

Called C Routine Example

```
#include <stdio.h>

char *my_croutine1 (int input) {
   static char temp[30];
   temp[0] = '\0';
   sprintf(temp, "in routine, you said %d", input);
  return temp;
}

void my_cprint (char *string) {
  printf ("cprint says %s\n", string);
 return;
}
```

The example above shows the C code used to call a Fortran routine. The Fortran routine returns a string that is then printed by the C program. As shown in the example, it is relatively straightforward to pass a string back to C from Fortran. Additionally, the string can be easily used by the C program because it is NULL-terminated.

In the above example, the following restrictions and behaviors apply:

- The function's length and result do not appear in the call statement; they are added by the compiler.
- The called routine must copy the result string into the location specified by `result`; it must not copy more than `length` characters.
- If fewer than `length` characters are returned, the return location should be padded on the right with blanks; Fortran does not use zeros to terminate strings.
- The called procedure is type `void`.

Example of Returning Character Types from Fortran to C

The following shows the C code used to call a Fortran routine; the Fortran routine returns a string that is then printed by the C program.

C Code Example

```
#include <stdio.h>

char *GetFortranWords(void);

int main() {

   printf ("Fortran says this: %s\n", GetFortranWords());
   return 0;
}
```

Called Fortran Routine Example

```
! this routine is called from C, returning a string that
!  can be printed by the caller

function GetFortranWords bind(c, name="GetFortranWords")
use, intrinsic :: iso_c_binding
type(C_ptr) :: GetFortranWords
character*30 returnval

returnval = "I like to type words!" // char(0)


GetFortranWords = C_LOC(returnval)
return
end
```

Legacy Extensions

ATTRIBUTES

Standard Fortran provides a syntax for interoperating with C, including applying the appropriate naming conventions. You can use BIND(C) for interoperability with C, instead of specifying ATTRIBUTES options.

The ATTRIBUTES options (also known as properties) C, STDCALL (Windows* only), REFERENCE, VALUE affect the calling convention of routines. You can specify:

- The C, STDCALL, and REFERENCE properties for an entire routine.
- The VALUE and REFERENCE properties for individual arguments.

By default, Fortran passes all data by reference (except the hidden length argument of strings, which is passed by value). If the C (or, for Windows*, STDCALL) option is used, the default changes to passing almost all data except arrays by value. However, in addition to the calling-convention options C and STDCALL, you can specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference, regardless of the calling convention option. Arrays can only be passed by reference.

Different Fortran calling conventions can be specified by declaring the Fortran procedure to have certain attributes.

The following table summarizes the effect of the most common Fortran calling convention directives.

Calling Conventions for ATTRIBUTES Options

Argument	Default	C	C, REFERENCE	STDCALL (Windows* IA-32 architecture)	STDCALL, REFERENCE (Windows* IA-32 architecture)
Scalar	Reference	Value	Reference	Value	Reference
Scalar [value]	Value	Value	Value	Value	Value
Scalar [reference]	Reference	Reference	Reference	Reference	Reference

Argument	Default	C	C, REFERENCE	STDCALL (Windows* IA-32 architecture)	STDCALL, REFERENCE (Windows* IA-32 architecture)
String	Reference, either Len:End or Len:Mixed	String(1:1)	Reference, either Len:End or Len:Mixed	String(1:1)	String(1:1)
String [value]	Error	String(1:1)	String(1:1)	String(1:1)	String(1:1)
String [reference]	Reference, either No Len or Len:Mixed	Reference, No Len	Reference, No Len	Reference, No Len	Reference, No Len
Array	Reference	Reference	Reference	Reference	Reference
Array [value]	Error	Error	Error	Error	Error
Array [reference]	Reference	Reference	Reference	Reference	Reference
Derived Type	Reference	Value, size dependent	Reference	Value, size dependent	Reference
Derived Type [value]	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent
Derived Type [reference]	Reference	Reference	Reference	Reference	Reference
F90 Pointer	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
F90 Pointer [value]	Error	Error	Error	Error	Error
F90 Pointer [reference]	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
Naming Conventions					
Prefix	_ (Windows* operating systems using IA-32 architecture, macOS* operating systems) *none* for all others	_ (Windows* operating systems using IA-32 architecture, macOS* operating systems) *none* for all others	_ (Windows* operating systems using IA-32 architecture, macOS* operating systems) *none* for all others	_	_
Suffix	none (Windows*)	*none*	*none*	@*n*	@*n*

Argument	Default	C	C, REFERENCE	STDCALL (Windows* IA-32 architecture)	STDCALL, REFERENCE (Windows* IA-32 architecture)
	_ (Linux*, macOS*)				
Case	Upper Case (Windows*) Lower Case (Linux*, macOS*)	Lower Case	Lower Case	Lower Case	Lower Case
Stack Cleanup	Caller	Caller	Caller	Callee	Callee

The terms in the above table mean the following:

[value]	Argument assigned the VALUE attribute.
[reference]	Argument assigned the REFERENCE attribute.
Value	The argument value is pushed on the stack. All values are padded to the next 4-byte boundary.
Reference	On systems using IA-32 architecture, the 4-byte argument address is pushed on the stack. On systems using Intel® 64 architecture, the 8-byte argument address is pushed on the stack.
Len:End or Len:Mixed	For certain string arguments: • Len:End applies when `-nomixed-str-len-arg` (Linux* and macOS*) or `/iface:nomixed_str_len_arg` (Windows*) is set. The length of the string is pushed (by value) on the stack after all of the other arguments. This is the default. • Len:Mixed applies when `-mixed-str-len-arg` (Linux* and macOS*) or `/iface:mixed_str_len_arg` (Windows*) is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
No Len or Len:Mixed	For certain string arguments: • No Len applies when `-nomixed-str-len-arg` (Linux* and macOS*) or `/iface:nomixed_str_len_arg` (Windows*) is set. The length of the string is not available to the called procedure. This is the default. • Len:Mixed applies when `-mixed-str-len-arg` (Linux* and macOS*) or `/iface:mixed_str_len_arg` (Windows*) is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
No Len	For string arguments, the length of the string is not available to the called procedure.
String(1:1)	For string arguments, the first character is converted to INTEGER(4) as in ICHAR(string(1:1)) and pushed on the stack by value.
Error	Produces a compiler error.
Descriptor	On systems using IA-32 architecture, the 4-byte address of the array descriptor. On systems using Intel® 64 architecture, the 8-byte address of the array descriptor.

@*n*	On systems using IA-32 architecture, the at sign (@) followed by the number of bytes (in decimal) required for the argument list.
Size dependent	On systems using IA-32 architecture, derived-type arguments specified by value are passed as follows: • Arguments from 1 to 4 bytes are passed by value. • Arguments from 5 to 8 bytes are passed by value in two registers (two arguments). • Arguments more than 8 bytes provide value semantics by passing a temporary storage address by reference.
Upper Case	Procedure name in all uppercase.
Lower Case	Procedure name in all lowercase.
Callee	The procedure being called is responsible for removing arguments from the stack before returning to the caller.
Caller	The procedure doing the call is responsible for removing arguments from the stack after the call is over.

You can use the BIND(C, name=<string>) attribute to resolve discrepancies with C. Alternatively, the ALIAS option can be used with any other Fortran calling-convention option to preserve mixed-case names. +

For Windows* operating systems, the compiler option `/iface` also establishes some default argument passing conventions. The `/iface` compiler option has the following choices:

Option	How are arguments passed?	Append `@n` to names on systems using IA-32 architecture?	Who cleans up stack?	Varargs support?
/iface:cref	By reference	No	Caller	Yes
/iface:stdref	By reference	Yes	Callee	No
/iface:default	By reference	No	Caller	Yes
/iface:c	By value	No	Caller	Yes
/iface:stdcall	By value	Yes	Callee	No
/iface:cvf	By reference	Yes	Callee	No

NOTE
The following applies to Windows*: When interfacing to the Windows* graphical user interface or making API calls, you will typically use STDCALL.

ALIAS

Use the ALIAS option for the ATTRIBUTES directive If the name of a routine appears as mixed-case in C and you need to preserve the case.

To use the ALIAS option, place the name in quotation marks exactly as it is to appear in the object file.

C Function `My_Proc` Example

```
!DIR$ ATTRIBUTES DECORATE,ALIAS:'My_Proc'
:: My_Proc
```

This example uses DECORATE to automatically reconcile external name declaration for the target platform.

Using the DECORATE option in combination with the ALIAS option specifies that the external name specified in ALIAS should have the correct prefix and postfix decorations for the calling mechanism in effect.

Compiler Options

Using the -nofor-main Compiler Option (Linux* and macOS*)

Intel® Fortran subprograms can be called by C/C++ main programs. Use the `-nofor-main` compiler option if your C/C++ program contains the `main()` entry point and is calling an Intel® Fortran subprogram.

See Also

`nofor-main` compiler option

Error Handling

Handling Compile Time Errors

Understanding Errors During the Build Process

The Intel® Fortran Compiler identifies syntax errors and violations of language rules in the source program.

Compiler Diagnostic Messages

These messages describe diagnostics that are reported during the processing of the source file. Compiler diagnostic messages usually provide enough information for you to determine the cause of an error and correct it. These messages generally have the following format:

```
filename(linenum): severity #error number: message
```

Diagnostic	Meaning
filename	Indicates the name of the source file currently being processed.
linenum	Indicates the source line where the compiler detects the condition.
severity	Indicates the severity of the diagnostic message: *Warning*, *Error*, or *Fatal error*.
message	Describes the problem.

The following is an example of an error message:

```
echar.for(7): error #6321: An unterminated block exits.
       DO I=1,5
-------^
```

The pointer (`---^`) indicates the place on the source program line where the error was found, in this case where an END DO statement was omitted.

To view the passes as they execute on the command line, specify the `watch` compiler option.

NOTE
You can perform compile-time procedure interface checking between routines with no explicit interfaces present. To do this, generate a module containing the interface for each compiled routine using the `gen-interfaces` option and check implicit interfaces using the `warn[:]interfaces` compiler option.

Controlling Compiler Diagnostic Warning and Error Messages

You can use a number of compiler options to control the diagnostic messages issued by the compiler. For example, compiler option `WB` turns compile time bounds errors into warnings. To control compiler diagnostic messages (such as warning messages), use the `warn` option. This option controls warnings issued by the compiler and supports a wide range of *keyword*s. Some of these are as follows:

- **[no]alignments:** Determines whether warnings occur or do not occur for data that is not naturally aligned.
- **[no]declarations:** Determines whether warnings occur or do not occur for any undeclared symbols.
- **[no]errors:** Determines whether warning-level messages are changed or are not changed to error-level messages.
- **[no]general:** Determines whether warning messages and informational messages are issued or are not issued by the compiler.
- **[no]interfaces:** Determines whether warnings about the interfaces for all called SUBROUTINES and invoked FUNCTIONS are issued or are not issued by the compiler.
- **[no]stderrors:** Determines whether warnings about Fortran standard violations are changed or are not changed tto errors.
- **[no]truncated_source:** Determines whether warnings occur or do not occur when source exceeds the maximum column width in fixed-format files.

For more information, see the warn compiler option.

You can also control the display of diagnostic information with variations of the `-diag` (Linux* and macOS*) or `/Qdiag` (Windows*) compiler option. This option accepts numerous arguments and values, allowing you wide control over displayed diagnostic messages and reports.

Some of the most common variations include the following:

Linux* and macOS*	Windows*	Description
`-diag-enable=`*list*	`/Qdiag-enable:`*list*	Enables a diagnostic message or a group of messages
`-diag-disable=`*list*	`/Qdiag-disable:`*list*	Disables a diagnostic message or a group of messages
`-diag-warning=`*list*	`/Qdiag-warning:`*list*	Tells the compiler to change diagnostics to warnings

Linux* and macOS*	Windows*	Description
`-diag-error=`*list*	`/Qdiag-error:`*list*	Tells the compiler to change diagnostics to errors
`-diag-remark=`*list*	`/Qdiag-remark:`*list*	Tells the compiler to change diagnostics to remarks (comments)

The *list* items can be one of the keywords `warn`, `remark`, or `error`, a keyword specifying a certain group (`par`, `vec`, `driver`, `cpu-dispatch`), or specific diagnostic IDs separated by commas. For more information, see diag, Qdiag.

Additionally, you can use the following related options:

Linux* and macOS*	Windows*	Description
`-diag-dump`	`/Qdiag-dump`	Tells the compiler to print all enabled diagnostic messages and stop compilation.
`-diag-file[=`*file*`]`	`/Qdiag-file[:`*file*`]`	Causes the results of diagnostic analysis to be output to a file.
`-diag-file-append[=`*file*`]`	`/Qdiag-file-append[:`*file*`]`	Causes the results of diagnostic analysis to be appended to a file.
`-diag-error-limit=`*n*	`/Qdiag-error-limit:`*n*	Specifies the maximum number of errors allowed before compilation stops.

Linker Diagnostic Errors

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors occur, the linker does not produce an executable file. Linker messages are descriptive, and you do not normally need additional information to determine the specific error.

To view the libraries being passed to the linker on the command line, specify the `watch` option.

Error Severity Levels

Comment Messages

These messages indicate valid but inadvisable use of the language being compiled. The compiler displays comments by default. You can suppress comment messages with the `warn[:]nousage` option.

Comment messages do not terminate translation or linking, they do not interfere with any output files either. Some examples of the comment messages are:

```
Null CASE construct
The use of a non-integer DO loop variable or expression
Terminating a DO loop with a statement other than CONTINUE or ENDDO
```

Warning Messages

These messages report valid but questionable use of the language being compiled. The compiler displays warnings by default. You can suppress warning messages by using the `nowarn` option. Warnings do not stop translation or linking. Warnings do not interfere with any output files. Some representative warning messages are:

```
constant truncated - precision too great
non-blank characters beyond column 72 ignored
Hollerith size exceeds that required by the context
```

Error Messages

These messages report syntactic or semantic misuse of Fortran. Errors suppress object code for the file containing the error and prevent linking, but they do not stop the compiler from scanning for any other errors. Some typical examples of error messages are:

```
line exceeds 132 characters
unbalanced parenthesis
incomplete string
```

Fatal Errors

Fatal messages indicate environmental problems. Fatal error conditions stop translation, assembly, and linking. If a fatal error ends compilation, the compiler displays a termination message on standard error output. Some representative fatal error messages are:

```
Disk is full, no space to write object file
Too many segments, object format cannot support this many segments
```

Using the Command Line

If you are using the command line, messages are written to the standard error output file. When using the command line:

- Make sure that the appropriate environment variables have been set by executing the `compilervars.sh` (Linux* and macOS*) or `compilervars.bat` (Windows*) file. For Windows*, these environment variables are preset if you use the Fortran Command Prompt window in the Intel® Visual Fortran program folder. For a list of environment variables used by the `ifort` command during compilation, see Supported Environment Variables.
- Specify the libraries to be linked against using compiler options.
- You can specify libraries (include the path, if needed) as file names on the command line.

Using the Visual Studio IDE (Windows*)

If you are using the Microsoft* Visual Studio* integrated development environment (IDE), compiler and linker errors are displayed in the **Build** pane of the **Output** window. To display the **Output** window, choose **View** > **Other Windows** > **Output**. You can also use the **Task List** window (**View** > **Other Windows** > **Task List**) to view display links to problems encountered during the build process. Click these tasks to jump to code that caused build problems. You can also check the Build log for more information.

To quickly locate the source line causing the error, follow these steps:

1. Double-click the error message text in the **Build** pane of the **Output** window.

   - or -
2. Press **F8**. The editor window appears with a marker in the left margin that identifies the line causing the error. You can continue to press **F8** to scroll through additional errors.

After you have corrected any compiler errors reported during the previous build, choose **Build project name** from the **Build** menu. The build engine recompiles only those files that have changed, or which refer to changed include or module files. If all files in your project compile without errors, the build engine links the object files and libraries to create your program or library.

You can force the build engine to recompile all source files in the project by selecting **Rebuild project name** from the **Build** menu. This is useful to verify that all of your source code is clean, especially if you are using a makefile, or if you use a new set of compiler options for all of the files in your project.

When using the IDE:

- Make sure that you have specified the correct path, library, and include directories. For more information, see Specifying Path Library and Include Directories.

- If a compiler option is not available through **Project** > **Properties** in the Intel® Fortran Property pages, you can type the option in the Command Line category. Use the lower part of the window under **Additional Options**, just as you would using the command line. For example, you can enter the linker option `/link /DEFAULTLIB` to specify an additional library.

> **NOTE**
> If you have a question about a compile-time error, please submit your question to the Intel® Fortran forum, which can be found at https://software.intel.com/en-us/forum.

See Also

Specifying Path Library and Include Directories
Supported Environment Variables
warn compiler option
diag, Qdiag compiler option

Handling Run-Time Errors

Understanding Run-Time Errors

During execution, your program may encounter errors or exception conditions. These conditions can result from any of the following:

- Errors that occur during I/O operations.
- Invalid input data.
- Argument errors in calls to the mathematical library.
- Arithmetic errors.
- Other system-detected errors.

The Intel® Fortran Run-Time Library (RTL) generates appropriate messages and takes action to recover from errors whenever possible.

For a description of each Intel® Fortran run-time error message, see List of Run-Time Error Messages.

There are a few tools and aids that are useful when an application fails and you need to diagnose the error. Compiler-generated machine code listings and linker-generated map files can help you understand the effects of compiler optimizations and to see how your application is laid out in memory. They may help you interpret the information provided in a stack trace at the time of the error. See Generating Listing and Map Files.

Forcing a Core Dump for Severe Errors

You can force a core dump for severe errors that do not usually cause a `core` file to be created. Before running the program, set the `FOR_DUMP_CORE_FILE` environment variable to TRUE to cause severe errors to create a `core` file. See the "Supported Environments Variables" topic for valid values of TRUE and FALSE.

For instance, the following C shell command sets the `FOR_DUMP_CORE_FILE` environment variable:

```
setenv FOR_DUMP_CORE_FILE y
```

The `core` file is written to the current directory and can be examined using a debugger.

NOTE
If you requested a core file to be created on severe errors and you do not get one when expected, the problem might be that your process limit for the allowable size of a core file is set too low (or to zero). See the man page for your shell for information on setting process limits. For example, the C shell command limit (with no arguments) will report your current settings, and `limit coredumpsize unlimited` will raise the allowable limit to your current system maximum.

See Also
Supported Environment Variables
Run-Time Default Error Processing
Run-Time Message Display and Format
Values Returned at Program Termination
Methods of Handling Errors
Using the END, EOR, and ERR Branch Specifiers
Using the IOSTAT Specifier and Fortran Exit Codes
Locating Run-Time Errors
List of Run-Time Error Messages
Generating Listing and Map Files
Signal Handling
Overriding the Default Run-Time Library Exception Handler
Traceback and related topics

Run-Time Default Error Processing

The Intel® Fortran run-time system processes a number of errors that can occur during program execution. A default action is defined for each error recognized by the Intel® Fortran run-time system. The default actions described throughout this section occur unless overridden by explicit error-processing methods.

The way in which the Intel® Fortran run-time system actually processes errors depends upon the following factors:

- The severity of the error. For instance, the program usually continues executing when an error message with a severity level of warning or info (informational) is detected.
- For certain errors associated with I/O statements, whether or not an I/O error-handling specifier was specified.
- For certain errors, whether or not the default action of an associated signal was changed.
- For certain errors related to arithmetic operations (including floating-point exceptions), compilation options can determine whether the error is reported and the severity of the reported error.

How arithmetic exception conditions are reported and handled depends on the cause of the exception and how the program was compiled. Unless the program was compiled to handle exceptions, the exception might not be reported until *after* the instruction that caused the exception condition.

See Also
Run-Time Message Display and Format
Values Returned at Program Termination
Locating Run-Time Errors
Traceback
Data Representation

See Also
Advanced Exception and Termination Handling Considerations
Setting Compiler Options in the Visual Studio* IDE Property Pages

Run-Time Message Display and Format

When errors occur during execution (run time) of a program, the Intel® Visual Fortran run-time system issues diagnostic messages.

The location where Fortran run-time messages are displayed depends on the Understanding Project Types:

Project Type	Where Messages Appear
Fortran Console applications	Run-time error messages are displayed on the standard error device (unless redirected).
Fortran QuickWin and Fortran Standard Graphics applications	Run-time error messages are displayed in a separate QuickWin message box.
Fortran Windows applications	Run-time error messages are displayed in a separate message box.

Fortran run-time messages have the following format:

```
forrtl:severity (number):message-text
```

where:

- ***forrtl*:** Identifies the source as the Intel® Fortran run-time system (Run-Time Library or RTL).
- ***severity*:** The severity levels are: *severe* , *error*, *warning*, or *info*
- ***number*:** This is the message number; see also the Using the IOSTAT Specifier and Fortran Exit Codes for I/O statements.
- ***message-text*:** Explains the event that caused the message.

The following table explains the severity levels of run-time messages, in the order of greatest to least severity. The severity of the run-time error message determines whether program execution continues:

Severity	Description
severe	Must be corrected. The program's execution is terminated when the error is encountered unless the program's I/O statements use the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier. (See Using the END EOR and ERR Branch Specifiers and Using the IOSTAT Specifier and Fortran Exit Codes and Methods of Handling Errors.) For severe errors, stack trace information is produced by default, unless environment variable `FOR_DISABLE_STACK_TRACE` is set. When that environment variable is set, no stack trace information is produced. If compiler option `traceback` is specified on the command line, the stack trace information contains program counters set to symbolic information. Otherwise, the information only contains hexadecimal program counter information. In some cases stack trace information is also produced by the compiled code at run-time to provide details about the creation of array temporaries.
error	Should be corrected. The program might continue execution, but the output from this execution might be incorrect. For errors of severity type error, stack trace information is produced by default, unless the environment variable `FOR_DISABLE_STACK_TRACE` is set. When that environment variable is set, no stack trace information is produced. If the command line option `traceback` is specified, the stack trace information contains program counters set to symbolic information. Otherwise, the information only contains hexadecimal program counter information.

Severity	Description
	In some cases stack trace information is also produced by the compiled code at run-time to provide details about the creation of array temporaries.
warning	Should be investigated. The program continues execution, but output from this execution might be incorrect.
info	For informational purposes only; the program continues.

For a description of each Intel® Fortran run-time error message, see Run-Time Default Error Processing and related topics.

The following example applies to Linux* and to macOS*:

In some cases, stack trace information is produced by the compiled code at run time to provide details about the creation of array temporaries.

The following program generates an error at line 12:

```
program ovf
real*4 x(5),y(5)
integer*4 i

 x(1) = -1e32
 x(2) = 1e38
 x(3) = 1e38
 x(4) = 1e38
 x(5) = -36.0

do i=1,5
y(i) = 100.0*(x(i))
print *, 'x = ', x(i), ' x*100.0 = ',y(i)
end do
end
```

The following command line produces stack trace information for the program executable.

```
> ifort -O0 -fpe0 -traceback ovf.f90 -o ovf.exe
> ovf.exe

x =  -1.0000000E+32  x*100.0
 =  -1.0000000E+34
forrtl: error (72): floating overflow
Image        PC         Routine       Line      Source
ovf.exe      08049E4A   MAIN__            14    ovf.f90
ovf.exe      08049F08   Unknown       Unknown   Unknown
ovf.exe      400B3507   Unknown       Unknown   Unknown
ovf.exe      08049C51   Unknown       Unknown   Unknown
Abort
```

The following suppresses stack trace information because the `FOR_DISABLE_STACK_TRACE` environment variable is set.

```
> setenv FOR_DISABLE_STACK_TRACE true> ovf.exe

x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34
forrtl: error (72): floating overflow
Abort
```

Run-Time Library Message Catalog File Location

The libifcore, libirc, and libm run-time libraries ship message catalogs. When a message by one of these libraries is to be displayed, the library searches for its message catalog in a directory specified by either the `NLSPATH` (Linux* and macOS*), or `%PATH%` (Windows*) environment variable. If the message catalog cannot be found, the message is displayed in English.

The names of the three message catalogs are as follows:

libifcore message catalogs and related text message files	Linux* and macOS*	`ifcore_msg.cat` `ifcore_msg.msg`
	Windows*	`ifcore_msg.dll` `ifcore_msg.mc`
libirc message catalogs and related text message files	Linux* and macOS*	`irc_msg.cat` `irc_msg.msg`
	Windows*	`irc_msg.dll` `irc_msg.mc`
libm message catalogs and related text message files	Linux* and macOS*	`libm.cat` `libm.msg`
	Windows*	`libmUI.dll` `libmUI.mc`

See Also

Understanding Project Types
Using the IOSTAT Specifier and Fortran Exit Codes
Using the END EOR and ERR Branch Specifiers
Methods of Handling Errors
Run-Time Default Error Processing

Values Returned at Program Termination

An Intel® Fortran program can terminate in a number of ways. On Linux* and macOS*, values are returned to the shell.

- The program runs to normal completion. A value of zero is returned.
- The program stops with a STOP or ERROR STOP statement. If an integer stop-code is specified, a status equal to the code is returned; if no stop-code is specified, a status of zero is returned.
- The program stops because of a signal that is caught but does not allow the program to continue. A value of '1' is returned.
- The program stops because of a severe run-time error. The error number for that run-time error is returned. See Understanding Run-Time Errors and related topics.
- The program stops with a CALL EXIT statement. The value passed to EXIT is returned.
- The program stops with a CALL ABORT statement. A value of '134' is returned.

See Also

Understanding Run-Time Errors

Methods of Handling Errors

Whenever possible, the Intel® Fortran RTL does certain error handling, such as generating appropriate messages and taking necessary action to recover from errors. You can explicitly supplement or override default actions by using the following methods:

- To transfer control to error-handling code within the program, use the END, EOR, and ERR branch specifiers in I/O statements. See Using the END EOR and ERR Branch Specifiers.
- To identify Fortran-specific I/O errors based on the value of Intel® Fortran RTL error codes, use the I/O status specifier (IOSTAT) in I/O statements (or call the ERRSNS subroutine). See Using the IOSTAT Specifier and Fortran Exit Codes.
- Obtain system-level error codes by using the appropriate library routines.
- For certain error conditions, use the signal handling facility to change the default action to be taken.

See Also

Using the IOSTAT Specifier and Fortran Exit Codes
Using the END EOR and ERR Branch Specifiers

Using the END, EOR, and ERR Branch Specifiers

When a severe error occurs during Intel® Fortran program execution, the default action is to display an error message and terminate the program. To override this default action, there are three branch specifiers you can use in I/O statements to transfer control to a specified point in the program:

- The END branch specifier handles an end-of-file condition.
- The EOR branch specifier handles an end-of-record condition for non-advancing reads.
- The ERR branch specifier handles all error conditions.

If you use the END, EOR, or ERR branch specifiers, no error message is displayed and execution continues at the designated statement, usually error-handling code.

You might encounter an unexpected error that the error-handling routine cannot handle. In this case, do one of the following:

- Modify the error-handling routine to display the error message number.
- Remove the END, EOR, or ERR branch specifiers from the I/O statement that causes the error.

After you modify the source code, compile, link, and run the program to display the error message. For example:

```
    READ (8,50,ERR=400)
```

If any severe error occurs during execution of this statement, the Intel® Visual Fortran RTL transfers control to the statement at label 400. Similarly, you can use the END specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

```
    READ (12,70,END=550)
```

When using non-advancing I/O, use the EOR specifier to handle the end-of-record condition. For example:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

You can also use ERR as a specifier in an OPEN, CLOSE, or INQUIRE statement. For example:

```
    OPEN (UNIT=10, FILE='FILNAME', STATUS='OLD', ERR=999)
```

If an error is detected during execution of this OPEN statement, control transfers to the statement at label 999.

Using the IOSTAT Specifier and Fortran Exit Codes

The IOSTAT specifier can be used to continue program execution after an I/O error, or an end-of-file or end-of-record condition occurs, and to return information about the status of I/O operations. Certain error conditions are *not* returned in IOSTAT.

The IOSTAT specifier can supplement or replace the use of the END=, EOR=, and ERR= branch specifiers.

Use of an IOSTAT= specifier in an I/O statement prevents initiation of error termination if an error occurs during the execution of the I/O statement. The integer variable specified in the IOSTAT= specifier becomes defined during the execution of the I/O statement with the following values:

- 0 for normal completion of the I/O statement (no error, end-of-file, or end-of-record condition occurs).
- The value of the (negative) default integer scalar constant IOSTAT_EOR defined in the intrinsic module ISO_FORTRAN_ENV if no error condition or end-of-file condition occurs, but an end-of-record condition does occur during the execution of an input statement.
- The value of the (negative) default integer scalar constant IOSTAT_END defined in ISO_FORTRAN_ENV if no error condition occurs, but an end-of-file condition does occur during the execution of an input statement.
- For an INQUIRE statement, the value of the default integer constant IOSTAT_INQUIRE_INTERNAL_UNIT defined in ISO_FORTRAN_ENV if a file-unit-number identifies an internal unit in the execution of the statement.
- A positive integer value if an error condition occurs. (This value is one of the Fortran-specific IOSTAT numbers listed in the run-time error message. See List of Run-Time Error Messages, which lists the messages.)

Note that the value assigned to the IOSTAT variable is the same value that would be returned as an exit code if error termination was initiated.

Following the execution of the I/O statement and assignment of an IOSTAT value, control transfers to the END=, EOR=, or ERR= statement label, if any. If there is no control transfer, normal execution continues.

The non-standard include file FOR_ISODEF.FOR and the non-standard module FORISODEF contain symbolic constants for the values returned through an IOSTAT= specifier.

The following example uses the IOSTAT specifier and the module FORIOSDEF to handle an OPEN statement error (in the FILE specifier).

```
    USE foriosdef
    IMPLICIT NONE
    CHARACTER(LEN=40) :: FILNM
    INTEGER IERR
    PRINT *, 'Type file name:'
    READ (*,*) FILNM
    OPEN (UNIT=1, FILE=FILNM, STATUS='OLD', IOSTAT=IERR, ERR=100)
    PRINT *, 'Opening file: ', FILNM
    ! process the input file
    ...
    CLOSE (UNIT=1)
    STOP
100 IF(IERR . EQ. FOR$IOS_FILNOTFOU) THEN
       PRINT *, 'File: ', FILNM, ' does not exist '
    ELSE
       PRINT *, 'Unrecoverable error, code =', IERR
    END IF
    END PROGRAM
```

Another way to obtain information about an error is the ERRSNS subroutine, which allows you to obtain the last I/O system error code associated with an Intel® Fortran RTL error (see the *Intel® Fortran Language Reference*).

See Also

List of Run-Time Error Messages
IS_IOSTAT_END intrinsic function
IS_IOSTAT_EOR intrinsic function

Locating Run-Time Errors

This topic provides some guidelines for locating the cause of exceptions and run-time errors. Intel® Fortran run-time error messages do not usually indicate the exact source location causing the error. The following compiler options are related to handling errors and exceptions:

- The `-check [keyword]` (Linux* and macOS*) or `/check[:keyword]` (Windows*) option generates extra code to catch certain conditions at run time.

  The keyword `bounds` generates code to perform compile-time and run-time checks on array subscript and character substring expressions. An error is reported if the expression is outside the dimension of the array or the length of the string.

  The keyword `uninit` generates code for dynamic checks of uninitialized variables. If a variable is read before written, a run-time error routine will be called.

  The keywords `noformat` and `nooutput_conversion`reduce the severity level of the associated run-time error to allow program continuation.

  The keyword `pointers` generates code to test for disassociated pointers and unallocatable arrays. The following `check[:]pointers` option examples result in various output messages.

  Example: Allocatable variable not allocated

```
real, allocatable:: a(:)
   !
   ! A is initially unallocated.  To allocate, use:
   !
   !    allocate(a(4))
   !
   ! Because A is unallocated, the next statement will
   ! issue an error in applications built with "check pointers".
   !
   a=17
   print  *,a
   end
Output 1:
forrtl: severe (408): fort: (8): Attempt to fetch from allocatable
                                 variable A when it is not allocated
```

  Example: Unassociated Pointer

```
   real, pointer:: a(:)
   allocate(a(5))
   a=17
   print  *,a
   deallocate(a)   ! Once A is deallocated, the next statement
                   ! issues an error in an application built
                   ! with "check pointers".
   a=20
   print  *,a
   end

Output 2:
```

```
   17.00000       17.00000       17.00000       17.00000       17.00000
forrtl: severe (408): fort: (7): Attempt to use pointer A when it is
                                 not associated with a target
```

Example: Cray pointer with zero value

```
   pointer(p,a)
   real, target:: b
   !
   ! P initially has no address assigned to it.  To assign an
   ! address, use:
   !
   !    p=loc(b)
   !
   ! Because P has no address assigned to it, the next
   ! statement will issue an error in an application built
   ! with "check pointers".
   !
   b=17.
   print  *,a
   end
Output 3:
forrtl: severe (408): fort: (9): Attempt to use pointee A when its
                                 corresponding integer
                                 pointer P has the value zero
```

- The `traceback` option generates extra information in the object file to provide source file traceback information when a severe error occurs at run time. This simplifies the task of locating the cause of severe run-time errors. Without the `traceback` option, you could try to locate the cause of the error using a map file and the hexadecimal addresses of the stack displayed when a severe error occurs. Certain traceback-related information accompanies severe run-time errors, as described in Traceback.
- The `fpe` option controls the handling of floating-point arithmetic exceptions (IEEE arithmetic) at run time. If you specify the `-fpe[:]3` compiler option, all floating-point exceptions are disabled, allowing IEEE exceptional values and program continuation. In contrast, specifying `-fpe[:]0` stops execution when an exceptional value (such as a NaN) is generated, when floating overflow or divide by zero occur, or when attempting to use a subnormal number, which usually allows you to localize the cause of the error. It also forces underflow to zero.
- The `warn` and `nowarn` options control compile-time warning messages, which, in some circumstances, can help determine the cause of a run-time error.
- On Linux* and on macOS*, the `-fexceptions` option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.
- On Windows*, the Compilation Diagnostics Options in the IDE control compile-time diagnostic messages, which, in some circumstances can help determine the cause of a run-time error.

See Also

Understanding Run-Time Errors
Traceback

List of Run-Time Error Messages

This section lists the errors processed by the compiler run-time library (RTL). For each error, the table provides the error number, the severity code, error message text, condition symbol name, and a detailed description of the error.

To define the condition symbol values (PARAMETER statements) in your program, include the following file:

```
for_iosdef.for
```

As described in the table, the severity of the message determines which of the following occurs:

- with *info* and *warning*, program execution continues
- with *error*, the results may be incorrect
- with *severe*, program execution stops (unless a recovery method is specified)

In the last case, to prevent program termination, you must include either an appropriate I/O error-handling specifier and recompile or, for certain errors, change the default action of a signal before you run the program again.

In the following table, the first column lists error numbers returned to IOSTAT variables when an I/O error is detected.

The first line of the second column provides the message as it is displayed (following *forrtl:*), including the severity level, message number, and the message text. The following lines of the second column contain the status condition symbol (such as FOR$IOS_INCRECTYP) and an explanation of the message.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
1[1]	**severe (1): Not a Fortran-specific error** FOR$IOS_NOTFORSPE. An error in the user program or in the RTL was not an Intel® Fortran-specific error and was not reportable through any other Intel® Fortran run-time messages.
8	**severe (8): Internal consistency check failure** FOR$IOS_BUG_CHECK. Internal error. Please check that the program is correct. Recompile if an error existed in the program. If this error persists, submit a problem report.
9	**severe (9): Permission to access file denied** FOR$IOS_PERACCFIL. Check the permissions of the specified file and whether the network device is mapped and available. Make sure the correct file and device was being accessed. Change the protection, specific file, or process used before rerunning the program.
10	**severe (10): Cannot overwrite existing file** FOR$IOS_CANOVEEXI. Specified file `xxx` already exists when OPEN statement specified STATUS='NEW' (create new file) using I/O unit *x*. Make sure correct file name, directory path, unit, and so forth were specified in the source program. Decide whether to: • Rename or remove the existing file before rerunning the program. • Modify the source file to specify different file specification, I/O unit, or OPEN statement STATUS.
11[1]	**info (11): Unit not connected** FOR$IOS_UNINOTCON. The specified unit was not open at the time of the attempted I/O operation. Check if correct unit number was specified. If appropriate, use an OPEN statement to explicitly open the file (connect the file to the unit number).
17	**severe (17): Syntax error in NAMELIST input** FOR$IOS_SYNERRNAM. The syntax of input to a namelist-directed READ statement was incorrect.
18	**severe (18): Too many values for NAMELIST variable** FOR$IOS_TOOMANVAL. An attempt was made to assign too many values to a variable during a namelist READ statement.
19	**severe (19): Invalid reference to variable in NAMELIST input** FOR$IOS_INVREFVAR. One of the following conditions occurred: • The variable was not a member of the namelist group. • An attempt was made to subscript a scalar variable. • A subscript of the array variable was out-of-bounds. • An array variable was specified with too many or too few subscripts for the variable.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	• An attempt was made to specify a substring of a non-character variable or array name. • A substring specifier of the character variable was out-of-bounds. • A subscript or substring specifier of the variable was not an integer constant. • An attempt was made to specify a substring by using an unsubscripted array variable.
20	**severe (20): REWIND error** FOR$IOS_REWERR. One of the following conditions occurred: • The file was not a sequential file. • The file was not opened for sequential or append access. • The Intel® Fortran RTL I/O system detected an error condition during execution of a REWIND statement.
21	**severe (21): Duplicate file specifications** FOR$IOS_DUPFILSPE. Multiple attempts were made to specify file attributes without an intervening close operation. A DEFINE FILE statement was followed by another DEFINE FILE statement or an OPEN statement.
22	**severe (22): Input record too long** FOR$IOS_INPRECTOO. A record was read that exceeded the explicit or default record length specified when the file was opened. To read the file, use an OPEN statement with a RECL=VALUE (record length) of the appropriate size.
23	**severe (23): BACKSPACE error** FOR$IOS_BACERR. The Intel® Fortran RTL I/O system detected an error condition during execution of a BACKSPACE statement.
24[1]	**severe (24): End-of-file during read** FOR$IOS_ENDDURREA. One of the following conditions occurred: • An Intel® Fortran RTL I/O system end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An end-of-file record written by the ENDFILE statement was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An attempt was made to read past the end of an internal file character string or array during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. This error is returned by END and ERRSNS.
25	**severe (25): Record number outside range** FOR$IOS_RECNUMOUT. A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was opened.
26	**severe (26): OPEN or DEFINE FILE required** FOR$IOS_OPEDEFREQ. A direct access READ, WRITE, or FIND statement was attempted for a file when no prior DEFINE FILE or OPEN statement with ACCESS='DIRECT' was performed for that file.
27	**severe (27): Too many records in I/O statement** FOR$IOS_TOOMANREC. An attempt was made to do one of the following: • Read or write more than one record with an `ENCODE` or `DECODE` statement. • Write more records than existed.
28	**severe (28): CLOSE error**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_CLOERR. An error condition was detected by the Intel® Fortran RTL I/O system during execution of a CLOSE statement.
29	**severe (29): File not found** FOR$IOS_FILNOTFOU. A file with the specified name could not be found during an OPEN operation.
30	**severe (30): Open failure** FOR$IOS_OPEFAI. An error was detected by the Intel® Fortran RTL I/O system while attempting to open a file in an OPEN, INQUIRE, or other I/O statement. This message is issued when the error condition is not one of the more common conditions for which specific error messages are provided. It can occur when an OPEN operation was attempted for one of the following: • Segmented file that was not on a disk or a raw magnetic tape • Standard I/O file that had been closed
31	**severe (31): Mixed file access modes** FOR$IOS_MIXFILACC. An attempt was made to use any of the following combinations: • Formatted and unformatted operations on the same unit • An invalid combination of access modes on a unit, such as direct and sequential • An Intel® Fortran RTL I/O statement on a logical unit that was opened by a program coded in another language
32	**severe (32): Invalid logical unit number** FOR$IOS_INVLOGUNI. A logical unit number greater than 2,147,483,647 or less than zero was used in an I/O statement.
33	**severe (33): ENDFILE error** FOR$IOS_ENDFILERR. One of the following conditions occurred: • The file was not a sequential organization file with variable-length records. • The file was not opened for sequential, append, or direct access. • An unformatted file did not contain segmented records. • The Intel® Fortran RTL I/O system detected an error during execution of an ENDFILE statement.
34	**severe (34): Unit already open** FOR$IOS_UNIALROPE. A DEFINE FILE statement specified a logical unit that was already opened.
35	**severe (35): Segmented record format error** FOR$IOS_SEGRECFOR. An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or was created by a program written in a language other than Fortran or Standard Fortran.
36	**severe (36): Attempt to access non-existent record** FOR$IOS_ATTACCNON. A direct-access READ or FIND statement attempted to access beyond the end of a relative file (or a sequential file on disk with fixed-length records) or access a record that was previously deleted from a relative file.
37	**severe (37): Inconsistent record length** FOR$IOS_INCRECLEN. An attempt was made to open a direct access file without specifying a record length.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
38	**severe (38): Error during write** FOR$IOS_ERRDURWRI. The Intel® Fortran RTL I/O system detected an error condition during execution of a WRITE statement.
39	**severe (39): Error during read** FOR$IOS_ERRDURREA. The Intel® Fortran RTL I/O system detected an error condition during execution of a READ statement.
40	**severe (40): Recursive I/O operation** FOR$IOS_RECIO_OPE. While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted, such as a function subprogram that performs I/O to the same logical unit that was referenced in an expression in an I/O list or variable format expression.
41	**severe (41): Insufficient virtual memory** FOR$IOS_INSVIRMEM. The Intel® Fortran RTL attempted to exceed its available virtual memory while dynamically allocating space. To overcome this problem, investigate increasing the data limit. Before you try to run this program again, wait until the new system resources take effect. **NOTE** This error can be returned by STAT in an ALLOCATE or a DEALLOCATE statement.
42	**severe (42): No such device** FOR$IOS_NO_SUCDEV. A pathname included an invalid or unknown device name when an OPEN operation was attempted.
43	**severe (43): File name specification error** FOR$IOS_FILNAMSPE. A pathname or file name given to an OPEN or INQUIRE statement was not acceptable to the Intel® Fortran RTL I/O system.
44	**severe (44): Inconsistent record type** FOR$IOS_INCRECTYP. The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.
45	**severe (45): Keyword value error in OPEN statement** FOR$IOS_KEYVALERR. An improper value was specified for an OPEN or CLOSE statement specifier requiring a value.
46	**severe (46): Inconsistent OPEN/CLOSE parameters** FOR$IOS_INCOPECLO. Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations follow: • READONLY or ACTION='READ' with STATUS='NEW' or STATUS='SCRATCH' • READONLY with STATUS='REPLACE', ACTION='WRITE', or ACTION='READWRITE' • ACCESS='APPEND' with READONLY, ACTION='READ', STATUS='NEW', or STATUS='SCRATCH' • DISPOSE='SAVE', 'PRINT', or 'SUBMIT' with STATUS='SCRATCH' • DISPOSE='DELETE' with READONLY • CLOSE statement STATUS='DELETE' with OPEN statement READONLY • ACCESS='DIRECT' with POSITION='APPEND' or 'ASIS'
47	**severe (47): Write to READONLY file**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_WRIREAFIL. A write operation was attempted to a file that was declared ACTION='READ' or READONLY in the OPEN statement that is currently in effect.
48	**severe (48): Invalid argument to Fortran Run-Time Library** FOR$IOS_INVARGFOR. The compiler passed an invalid or improperly coded argument to the Intel® Fortran RTL. This can occur if the compiler is newer than the RTL in use.
51	**severe (51): Inconsistent file organization** FOR$IOS_INCFILORG. The file organization specified in an OPEN statement did not match the organization of the existing file.
53	**severe (53): No current record** FOR$IOS_NO_CURREC. Attempted to execute a REWRITE statement to rewrite a record when the current record was undefined. To define the current record, execute a successful READ statement. You can optionally perform an INQUIRE statement on the logical unit after the READ statement and before the REWRITE statement. No other operations on the logical unit may be performed between the READ and REWRITE statements.
55	**severe (55): DELETE error** FOR$IOS_DELERR. An error condition was detected by the Intel® Fortran RTL I/O system during execution of a DELETE statement.
57	**severe (57): FIND error** FOR$IOS_FINERR. The Intel® Fortran RTL I/O system detected an error condition during execution of a FIND statement.
58[1]	**info (58): Format syntax error at or near xx** FOR$IOS_FMTSYN. Check the statement containing *xx*, a character substring from the format string, for a format syntax error. For more information, see the FORMAT statement.
59	**severe (59): List-directed I/O syntax error** FOR$IOS_LISIO_SYN. The data in a list-directed input record had an invalid format, or the type of the constant was incompatible with the corresponding variable. The value of the variable was unchanged. **NOTE** Note: The ERR transfer is taken after completion of the I/O statement for error number 59. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
60	**severe (60): Infinite format loop** FOR$IOS_INFFORLOO. The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.
61	**severe or info(61): Format/variable-type mismatch** FOR$IOS_FORVARMIS. An attempt was made either to read or write a real variable with an integer field descriptor (I, L, O, Z, B), or to read or write an integer or logical variable with a real field descriptor (D, E, or F). To suppress this error message, see the description of `check[:]noformat`.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

NOTE
Note: The severity depends on the `check[:]keywords` option used during the compilation command. The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

62 **severe (62): Syntax error in format**

FOR$IOS_SYNERRFOR. A syntax error was encountered while the RTL was processing a format stored in an array or character variable.

63 **error or info(63): Output conversion error**

FOR$IOS_OUTCONERR. During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. When this situation is encountered, the overflowed field is filled with asterisks to indicate the error in the output record. If no ERR address has been defined for this error, the program continues after the error message is displayed. To suppress this error message, see the description of `check[:]nooutput_conversion`.

NOTE
Note: The severity depends on the `check[:]keywords` option used during the compilation command. The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

64 **severe (64): Input conversion error**

FOR$IOS_INPCONERR. During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable was set to zero.

NOTE
The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

65 **error (65): Floating invalid**

FOR$IOS_FLTINV. During an arithmetic operation, the floating-point values used in a calculation were invalid for the type of operation requested or invalid exceptional values. For example, the error can occur if you request a log of the floating-point values 0.0 or a negative number. For certain arithmetic expressions, specifying the `check[:]nopower` option can suppress this message.

66 **severe (66): Output statement overflows record**

FOR$IOS_OUTSTAOVE. An output statement attempted to transfer more data than would fit in the maximum record size.

67 **severe (67): Input statement requires too much data**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_INPSTAREQ. Attempted to read more data than exists in a record with an unformatted READ statement or with a formatted sequential READ statement from a file opened with a PAD='NO' specifier.
68	**severe (68): Variable format expression value error** FOR$IOS_VFEVALERR. The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of 1 was assumed, except for a P edit descriptor, for which a value of zero was assumed.
	NOTE The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
69[1]	**error (69): Process interrupted (SIGINT)** FOR$IOS_SIGINT. The process received the signal SIGINT. Determine source of this interrupt signal (described in signal(3)).
72[1]	**error (72): Floating overflow** FOR$IOS_FLTOVF. During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type. See Data Representation for ranges of the various data types.
73[1]	**error (73): Floating divide by zero** FOR$IOS_FLTDIV. During a floating-point arithmetic operation, an attempt was made to divide by zero.
74[1]	**error (74): Floating underflow** FOR$IOS_FLTUND. During an arithmetic operation, a floating-point value became less than the smallest finite value for that data type. Depending on the values of the `fpe[:]`*n* option, the underflowed result was either set to zero or allowed to gradually underflow. See the Data Representation for ranges of the various data types.
75[1]	**error (75): Floating point exception** FOR$IOS_SIGFPE. A floating-point exception occurred. Possible causes include: • Division by zero. • Overflow. • An invalid operation, such as subtraction of infinite values, multiplication of zero by infinity without signs), division of zero by zero or infinity by infinity. • Conversion of floating-point to fixed-point format when an overflow prevents conversion.
76[1]	**error (76): IOT trap signal** FOR$IOS_SIGIOT. Core dump file created. Examine core dump for possible cause of this `IOT` signal.
77[1]	**severe (77): Subscript out of range** FOR$IOS_SUBRNG. An array reference was detected outside the declared array bounds.
78[1]	**error (78): Process killed** FOR$IOS_SIGTERM. The process received a signal requesting termination of this process. Determine the source of this software termination signal.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
79[1]	**error (79): Process quit** FOR$IOS_SIGQUIT. The process received a signal requesting termination of itself. Determine the source of this quit signal.
90[1]	**info (90): INQUIRE of internal unit-number is always an error (NOTE: unit identifies a file)** FOR$IOS_IOSTAT_INTERNAL. The file name specified for an INQUIRE statement is an internal unit-number.
91[1]	**info (91): INQUIRE of internal unit-number is always an error (NOTE: unit does not identify a file)** FOR$IOS_IOSTAT_INTERNAL_UNIT. The unit number specified for an INQUIRE statement is an internal unit-number.
95[1]	**info (95): Floating-point conversion failed** FOR$IOS_FLOCONFAI. The attempted unformatted read or write of nonnative floating-point data failed because the floating-point value: • Exceeded the allowable maximum value for the equivalent native format and was set equal to infinity (plus or minus). • Was infinity (plus or minus) and was set to infinity (plus or minus). • Was invalid and was set to not a number (NaN). Very small numbers are set to zero (0). This error could be caused by the specified nonnative floating-point format not matching the floating-point format found in the specified file. Check the following: • The correct file was specified. • The record layout matches the format Intel® Fortran is expecting. • The ranges for the data being used (see Data Representation). • The correct nonnative floating-point data format was specified (see Supported Native and Nonnative Numeric Formats).
96	**info (96): F_UFMTENDIAN environment variable was ignored:erroneous syntax** FOR$IOS_UFMTENDIAN. Syntax for specifying whether little endian or big endian conversion is performed for a given Fortran unit was incorrect. Even though the program will run, the results might not be correct if you do not change the value of `F_UFMTENDIAN`. For correct syntax, see Environment Variable F_UFMTENDIAN Method.
98	**Severe (98): cannot allocate memory for the file buffer - out of memory** FOR_S_NOMEMFORIO. This error often occurs during a file I-O operation such as OPEN, READ, or WRITE. Either increase the amount of memory available to the program, or reduce its demand.
104	**Severe (104): incorrect "XXXXX=" specifier value for connected file** FOR$IOS_BADSPECVAL. The listed specifier value is invalid for the connected file. Possible specifier values are one of the following: • ACTION= • ASSOCIATEVARIABLE= • ASYNCHRONOUS= • BUFFERED= • DISPOSE= • FORM= • IOFOCUS= • MAXREC=

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	• MODE= • ORGANIZATION= • POSITION= • RECL= • RECORDTYPE= • SHARE= • STATUS= • TITLE=
106	**Severe (106): FORT_BLOCKSIZE environment variable has erroneous syntax** FOR_S_INVBLOCKSIZE. Syntax for specifying the default block size value was incorrect. For correct syntax, see Environment Variable `FORT_BLOCKSIZE`.
107	**Severe (107): FORT_BUFFERCOUNT environment variable has erroneous syntax** FOR_S_INVBUFRCNT. Syntax for specifying the default buffer count value was incorrect. For correct syntax, see Environment Variable `FORT_BUFFERCOUNT`.
108	**Severe (108): Cannot stat file** FOR$IOS_CANSTAFILE. Make sure correct file and unit were specified.
109	**info (109): stream data transfer statement is not allowed to an unopened unit** FOR$IOS_SIONOTOPEN. Stream data transfer statement is not allowed to an unopened unit.
118	**severe (118): The 'RECL=' value in an OPEN statement exceeds the maximum allowed for the file's record type.** FOR$IOS_MAX_FXD_RECL. The 'RECL=' value in an OPEN statement is too large for the file's record type.
119[1]	**severe (119): The FORT_BUFFERING_THRESHOLD environment variable has erroneous syntax** FOR$ IOS_INVTHRESHOLD. If specified, the FORT_BUFFERING_THRESHOLD environment variable must be an integer value greater than zero and less than 2147483647.
120	**severe (120): Operation requires seek ability** FOR$IOS_OPEREQSEE. Attempted an operation on a file that requires the ability to perform seek operations on that file. Make sure the correct unit, directory path, and file were specified.
121	**Severe (121): Cannot access current working directory** FOR$IOS_CWDERROR. Cannot access the current working directory to open or access a file. One of the following conditions occurred: • The directory has been deleted or moved by another task. • The getcwd system function failed due to an OS or file-system problem.
122	**Severe (122): invalid attempt to assign into a pointer that is not associated** FOR$IOS_UNASSOCPTR. Invalid attempt to assign into a pointer that is not associated.
124[1]	**severe (124): Invalid command supplied to EXECUTE_COMMAND_LINE** FOR$ IOS_INVCMDECL. The command line passed to the EXECUTE_COMMAND_LINE intrinsic is invalid.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

NOTE
This error can be returned by CMDSTAT in an EXECUTE_COMMAND_LINE statement.

127 **Severe (127): User Defined I/O procedure returned error <user-iostat, message: '<user-iomsg>'**

FOR$IOS_UDIO_CHILD_IOSTAT_IOMSG. A child I/O action has set the IOMSG and IOSTAT shown in this message.

128 **info (128): Error stop – program terminated**

FOR$IOS_ERROR_STOP. The user program has executed an ERROR STOP statement.

129 **info (129): User Defined I/O procedure's IOMSG was truncated to fit IOMSG variable**

FOR$IOS_UDIO_IOMSG_TRUNCATED. A child I/O action has set IOMSG and IOSTAT. The result was too large to be held by the parent's IOMSG variable; non-blank text has been truncated.

134 **No associated message**

The program was terminated internally through abort().

138[1] **severe (138): Array index out of bounds**

FOR$IOS_BRK_RANGE. An array subscript is outside the dimensioned boundaries of that array. Set the `check[:]bounds` option and recompile.

139[1] **severe: (139): Array index out of bounds for index nn**

FOR$IOS_BRK_RANGE2. An array subscript is outside the dimensioned boundaries of that array. Set the `check[:]bounds` option and recompile.

140[1] **error (140): Floating inexact**

FOR$IOS_FLTINE. A floating-point arithmetic or conversion operation gave a result that differs from the mathematically exact result. This trap is reported if the rounded result of an IEEE operation is not exact.

144[1] **severe (144): Reserved operand**

FOR$IOS_ROPRAND. The Intel® Fortran RTL encountered a reserved operand while executing your program. Please report the problem to Intel.

145[1] **severe (145): Assertion error**

FOR$IOS_ASSERTERR. The Intel® Fortran RTL encountered an assertion error. Please report the problem to Intel.

146[1] **severe (146): Null pointer error**

FOR$IOS_NULPTRERR. Attempted to use a pointer that does not contain an address. Modify the source program, recompile, and relink.

147[1] **severe (147): Stack overflow**

FOR$IOS_STKOVF. The Intel® Fortran RTL encountered a stack overflow while executing your program.

148[1] **severe (148): String length error**

FOR$IOS_STRLENERR. During a string operation, an integer value appears in a context where the value of the integer is outside the permissible string length range. Set the `check[:]bounds` option and recompile.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
149[1]	**severe (149): Substring error** FOR$IOS_SUBSTRERR. An array subscript is outside the dimensioned boundaries of an array. Set the `check[:]bounds` option and recompile.
150[1]	**severe (150): Range error** FOR$IOS_RANGEERR. An integer value appears in a context where the value of the integer is outside the permissible range.
151[1]	**severe (151): Allocatable array is already allocated** FOR$IOS_INVREALLOC. An allocatable array must not already be allocated when you attempt to allocate it. You must deallocate the array before it can again be allocated. **NOTE** This error can be returned by STAT in an ALLOCATE statement.
152[1]	**severe (152): Unresolved contention for Intel Fortran RTL global resource** FOR$IOS_RESACQFAI. Failed to acquire an Intel® Fortran RTL global resource for a reentrant routine. For a multithreaded program, the requested global resource is held by a different thread in your program. For a program using asynchronous handlers, the requested global resource is held by the calling part of the program (such as main program) and your asynchronous handler attempted to acquire the same global resource.
153[1]	**severe (153): Allocatable array or pointer is not allocated** FOR$IOS_INVDEALLOC. A Standard Fortran allocatable array or pointer must already be allocated when you attempt to deallocate it. You must allocate the array or pointer before it can again be deallocated. **NOTE** This error can be returned by STAT in an DEALLOCATE statement.
154[1]	**severe(154): Array index out of bounds** FOR$IOS_RANGE. An array subscript is outside the dimensioned boundaries of that array. Set the `check[:]bounds` option and recompile.
155[1]	**severe(155): Array index out of bounds for index nn** FOR$IOS_RANGE2. An array subscript is outside the dimensioned boundaries of that array. Set the `check[:]bounds` option and recompile.
156[1]	**severe(156): GENTRAP code = hex dec** FOR$IOS_DEF_GENTRAP. The Intel® Fortran RTL has detected an unknown `GENTRAP` code. The cause is most likely a software problem due to memory corruption, or software signaling an exception with an incorrect exception code. Try setting the `check[:]bounds` option and recompile to see if that finds the problem.
157[1]	**severe(157): Program Exception - access violation** FOR$IOS_ACCVIO. The program tried to read from or write to a virtual address for which it does not have the appropriate access. Try setting the `check[:]bounds` option and recompile to see if the problem is an out-of-bounds memory reference or a argument mismatch that causes data to be treated as an address.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	Other possible causes of this error include: • Mismatches in C vs. STDCALL calling mechanisms, causing the stack to become corrupted. • References to unallocated pointers. • Attempting to access a protected (for example, read-only) address.
158[1]	**severe(158): Program Exception - datatype misalignment** FOR$IOS_DTYPE_MISALIGN. The Intel® Fortran RTL has detected data that is not aligned on a natural boundary for the data type specified. For example, a REAL(8) data item aligned on natural boundaries has an address that is a multiple of 8. To ensure naturally aligned data, use the `align` option. This is an operating system error. See your operating system documentation for more information.
159[1]	**severe(159): Program Exception - breakpoint** FOR$IOS_PGM_BPT. The Intel® Fortran RTL has encountered a breakpoint in the program. This is an operating system error. See your operating system documentation for more information.
160[1]	**severe(160): Program Exception - single step** FOR$IOS_PGM_SS. A trace trap or other single-instruction mechanism has signaled that one instruction has been executed. This is an operating system error. See your operating system documentation for more information.
161[1]	**severe(161): Program Exception - array bounds exceeded** FOR$IOS_PGM_BOUNDS. The program tried to access an array element that is outside the specified boundaries of the array. Set the `check[:]bounds` option and recompile.
162[1]	**severe(162): Program Exception - denormal floating-point operand** FOR$IOS_PGM_DENORM. A floating-point arithmetic or conversion operation has a subnormalized number as an operand. A subnormalized number is smaller than the lowest value in the normal range for the data type specified. See Data Representation for ranges for floating-point types. Either locate and correct the source code causing the subnormalized value or, if a subnormalized value is acceptable, specify a different value for the `fpe` compiler option to allow program continuation.
163[1]	**severe(163): Program Exception - floating stack check** FOR$IOS_PGM_FLTSTK. During a floating-point operation, the floating-point register stack on systems using IA-32 architecture overflowed or underflowed. This is a fatal exception. The most likely cause is calling a REAL function as if it were an INTEGER function or subroutine, or calling an INTEGER function or subroutine as if it were a REAL function. Carefully check that the calling code and routine being called agree as to how the routine is declared. If you are unable to resolve the issue, please send a problem report with an example to Intel.
164[1]	**severe(164): Program Exception - integer divide by zero** FOR$IOS_PGM_INTDIV. During an integer arithmetic operation, an attempt was made to divide by zero. Locate and correct the source code causing the integer divide by zero.
166[1]	**severe(166): Program Exception - privileged instruction**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_PGM_PRIVINST. The program tried to execute an instruction whose operation is not allowed in the current machine mode. This is an operating system error. See your operating system documentation for more information.
167[1]	**severe(167): Program Exception - in page error** FOR$IOS_PGM_INPGERR. The program tried to access a page that was not present, so the system was unable to load the page. For example, this error might occur if a network connection was lost while trying to run a program over the network. This is an operating system error. See your operating system documentation for more information.
168[1]	**severe(168): Program Exception - illegal instruction** FOR$IOS_PGM_ILLINST. The program tried to execute an invalid instruction. This is an operating system error. See your operating system documentation for more information.
169[1]	**severe(169): Program Exception - noncontinuable exception** FOR$IOS_PGM_NOCONTEXCP. The program tried to continue execution after a noncontinuable exception occurred. This is an operating system error. See your operating system documentation for more information.
170[1]	**severe(170): Program Exception - stack overflow** FOR$IOS_PGM_STKOVF. The Intel® Fortran RTL has detected a stack overflow while executing your program. See your Release Notes for information on how to increase stack size.
171[1]	**severe(171): Program Exception - invalid disposition** FOR$IOS_PGM_INVDISP. An exception handler returned an invalid disposition to the exception dispatcher. Programmers using a high-level language should never encounter this exception. This is an operating system error. See your operating system documentation for more information.
172[1]	**severe(172): Program Exception - exception code = hex dec** FOR$IOS_PGM_EXCP_CODE. The Intel® Fortran RTL has detected an unknown exception code. This is an operating system error. See your operating system documentation for more information.
173[1]	**severe(173): A pointer passed to DEALLOCATE points to an array that cannot be deallocated** FOR$IOS_INVDEALLOC2. A pointer that was passed to DEALLOCATE pointed to an explicit array, an array slice, or some other type of memory that could not be deallocated in a DEALLOCATE statement. Only whole arrays previous allocated with an ALLOCATE statement may be validly passed to DEALLOCATE. **NOTE** This error can be returned by STAT in a DEALLOCATE statement.
174[1]	**severe (174): SIGSEGV, message-text** FOR$IOS_SIGSEGV. One of two possible messages occurs for this error number:

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

```
severe (174): SIGSEGV, segmentation fault occurred
```

This message indicates that the program attempted an invalid memory reference. Check the program for possible errors.

```
severe (174): SIGSEGV, possible program stack overflow occurred
```

The following explanatory text also appears:

```
Program requirements exceed current stacksize resource limit.
```

175[1] **severe(175): DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8**

FOR$IOS_SHORTDATEARG. The number of characters associated with the DATE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least eight characters in length. Verify that the TIME and ZONE arguments also meet their minimum lengths.

176[1] **severe(176): TIME argument to DATE_AND_TIME is too short (LEN=n), required LEN=10**

FOR$IOS_SHORTTIMEARG. The number of characters associated with the TIME argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least ten characters in length. Verify that the DATE and ZONE arguments also meet their minimum lengths.

177[1] **severe(177): ZONE argument to DATE_AND_TIME is too short (LEN=n), required LEN=5**

FOR$IOS_SHORTZONEARG. The number of characters associated with the ZONE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least five characters in length. Verify that the DATE and TIME arguments also meet their minimum lengths.

178[1] **severe(178): Divide by zero**

FOR$IOS_DIV. A floating-point or integer divide-by-zero exception occurred.

179[1] **severe(179): Cannot allocate array - overflow on array size calculation**

FOR$IOS_ARRSIZEOVF. An attempt to dynamically allocate storage for an array failed because the required storage size exceeds addressable memory.

NOTE
This error can be returned by STAT in an ALLOCATE statement.

182[1] **severe (182): floating invalid - possible uninitialized real/complex variable.**

FOR$ IOS_FLTINV_UNINIT. An invalid floating-point operation failed invalid – likely caused by an uninitialized real/complex variable.

183 **warning (183): FASTMEM allocation is requested but the libmemkind library is not linked in, so using the default allocator.**

FOR$IOS_NOLIBMEMKINDWARN. An allocation requested FASTMEM but the libmemkind library is not linked into the executable, so memory will be allocated from the default memory allocator for that platform.

184 **severe (184): FASTMEM allocation is requested but the libmemkind library is not linked into the executable.**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

FOR$IOS_NOLIBMEMKINDWARN. An allocation request for FASTMEM failed because the libmemkind library is not linked into the executable.

NOTE
This error can be returned by STAT in an ALLOCATE statement.

185 **warning (185): FASTMEM allocation is requested but HBW memory is not available, so using the default allocator.**

FOR$IOS_NOFASTMEMWARN. An allocation requested FASTMEM but HBW memory is not available on the node, so memory will be allocated from the default memory allocator for that platform.

186 **severe (186): FASTMEM allocation is requested but HBW memory is not available on this node.**

FOR$IOS_NOLIBMEMKINDWARN. An allocation request for FASTMEM failed because HBW memory is not available on this node.

NOTE
This error can be returned by STAT in an ALLOCATE statement.

188 **severe (188): An assignment was made from an object of one size to an object of a different size that cannot be deallocated.**

FOR$ IOS_INCOMPATIBLE_SIZES. An assignment was made from an object of one size to an object of a different size that cannot be deallocated.

NOTE
This error can be returned by STAT in an ALLOCATE statement.

189 **severe (189): LHS and RHS of an assignment statement have incompatible types.**

FOR$ IOS_INCOMPATIBLE_TYPES. The left-hand side (LHS) of an assignment statement is not type compatible with the right-hand side (RHS) of the assignment statement.

NOTE
This error can be returned by STAT in an ALLOCATE statement.

190 **severe (190): For allocate(source=), source needs to be allocated.**

FOR$ IOS_ALLOC_INVSOURCE. For allocate(source=), if source is a pointer then it should be associated with a target. If it is allocatable, it should be allocated.

NOTE
This error can be returned by STAT in an ALLOCATE statement.

194[1] **severe (194): Run-Time Check Failure. The variable \'%s\' is being used in \'%s\' without being defined.**

FOR$ IOS_RTC_UNINIT_USE_SRC. The named variable in the named source file is being used without first being initialized.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
256	**severe (256): Unformatted I/O to unit open for formatted transfers** FOR$IOS_UNFIO_FMT. Attempted unformatted I/O to a unit where the OPEN statement (FORM specifier) indicated the file was formatted. Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains unformatted data, specify FORM='UNFORMATTED' in the OPEN statement. Otherwise, if appropriate, use formatted I/O (such as list-directed or namelist I/O).
257	**severe (257): Formatted I/O to unit open for unformatted transfers** FOR$IOS_FMTIO_UNF. Attempted formatted I/O (such as list-directed or namelist I/O) to a unit where the OPEN statement indicated the file was unformatted (FORM specifier). Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains formatted data, specify FORM='FORMATTED' in the OPEN statement. Otherwise, if appropriate, use unformatted I/O.
259	**severe (259): Sequential-access I/O to unit open for direct access** FOR$IOS_SEQIO_DIR. The OPEN statement for this unit number specified direct access and the I/O statement specifies sequential access. Check the OPEN statement and make sure the I/O statement uses the correct unit number and type of access.
264	**severe (264): operation requires file to be on disk or tape** FOR$IOS_OPEREQDIS. Attempted to use a BACKSPACE statement on such devices as a terminal.
265	**severe (265): operation requires sequential file organization and access** FOR$IOS_OPEREQSEQ. Attempted to use a BACKSPACE statement on a file whose organization was not sequential or whose access was not sequential. A BACKSPACE statement can only be used for sequential files opened for sequential access.
266[1]	**error (266): Fortran abort routine called** FOR$IOS_PROABOUSE. The program called the abort routine to terminate itself.
268[1]	**severe (268): End of record during read** FOR$IOS_ENDRECDUR. An end-of-record condition was encountered during execution of a non-advancing I/O READ statement that did not specify the EOR branch specifier.
296[1]	**info(296): nn floating inexact traps** FOR$IOS_FLOINEEXC. The total number of floating-point inexact data traps encountered during program execution was *nn*. This summary message appears at program completion.
297[1]	**info (297): nn floating invalid traps** FOR$IOS_FLOINVEXC. The total number of floating-point invalid data traps encountered during program execution was *nn*. This summary message appears at program completion.
298[1]	**info (298): nn floating overflow traps** FOR$IOS_FLOOVFEXC. The total number of floating-point overflow traps encountered during program execution was *nn*. This summary message appears at program completion.
299[1]	**info (299): nn floating divide-by-zero traps** FOR$IOS_FLODIV0EXC. The total number of floating-point divide-by-zero traps encountered during program execution was *nn*. This summary message appears at program completion.
300[1]	**info (300): nn floating underflow traps** FOR$IOS_FLOUNDEXC. The total number of floating-point underflow traps encountered during program execution was *nn*. This summary message appears at program completion.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
408	**Severe (408): fort: (19): Dummy character variable \'%s\' has length 1 which is greater then actual variable length**
529	**Info (529): Attempt to close a unit which was not open.** FOR$IOS_SYNERRNAM. The unit number specified on a CLOSE statement is not currently open.
540	**severe (540): Array or substring subscript expression out of range** FOR$IOS_F6096. An expression used to index an array was smaller than the lower dimension bound or larger than the upper dimension bound.
541	**severe (541): CHARACTER substring expression out of range** FOR$IOS_F6097. An expression used to index a character substring was illegal.
542	**severe (542): Label not found in assigned GOTO list** FOR$IOS_F6098. The label assigned to the integer-variable name was not specified in the label list of the assigned GOTO statement.
545	**severe (545): Invalid INTEGER** FOR$IOS_F6101. Either an illegal character appeared as part of an integer, or a numeric character larger than the radix was used in an alternate radix specifier.
546	**severe (546): REAL indefinite (uninitialized or previous error)** FOR$IOS_F6102. An invalid real number was read from a file, an internal variable, or the console. This can happen if an invalid number is generated by passing an illegal argument to an intrinsic function -- for example, SQRT(-1) or ASIN(2). If the invalid result is written and then later read, the error will be generated.
547	**severe (547): Invalid REAL** FOR$IOS_F103. An illegal character appeared as part of a real number.
548	**severe (548): REAL math overflow** FOR$IOS_F6104. A real value was too large. Floating-point overflows in either direct or emulated mode generate NaN (Not-A-Number) exceptions, which appear in the output field as asterisks (*) or the letters NAN.
551	**severe (551): Formatted I/O not consistent with OPEN options** FOR$IOS_F6200. The program tried to perform formatted I/O on a unit opened with FORM='UNFORMATTED' or FORM='BINARY'.
552	**severe (552): List-directed I/O not consistent with OPEN options** FOR$IOS_F6201. The program tried to perform list-directed I/O on a file that was not opened with FORM='FORMATTED' and ACCESS='SEQUENTIAL'.
553	**severe (553): Terminal I/O not consistent with OPEN options** FOR$IOS_F6202. When a special device such as CON, LPT1, or PRN is opened in an OPEN statement, its access must be sequential and its format must be either formatted or binary. By default ACCESS='SEQUENTIAL' and FORM='FORMATTED' in OPEN statements. To generate this error the device's OPEN statement must contain an option not appropriate for a terminal device, such as ACCESS='DIRECT' or FORM='UNFORMATTED'.
554	**severe (554): Direct I/O not consistent with OPEN options** FOR$IOS_F6203. A REC=*option* was included in a statement that transferred data to a file that was opened with the ACCESS='SEQUENTIAL' option.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
555	**severe (555): Unformatted I/O not consistent with OPEN options** FOR$IOS_F6204. If a file is opened with FORM='FORMATTED', unformatted or binary data transfer is prohibited.
556	**severe (556): A edit descriptor expected for CHARACTER** FOR$IOS_F6205. The A edit descriptor was not specified when a character data item was read or written using formatted I/O.
557	**severe (557): E, F, D, or G edit descriptor expected for REAL** FOR$IOS_F6206. The E, F, D, or G edit descriptor was not specified when a real data item was read or written using formatted I/O.
558	**severe (558): I edit descriptor expected for INTEGER** FOR$IOS_F6207. The I edit descriptor was not specified when an integer data item was read or written using formatted I/O.
559	**severe (559): L edit descriptor expected for LOGICAL** FOR$IOS_F6208. The L edit descriptor was not specified when a logical data item was read or written using formatted I/O.
560	**severe (560): File already open: parameter mismatch** FOR$IOS_F6209. An OPEN statement specified a connection between a unit and a filename that was already in effect. In this case, only the BLANK= specifier can have a different setting.
561	**severe (561): Namelist I/O not consistent with OPEN options** FOR$IOS_F6210. The program tried to perform namelist I/O on a file that was not opened with FORM='FORMATTED' and ACCESS='SEQUENTIAL'.
562	**severe (562): IOFOCUS option illegal with non-window unit** FOR$IOS_F6211. IOFOCUS was specified in an OPEN or INQUIRE statement for a non-window unit. The IOFOCUS option can only be used when the unit opened or inquired about is a QuickWin child window.
563	**severe (563): IOFOCUS option illegal without QuickWin** FOR$IOS_F6212. IOFOCUS was specified in an OPEN or INQUIRE statement for a non-QuickWin application. The IOFOCUS option can only be used when the unit opened or inquired about is a QuickWin child window.
564	**severe (564): TITLE illegal with non-window unit** FOR$IOS_F6213. TITLE was specified in an OPEN or INQUIRE statement for a non-window unit. The TITLE option can only be used when the unit opened or inquired about is a QuickWin child window.
565	**severe (565): TITLE illegal without QuickWin** FOR$IOS_F6214. TITLE was specified in an OPEN or INQUIRE statement for a non-QuickWin application. The TITLE option can only be used when the unit opened or inquired about is a QuickWin child window.
566	**severe (566): KEEP illegal for scratch file** FOR$IOS_F6300. STATUS='KEEP' was specified for a scratch file. This is illegal because scratch files are automatically deleted at program termination.
567	**severe (567): SCRATCH illegal for named file**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6301. STATUS='SCRATCH' should not be used in a statement that includes a filename.
568	**severe (568): Multiple radix specifiers** FOR$IOS_F6302. More than one alternate radix for numeric I/O was specified. F6302 can indicate an error in spacing or a mismatched format for data of different radices.
569	**severe (569): Illegal radix specifier** FOR$IOS_F6303. A radix specifier was not between 2 and 36, inclusive. Alternate radix constants must be of the form n#ddd... where n is a radix from 2 to 36 inclusive and ddd... are digits with values less than the radix. For example, 3#12 and 34#7AX are valid constants with valid radix specifiers. 245#7A and 39#12 do not have valid radix specifiers and generate error 569 if input.
570	**severe (570): Illegal STATUS value** FOR$IOS_F6304. An illegal value was used with the STATUS option. STATUS accepts the following values: • 'KEEP' or 'DELETE'' when used with CLOSE statements • 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN' when used with OPEN statements
571	**severe (571): Illegal MODE value** FOR$IOS_F6305. An illegal value was used with the MODE option. MODE accepts the values 'READ', 'WRITE', or 'READWRITE'.
572	**severe (572): Illegal ACCESS value** FOR$IOS_F6306. An illegal value was used with the ACCESS option. ACCESS accepts the values 'SEQUENTIAL' and 'DIRECT'.
573	**severe (573): Illegal BLANK value** FOR$IOS_F6307. An illegal value was used with the BLANK option. BLANK accepts the values 'NULL' and 'ZERO'.
574	**severe (574): Illegal FORM value** FOR$IOS_F6308. An illegal value was used with the FORM option. FORM accepts the following values: 'FORMATTED', 'UNFORMATTED', and 'BINARY'.
575	**severe (575): Illegal SHARE value** FOR$IOS_F6309. An illegal value was used with the SHARE option. SHARE accepts the values 'COMPAT', 'DENYRW', 'DENYWR', 'DENYRD', and 'DENYNONE'.
577	**severe (577): Illegal record number** FOR$IOS_F6311. An invalid number was specified as the record number for a direct-access file. The first valid record number for direct-access files is 1.
578	**severe (578): No unit number associated with *** FOR$IOS_F6312. In an INQUIRE statement, the NUMBER option was specified for the file associated with * (console).
580	**severe (580): Illegal unit number** FOR$IOS_F6314. An illegal unit number was specified.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	Legal unit numbers can range from 0 through 2**31-1, inclusive.
581	**severe (581): Illegal RECL value** FOR$IOS_F6315. A negative or zero record length was specified for a direct file. The smallest valid record length for direct files is 1.
582	**severe (582): Array already allocated** FOR$IOS_F6316. The program attempted to ALLOCATE an already allocated array.
583	**severe (583): Array size zero or negative** FOR$IOS_F6317. The size specified for an array in an ALLOCATE statement must be greater than zero.
584	**severe (584): Non-HUGE array exceeds 64K** FOR$IOS_F6318.
585	**severe (585): Array not allocated** FOR$IOS_F6319. The program attempted to DEALLOCATE an array that was never allocated.
586	**severe (586): BACKSPACE illegal on terminal device** FOR$IOS_F6400. A BACKSPACE statement specified a unit connected to a terminal device such as a terminal or printer.
587	**severe (587): EOF illegal on terminal device** FOR$IOS_F6401. An EOF intrinsic function specified a unit connected to a terminal device such as a terminal or printer.
588	**severe (588): ENDFILE illegal on terminal device** FOR$IOS_F6402. An ENDFILE statement specified a unit connected to a terminal device such as a terminal or printer.
589	**severe (589): REWIND illegal on terminal device** FOR$IOS_F6403. A REWIND statement specified a unit connected to a terminal device such as a terminal or printer.
590	**severe (590): DELETE illegal for read-only file** FOR$IOS_F6404. A CLOSE statement specified STATUS='DELETE' for a read-only file.
591	**severe (591): External I/O illegal beyond end of file** FOR$IOS_F6405. The program tried to access a file after executing an ENDFILE statement or after it encountered the end-of-file record during a read operation. A BACKSPACE, REWIND, or OPEN statement must be used to reposition the file before execution of any I/O statement that transfers data.
592	**severe (592): Truncation error: file closed** FOR$IOS_F6406.
593	**severe (593): Terminal buffer overflow** FOR$IOS_F6407. More than 131 characters were input to a record of a unit connected to the terminal (keyboard). Note that the operating system may impose additional limits on the number of characters that can be input to the terminal in a single record.
594	**severe (594): Comma delimiter disabled after left repositioning**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

FOR$IOS_F6408. If you have record lengths that exceed the buffer size associated with the record, (for instance, the record is a file with the buffer set by BLOCKSIZE in the OPEN statement), either you should not do left tabbing within the record, or you should not use commas as field delimiters. This is because commas are disabled as input field delimiters if left tabbing leaves the record positioned in a previous buffer.

For example, consider you have a file `LONG.DAT` that is one continuous record with data fields separated by commas. You then set the buffer associated with the file to 512 bytes, read more than one buffer size of data, tab left to data in the previous buffer, and attempt to read further data, as follows:

```
    INTEGER value(300)
   OPEN (1, FILE = 'LONG.DAT', BLOCKSIZE = 512)s
   READ (1, 100) (value(i), i = 1, 300)s
100 FORMAT (290I2,TL50,10I2)
```

In this case, error 594 occurs.

599 **severe (599): File already connected to a different unit**

FOR$IOS_F6413. The program tried to connect an already connected file to a new unit.

A file can be connected to only one unit at a time.

600 **severe (600): Access not allowed**

FOR$IOS_F6414.

This error can be caused by one of the following:

- The filename specified in an OPEN statement was a directory.
- An OPEN statement tried to open a read-only file for writing.
- The file was opened with SHARE='DENYRW' by another process.

601 **severe (601): File already exists**

FOR$IOS_F6415. An OPEN statement specified STATUS='NEW' for a file that already exists.

602 **severe (602): File not found**

FOR$IOS_F6416. An OPEN statement specified STATUS='OLD' for a specified file or a directory path that does not exist.

603 **severe (603): Too many open files**

FOR$IOS_F6417. The program exceeded the number of open files the operating system allows.

604 **severe (604): Too many units connected**

FOR$IOS_F6418. The program exceeded the number of units that can be connected at one time. Units are connected with the OPEN statement.

605 **severe (605): Illegal structure for unformatted file**

FOR$IOS_F6419. The file was opened with FORM='UNFORMATTED' and ACCESS='SEQUENTIAL', but its internal physical structure was incorrect or inconsistent. Possible causes: the file was created in another mode or by a non-Fortran program.

606 **severe (606): Unknown unit number**

FOR$IOS_F6420. A statement such as BACKSPACE or ENDFILE specified a file that had not yet been opened. (The READ and WRITE statements do not cause this problem because they prompt you for a file if the file has not been opened yet.)

607 **severe (607): File read-only or locked against writing**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6421. The program tried to transfer data to a file that was opened in read-only mode or locked against writing. The error message may indicate a CLOSE error when the fault is actually coming from WRITE. This is because the error is not discovered until the program tries to write buffered data when it closes the file.
608	**severe (608): No space left on device** FOR$IOS_F6422. The program tried to transfer data to a file residing on a device (such as a hard disk) that was out of storage space.
609	**severe (609): Too many threads** FOR$IOS_F6423. Too many threads were active simultaneously. At most, 32 threads can be active at one time. Close any unnecessary processes or child windows within your application.
610	**severe (610): Invalid argument** FOR$IOS_F6424.
611	**severe (611): BACKSPACE illegal for SEQUENTIAL write-only files** FOR$IOS_F6425. The BACKSPACE statement is not allowed in files opened with MODE='WRITE' (write-only status) because BACKSPACE requires reading the previous record in the file to provide positioning. Resolve the problem by giving the file read access or by avoiding the BACKSPACE statement. Note that the REWIND statement is valid for files opened as write-only.
612	**severe (612): File not open for reading or file locked** FOR$IOS_F6500. The program tried to read from a file that was not opened for reading or was locked.
613	**severe (613): End of file encountered** FOR$IOS_F6501. The program tried to read more data than the file contains.
614	**severe (614): Positive integer expected in repeat field** FOR$IOS_F6502. When the *i**c* form is used in list-directed input, the *i* must be a positive integer. For example, consider the following statement: `READ(*,*) a, b` Input 2*56.7 is accepted, but input 2.1*56.7 returns error 614.
615	**severe (615): Multiple repeat field** FOR$IOS_F6503. In list-directed input of the form *i**c*, an extra repeat field was used. For example, consider the following: `READ(*,*) I, J, K` Input of 2*1*3 returns this error. The 2*1 means send two values, each 1; the *3 is an error.
616	**severe (616): Invalid number in input** FOR$IOS_F6504. Some of the values in a list-directed input record were not numeric. For example, consider the following: `READ(*,*) I, J` The preceding statement would cause this error if the input were: 123 'abc'.
617	**severe (617): Invalid string in input**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6505. A string item was not enclosed in single quotation marks.
618	**severe (618): Comma missing in COMPLEX input**
	FOR$IOS_F6506. When using list-directed input, the real and imaginary components of a complex number were not separated by a comma.
619	**severe (619): T or F expected in LOGICAL read**
	FOR$IOS_F6507. The wrong format was used for the input field for logical data. The input field for logical data consists of optional blanks, followed by an optional decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field, so that .TRUE. and .FALSE. are acceptable input forms.
620	**severe (620): Too many bytes read from unformatted record**
	FOR$IOS_F6508. The program tried to read more data from an unformatted file than the current record contained. If the program was reading from an unformatted direct file, it tried to read more than the fixed record length as specified by the RECL option. If the program was reading from an unformatted sequential file, it tried to read more data than was written to the record.
621	**severe (621): H or apostrophe edit descriptor illegal on input**
	FOR$IOS_F6509. Hollerith (H) or apostrophe edit descriptors were encountered in a format used by a READ statement.
622	**severe (622): Illegal character in hexadecimal input**
	FOR$IOS_F6510. The input field contained a character that was not hexadecimal. Legal hexadecimal characters are 0 - 9 and A - F.
623	**severe (623): Variable name not found**
	FOR$IOS_F6511. A name encountered on input from a namelist record is not declared in the corresponding NAMELIST statement.
624	**severe (624): Invalid NAMELIST input format**
	FOR$IOS_F6512. The input record is not in the correct form for NAMELIST input.
625	**severe (625): Wrong number of array dimensions**
	FOR$IOS_F6513. In NAMELIST input, an array name was qualified with a different number of subscripts than its declaration, or a non-array name was qualified.
626	**severe (626): Array subscript exceeds allocated area**
	FOR$IOS_F6514. A subscript was specified in NAMELIST input which exceeded the declared dimensions of the array.
627	**severe (627): Invalid subrange in NAMELIST input**
	FOR$IOS_F6515. A character item in namelist input was qualified with a subrange that did not meet the requirement that 1 <= e1 <= e2 <= len (where "len" is the length of the character item, "e1" is the leftmost position of the substring, and "e2" is the rightmost position of the substring).
628	**severe (628): Substring range specified on non-CHARACTER item**
	FOR$IOS_F6516. A non-CHARACTER item in namelist input was qualified with a substring range.
629	**severe (629): Internal file overflow**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6600. The program either overflowed an internal-file record or tried to write to a record beyond the end of an internal file.
630	**severe (630): Direct record overflow**
	FOR$IOS_F6601. The program tried to write more than the number of bytes specified in the RECL option to an individual record of a direct-access file.
631	**severe (631):Numeric field bigger than record size**
	FOR$IOS_F6602. The program tried to write a non-CHARACTER item across a record boundary in list-directed or namelist output. Only character constants can cross record boundaries.
632	**severe (632): Heap space limit exceeded**
	FOR$IOS_F6700. The program ran out of heap space. The ALLOCATE statement and various internal functions allocate memory from the heap. This error will be generated when the last of the heap space is used up.
633	**severe (633): Scratch file name limit exceeded**
	FOR$IOS_F6701. The program exhausted the template used to generate unique scratch-file names. The maximum number of scratch files that can be open at one time is 26.
634	**severe (634): D field exceeds W field in ES edit descriptor**
	FOR$IOS_F6970. The specified decimal length `d` exceeds the specified total field width `w` in an ES edit descriptor.
635	**severe (635): D field exceeds W field in EN edit descriptor**
	FOR$IOS_F6971. The specified decimal length `d` exceeds the specified total field width `w` in an EN edit descriptor.
636	**severe (636): Exponent of 0 not allowed in format**
	FOR$IOS_F6972.
637	**severe (637): Integer expected in format**

	FOR$IOS_F6980. An edit descriptor lacked a required integer value. For example, consider the following:
	`WRITE(*, 100) I, J`
	`100 FORMAT (I2, TL, I2)`
	The preceding code will cause this error because an integer is expected after TL.

638	**severe (638): Initial left parenthesis expected in format**
	FOR$IOS_F6981. A format did not begin with a left parenthesis (( ).
639	**severe (639): Positive integer expected in format**
	FOR$IOS_F6982. A zero or negative integer value was used in a format. Negative integer values can appear only with the P edit descriptor. Integer values of 0 can appear only in the `d` and `m` fields of numeric edit descriptors.
640	**severe (640): Repeat count on nonrepeatable descriptor**
	FOR$IOS_F6983. One or more BN, BZ, S, SS, SP, T, TL, TR, /, $, :, or apostrophe (') edit descriptors had repeat counts associated with them.
641	**severe (641): Integer expected preceding H, X, or P edit descriptor**
	FOR$IOS_F6984. An integer did not precede a (nonrepeatable) H, X, or P edit descriptor.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

The correct formats for these descriptors are *n*H, *n*X, and *k*P, respectively, where *n* is a positive integer and *k* is an optionally signed integer.

642 **severe (642): N or Z expected after B in format**

FOR$IOS_F6985. To control interpretation of embedded and trailing blanks within numeric input fields, you must specify BN (to ignore them) or BZ (to interpret them as zeros).

643 **severe (643): Format nesting limit exceeded**

FOR$IOS_F6986. More than sixteen sets of parentheses were nested inside the main level of parentheses in a format.

644 **severe (644): '.' expected in format**

FOR$IOS_F6987. No period appeared between the w and d fields of a D, E, F, or G edit descriptor.

645 **severe (645): Unexpected end of format**

FOR$IOS_F6988. An incomplete format was used.

Improperly matched parentheses, an unfinished Hollerith (H) descriptor, or another incomplete descriptor specification can cause this error.

646 **severe (646): Unexpected character in format**

FOR$IOS_F6989. A character that cannot be interpreted as part of a valid edit descriptor was used in a format. For example, consider the following:

```
      WRITE(*, 100) I, J
  100 FORMAT (I2, TL4.5, I2)
```

The code will generate this error because TL4.5 is not a valid edit descriptor. An integer must follow TL.

647 **severe (647): M field exceeds W field in I edit descriptor**

FOR$IOS_F6990. In syntax Iw.m, the value of m cannot exceed the value of w.

648 **severe (648): Integer out of range in format**

FOR$IOS_F6991. An integer value specified in an edit descriptor was too large to represent as a 4-byte integer.

649 **severe (649): format not set by ASSIGN**

FOR$IOS_F6992. The format specifier in a READ, WRITE, or PRINT statement was an integer variable, but an ASSIGN statement did not properly assign it the statement label of a FORMAT statement in the same program unit.

650 **severe (650): Separator expected in format**

FOR$IOS_F6993. Within format specifications, edit descriptors must be separated by commas or slashes (/).

651 **severe (651): %c or $: nonstandard edit descriptor in format**

FOR$IOS_F6994.

652 **severe (652): Z: nonstandard edit descriptor in format**

FOR$IOS_F6995. Z is not a standard edit descriptor in format.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	If you want to transfer hexadecimal values, you must use the edit descriptor form Z`w[.m]`, where `w` is the field width and `m` is the minimum number of digits that must be in the field (including leading zeros).
653	**severe (653): DOS graphics not supported under Windows NT** FOR$IOS_F6996.
654	**severe (654): Graphics error** FOR$IOS_F6997. An OPEN statement in which IOFOCUS was TRUE, either explicitly or by default, failed because the new window could not receive focus. The window handle may be invalid, or closed, or there may be a memory resource problem.
655	**severe (655): Using QuickWin is illegal in console application** FOR$IOS_F6998. A call to QuickWin from a console application was encountered during execution.
656	**severe (656): Illegal 'ADVANCE' value** FOR$IOS_F6999. The ADVANCE option can only take the values 'YES' and 'NO'. ADVANCE='YES' is the default. ADVANCE is a READ statement option.
657	**severe (657): DIM argument to SIZE out of range** FOR$IOS_F6702. The argument specified for DIM must be greater than or equal to 1, and less than or equal to the number of dimensions in the specified array. Consider the following: `i = SIZE (array, DIM = dim)` In this case, `1 <= dim <= n`, where *n* is the number of dimensions in array.
658	**severe (657): Undefined POINTER used as argument to ASSOCIATED function** FOR$IOS_F6703. A POINTER used as an argument to the ASSOCIATED function must be defined; that is, assigned to a target, allocated, or nullified.
659	**severe (659): Reference to uninitialized POINTER** FOR$IOS_F6704. Except in an assignment statement, a pointer must not be referenced until it has been initialized: assigned to a target, allocated or nullified.
660	**severe (660): Reference to POINTER which is not associated** FOR$IOS_F6705. Except in an assignment statement and certain procedure references, a pointer must not be referenced until it has been associated: either assigned to a target or allocated.
661	**severe (661): Reference to uninitialized POINTER 'pointer'** FOR$IOS_F6706. Except in an assignment statement, a pointer must not be referenced until it has been initialized: assigned to a target, allocated or nullified.
662	**severe (662): reference to POINTER 'pointer' which is not associated** FOR$IOS_F6707. Except in an assignment statement and certain procedure references, a pointer must not be referenced until it has been associated: either assigned to a target or allocated.
663	**severe (663): Out of range: substring starting position 'pos' is less than 1** FOR$IOS_F6708. A substring starting position must be a positive integer variable or expression that indicates a position in the string: at least 1 and no greater than the length of the string.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
664	**severe (664): Out of range: substring ending position 'pos' is greater than string length 'len'** FOR$IOS_F6709. A substring ending position must be a positive integer variable or expression that indicates a position in the string: at least 1 and no greater than the length of the string.
665	**severe (665): Subscript 'n' of 'str' (value 'val') is out of range ('first:last')** FOR$IOS_F6710. The subscript for a substring within a string is not a valid string position: at least 1 and no greater than the length of the string.
666	**severe (666): Subscript 'n' of 'str' (value 'val') is out of range ('first:*')** FOR$IOS_F6711. The subscript for a substring within a string is not a valid string position: at least 1 and no greater than the length of the string.
667	**severe (667): VECTOR argument to PACK has incompatible character length** FOR$IOS_F6712. The character length of elements in the VECTOR argument to PACK is not the same as the character length of elements in the array to be packed.
668	**severe (668): VECTOR argument to PACK is too small** FOR$IOS_F6713. The VECTOR argument to PACK must have at least as many elements as there are true elements in MASK (the array that controls packing).
669	**severe (669): SOURCE and PAD arguments to RESHAPE have different character lengths** FOR$IOS_F6714. The character length of elements in the SOURCE and PAD arguments to PACK must be the same.
670	**severe (670): Element 'n' of SHAPE argument to RESHAPE is negative** FOR$IOS_F6715. The SHAPE vector specifies the shape of the reshaped array. Since an array cannot have a negative dimension, SHAPE cannot have a negative element.
671	**severe (671): SOURCE too small for specified SHAPE in RESHAPE, and no PAD** FOR$IOS_F6716. If there is no PAD array, the SOURCE argument to RESHAPE must have enough elements to make an array of the shape specified by SHAPE.
672	**severe (672): Out of memory** FOR$IOS_F6717. The system ran out of memory while trying to make the array specified by RESHAPE. If possible, reset your virtual memory size through the Windows* Control Panel, or close unnecessary applications and deallocate all allocated arrays that are no longer needed.
673	**severe (673): SHAPE and ORDER arguments to RESHAPE have different sizes ('size1' and 'size2')** FOR$IOS_F6718. ORDER specifies the order of the array dimensions given in SHAPE, and they must be vectors of the same size.
674	**severe (674): Element 'n' of ORDER argument to RESHAPE is out of range ('range')** FOR$IOS_F6719. The ORDER argument specifies the order of the dimensions of the reshaped array, and it must be a permuted list of (1, 2, ..., *n*) where *n* is the highest dimension in the reshaped array.
675	**severe (675): Value 'val' occurs twice in ORDER argument to RESHAPE** FOR$IOS_F6720. The ORDER vector specifies the order of the dimensions of the reshaped array, and it must be a permuted list of (1, 2, ..., *n*) where *n* is the highest dimension in the reshaped array. No dimension can occur twice.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
676	**severe (676): Impossible nextelt overflow in RESHAPE** FOR$IOS_F6721.
677	**severe (677): Invalid value 'dim' for argument DIM for SPREAD of rank 'rank' source** FOR$IOS_F6722. The argument specified for DIM to SPREAD must be greater than or equal to 1, and less than or equal to one larger than the number of dimensions (rank) of SOURCE. Consider the following statement: `result = SPREAD (SOURCE= array, DIM = dim, NCOPIES = k)` In this case, 1 <= dim <= *n*+ 1, where *n*is the number of dimensions in array.
678	**severe (678): Complex zero raised to power zero** FOR$IOS_F6723. Zero of any type (complex, real, or integer) cannot be raised to zero power.
679	**severe (679): Complex zero raised to negative power** FOR$IOS_F6724. Zero of any type (complex, real, or integer) cannot be raised to a negative power. Raising to a negative power inverts the operand.
680	**severe (680): Impossible error in NAMELIST input** FOR$IOS_F6725.
681	**severe (681):DIM argument to CSHIFT ('dim') is out of range** FOR$IOS_F6726. The optional argument DIM specifies the dimension along which to perform the circular shift, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array to be shifted. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in the array to be shifted.
682	**severe (682): DIM argument ('dim') to CSHIFT is out of range (1:'n')** FOR$IOS_F6727. The optional argument DIM specifies the dimension along which to perform the circular shift, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array to be shifted. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in the array to be shifted.
683	**severe (683): Shape mismatch (dimension 'dim') between ARRAY and SHIFT in CSHIFT** FOR$IOS_F6728. The SHIFT argument to CSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the SHIFT must conform to the shape of the array being shifted in every dimension except the one being shifted along.
684	**severe (684): Internal error - bad arguments to CSHIFT_CA** FOR$IOS_F6729.
685	**severe (685): Internal error - bad arguments to CSHIFT_CAA** FOR$IOS_F6730.
686	**severe (686): DATE argument to DATE_AND_TIME is too short (LEN='len')** FOR$IOS_F6731. The character DATE argument must have a length of at least eight to contain the complete value.
687	**severe (687): TIME argument to DATE_AND_TIME is too short (LEN='len')** FOR$IOS_F6732. The character TIME argument must have a length of at least ten to contain the complete value.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
688	**severe (688): ZONE argument to DATE_AND_TIME is too short (LEN='len')** FOR$IOS_F6733. The character ZONE argument must have a length of at least five to contain the complete value.
689	**severe (689): VALUES argument to DATE_AND_TIME is too small ('size' elements)** FOR$IOS_F6734. The integer VALUES argument must be a one-dimensional array with a size of at least eight to hold all returned values.
690	**severe (690): Out of range: DIM argument to COUNT has value 'dim'** FOR$IOS_F6735. The optional argument DIM specifies the dimension along which to count true elements of MASK, and must be greater than or equal to 1 and less than or equal to the number of dimensions in MASK. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in MASK.
691	**severe (691): Out of range: DIM argument to COUNT has value 'dim' with MASK of rank 'rank'** FOR$IOS_F6736. The optional argument DIM specifies the dimension along which to count true elements of MASK, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in MASK. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in MASK.
692	**severe (692): Out of range: DIM argument to PRODUCT has value 'dim'** FOR$IOS_F6737. The optional argument DIM specifies the dimension along which to compute the product of elements in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array holding the elements to be multiplied.
693	**severe (693): Out of range: DIM argument to PRODUCT has value 'dim' with ARRAY of rank 'rank'** FOR$IOS_F6738. The optional argument DIM specifies the dimension along which to compute the product of elements in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) of the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array holding the elements to be multiplied.
694	**severe (694): Out of range: DIM argument to SUM has value 'dim' with ARRAY of rank 'rank'** FOR$IOS_F6739. The optional argument DIM specifies the dimension along which to sum the elements of an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) of the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array holding the elements to be summed.
695	**severe (695): Real zero raised to zero power** FOR$IOS_F6740. Zero of any type (real, complex, or integer) cannot be raised to zero power.
696	**severe (696): Real zero raised to negative power** FOR$IOS_F6741. Zero of any type (real, complex, or integer) cannot be raised to a negative power. Raising to a negative power inverts the operand.
697	**severe (697): Out of range: DIM argument to SUM has value 'dim'** FOR$IOS_F6742. The optional argument DIM specifies the dimension along which to sum the elements of an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array holding the elements to be summed.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
698	**severe (698): DIM argument ('dim') to EOSHIFT is out of range (1:'n')** FOR$IOS_F6743. The optional argument DIM specifies the dimension along which to perform an end-off shift in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array holding the elements to be shifted.
699	**severe (699): Shape mismatch (dimension 'dim') between ARRAY and BOUNDARY in EOSHIFT** FOR$IOS_F6744. The BOUNDARY argument to EOSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the BOUNDARY must conform to the shape of the array being shifted in every dimension except the one being shifted along.
700	**severe (700): DIM argument to EOSHIFT is out of range ('dim')** FOR$IOS_F6745. The optional argument DIM specifies the dimension along which to perform an end-off shift in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array holding the elements to be shifted.
701	**severe (701): Shape mismatch (dimension 'dim') between ARRAY and SHIFT in EOSHIFT** FOR$IOS_F6746. The SHIFT argument to EOSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the SHIFT must conform to the shape of the array being shifted in every dimension except the one being shifted along.
702	**severe (702): BOUNDARY argument to EOSHIFT has wrong LEN ('len1 instead of len2')** FOR$IOS_F6747. The character length of elements in the BOUNDARY argument and in the array being end-off shifted must be the same.
703	**severe (703): BOUNDARY has LEN 'len' instead of 'len' to EOSHIFT** FOR$IOS_F6748.
704	**severe (704): Internal error - bad arguments to EOSHIFT** FOR$IOS_F6749.
705	**severe (705): GETARG: value of argument 'num' is out of range** FOR$IOS_F6750. The value used for the number of the command-line argument to retrieve with GETARG must be 0 or a positive integer. If the number of the argument to be retrieved is greater than the actual number of arguments, blanks are returned, but no error occurs.
706	**severe (706): FLUSH: value of LUNIT 'num' is out of range** FOR$IOS_F6751. The unit number specifying which I/O unit to flush to its associated file must be an integer between 0 and 2**31-1, inclusive. If the unit number is valid, but the unit is not opened, error F6752 is generated.
707	**severe (707): FLUSH: Unit 'n' is not connected** FOR$IOS_F6752. The I/O unit specified to be flushed to its associated file is not connected to a file.
708	**severe (708): Invalid string length ('len') to ICHAR** FOR$IOS_F6753. The character argument to ICHAR must have length of one.
709	**severe (709): Invalid string length ('len') to IACHAR**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6754. The character argument to IACHAR must have length of one.
710	**severe (710): Integer zero raised to negative power**
	FOR$IOS_F6755. Zero of any type (integer, real, or complex) cannot be raised to a negative power. Raising to a negative power inverts the operand.
711	**severe (711): INTEGER zero raised to zero power**
	FOR$IOS_F6756. Zero of any type (integer, real, or complex) cannot be raised to zero power.
712	**severe (712): SIZE argument ('size') to ISHFTC intrinsic out of range**
	FOR$IOS_F6757. The argument SIZE must be positive and must not exceed the bit size of the integer being shifted. The bit size of this integer can be determined with the function BIT_SIZE.
713	**severe (713): SHIFT argument ('shift') to ISHFTC intrinsic out of range**
	FOR$IOS_F6758. The argument SHIFT to ISHFTC must be an integer whose absolute value is less than or equal to the number of bits being shifted: either all bits in the number being shifted or a subset specified by the optional argument SIZE.
714	**severe (714): Out of range: DIM argument to LBOUND has value 'dim'**
	FOR$IOS_F6759. The optional argument DIM specifies the dimension whose lower bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
715	**severe (715): Out of range: DIM argument ('dim') to LBOUND greater than ARRAY rank 'rank'**
	FOR$IOS_F6760. The optional argument DIM specifies the dimension whose lower bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.
716	**severe (716): Out of range: DIM argument to MAXVAL has value 'dim'**
	FOR$IOS_F6761. The optional argument DIM specifies the dimension along which maximum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
717	**severe (717): Out of range: DIM argument to MAXVAL has value 'dim' with ARRAY of rank 'rank'**
	FOR$IOS_F6762. The optional argument DIM specifies the dimension along which maximum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.
718	**severe (718): Cannot allocate temporary array -- out of memory**
	FOR$IOS_F6763. There is not enough memory space to hold a temporary array. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you may need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
719	**severe (719): Attempt to DEALLOCATE part of a larger object**
	FOR$IOS_F6764. An attempt was made to DEALLOCATE a pointer to an array subsection or an element within a derived type. The whole data object must be deallocated; parts cannot be deallocated.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
720	**severe (720): Pointer in DEALLOCATE is ASSOCIATED with an ALLOCATABLE array** FOR$IOS_F6765. Deallocating a pointer associated with an allocatable target is illegal. Instead, deallocate the target the pointer points to, which frees memory and disassociates the pointer.
721	**severe (721): Attempt to DEALLOCATE an object which was not allocated** FOR$IOS_F6766. You cannot deallocate an array unless it has been previously allocated. You cannot deallocate a pointer whose target was not created by allocation. The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated.
722	**severe (722): Cannot ALLOCATE scalar POINTER -- out of memory** FOR$IOS_F6767. There is not enough memory space to allocate the pointer. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you may need to reset your virtual memory size through the Windows* Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
723	**severe (723): DEALLOCATE: object not allocated/associated** FOR$IOS_F6768. You cannot deallocate an array unless it has been previously allocated. You cannot deallocate a pointer whose target was not created by allocation, or a pointer that has undefined association status. The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated.
724	**severe (724): Cannot ALLOCATE POINTER array -- out of memory** FOR$IOS_F6769. There is not enough memory space to allocate the POINTER array. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you may need to reset your virtual memory size through the Windows* Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
725	**severe (725): DEALLOCATE: Array not allocated** FOR$IOS_F6770. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.
726	**severe (726): DEALLOCATE: Character array not allocated** FOR$IOS_F6771. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.
727	**severe (727): Cannot ALLOCATE allocatable array -- out of memory** FOR$IOS_F6772. There is not enough memory space to hold the array. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you may need to reset your virtual memory size through the Windows* Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
728	**severe (728): Cannot allocate automatic object -- out of memory** FOR$IOS_F6773. There is not enough memory space to hold the automatic data object. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you may need to reset your virtual memory size through the Windows* Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

An automatic data object is an object that is declared in a procedure subprogram or interface, is not a dummy argument, and depends on a nonconstant expression. For example:

```
SUBROUTINE  EXAMPLE (N)
  DIMENSION A (N, 5), B(10*N)
```

The arrays `A` and `B` in the example are automatic data objects.

729 **severe (729): DEALLOCATE failure: ALLOCATABLE array is not ALLOCATED**

FOR$IOS_F6774. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.

730 **severe (730): Out of range: DIM argument to MINVAL has value 'dim'**

FOR$IOS_F6775. The optional argument DIM specifies the dimension along which minimum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.

731 **severe (731): Out of range: DIM argument to MINVAL has value 'dim' with ARRAY of rank 'rank'**

FOR$IOS_F6776. The optional argument DIM specifies the dimension along which minimum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.

732 **severe (732): P argument to MOD is double precision zero**

FOR$IOS_F6777. `MOD(A, P)` is computed as *A* - INT(*A*,*P*) * *P*. *P* cannot be zero.

733 **severe (733): P argument to MOD is integer zero**

FOR$IOS_F6778. `MOD(A, P)` is computed as *A* - INT(*A*,*P*) * *P*. *P* cannot be zero.

734 **severe (734): P argument to MOD is real zero**

FOR$IOS_F6779. `MOD(A, P)` is computed as *A* - INT(*A*,*P*) * *P*. *P* cannot be zero.

735 **severe (735): P argument to MODULO is real zero**

FOR$IOS_F6780. `MODULO(A, P)` for real numbers is computed as *A* - FLOOR(*A*,*P*) * P. So, *P* cannot be zero.

736 **severe (736): P argument to MODULO is zero**

FOR$IOS_F6781. In the function, `MODULO(A, P)`, *P* cannot be zero.

737 **severe (737): Argument S to NEAREST is zero**

FOR$IOS_F6782. The sign of the *S* argument to `NEAREST(X, S)` determines the direction of the search for the nearest number to *X*, and cannot be zero.

738 **severe (738): Heap storage exhausted**

FOR$IOS_F6783.

739 **severe (739): PUT argument to RANDOM_SEED is too small**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

FOR$IOS_F6784. The integer array `PUT` must be greater than or equal to the number of integers the processor uses to set the seed value. This number can be determined by calling RANDOM_SEED with the SIZE argument. For example:

```
INTEGER, ALLOCATABLE SEED
CALL RANDOM_SEED( )           ! initialize processor
CALL RANDOM_SEED(SIZE = K)    ! get size of seed
ALLOCATE SEED(K)              ! allocate array
CALL RANDOM_SEED(PUT = SEED)  ! set the seed
```

NOTE
RANDOM_SEED can be called with at most one argument at a time.

740 **severe (740): GET argument to RANDOM_SEED is too small**

FOR$IOS_F6785. The integer array GET must be greater than or equal to the number of integers the processor uses to set the seed value. This number can be determined by calling RANDOM_SEED with the SIZE argument. For example:

```
INTEGER, ALLOCATABLE SEED
CALL RANDOM_SEED( )           ! initialize processor
CALL RANDOM_SEED(SIZE = K)    ! get size of seed
ALLOCATE SEED(K)              ! allocate array
CALL RANDOM_SEED(GET = SEED)  ! get the seed
```

NOTE
RANDOM_SEED can be called with at most one argument at a time.

741 **severe (741): Recursive I/O reference**

FOR$IOS_F6786.

742 **severe (742): Argument to SHAPE intrinsic is not PRESENT**

FOR$IOS_F6787.

743 **severe (743): Out of range: DIM argument to UBOUND had value 'dim'**

FOR$IOS_F6788. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.

744 **severe (744): DIM argument ('dim') to UBOUND greater than ARRAY rank 'rank'**

FOR$IOS_F6789. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.

745 **severe (745): Out of range: UBOUND of assumed-size array with DIM==rank ('rank')**

FOR$IOS_F6790. The optional argument DIM specifies the dimension whose upper bound is to be returned.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	An assumed-size array is a dummy argument in a subroutine or function, and the upper bound of its last dimension is determined by the size of actual array passed to it. Assumed-size arrays have no determined shape, and you cannot use UBOUND to determine the extent of the last dimension. You can use UBOUND to determine the upper bound of one of the fixed dimensions, in which case you must pass the dimension number along with the array name.
746	**severe (746): Out of range: DIM argument ('dim') to UBOUND greater than ARRAY rank** FOR$IOS_F6791. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
747	**severe (747): Shape mismatch: Dimension 'shape' extents are 'ext1' and 'ext2'** FOR$IOS_F6792.
748	**severe (748): Illegal POSITION value** FOR$IOS_F6793. An illegal value was used with the POSITION specifier. POSITION accepts the following values: • 'ASIS' (the default) • 'REWIND' - on Fortran I/O systems, this is the same as 'ASIS' • 'APPEND'
749	**severe (749): Illegal ACTION value** FOR$IOS_F6794. An illegal value was used with the ACTION specifier. ACTIO accepts the following values: • 'READ' • 'WRITE' • 'READWRITE' - the default
750	**severe (750): DELIM= specifier not allowed for an UNFORMATTED file** FOR$IOS_F6795. The DELIM specifier is only allowed for files connected for formatted data transfer. It is used to delimit character constants in list-directed an namelist output.
751	**severe (751): Illegal DELIM value** FOR$IOS_F6796. An illegal value was used with the DELIM specifier. DELIM accepts the following values: • 'APOSTROPHE' • 'QUOTE' • 'NONE' - the default
752	**severe (752): PAD= specifier not allowed for an UNFORMATTED file** FOR$IOS_F6797. The PAD specifier is only allowed for formatted input records. It indicates whether the formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.
753	**severe (753): Illegal PAD= value** FOR$IOS_F6798. An illegal value was used with the PAD specifier. PAD accepts the following values: • 'NO'

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	• 'YES' - the default
754	**severe (754): Illegal CARRIAGECONTROL=value** FOR$IOS_F6799. An illegal value was used with the CARRIAGECONTROL specifier. CARRIAGECONTROL accepts the following values: • 'FORTRAN' - default if the unit is connected to a terminal or console • 'LIST' - default for formatted files • 'NONE' - default for unformatted files
755	**severe (755): SIZE= specifier only allowed with ADVANCE='NO'** FOR$IOS_F6800. The SIZE specifier can only appear in a formatted, sequential READ statement that has the specifier ADVANCE='NO' (indicating non-advancing input).
756	**severe (756): Illegal character in binary input** FOR$IOS_F6801.
757	**severe (757): Illegal character in octal input** FOR$IOS_F6802.
758	**severe (758): End of record encountered** FOR$IOS_F6803.
759	**severe (759): Illegal subscript in namelist input record** FOR$IOS_F6804.
760	**severe (760): Error reported by 'EnumSystemLocales'** FOR$IOS_F6805.
761	**severe (761): Cannot set environment variable %s** FOR$IOS_F6806.
762	**info(762): Error freeing internal data structure.** FOR$IOS_F6807.
772	**severe (772): Image number %d is not a valid image number; valid numbers are 1 to %d** FOR$IOS_F6817. A reference has been made to an image number that is not a valid image number.
774	**severe (774): Image-set array expression must not contain repeated values** FOR$IOS_F6819. A SYNC IMAGES <image-list> statement was attempted. The <image-list> contains duplicate values. This is not permitted by the standard.
775	**severe (775): The lock variable in a LOCK statement is already locked by the executing image** FOR$IOS_F6820. An attempt was made to use a LOCK statement on a lock. However, that lock has already been locked by this image.
776	**severe (776): The lock variable in an UNLOCK statement is not already locked by the executing image** FOR$IOS_F6821. An attempt was made to use an UNLOCK statement on a lock. However, that lock is currently unlocked; it has not been locked by this image.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
777	**severe (777): The lock variable in a LOCK statement is already locked by another image** FOR$IOS_F6822. An attempt was made to use a LOCK statement on a lock. However, that lock is already locked by another image.
778	**severe (778): One of the images to be synchronized with has terminated** FOR$IOS_F6823. The code has tried to synchronize with a set of images using a SYNC statement. One of the images to be synchronized with has already terminated.
779	**info (779): In coarray image %d\n** FOR$IOS_F6824. This is issued as part of the stack traceback from a fatal error. The image listed is the one in which the error happened.
780	**severe (780): The lock variable in an UNLOCK statement is locked by another image** FOR$IOS_F6825. An attempt was made to use a coarray UNLOCK operation on a lock. However, that lock is locked by another image. This may be the result of an error in the program that causes the code to think it has a lock when it does not.
781	**error (781): Only image 1 may read from unit '*'** FOR$IOS_F6826. It is a coarray rule that only image 1 may read from the 'console'.

Footnotes:

[1] Identifies errors not returned by IOSTAT.

Signal Handling (Linux* and macOS* only)

A `signal` is an abnormal event generated by one of various sources, such as:

- A user of a terminal.
- Program or hardware error.
- Request of another program.
- When a process is stopped to allow access to the control terminal.

You can optionally set certain events to issue signals, for example:

- When a process resumes after being stopped
- When the status of a child process changes
- When input is ready at the terminal

Some signals terminate the receiving process if no action is taken (optionally creating a `core` file), while others are simply ignored unless the process has requested otherwise.

Except for certain signals, calling the `signal` or `sigaction` routine allows specified signals to be ignored or causes an interrupt (transfer of control) to the location of a user-written signal handler.

You can establish one of the following actions for a signal with a call to `signal`:

- Ignore the specified signal (identified by number).
- Use the default action for the specified signal, which can reset a previously established action.
- Transfer control from the specified signal to a procedure to receive the signal, specified by name.

Calling the `signal` routine lets you change the action for a signal, such as intercepting an operating system signal and preventing the process from being stopped.

The table below shows the signals that the Intel® Fortran RTL arranges to catch when a program is started:

Signal	Intel® Fortran RTL message
SIGFPE	`Floating-point exception` (number 75)
SIGINT	`Process interrupted` (number 69)
SIGIOT	`IOT trap signal` (number 76)
SIGQUIT	`Process quit` (number 79)
SIGSEGV	`Segmentation fault` (number 174)
SIGTERM	`Process killed` (number 78)

Calling the `signal` routine (specifying the numbers for these signals) results in overwriting the signal-handling facility set up by the Intel® Fortran RTL. The only way to restore the default action is to save the returned value from the first call to `signal`.

When using a debugger, it may be necessary to enter a command to allow the Intel® Fortran RTL to receive and handle the appropriate signals.

Overriding the Default Run-Time Library Exception Handler

To override the default run-time library exception handler on Linux* and macOS*, your application must call `signal` to change the action for the signal of interest.

For example, assume that you want to change the signal action to cause your application to call `abort()` and generate a `core` file.

The following example adds a function named `clear_signal_` to call `signal()` and change the action for the `SIGABRT` signal:

```
#include <signal.h>
  void clear_signal_() { signal (SIGABRT, SIG_DFL); }

  int myabort_() {
    abort();
    return 0;
}
```

A call to the `clear_signal()` local routine must be added to main. Make sure that the call appears before any call to the local `myabort()` routine:

```
program aborts
integer i

call clear_signal()

i = 3
if (i < 5) then
call myabort()
end if
end
```

Advanced Exception and Termination Handling

Advanced Exception and Termination Handling Overview

This section provides information about exception and termination handling issues.

To employ some of the exception handling techniques presented, you will need a C language compiler, which has support for `try-except` constructs or some other form of support for structured exception handling.

See Also

General Default Exception Handling
Default Console Event Handling
General Default Termination Handling
Handlers for the Application (Project) Types
Providing Your Own Exception Termination Handler

General Default Exception Handling

The Intel® Visual Fortran run-time system provides minimal default support for exception handling, console event handling, and application termination rundown.

The default exception handling support provided depends on the type of application (project type) being developed:

- A default exception handler is included with Fortran Console, Fortran QuickWin, and Fortran Standard Graphics applications.
- No default exception handler is included with Fortran Windows or Fortran DLL applications or with C Console applications that contain Fortran procedures.

When you use the default exception handler, all events are enabled. You cannot connect or disconnect individual events from the default Fortran exception handler. However, you can disable the Fortran exception handler, and enable your own. To disable the exception handler, set the `FOR_IGNORE_EXCEPTIONS` environment variable to TRUE.

Most exceptions captured by the Intel® Visual Fortran default handler are dealt with as severe errors. When an exception occurs, the Fortran run-time system will display an error message and traceback output as described in Understanding Run-Time Errors. A run-time error with a severe error (as described in Run-Time Message Display and Format) causes the Intel® Visual Fortran run-time system to terminate the application. Most I/O programming errors are also severe and will terminate an application.

I/O programming errors are not exceptions and cannot be caught by an exception handler. An unhandled I/O programming error is reported through a different mechanism in the Intel® Visual Fortran run-time system. Regardless of the application (project) type, unhandled I/O programming error will generate an error message and traceback output.

See Also

Run-Time Message Display and Format
Understanding Run-Time Errors
project type

Default Console Event Handling

When the Fortran run-time system is initialized, it establishes a default console event handler through the `SetConsoleCtrlHandler` Windows API routine. The default handler will respond to the following event types:

- `CTRL_C_EVENT`
- `CTRL_BREAK_EVENT`
- `CTRL_CLOSE_EVENT`

These event types will result in an orderly program abort with an appropriate diagnostic message. To disable this default call to `SetConsoleCtrlHandler`, set an environment variable named `FOR_DISABLE_CONSOLE_CTRL_HANDLER` to the value YES (Y or y), or TRUE (T or t), or a number greater than zero.

Other console events such as a `CTRL_LOGOFF_EVENT` or `CTRL_SHUTDOWN_EVENT` are not handled by the default handler. The handler is notified of these events but returns FALSE to the operating system. This allows an Intel® Visual Fortran application activated as a Windows* service to continue execution when a user logs off.

See Also

Establishing Console Event Handlers

General Default Termination Handling

When a Fortran Console, Fortran QuickWin, or Fortran Standard Graphics application terminates execution, either by normal termination or due to a severe error or exception, the following actions are taken by the Fortran run-time system:

- Any Open files are closed and the requested `DISPOSITION` operations are performed.
- With a QuickWin application, any open QuickWin windows are closed.
- The C run-time `exit()` routine is called with the status code to return to the operating system. The C run-time `exit()` routine will call the Windows API routine `ExitProcess` to terminate the process. (See `crt0dat.c` in the C run-time sources).

In a Fortran DLL or Fortran Windows* application, any unhandled I/O programming errors will cause the following actions:

- Any Open files are closed and the requested `DISPOSITION` operations are performed.
- The C run-time `exit()` routine is called with the status code to return to the operating system. The C run-time `exit()` routine will call the Windows API routine `ExitProcess` to terminate the process. (See `crt0dat.c` in the C run-time sources.)

Any unhandled exceptions that occur in a Fortran DLL or Fortran Windows application will have application dependent behavior. Since there is no Fortran default handler present, the behavior depends on what you provide for a handler. If you do not explicitly provide a handler, the default mechanisms provided in your main program will determine the behavior. In a Fortran Windows* application, the C run-time system will terminate the application.

Handlers for the Application (Project) Types

To understand how Intel® Fortran handlers are incorporated into your application, and how you might incorporate your own handlers, you should understand how each application type is constructed. This section describes handlers for the various application (project) types.

Fortran Console Applications

Fortran Console applications resemble C applications, with the Intel® Fortran run-time system providing the C `main()` function.

The entry point for a console application is specified as the C library's `mainCRTStartup()` routine (see module `crt0.c` in the C run-time sources). This initializes the C run-time system, wraps the Fortran run-time system `main()` in a try-except construct using the C run-time's exception filter (`_XcptFilter()`), and calls the Fortran run-time system `main()` routine in run-time module `for_main.c`. In simplified form, it looks like this:

```
 mainCRTStartup() {
  C initialization code here
  __try {
       more initialization code here
       mainret = main()      /* calls Fortran run-time main() */
       exit(mainret)
  } __except ( _XcptFilter() )
         { _exit ( GetExceptionCode() ) }
}
```

In the Fortran run-time system, `main()` initializes the Fortran run-time system (if not already initialized), wraps a try-except construct around `MAIN__` (the entry point to the Fortran code) with a filter expression that invokes the Fortran run-time system default handler on exceptions, and calls `MAIN__`. It also wraps a try-finally construct around all of this so run-time system clean up gets done (with for_rtl_finish_) when the program exits. In simplified form, it looks like this:

```
main() {
  __try {
     __try {
           for_rtl_init()
           MAIN__
     } __except ( expression-invoking-fortran-default-handler )
              { }
  } __finally { for_rtl_finish() }
}
```

In the Fortran code, symbol `MAIN__` is the entry point called by the run-time system's main() routine. `MAIN__` has the code to do any further run-time initialization or checks. For example, if the user compiled with the non-default `fpe[:]0` option, there would be a call to FOR_SET_FPE to tell the run-time system how to setup/react to floating-point exceptions.

Fortran QuickWin and Standard Graphics Applications

A Fortran QuickWin (including Fortran Standard Graphics) application is a specialized windows application where Intel® Visual Fortran provides the `WinMain()` function.

The entry point for a QuickWin application is specified as the C library `WinMainCRTStartup()` routine (see module `crt0.c` in the C run-time sources). This gets the C run-time initialized, wraps the Intel® Visual Fortran defined `WinMain()` in a try-except construct using the C run-time exception filter (`_XcptFilter()`) and calls the Intel® Visual Fortran defined `WinMain()` routine. In simplified form, it looks like this:

```
WinMainCRTStartup() {
  C initialization code here
  __try {
        more initialization code here
        mainret = WinMain()      /* calls qwin library WinMain() */
        exit(mainret)
  } __except ( _XcptFilter() )
           { _exit ( GetExceptionCode() ) }
}
```

In the QuickWin library, `WinMain()` performs some initialization specific to QuickWin, creates a new thread which begins execution at `QWINForkMain`, and then sits in a message loop directing the action. The message loop is wrapped in a try-except-finally construct which invokes the Fortran run-time system default handler if an exception occurs, and calls for_rtl_finish_ at exit. `QWINForkMain()` running in the other thread calls the Fortran run-time system `main()`, which in turn calls `MAIN__`. In simplified form, it looks like this:

```
WinMain() {
  Initialization code here
  BeginThreadEx (..., QWINForkMain, ... )
  __try {
     __try {
           the message loop...
           for_rtl_finish()
           return (msg.wParam)
     } __except ( expression-invoking-default-fortran-handler )
              { }
  } __finally {
        for_rtl_finish()
```

```
        return (msg.wParam)
    }
  }
```

`QWINForkMain` resembles the following:

```
  QWINForkMain() {
    main()      /* calls the CVF rtl main() which calls MAIN__ */
    cleanup and exit...
  }
```

The routines `main()` and `MAIN__` are the same as previously described for a Fortran Console application.

Fortran DLL Applications

A Fortran DLL is a collection of one or more routines that you generally call from some other main program. As such, the routines execute in the structure and environment created by the code which calls into the DLL. You can provide DLL initialization through a `DllMain()` function, but you probably would control general application initialization from the main program.

There are no automatic provisions for any exception handler in a DLL. There is no environment initialization except what you provide. Of course, if your main application is also written in Fortran, you will get the default Fortran handlers provided by that application type.

Fortran Windows* Applications

A Fortran Windows* application has as its entry point `WinMainCRTStartup()` and each user writes the code for the `WinMain` function declaration and interface. Examples are provided to show how to do this in Fortran code. The compiler still generates symbol `MAIN__` with the initialization code in place, but nothing calls `MAIN__`. Also, nothing connects up to the run-time system's `main()` so there's no try-except construct to hook in the default Intel® Fortran handler, and no run-time system initialization or cleanup. In simplified form, it looks like this:

```
  WinMainCRTStartup() {
    C initialization code
    __try {
          more initialization code
          mainret = WinMain()      /* calls the user's WinMain() */
          exit(mainret)
    } __except ( _XcptFilter() )
              { _exit ( GetExceptionCode() ) }
  }
```

The Fortran code contains:

```
  integer(4) function WinMain( HANDLE, HANDLE, LPSTR, int )
  ...
  ! whatever Fortran the user codes here...
  ...
  end
```

See Also

for_rtl_finish_
FOR_SET_FPE

Providing Your Own Exception/Termination Handler

For Fortran Console, Fortran QuickWin, and Fortran Standard Graphics applications, the default exception and termination handlers are probably sufficient to meet most needs. As described in Handlers for the Application (Project) Types, Fortran DLL and Fortran Windows* applications do not have default handlers.

Whenever the default exception and termination handlers do not meet all your needs, consider providing your own handler. This is really a question you need to answer for each specific application. Some examples:

- Suppose your application creates some files during the course of its execution and you do not want to leave them on the disk if an unexpected error or exception occurs. The default termination actions only cause the files to be closed if you specifically opened them with DISPOSE='DELETE'. But suppose you do not want them deleted under normal termination. If an unexpected event occurs, you need to get control so you can clean up these files as needed.
- Perhaps your application can recover from a particular situation, for example, an integer divide-by-zero operation. You want to gain control if that exception occurs and deal with it.
- Perhaps you just want to output an application-specific error message when an exception occurs.
- You are building a Fortran DLL to run under a Visual Basic* GUI and you do not want the DLL to crash the application if an exception occurs in the DLL.
- Your code takes a lock on a global resource and you want to be sure and release the resource if an unexpected event occurs.

The list of possibilities is endless and only the application developer can anticipate particular needs.

The most general way to establish your own handler is to use Windows* structured exception handling capabilities (SEH). For lighter-weight exception handling requirements, you can use SIGNALQQ.

See Also

Using SIGNALQQ
SIGNALQQ
Handlers for the Application (Project) Types
Establishing Console Event Handlers
Using Windows* Structured Exception Handling (SEH) Overview
List of Run-Time Error Messages

Using Windows* Structured Exception Handling (SEH)

Using Windows* Structured Exception Handling (SEH) Overview

Windows* provides a robust exception and termination handling mechanism called Structured Exception Handling (SEH). Structured exception handling requires support in both the operating system and compilers. Unfortunately, Intel® Fortran does not include extensions for SEH support, but you can still take advantage of this powerful tool. By introducing a bit of C code in your application, you can use SEH to meet your exception handling needs.

A good reference on this subject is Chapter 16 in the book *Advanced Windows* (Third Edition) by Jeffrey Richter.

Custom Handlers for Fortran Console, Fortran QuickWin, and Fortran Standard Graphics Applications

Fortran Console and Fortran QuickWin (and Fortran Standard Graphics) applications have the full benefit of the Fortran default exception and error handling processing facilities. You may, however, want to supplement or replace the default facilities.

Custom Handlers for Fortran DLL Applications

There are two aspects to creating custom handlers for Fortran DLL applications:

- Containing errors and exceptions
- Enabling floating-point traps

Containing Errors and Exceptions in Fortran DLLs

If you are building a Fortran DLL and intend to call it from a main program written in some other language, you want to be careful that errors and exceptions in the DLL do not crash your main application.

Here are a few basic principles to keep in mind if you are building a Fortran DLL:

- Construct your library routines so that they return a status to the caller and let the caller decide what to do.
- To return an expected status to the caller, you need to be defensive in your library code, so consider these other principles:
- Where it makes sense, have the library code check input arguments passed in from the caller to make sure they are valid for whatever the library routine is going to do with them. For example, suppose the routine implements some numerical algorithm that has a valid domain of inputs it can act on and still produce well defined behavior. You can check the input arguments before you execute the algorithm and avoid unexpected behavior that might otherwise result (like unexpected floating-point exceptions). You might use Fortran intrinsic procedures like ISNAN and FP_CLASS to detect exceptional IEEE numbers. Your DLL code needs to return a status to the caller indicating the problem and let the caller take the appropriate action (gracefully shut down the application, try again with different input, etc.).
- In your library code, *always* check the success or failure of calls to I/O routines and dynamic memory allocation/deallocation. In Fortran, the I/O statements have optional ERR, END, EOR, and IOSTAT arguments that you can use to determine if the I/O requested was successful. Dynamic memory ALLOCATE and DEALLOCATE statements have an optional STAT specifier that allows you to obtain the status of the dynamic memory allocation/deallocation and prevent program termination.
- If you do not specify an action to take on an error, the Fortran run-time system has no choice but to deal with the error as an unhandled severe error and terminate the program. For a specific example of using IOSTAT and ERR to deal gracefully with an OPEN statement that gets a file-not-found error, see Using the IOSTAT Specifier and Fortran Exit Codes. You can do the same sort of thing in your code, but just return the status back to your Visual Basic* or other non-Fortran main program and let it decide what to do.
- Try to write your DLL code so unexpected program exceptions cannot occur, but devise a strategy for dealing with unexpected exceptions if they do happen. The most effective alternative for dealing with an exception is to use Windows Structured Exception Handling support to gain control when an exception happens. Wrap all your DLL routine calls in C try/except constructs and have the `except()` filter expression call a routine you define which determines how to respond.

Enabling Floating-Point Traps in Fortran DLLs

Before you can worry about how you will handle a floating-point trap condition occurring in a DLL, you have to consider the problem of unmasking those traps so they can occur. If you are compiling with `fpe[:]3` and polling the floating-point status word to check for exceptions, you do not have to worry about the problem of unmasking traps. You do not want traps unmasked in that case.

If your strategy is to compile with `fpe[:]0` and allow traps on floating-point exceptions, you need to take action to unmask the traps in the floating-point control word because most other languages mask traps by default.

Recall that a Fortran Console or Fortran QuickWin (or Standard Graphics) application would have unmasked traps for you automatically because the Fortran run-time system provides the main program and calls your `MAIN__` which executes some prolog code before the actual application code starts. You do not have that in a Fortran DLL called by some other language. Different languages establish different initial environments. You must provide the desired initial environment yourself.

See Also

ISNAN
FP_CLASS
ALLOCATE
DEALLOCATE

Custom Handlers for Fortran Windows* Applications

Fortran Windows* applications are not hooked up to the Fortran default exception handling processing facilities. Fortran Windows* applications are considered to be an area devoted to full customization, and the Fortran run-time system tries to "stay out of the way," so you can do whatever you want in your code.

Establishing Console Event Handlers

Control-C event handling is basically not reliable due to the threaded nature of processes executing on the Windows operating systems. Depending on what is happening at the instant a user types the Control-C, an event handler may or may not get the opportunity to execute. In any case, there are two ways to establish a handler if you want to do so. You can use the Windows* API routine `SetConsoleCtrlHandler` directly or you can use SIGNALQQ to establish a handler for the C `SIGINT` or `SIGBREAK` signals.

The Fortran run-time system establishes a console event handler through a call to `SetConsoleCtrlHandler` as part of its run-time initialization processing. See Default Console Event Handling for a description of this handler's behavior.

If you call `SetConsoleCtrlHandler` to establish your own event handler, your handler will be called first on console events.

If you establish a handler through SIGNALQQ with `SIGINT` or `SIGBREAK`, the C run-time system will establish its own internal handler for console events through a call to `SetConsoleCtrlHandler`, and it will record your routine as the desired action to take upon occurrence of an event. When an event is delivered to the C run-time handler, it will reset the action for the signal to `SIG_DFL` and then call your handler routine.

You must call SIGNALQQ again to reset the action to your routine if you want to continue from the control event. Your handler is called with the signal code (either `SIGINT` or `SIGBREAK`) as the argument. After your routine returns to the C run-time event handler, the C handler will return the value TRUE to the operating system indicating the event has been handled.

See Also

SIGNALQQ
Default Console Event Handling

Using SIGNALQQ

For lightweight exception handling requirements, a handler established with SIGNALQQ may meet your needs. This section describes how signal handling with SIGNALQQ works in detail and also how the Fortran run-time routine GETEXCEPTIONPTRSQQ works.

Reference is made to C run-time sources provided with Microsoft* Visual C++*. The discussion is worth reviewing even if you do not have Visual C++* available.

C-Style Signal Handling Overview

Many Fortran applications were developed on UNIX* systems where C-style signal handling was the usual way of dealing with exceptions. When ported to Windows*, these applications can continue to use the C signal interface. SIGNALQQ will work with any application type using pure Fortran or mixed Fortran and C code.

SIGNALQQ is just a Fortran jacket to the C run-time `signal()` function. When you call SIGNALQQ, you are actually registering your signal handler (or action) for a particular signal with the C run-time system. The C run-time system simply stores your handler (or action) in an internal exception action table or variable where it associates your handler with the desired signal. The operating system has no knowledge of this association.

If you have Visual C++* available, you can look at the code for the C run-time signal routine in `. . . \MICROSOFT VISUAL STUDIO .NET\VC7\CRT\SRC\WINSIG.C` and see how the table is managed. The table itself is defined and initialized in source file `WINXFLTR.C`, available in the same folder. When a signal occurs, the C run-time system checks its internal table to see if you have registered a handler for the particular signal. It calls your routine if you have assigned a handler.

Signal is Really SEH Again

Notice that it is the C run-time system that calls your handler when a signal occurs, not the operating system. So how did the C run-time get the exception delivered to it? Recall that the entry point of your image is either mainCRTStartup or WinMainCRTStartup, depending on the application type. Refer to Handlers for the Application (Project) Types and look at these entry points (or look at source file `Crt0.c` in the C run-time sources). Notice that they wrap a try-except construct around a call to either `main()` or `WinMain()` and that the filter expression associated with the __except construct calls a function `_XcptFilter`. `_XcptFilter` is passed two arguments which are the operating system supplied exception information.

When an exception occurs, the operating system looks at the list of exception filters and, starting with the inner-most nested try-except construct, evaluates except filter expressions until it finds one which does not return `EXCEPTION_CONTINUE_SEARCH`. If your application type includes main from the Fortran run-time system and thus the except construct associated with main, the Fortran run-time filter will be evaluated before the C run-time filter. The Fortran filter expression will check to see if you have established your own handler with SIGNALQQ. If it finds there is such a handler, or if you have set the environment variable `FOR_IGNORE_EXCEPTIONS`, it will return `EXCEPTION_CONTINUE_SEARCH` to allow the C run-time exception filter the opportunity to deal with the exception and find your handler. If you have not established your own handler or set the environment variable, the Fortran run-time will perform its default exception handling processing.

The C filter function, `_XcptFilter`, compares the exception code from the operating system with its mapping of operating system exceptions to C signal codes. If it finds a match in the table, it uses the exception action entry in the table corresponding to the signal code. This is the same table where your SIGNALQQ handler is recorded as the action for the requested signal code. If you have established a handler, it will be called from `_XcptFilter`. Before your handler is called, `_XcptFilter` resets the specified action for the signal to SIG_DFL in the exception action table. If you try to continue from the exception and you want your handler invoked on the next occurrence of the signal, you must call SIGNALQQ again to reestablish your handler as the action for that signal. When your handler routine is finished executing and returns to `_XcptFilter`, the value `EXCEPTION_CONTINUE_EXECUTION` is returned to the operating system by `_XcptFilter`. The operating system will then resume execution at the point of the exception. If you do not want to continue execution, your handler should take appropriate action to shut down the application.

Not every operating system exception code maps to a C signal code. You can see the mapping in source `WINXFLTR.C` if you have it. Here is the list if you do not have `WINXFLTR.C`:

Operating System Exception Code	C Signal Number
STATUS_ACCESS_VIOLATION	SIGSEGV
STATUS_ILLEGAL_INSTRUCTION	SIGILL
STATUS_PRIVILEGED_INSTRUCTION	SIGILL
STATUS_FLOAT_DENORMAL_OPERAND	SIGFPE
STATUS_FLOAT_DIVIDE_BY_ZERO	SIGFPE
STATUS_FLOAT_INEXACT_RESULT	SIGFPE
STATUS_FLOAT_INVALID_OPERATION	SIGFPE
STATUS_FLOAT_OVERFLOW	SIGFPE
STATUS_FLOAT_STACK_CHECK	SIGFPE
STATUS_FLOAT_UNDERFLOW	SIGFPE

How GETEXCEPTIONPTRSQQ Works

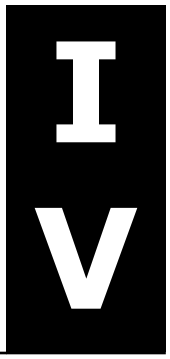
When the C run-time exception filter function `_XcptFilter` calls your handler that you established with SIGNALQQ, the only argument passed to your handler is the C signal number. The C run-time system also saves a pointer to the exception information supplied by the operating system. This pointer is named `_pxcptinfoptrs` and you can retrieve it through the Fortran run-time routine GETEXCEPTIONPTRSQQ (W*32). See C header file `signal.h` for the public definition of `_pxcptinfoptrs`.

The value returned by GETEXCEPTIONPTRSQQ can be used in your handler routine to generate a traceback with TRACEBACKQQ. GETEXCEPTIONPTRSQQ just returns `_pxcptinfoptrs`. This pointer is only valid while you are executing within the evaluation of the C run-time filter function `_XcptFilter` because the exception information is on the program stack, so do not use GETEXCEPTIONPTRSQQ in any other context.

See Also

GETEXCEPTIONPTRSQQ (W*32)
Handlers for the Application (Project) Types
SIGNALQQ

Part



Language Reference

This document contains the complete description of the Intel® Fortran programming language, which includes Fortran 2008, Fortran 2003, Fortran 95, Fortran 90, and some Fortran 2018 language features. It contains information on language syntax and semantics, on adherence to various Fortran standards, and on extensions to those standards.

For information about the Fortran standards, visit the Fortran standards technical committee website at http://j3-fortran.org/.

This manual is intended for experienced applications programmers who have a basic understanding of Fortran concepts and the Standard Fortran language.

Some familiarity with your operating system is helpful. This manual is not a Fortran or programming tutorial.

This document contains the following sections:

New Language Features	Describes the major new features for this release.
Program Elements and Source Forms	Describes program elements, the Fortran standard character set, and source forms.
Data Types, Constants, and Variables	Describes intrinsic and derived data types, constants, variables (scalars and arrays), and substrings.
Expressions and Assignment Statements	Summarizes Fortran expressions and assignment statements, which are used to define or redefine variables.
Specification Statements	Summarizes specification statements, which are used to declare the attributes of data objects.
Dynamic Allocation	Summarizes statements used in dynamic allocation: ALLOCATE, DEALLOCATE, and NULLIFY. It also describes the effects of allocation and deallocation.
Execution Control	Summarizes constructs and statements that can transfer control within a program.
Program Units and Procedures	Describes program units (including modules), subroutines and functions, and procedure interfaces.
Intrinsic Procedures	Describes argument keywords used in intrinsic procedures and provides an overview of intrinsic procedures.
Data Transfer I/O Statements	Summarizes data transfer input/output (I/O) statements.

I/O Formatting	Describes the rules for I/O formatting.
File Operation I/O Statements	Summarizes auxiliary I/O statements you can use to perform file operations.
Compilation Control Statements	Summarizes compilation control statements INCLUDE and OPTIONS.
Directive Enhanced Compilation	Describes general directives and OpenMP* Fortran compiler directives.
Scope and Association	Describes scope, which refers to the area in which a name is recognized, and association, which is the language concept that allows different names to refer to the same entity in a particular region of a program.
Deleted and Obsolescent Language Features	Describes deleted features in Fortran 2003 and obsolescent language features in Fortran 2003.
Additional Language Features	Describes some statements and language features supported for programs written in older versions of Fortran.
Additional Character Sets	Describes the additional character sets available on Windows*, Linux*, and macOS* systems.
Data Representation Models	Describes data representation models for numeric intrinsic functions.
Library Modules and Run-Time Library Routines	Summarizes the library modules and run-time library routines.
Summary of Language Extensions	Summarizes Intel Fortran extensions to the Fortran 2003 Standard.
A to Z Reference	Contains language summary tables and descriptions of all Intel® Fortran statements, intrinsics, directives, and module library routines, which are listed in alphabetical order.
Glossary	Contains abbreviated definitions of some commonly used terms in this manual.

Other books in this documentation set provide details about the features of the compilers, how to improve the run-time performance of Fortran programs, floating-point support, and compiler options.

For information on conventions used in this document, see Conventions.

For a summary of the latest Fortran 2003 features in this release, see Fortran 2003 Features.

For a summary of Fortran 2008 features in this release, see Fortran 2008 Features.

For a summary of Fortran 2018 features in this release, see Fortran 2018 Features.

New Language Features

The major new features for this release are as follows:

- Compiler option `assume [no]old_inquire_recl`

  Determines the value of the RECL= specifier on an INQUIRE statement for an unconnected unit or a unit connected for stream access. For more information, see assume.

- Compiler option `assume [no]old_ldout_zero`

  Determines the format of a floating-point zero produced by list-directed output. old_ldout_zero uses exponential format, noold_ldout_zero uses fractional format. For more information, see assume.
- Compiler option `check [no]udio_iostat`

  Determines whether standard conformance checking occurs when user defined derived type input/output procedures are executed. For more information, see check.
- Compiler option `warn [no]externals`

  Determines whether warnings occur for any dummy procedures or procedure calls that have on explicit interface or have not been declared EXTERNAL. For more information, see warn.

The following are new OpenMP* features:

- Extension to the OpenMP* SIMD directive

  You can now specify an IF clause for this directive. For more information, see SIMD Directive (OpenMP* API).
- Extension to the OpenMP* SIMD directive

  You can now specify a NONTEMPORAL clause for this directive. For more information, see SIMD Directive (OpenMP* API).
- The SCAN directive

  You can now specify the SCAN directive, which specifies a scan computation that updates each list item in each iteration of the loop the directive appears in. For more information, see SCAN.

For a summary of new Fortran 2018 features, see Fortran 2018 Features.

For a summary of new Fortran 2008 features, see Fortran 2008 Features.

For a summary of Fortran 2003 features, see Fortran 2003 Features.

For information on new compiler options in this release, see New Options in the *Compiler Options* reference.

For information about the Fortran standards, visit the Fortran standards technical committee website at http://j3-fortran.org/.

For information about the OpenMP* standards, see the OpenMP website at http://www.openmp.org/.

Program Elements and Source Forms

A Fortran program consists of one or more program units. A program unit is usually a sequence of statements that define the data environment and the steps necessary to perform calculations; it is terminated by an END statement.

A keyword in a Fortran program can either be a part of the syntax of a statement (statement keyword), or it can be the name of a dummy argument (argument keyword).

Names identify entities within a Fortran program unit. In earlier versions of Fortran, names were called "symbolic names".

Character sets show the characters you can use in Fortran programs.

Fortran programs can be in free, fixed, or tab format.

For more information, see the individual topics in this section.

Program Units

A Fortran program consists of one or more program units. A *program unit* is usually a sequence of statements that define the data environment and the steps necessary to perform calculations; it is terminated by an END statement.

A program unit can be either a main program, an external subprogram, a module, a submodule, or a block data program unit. An executable program contains one main program, and, optionally, any number of the other kinds of program units. Program units can be separately compiled.

An *external subprogram* is a function or subroutine that is not contained within a main program, a module, a submodule, or another subprogram. It defines a procedure to be performed and can be invoked from other program units of the Fortran program. Modules, submodules, and block data program units are not executable, so they are not considered to be procedures. (Modules and submodules can contain module procedures, though, which are executable.)

Modules contain definitions that can be made accessible to other program units: data and type definitions, definitions of procedures (called *module subprograms*), and *procedure interfaces*. Module subprograms can be either functions or subroutines. They can be invoked by other module subprograms in the module, or by other program units that access the module.

A *submodule* extends a module or another submodule. It can contain the definitions of procedures declared in a module or submodule.

A *block data program unit* specifies initial values for data objects in named common blocks. In Standard Fortran, this type of program unit can be replaced by a module program unit.

Main programs, external subprograms, and module subprograms can contain *internal subprograms*. The entity that contains the internal subprogram is its *host*. Internal subprograms can be invoked only by their host or by other internal subprograms in the same host. Internal subprograms must not contain internal subprograms.

The following sections discuss Statements, Names, and Keywords.

See Also

Program Units and Procedures

Statements

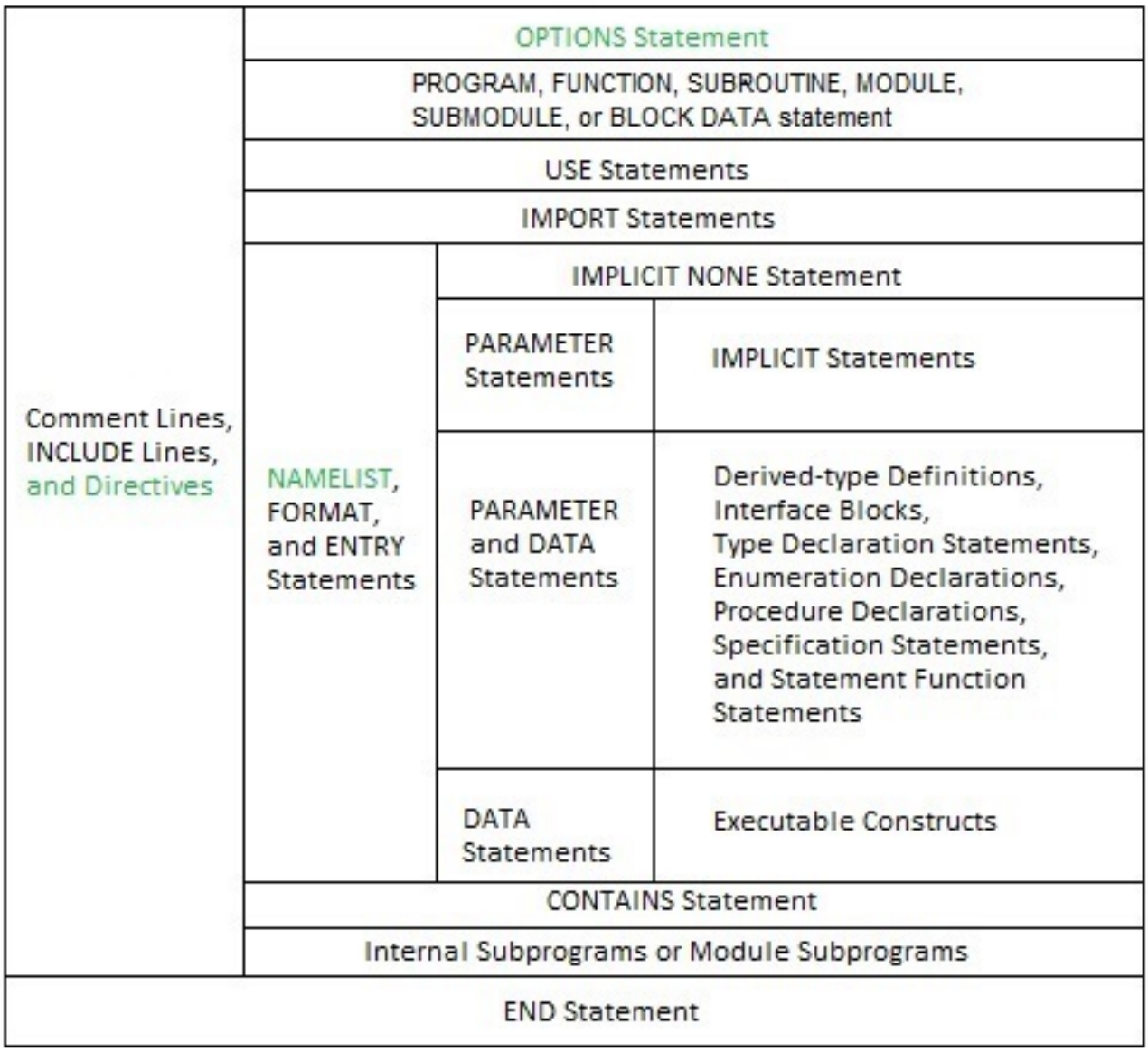
Program statements are grouped into two general classes: executable and nonexecutable. An *executable statement* specifies an action to be performed. A *nonexecutable statement* describes program attributes, such as the arrangement and characteristics of data, as well as editing and data-conversion information.

Order of Statements in a Program Unit

The following figure shows the required order of statements in a Fortran program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, you can intersperse DATA statements with executable constructs.

Horizontal lines indicate statement types that cannot be interspersed. For example, you cannot intersperse DATA statements with CONTAINS statements.

Required Order of Statements



PUBLIC and PRIVATE statements are only allowed in the scoping units of modules. In Standard Fortran, NAMELIST statements can appear only among specification statements. However, Intel® Fortran allows them to also appear among executable statements.

The following table shows other statements restricted from different types of scoping units.

Statements Restricted in Scoping Units

Scoping Unit	Restricted Statements
Main program	ENTRY, IMPORT, and RETURN statements
Module[1]	ENTRY, FORMAT, IMPORT, OPTIONAL, and INTENT statements, statement functions, and executable statements

Scoping Unit	Restricted Statements
Submodule[1]	ENTRY, FORMAT, IMPORT, OPTIONAL, and INTENT statements, statement functions, and executable statements
Block data program unit	CONTAINS, ENTRY, IMPORT, and FORMAT statements, interface blocks, statement functions, and executable statements
Internal subprogram	CONTAINS, IMPORT, and ENTRY statements
Interface body	CONTAINS, DATA, ENTRY, IMPORT[2], SAVE, and FORMAT statements, statement functions, and executable statements
BLOCK construct	CONTAINS, DATA, ENTRY, and IMPORT statements, statement functions, and these specification statements: COMMON, EQUIVALENCE, IMPLICIT, INTENT (or its equivalent attribute), NAMELIST, OPTIONAL (or its equivalent attribute), and VALUE (or its equivalent attribute)

[1] The scoping unit of a module does not include any module subprograms that the module contains.

[2] An IMPORT statement can appear only in an interface body that is not a separate module procedure interface body.

See Also

Scope for details on scoping units

Keywords

A keyword can either be a part of the syntax of a statement (statement keyword), or it can be the name of a dummy argument (argument keyword). Examples of statement keywords are WRITE, INTEGER, DO, and OPEN. Examples of argument keywords are arguments to the intrinsic functions.

In the intrinsic function UNPACK (*vector*, *mask*, *field*), for example, *vector*, *mask*, and *field* are argument keywords. They are dummy argument names, and any variable may be substituted in their place. Dummy argument names and real argument names are discussed in topic Program Units and Procedures.

Keywords are not reserved. The compiler recognizes keywords by their context. For example, a program can have an array named IF, read, or Goto, even though this is not good programming practice. The only exception is the keyword PARAMETER. If you plan to use variable names beginning with PARAMETER in an assignment statement, you need to use compiler option altparam.

Using keyword names for variables makes programs harder to read and understand. For readability, and to reduce the possibility of hard-to-find bugs, avoid using names that look like parts of Fortran statements. Rules that describe the context in which a keyword is recognized are discussed in topic Program Units and Procedures.

Argument keywords are a feature of Standard Fortran that let you specify dummy argument names when calling intrinsic procedures, or anywhere an interface (either implicit or explicit) is defined. Using argument keywords can make a program more readable and easy to follow. This is described more fully in topic Program Units and Procedures. The syntax statements in the A-Z Reference show the dummy keywords you can use for each Fortran procedure.

See Also

altparam compiler option

Program Units and Procedures

Names

Names identify entities within a Fortran program unit (such as variables, function results, common blocks, named constants, procedures, program units, namelist groups, and dummy arguments). In FORTRAN 77, names were called "symbolic names".

A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign.

In Fortran 2008, a name can contain up to 63 characters.

The length of a module name (in MODULE and USE statements) may be restricted by your file system.

NOTE
Be careful when defining names that contain dollar signs. A dollar sign can be a symbol for command or symbol substitution in various shell and utility commands.

In an executable program, the names of the following entities are global and must be unique in the entire program:

- Program units
- External procedures
- Common blocks
- Modules

Examples

The following examples demonstrate valid and invalid names

Valid Names

```
NUMBER

FIND_IT

X
```

Invalid Names

`5Q`	Begins with a numeral.
`B.4`	Contains a special character other than _ or $.
`_WRONG`	Begins with an underscore.

The following are all valid examples of using names:

```
INTEGER (SHORT) K       !K names an integer variable
SUBROUTINE EXAMPLE      !EXAMPLE names the subroutine
LABEL: DO I = 1,N       !LABEL names the DO block
```

Character Sets

Intel® Fortran supports the following characters:

- The Fortran character set, which consists of the following:
  - All uppercase and lowercase letters (A through Z and a through z)
  - The numerals 0 through 9
  - The underscore (_)
  - The following special characters:

Character	Name	Character	Name
blank or <Tab>	Blank (space) or tab	;	Semicolon
=	Equal sign	!	Exclamation point
+	Plus sign	"	Quotation mark or quote
-	Minus sign	%	Percent sign
*	Asterisk	&	Ampersand
/	Slash	~	Tilde
\	Backslash	<	Less than
(	Left parenthesis	>	Greater than
)	Right parenthesis	?	Question mark
[	Left square bracket	'	Apostrophe
]	Right square bracket	`	Grave accent
{	Left curly bracket	^	Circumflex accent
}	Right curly bracket	|	Vertical line
,	Comma	$	Dollar sign (currency symbol)
.	Period or decimal point	#	Number sign
:	Colon	@	Commercial at

- Other printable characters

  Printable characters include the tab character (09 hex), ASCII characters with codes in the range 20(hex) through 7E(hex), and characters in certain special character sets.

  Printable characters that are not in the Standard Fortran character set can only appear in comments, character constants, Hollerith constants, character string edit descriptors, and input/output records.

Uppercase and lowercase letters are treated as equivalent when used to specify program behavior (except in character constants and Hollerith constants).

See Also

Data Types, Constants, and Variables for further details on character sets and default character types

ASCII and Key Code Charts for Windows*

ASCII Character Set for Linux* and macOS*

Source Forms

Within a program, source code can be in free, fixed, or tab form. Fixed or tab forms must not be mixed with free form in the same source program, but different source forms can be used in different source programs.

All source forms allow lowercase characters to be used as an alternative to uppercase characters.

Several characters are indicators in source code (unless they appear within a comment or a Hollerith or character constant). The following are rules for indicators in all source forms:

- Comment indicator

A comment indicator can precede the first statement of a program unit and appear anywhere within a program unit. If the comment indicator appears within a source line, the comment extends to the end of the line.

An all blank line is also a comment line.

Comments have no effect on the interpretation of the program unit.

For more information, see comment indicators in free source form, or fixed and tab source forms.

- Statement separator

  More than one statement (or partial statement) can appear on a single source line if a statement separator is placed between the statements. The statement separator is a semicolon character (;).

  Consecutive semicolons (with or without intervening blanks) are considered to be one semicolon.

  If a semicolon is the first character on a line, the last character on a line, or the last character before a comment, it is ignored.
- Continuation indicator

  A statement can be continued for more than one line by placing a continuation indicator on the line. Intel Fortran allows at least 511 continuation lines for a fixed or tab source program. Although Standard Fortran permits up to 256 continuation lines in free-form programs, Intel® Fortran allows up to 511 continuation lines.

  Comments can occur within a continued statement, but comment lines cannot be continued.

  For more information, see continuation indicators in free source form, or fixed and tab source forms.

The following table summarizes characters used as indicators in source forms.

Indicators in Source Forms

Source Item	Indicator [1]	Source Form	Position
Comment	!	All forms	Anywhere in source code
Comment line	!	Free	At the beginning of the source line
	!, C, or *	Fixed	In column 1
		Tab	In column 1
Continuation line [2]	&	Free	At the end of the source line
	Any character except zero or blank	Fixed	In column 6
	Any digit except zero	Tab	After the first tab
Statement separator	;	All forms	Between statements on the same line
Statement label	1 to 5 decimal digits	Free	Before a statement
		Fixed	In columns 1 through 5
		Tab	Before the first tab
A debugging statement[3]	D	Fixed	In column 1
		Tab	In column 1

Source Item	Indicator [1]	Source Form	Position

[1] If the character appears in a Hollerith or character constant, it is not an indicator and is ignored.

[2] For fixed or tab source form, at least 511 continuation lines are allowed. For free source form, at least 255 continuation lines are allowed.

[3] Fixed and tab forms only.

Source form and line length can be changed at any time by using the FREEFORM, NOFREEFORM, or FIXEDFORMLINESIZE directives. The change remains in effect until the end of the file, or until changed again.

You can also select free source form by using compiler option free.

Source code can be written so that it is useable for all source forms.

Statement Labels

A *statement label* (or statement number) identifies a statement so that other statements can refer to it, either to get information or to transfer control. A label can precede any statement that is not part of another statement.

A statement label must be one to five decimal digits long; blanks and leading zeros are ignored. An all-zero statement label is invalid, and a blank statement cannot be labeled.

Labeled FORMAT and labeled executable statements are the only statements that can be referred to by other statement. FORMAT statements are referred to only in the format specifier of an I/O statement or in an ASSIGN statement. Two statements within a scoping unit cannot have the same label.

See Also

free compiler option

Free Source Form

In free source form, statements are not limited to specific positions on a source line. In Standard Fortran, a free form source line can contain from 0 to 132 characters. Intel® Fortran allows the line to be of any length.

Blank characters are significant in free source form. The following are rules for blank characters:

- Blank characters must not appear in lexical tokens, except within a character context. For example, there can be no blanks between the exponentiation operator **. Blank characters can be used freely between lexical tokens to improve legibility.
- Blank characters must be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels. For example, consider the following statements:

```
INTEGER NUM
GO TO 40
20 DO K=1,8
```

  The blanks are required after INTEGER, TO, 20, and DO.
- Some adjacent keywords must have one or more blank characters between them. Others do not require any; for example, BLOCK DATA can also be spelled BLOCKDATA. The following list shows which keywords have optional or required blanks:

Optional Blanks	Required Blanks
BLOCK DATA	ABSTRACT INTERFACE
DOUBLE COMPLEX	CASE DEFAULT
DOUBLE PRECISION	CLASS DEFAULT

Optional Blanks	Required Blanks
ELSE IF	CLASS IS
ELSE WHERE	DO CONCURRENT
END ASSOCIATE	DO WHILE
END BLOCK	ERROR STOP
END BLOCK DATA	EVENT POST
END CRITICAL	EVENT WAIT
END DO	FAIL IMAGE
END ENUM	IMPLICIT *type-specifier*
END FILE	IMPLICIT NONE
END FORALL	INTERFACE ASSIGNMENT
END FUNCTION	INTERFACE OPERATOR
END IF	MODULE PROCEDURE
END INTERFACE	*prefix*[1] [*prefix* ...] FUNCTION
END MODULE	*prefix*[1] [*prefix* ...] *type-specifier* [*prefix*...] FUNCTION
END PROCEDURE	*prefix*[1] [*prefix* ...] SUBROUTINE
END PROGRAM	SYNC ALL
END SELECT	SYNC IMAGES
END SUBMODULE	SYNC MEMORY
END SUBROUTINE	*type-specifier* FUNCTION
END TYPE	*type-specifier prefix*[1] [*prefix* ...] FUNCTION
END WHERE	
GO TO	
IN OUT	
SELECT CASE	
SELECT TYPE	

[1]*prefix* is ELEMENTAL or IMPURE or MODULE or NON_RECURSIVE or PURE or RECURSIVE. No *prefix* can be specified more than once. You cannot specify both IMPURE and PURE. You cannot specify both NON_RECURSIVE and RECURSIVE.

For information on statement separators (;) in all forms, see Source Forms.

Comment Indicator

In free source form, the exclamation point character (!) indicates a comment if it is within a source line, or a comment line if it is the first character in a source line.

Continuation Indicator

In free source form, the ampersand character (&) indicates a continuation line (unless it appears in a Hollerith or character constant, or within a comment). The continuation line is the first noncomment line following the ampersand. Although Standard Fortran permits up to 256 continuation lines in free-form programs, Intel® Fortran allows up to 511 continuation lines.

The following shows a continued statement:

```
TCOSH(Y) = EXP(Y) + &    ! The initial statement line
  EXP(-Y)                ! A continuation line
```

If the first nonblank character on the next noncomment line is an ampersand, the statement continues at the character following the ampersand. For example, the preceding example can be written as follows:

```
TCOSH(Y) = EXP(Y) + &
 & EXP(-Y)
```

If a lexical token must be continued, the first nonblank character on the next noncomment line must be an ampersand followed immediately by the rest of the token. For example:

```
   TCOSH(Y) = EXP(Y) + EX&
&P(-Y)
```

If you continue a character constant, an ampersand must be the first non-blank character of the continued line; the statement continues with the next character following the ampersand. For example:

```
ADVERTISER = "Davis, O'Brien, Chalmers & Peter&
        &son"
ARCHITECT  = "O'Connor, Emerson, and Dickinson&
      &  Associates"
```

If the ampersand is omitted on the continued line, the statement continues with the first non-blank character in the continued line. So, in the preceding example, the whitespace before "Associates" would be ignored.

The ampersand cannot be the only nonblank character in a line, or the only nonblank character before a comment; an ampersand in a comment is ignored.

See Also

Source Forms for details on the general rules for all source forms

Fixed and Tab Source Forms

In the Fortran standard, fixed source form is identified as obsolescent.

In fixed and tab source forms, there are restrictions on where a statement can appear within a line.

By default, a statement can extend to character position 72. In this case, any text following position 72 is ignored and no warning message is printed. You can specify compiler option extend-source to extend source lines to character position 132.

Except in a character context, blanks are not significant and can be used freely throughout the program for maximum legibility.

Some Fortran compilers use blanks to pad short source lines out to 72 characters. By default, Intel® Fortran does not. If portability is a concern, you can use the concatenation operator to prevent source lines from being padded by other Fortran compilers (see the example in "Continuation Indicator" below) or you can force short source lines to be padded by using compiler option pad-source.

Comment Indicator

In fixed and tab source forms, the exclamation point character (!) indicates a comment if it is within a source line. It must not appear in column 6 of a fixed form line; that column is reserved for a continuation indicator.

The letter C (or c), an asterisk (*), or an exclamation point (!) indicates a comment line when it appears in column 1 of a source line.

Continuation Indicator

In fixed and tab source forms, a continuation line is indicated by one of the following:

- For fixed form: Any character (except a zero or blank) in column 6 of a source line
- For tab form: Any digit (except zero) after the first tab

The compiler considers the characters following the continuation indicator to be part of the previous line. Although Standard Fortran permits up to 19 continuation lines in a fixed-form program, Intel® Fortran allows up to 511 continuation lines.

If a zero or blank is used as a continuation indicator, the compiler considers the line to be an initial line of a Fortran statement.

The statement label field of a continuation line must be blank (except in the case of a debugging statement).

When long character or Hollerith constants are continued across lines, portability problems can occur. Use the concatenation operator to avoid such problems. For example:

```
      PRINT *, 'This is a very long character constant '//
     +         'which is safely continued across lines'
```

Use this same method when initializing data with long character or Hollerith constants. For example:

```
      CHARACTER*(*) LONG_CONST
      PARAMETER (LONG_CONST = 'This is a very long '//
     + 'character constant which is safely continued '//
     + 'across lines')
      CHARACTER*100 LONG_VAL
      DATA LONG_VAL /LONG_CONST/
```

Hollerith constants must be converted to character constants before using the concatenation method of line continuation.

The Fortran Standard requires that, within a program unit, the END statement cannot be continued, and no other statement in the program unit can have an initial line that appears to be the program unit END statement. In these instances, Intel Fortran produces warnings when standards checking is requested.

Debugging Statement Indicator

In fixed and tab source forms, the statement label field can contain a statement label, a comment indicator, or a debugging statement indicator.

The letter D indicates a debugging statement when it appears in column 1 of a source line. The initial line of the debugging statement can contain a statement label in the remaining columns of the statement label field.

If a debugging statement is continued onto more than one line, every continuation line must begin with a D and a continuation indicator.

By default, the compiler treats debugging statements as comments. However, you can specify compiler option d-lines to force the compiler to treat debugging statements as source text to be compiled.

See Also

Fixed-format lines
Tab-format lines
Source Forms for details on the general rules for all source forms
d-lines compiler option
extend-source compiler option
pad-source compiler option

Fixed-Format Lines

In fixed source form, a source line has columns divided into fields for statement labels, continuation indicators, statement text, and sequence numbers. Each column represents a single character.

The column positions for each field follow:

Field	Column
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (or 132 with compiler option extend-source)
Sequence number	73 through 80

By default, a sequence number or other identifying information can appear in columns 73 through 80 of any fixed-format line in an Intel® Fortran program. The compiler ignores the characters in this field.

If you extend the statement field to position 132, the sequence number field does not exist.

> **NOTE**
> If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

See Also

Source Forms
Fixed and Tab Source Forms
extend-source compiler option

Tab-Format Lines

In tab source form, you can specify a statement label field, a continuation indicator field, and a statement field, but not a sequence number field.

The following figure shows equivalent source lines coded with tab and fixed source form.

Line Formatting Example

Format using TAB Character

C TAB FIRST VALUE

10 TAB I = J + 5*K +

TAB 1 L*M

TAB IVAL = I+2

Character-per-Column Format

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C						F	I	R	S	T		V	A	L	U	E			
1	0					I		=		J		+		5	*	K		+	
					1		L	*	M										
						I	V	A	L		=	I	+	2					

ZK-0614-GE

The statement label field precedes the first tab character. The continuation indicator field and statement field follow the first tab character.

The continuation indicator is any nonzero digit. The statement field can contain any Fortran statement. A Fortran statement cannot start with a digit.

If a statement is continued, a continuation indicator must be the first character (following the first tab) on the continuation line.

Many text editors and terminals advance the terminal print carriage to a predefined print position when you press the <Tab> key. However, the Intel® Fortran compiler does not interpret the tab character in this way. It treats the tab character in a statement field the same way it treats a blank character. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (usually located at columns 9, 17, 25, 33, and so on).

NOTE
If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

See Also

Source Forms for details on the general rules for all source forms
Fixed and Tab Source Forms for details on the general rules for fixed and tab source forms

Source Code Useable for All Source Forms

To write source code that is useable for all source forms (free, fixed, or tab), follow these rules:

Blanks	Treat as significant (see Free Source Form).

Statement labels	Place in column positions 1 through 5 (or before the first tab character).
Statements	Start in column position 7 (or after the first tab character).
Comment indicator	Use only !. Place anywhere *except* in column position 6 (or immediately after the first tab character).
Continuation indicator	Use only &. Place in column position 73 of the initial line and each continuation line, and in column 6 of each continuation line (no tab character can precede the ampersand in column 6).

The following example is valid for all source forms:

```
Column:
12345678...
                                                                      73
_______________________________________________________________________

! Define the user function MY_SIN
      DOUBLE PRECISION FUNCTION MY_SIN(X)
        MY_SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)                    &
     &           - X**7/FACTOR(7)
      CONTAINS
        INTEGER FUNCTION FACTOR(N)
         FACTOR = 1
         DO 10 I = N, 1, -1
  10     FACTOR = FACTOR * I
        END FUNCTION FACTOR
      END FUNCTION MY_SIN
```

Data Types, Constants, and Variables

Each constant, variable, array, expression, or function reference in a Fortran statement has a data type. The data type of these items can be inherent in their construction, implied by convention, or explicitly declared.

Each *data type* has the following properties:

- A name

  The names of the intrinsic data types are predefined, while the names of derived types are defined in derived-type definitions. Data objects (constants, variables, or parts of constants or variables) are declared using the name of the data type.
- A set of associated values

  Each data type has a set of valid values. Integer and real data types have a range of valid values. Complex and derived types have sets of values that are combinations of the values of their individual components.
- A way to represent constant values for the data type

  A *constant* is a data object with a fixed value that cannot be changed during program execution. The value of a constant can be a numeric value, a logical value, or a character string.

  A constant that does not have a name is a *literal constant*. A literal constant must be of intrinsic type and it cannot be array-valued.

A constant that has a name is a *named constant*. A named constant can be of any type, including derived type, and it can be array-valued. A named constant has the PARAMETER attribute and is specified in a type declaration statement or PARAMETER statement.

- A set of operations to manipulate and interpret these values

  The data type of a variable determines the operations that can be used to manipulate it. Besides intrinsic operators and operations, you can also define operators and operations.

See Also

Type Declarations

Defined Operations

PARAMETER attribute and statement

Expressions for details on valid operations for data types

Intrinsic Data Types

Intel® Fortran provides the following intrinsic data types:

- INTEGER

  The following kind parameters are available for data of type integer:

  - INTEGER([KIND=]1) or INTEGER*1
  - INTEGER([KIND=]2) or INTEGER*2
  - INTEGER([KIND=]4) or INTEGER*4
  - INTEGER([KIND=]8) or INTEGER*8
- REAL

  The following kind parameters are available for data of type real:

  - REAL([KIND=]4) or REAL*4
  - REAL([KIND=]8) or REAL*8
  - REAL([KIND=]16) or REAL*16
- DOUBLE PRECISION

  No kind parameter is permitted for data declared with type DOUBLE PRECISION. This data type is the same as REAL([KIND=]8).
- COMPLEX

  The following kind parameters are available for data of type complex:

  - COMPLEX([KIND=]4) or COMPLEX*8
  - COMPLEX([KIND=]8) or COMPLEX*16
  - COMPLEX([KIND=]16) or COMPLEX*32
- DOUBLE COMPLEX

  No kind parameter is permitted for data declared with type DOUBLE COMPLEX. This data type is the same as COMPLEX([KIND=]8).
- LOGICAL

  The following kind parameters are available for data of type logical:

  - LOGICAL([KIND=]1) or LOGICAL*1
  - LOGICAL([KIND=]2) or LOGICAL*2
  - LOGICAL([KIND=]4) or LOGICAL*4
  - LOGICAL([KIND=]8) or LOGICAL*8
- CHARACTER

  There is one kind parameter available for data of type character: CHARACTER([KIND=]1).
- BYTE

  This is a 1-byte value; the data type is equivalent to INTEGER([KIND=]1).

The intrinsic function KIND can be used to determine the kind type parameter of a representation method.

For more portable programs, you should not use the forms INTEGER([KIND=]n) or REAL([KIND=]n). You should instead define a PARAMETER constant using the SELECTED_INT_KIND or SELECTED_REAL_KIND function, whichever is appropriate. For example, the following statements define a PARAMETER constant for an INTEGER kind that has 9 digits:

```
INTEGER, PARAMETER :: MY_INT_KIND = SELECTED_INT_KIND(9)
  ...
INTEGER(MY_INT_KIND) :: J
  ...
```

Note that the syntax :: is used in type declaration statements.

The following sections describe the intrinsic data types and forms for literal constants for each type.

See Also

KIND intrinsic function

Declaration Statements for Noncharacter Types for details on declaration statements for intrinsic data types

Declaration Statements for Character Types for details on declaration statements for intrinsic data types

Expressions for details on operations for intrinsic data types

Data Type Storage Requirements table for details on storage requirements for intrinsic data types

Integer Data Types

Integer data types can be specified as follows:

INTEGER

INTEGER([KIND=]*n*)

INTEGER**n*

n	Is a constant expression that evaluates to kind 1, 2, 4, or 8.

If a kind parameter is specified, the integer has the kind specified. If a kind parameter is not specified, integer constants are interpreted as follows:

- If the integer constant is within the default integer kind range, the kind is default integer.
- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind that holds the constant.

Default integer is affected by compiler option integer-size, the INTEGER compiler directive, and the OPTIONS statement.

The intrinsic inquiry function KIND returns the kind type parameter, if you do not know it. You can use the intrinsic function SELECTED_INT_KIND to find the kind values that provide a given range of integer values. The decimal exponent range is returned by the intrinsic function RANGE.

Examples

The following examples show ways an integer variable can be declared.

An entity-oriented example is:

```
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER(2), POINTER :: k, limit
INTEGER(1), DIMENSION(10) :: min
```

An attribute-oriented example is:

```
INTEGER days, hours
INTEGER(2) k, limit
INTEGER(1) min
DIMENSION days(:), hours(:), min (10)
POINTER days, hours, k, limit
```

An integer can be used in certain cases when a logical value is expected, such as in a logical expression evaluating a condition, as in the following:

```
INTEGER I, X
READ (*,*) I
IF (I) THEN
  X = 1
END IF
```

See Also

Integer Constants
integer-size compiler option

Integer Constants

An *integer constant* is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

Integer constants take the following form:

[*s*]*n*[*n*...][_*k*]

s	Is a sign; required if negative (-), optional if positive (+).
n	Is a decimal digit (0 through 9). Any leading zeros are ignored.
k	Is the optional kind parameter: 1 for INTEGER(1), 2 for INTEGER(2), 4 for INTEGER(4), or 8 for INTEGER(8). It must be preceded by an underscore (_).

An unsigned constant is assumed to be nonnegative.

Integer constants are interpreted as decimal values (base 10) by default. To specify a constant that is not in base 10, use the following extension syntax:

[*s*] [[*base*] #] *nnn*...

s	Is an optional plus (+) or minus (-) sign.
base	Is any constant from 2 through 36. If *base* is omitted but # is specified, the integer is interpreted in base 16. If both *base* and # are omitted, the integer is interpreted in base 10. For bases 11 through 36, the letters A through Z represent numbers greater than 9. For example, for base 36, A represents 10, B represents 11, C represents 12, and so on, through Z, which represents 35. The case of the letters is not significant. The value of *nnn* cannot be bigger than 2**31-1. The value is extended with zeroes on the left or truncated on the left to make it the correct size. A minus sign for *s* negates the value.

Note that compiler option `integer-size` can affect the KIND type parameter of INTEGER data and integer constants.

Examples

Valid Integer (base 10) Constants

0
-127
+32123
47_2

Invalid Integer (base 10) Constants

9999999999999999999	Number too large.
3.14	Decimal point not allowed; this is a valid REAL constant.
32,767	Comma not allowed.
33_3	3 is not a valid kind type for integers.

The following seven integers are all assigned a value equal to 3,994,575 decimal:

```
I = 2#11110011111001111001111
m = 7#45644664
J = +8#17171717
K = #3CF3CF
n = +17#2DE110
L = 3994575
index = 36#2DM8F
```

The following seven integers are all assigned a value equal to -3,994,575 decimal:

```
I = -2#11110011111001111001111
m = -7#45644664
J = -8#17171717
K = -#3CF3CF
n = -17#2DE110
L = -3994575
index = -36#2DM8F
```

You can use integer constants to assign values to data. The following table shows assignments to different data and lists the integer and hexadecimal values in the data:

```
Fortran Assignment     Integer Value in Data    Hexadecimal Value in Data

LOGICAL(1)X
INTEGER(1)X
X = -128                         -128                Z'80'
X =  127                          127                Z'7F'
X =  255                           -1                Z'FF'

LOGICAL(2)X
INTEGER(2)X
X = 255                           255                Z'FF'
X = -32768                     -32768                Z'8000'
X = 32767                       32767                Z'7FFF'
X = 65535                          -1                Z'FFFF'
```

See Also

Numeric Expressions
integer-size compiler option

Real Data Types

Real data types can be specified as follows:

REAL

REAL([KIND=]*n*)

REAL**n*

DOUBLE PRECISION

n	Is a constant expression that evaluates to kind 4, 8, or 16.

If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is default real.

Default real is affected by compiler options specifying real size and by the REAL directive.

The default KIND for DOUBLE PRECISION is affected by compiler option double-size. If this compiler option is not specified, default DOUBLE PRECISION is REAL(8).

No kind parameter is permitted for data declared with type DOUBLE PRECISION.

The intrinsic inquiry function KIND returns the kind type parameter. The intrinsic inquiry function RANGE returns the decimal exponent range, and the intrinsic function PRECISION returns the decimal precision. You can use the intrinsic function SELECTED_REAL_KIND to find the kind values that provide a given precision and exponent range.

Examples

The following examples show how real variables can be declared.

An entity-oriented example is:

```
REAL (KIND = high), OPTIONAL :: testval
REAL, SAVE :: a(10), b(20,30)
```

An attribute-oriented example is:

```
REAL (KIND = high) testval
REAL a(10), b(20,30)
OPTIONAL testval
SAVE a, b
```

See Also

General Rules for Real Constants
REAL(4) Constants
REAL(8) or DOUBLE PRECISION Constants
REAL(16) Constants
real-size

General Rules for Real Constants

A *real constant* approximates the value of a mathematical real number. The value of the constant can be positive, zero, or negative.

The following is the general form of a real constant with no exponent part:

[*s*]*n*[*n*...][_*k*]

A real constant with an exponent part has one of the following forms:

[*s*]*n*[*n*...]E[*s*]*nn*...[_*k*]

[*s*]*n*[*n*...]D[*s*]*nn*...

[*s*]*n*[*n*...]Q[*s*]*nn*...

s	Is a sign; required if negative (-), optional if positive (+).
n	Is a decimal digit (0 through 9). A decimal point must appear if the real constant has no exponent part.
k	Is the optional kind parameter: 4 for REAL(4), 8 for REAL(8), or 16 for REAL(16). It must be preceded by an underscore (_).

Description

Leading zeros (zeros to the left of the first nonzero digit) are ignored in counting significant digits. For example, in the constant 0.00001234567, all of the nonzero digits, and none of the zeros, are significant. (See the following sections for the number of significant digits each kind type parameter typically has).

The exponent represents a power of 10 by which the preceding real or integer constant is to be multiplied (for example, 1.0E6 represents the value 1.0 * 10**6).

A real constant with no exponent part and no kind type parameter is (by default) a single-precision (REAL(4)) constant. You can change the default behavior by specifying compiler option fpconstant.

If the real constant has no exponent part, a decimal point must appear in the string (anywhere before the optional kind parameter). If there is an exponent part, a decimal point is optional in the string preceding the exponent part; the exponent part must not contain a decimal point.

The exponent letter E denotes a single-precision real (REAL(4)) constant, unless the optional kind parameter specifies otherwise. For example, -9.E2_8 is a double-precision constant (which can also be written as -9.D2).

The exponent letter D denotes a double-precision real (REAL(8)) constant.

The exponent letter Q denotes a quad-precision real (REAL(16)) constant.

A minus sign must appear before a negative real constant; a plus sign is optional before a positive constant. Similarly, a minus sign must appear between the exponent letter (E, D, or Q) and a negative exponent, whereas a plus sign is optional between the exponent letter and a positive exponent.

If the real constant includes an exponent letter, the exponent field cannot be omitted, but it can be zero.

To specify a real constant using both an exponent letter and a kind parameter, the exponent letter must be E, and the kind parameter must follow the exponent part.

See Also

fpconstant compiler option

REAL(4) Constants

A single-precision REAL constant occupies four bytes of memory. The number of digits is unlimited, but typically only the leftmost seven digits are significant.

IEEE* IEEE binary32 format is used.

Note that compiler option real-size can affect REAL data.

Examples

Valid REAL(4) Constants

```
3.14159

3.14159_4
```

```
621712._4

-.00127

+5.0E3

2E-3_4
```

Invalid REAL(4) Constants

`1,234,567.`	Commas not allowed.
`325E-47`	Too small for REAL; this is a valid DOUBLE PRECISION constant.
`-47.E47`	Too large for REAL; this is a valid DOUBLE PRECISION constant.
`625._6`	6 is not a valid kind for reals.
`100`	Decimal point is missing; this is a valid integer constant.
`$25.00`	Special character not allowed.

See Also

General Rules for Real Constants

real-size compiler option

Compiler Reference > Data and I/O section: *Data Representation*

REAL(8) or DOUBLE PRECISION Constants

A REAL(8) or DOUBLE PRECISION constant has more than twice the accuracy of a REAL(4) number, and greater range.

A REAL(8) or DOUBLE PRECISION constant occupies eight bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 15 digits are significant.

IEEE* binary64 format is used.

Note that compiler option double-size can affect DOUBLE PRECISION data.

The default KIND for DOUBLE PRECISION is affected by compiler option double-size.

Examples

Valid REAL(8) or DOUBLE PRECISION Constants

```
123456789D+5

123456789E+5_8

+2.7843D00

-.522D-12

2E200_8

2.3_8

3.4E7_8
```

Invalid REAL(8) or DOUBLE PRECISION Constants

`-.25D0_2`	2 is not a valid kind for reals.
`+2.7182812846182`	No D exponent designator is present; this is a valid single-precision constant.
`123456789.D400`	Too large for any double-precision format.
`123456789.D-400`	Too small for any double-precision format.

See Also

General Rules for Real Constants

double-size compiler option

Compiler Reference > Data and I/O section: *Data Representation*

REAL(16) Constants

A REAL(16) constant has more than four times the accuracy of a REAL(4) number, and a greater range.

A REAL(16) constant occupies 16 bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 33 digits are significant.

IEEE* binary128 format is used.

Examples

Valid REAL(16) Constants

```
123456789Q4000

-1.23Q-400

+2.72Q0

1.88_16
```

Invalid REAL(16) Constants

`1.Q5000`	Too large.
`1.Q-5000`	Too small.

See Also

General Rules for Real Constants

Compiler Reference > Data and I/O section: *Data Representation*

Complex Data Types

Complex data types can be specified as follows:

COMPLEX

COMPLEX([KIND=]*n*)

COMPLEX**s*

DOUBLE COMPLEX

n	Is a constant expression that evaluates to kind 4, 8, or 16.

s	Is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type default complex.

Default real is affected by compiler option real-size and by the REAL directive.

The default KIND for DOUBLE COMPLEX is affected by compiler option `double-size`. If the compiler option is not specified, default DOUBLE COMPLEX is COMPLEX(8).

No kind parameter is permitted for data declared with type DOUBLE COMPLEX.

A complex number of any kind is made up of a real part and an imaginary part. The REAL and AIMAG intrinsic functions return the real and imaginary parts of a complex number respectively. The CMPLX intrinsic constructs a complex number from two real numbers. The %re and %im complex part designators access the real and imaginary parts of a complex number respectively.

Examples

The following examples show how complex variables can be declared.

An entity-oriented example is:

```
COMPLEX (4), DIMENSION (8) :: cz, cq
```

An attribute-oriented example is:

```
COMPLEX(4) cz, cq
DIMENSION(8) cz, cq
```

The following shows an example of the parts of a complex number:

```
COMPLEX (4) :: ca = (1.0, 2.0)
REAL (4) :: ra = 3.0, rb = 4.0
PRINT *, REAL (ca), ca%RE            ! prints 1.0, 1.0
PRINT *, AIMAG (ca), ca%IM           ! prints 2.0, 2.0
PRINT *, CMPLX (ra, rb)              ! prints (3.0, 4.0)
ca = CMPLX (ra, AIMAG (ca))
PRINT *, ca                          ! prints (3.0, 2.0)
ca%im = rb
PRINT *, ca                          ! prints (3.0, 4.0)
```

See Also

General Rules for Complex Constants
AIMAG intrinsic
CMPLX intrinsic
COMPLEX(4) Constants
COMPLEX(8) or DOUBLE COMPLEX Constants
REAL intrinsic

General Rules for Complex Constants

A *complex constant* approximates the value of a mathematical complex number. The constant is a pair of real or integer values, separated by a comma, and enclosed in parentheses. The first constant represents the real part of that number; the second constant represents the imaginary part.

The following is the general form of a complex constant:

(*c*,*c*)

c	Is as follows:

- For COMPLEX(4) constants, *c* is an integer or REAL(4) constant.
- For COMPLEX(8) constants, *c* is an integer, REAL(4) constant, or DOUBLE PRECISION (REAL(8)) constant. At least one of the pair must be DOUBLE PRECISION.
- For COMPLEX(16) constants, *c* is an integer, REAL(4) constant, REAL(8) constant, or REAL(16) constant. At least one of the pair must be a REAL(16) constant.

Note that the comma and parentheses are required.

COMPLEX(4) Constants

A COMPLEX(4) constant is a pair of integer or single-precision real constants that represent a complex number.

A COMPLEX(4) constant occupies eight bytes of memory and is interpreted as a complex number.

If the real and imaginary part of a complex literal constant are both real, the kind parameter value is that of the part with the greater decimal precision.

The rules for REAL(4) constants apply to REAL(4) constants used in COMPLEX constants. (See General Rules for Complex Constants and REAL(4) Constants for the rules on forming REAL(4) constants.)

The REAL(4) constants in a COMPLEX constant have IEEE* binary32 format.

Note that compiler option real-size can affect REAL data.

Examples

Valid COMPLEX(4) Constants

```
(1.7039,-1.70391)

(44.36_4,-12.2E16_4)

(+12739E3,0.)

(1,2)
```

Invalid COMPLEX(4) Constants

`(1.23,)`	Missing second integer or single-precision real constant.
`(1.0, 2H12)`	Hollerith constant not allowed.

See Also

General Rules for Complex Constants
real-size compiler option

COMPLEX(8) or DOUBLE COMPLEX Constants

A COMPLEX(8) or DOUBLE COMPLEX constant is a pair of constants that represents a complex number. One of the pair must be a double-precision real constant, the other can be an integer, single-precision real, or double-precision real constant.

A COMPLEX(8) or DOUBLE COMPLEX constant occupies 16 bytes of memory and is interpreted as a complex number.

The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(8) or DOUBLE COMPLEX constants. (See General Rules for Complex Constants and REAL(8) or DOUBLE PRECISION Constants for the rules on forming DOUBLE PRECISION constants.)

The DOUBLE PRECISION constants in a COMPLEX(8) or DOUBLE COMPLEX constant have IEEE* binary64 format.

The default KIND for DOUBLE COMPLEX is affected by compiler option `double-size`.

Examples

Valid COMPLEX(8) or DOUBLE COMPLEX Constants

```
(1.7039,-1.7039D0)

(547.3E0_8,-1.44_8)

(1.7039E0,-1.7039D0)

(+12739D3,0.D0)
```

Invalid COMPLEX(8) or DOUBLE COMPLEX Constants

`(1.23D0,)`	Second constant missing.
`(1D1,2H12)`	Hollerith constants not allowed.
`(1,1.2)`	Neither constant is DOUBLE PRECISION; this is a valid single-precision constant.

See Also

General Rules for Complex Constants
double-size

COMPLEX(16) Constants

A COMPLEX(16) constant is a pair of constants that represents a complex number. One of the pair must be a REAL(16) constant, the other can be an integer, single-precision real, double-precision real, or REAL(16) constant.

A COMPLEX(16) constant occupies 32 bytes of memory and is interpreted as a complex number.

The rules for REAL(16) constants apply to REAL(16) constants used in COMPLEX constants. (See General Rules for Complex Constants and REAL(16) Constants for the rules on forming REAL(16) constants.)

The REAL(16) constants in a COMPLEX constant have IEEE* binary128 format.

Note that compiler option real-size can affect REAL data.

Examples

Valid COMPLEX(16) Constants

```
(1.7039,-1.7039Q2)

(547.3E0_16,-1.44)

(+12739D3,0.Q0)
```

Invalid COMPLEX(16) Constants

`(1.23Q0,)`	Second constant missing.
`(1D1,2H12)`	Hollerith constants not allowed.
`(1.7039E0,-1.7039D0)`	Neither constant is REAL(16); this is a valid double-precision constant.

See Also

General Rules for Complex Constants

real-size compiler option

Logical Data Types

Logical data types can be specified as follows:

LOGICAL

LOGICAL([KIND=]*n*)

LOGICAL*_n_

n	Is a constant expression that evaluates to kind 1, 2, 4, or 8.

If a kind parameter is specified, the logical constant has the kind specified. If no kind parameter is specified, the kind of the constant is default logical.

Examples

The following examples show how logical variables can be declared.

An entity-oriented example is:

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (KIND = byte), SAVE :: doit, dont
```

An attribute-oriented example is:

```
LOGICAL flag1, flag2
LOGICAL (KIND = byte) doit, dont
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

See Also

Logical Constants

Logical Constants

A logical constant represents only the logical values true or false, and takes one of the following forms:

.TRUE.[_*k*]

.FALSE.[_*k*]

k	Is the optional kind parameter: 1 for LOGICAL(1), 2 for LOGICAL(2), 4 for LOGICAL(4), or 8 for LOGICAL(8). It must be preceded by an underscore (_).

The numeric value of .TRUE. and .FALSE. can be -1 and 0 or 1 and 0 depending on compiler option fpscomp [no]logicals. Logical data can take on integer data values. Logical data type ranges correspond to their comparable integer data type ranges. For example, the LOGICAL(2) range is the same as the INTEGER(2) range.

Character Data Type

The character data type can be specified as follows:

CHARACTER ([LEN=] *len*)

CHARACTER (LEN= *len*, KIND= *n*)

CHARACTER (len, [KIND=] *n*)

CHARACTER (KIND= *n* [, LEN= *len*])

CHARACTER* *len* [,]

n	Is a constant expression that evaluates to kind 1.
len	Is a string length (not a kind). For more information, see Declaration Statements for Character Types.

If no kind type parameter is specified, the kind of the constant is default character.

On Windows systems, several Multi-Byte Character Set (MBCS) functions are available to manipulate special non-English characters.

See Also

Character Constants
C Strings
Character Substrings
Character Data Type

Character Constants

A *character constant* is a character string enclosed in delimiters (apostrophes or quotation marks). It takes one of the following forms:

[*k*_]'[*ch*...]' [*C*]

[*k*_]"[*ch*...]" [*C*]

k	Is the optional kind parameter: 1 (the default). It must be followed by an underscore (_). Note that in character constants, the kind must precede the constant.
ch	Is an ASCII character.
C	Is a C string specifier. C strings can be used to define NUL-terminated strings. For more information, see C Strings in Character Constants.

Description

The value of a character constant is the string of characters between the delimiters. The value does not include the delimiters, but does include all blanks or tabs within the delimiters.

If a character constant is delimited by apostrophes, use two consecutive apostrophes (`''`) to place an apostrophe character in the character constant.

Similarly, if a character constant is delimited by quotation marks, use two consecutive quotation marks ("") to place a quotation mark character in the character constant.

The length of the character constant is the number of characters between the delimiters, but two consecutive delimiters are counted as one character.

The length of a character constant must be in the range of 0 to 7188. Each character occupies one byte of memory.

If a character constant appears in a numeric context (such as an expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant.

A zero-length character constant is represented by two consecutive apostrophes or quotation marks.

Examples

Valid Character Constants

```
"WHAT KIND TYPE? "

'TODAY''S DATE IS: '
```

```
"The average is: "

''
```

Invalid Character Constants

`'HEADINGS`	No trailing apostrophe.
`'Map Number:"`	Beginning delimiter does not match ending delimiter.

See Also

Declaration Statements for Character Types

C Strings in Character Constants

String values in the C language are terminated with null characters (CHAR(0)) and can contain nonprintable characters (such as backspace).

Nonprintable characters are specified by escape sequences. An escape sequence is denoted by using the backslash (\) as an escape character, followed by a single character indicating the nonprintable character desired.

This type of string is specified by using a standard string constant followed by the character C. The standard string constant is then interpreted as a C-language constant. Backslashes are treated as escapes, and a null character is automatically appended to the end of the string (even if the string already ends in a null character).

The following table shows the escape sequences that are allowed in character constants:

C-Style Escape Sequences

Escape Sequence	Represents
\a or \A	A bell
\b or \B	A backspace
\f or \F	A formfeed
\n or \N	A new line
\r or \R	A carriage return
\t or \T	A horizontal tab
\v or \V	A vertical tab
\x*hh* or \X*hh*	A hexadecimal bit pattern
ooo	An octal bit pattern
\0	A null character
\\	A backslash

If a string contains an escape sequence that isn't in this table, the backslash is ignored.

A C string must also be a valid Fortran string. If the string is delimited by apostrophes, apostrophes in the string itself must be represented by two consecutive apostrophes (`''`).

For example, the escape sequence `\'string` causes a compiler error because Fortran interprets the apostrophe as the end of the string. The correct form is `\''string`.

If the string is delimited by quotation marks, quotation marks in the string itself must be represented by two consecutive quotation marks ("").

The sequences \ooo and \x*hh* allow any ASCII character to be given as a one- to three-digit octal or a one- to two-digit hexadecimal character code. Each octal digit must be in the range 0 to 7, and each hexadecimal digit must be in the range 0 to F. For example, the C strings `'\010'C` and `'\x08'C` both represent a backspace character followed by a null character.

The C string `'\\abcd'C` is equivalent to the string `'\abcd'` with a null character appended. The string `''C` represents the ASCII null character.

Character Substrings

A *character substring* is a contiguous segment of a character string. It takes one of the following forms:

v ([*e1*]:[*e2*])

a (*s* [, *s*] . . .) ([*e1*]:[*e2*])

v	Is a character scalar constant, or the name of a character scalar variable or character structure component.
e1	Is a scalar integer (or other numeric) expression specifying the leftmost character position of the substring; the *starting* point.
e2	Is a scalar integer (or other numeric) expression specifying the rightmost character position of the substring; the *ending* point.
a	Is the name of a character array.
s	Is a subscript expression.

Both *e1* and *e2* must be within the range 1,2, ..., *len*, where *len* is the length of the parent character string. If *e1* exceeds *e2*, the substring has length zero.

Description

Character positions within the parent character string are numbered from left to right, beginning at 1.

If the value of the numeric expression *e1* or *e2* is not of type integer, it is converted to integer before use (any fractional parts are truncated).

If *e1* is omitted, the default is 1. If *e2* is omitted, the default is *len*. For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

Examples

Consider the following example:

```
CHARACTER*8 C, LABEL
LABEL = 'XVERSUSY'
C = LABEL(2:7)
```

LABEL(2:7) specifies the substring starting with the second character position and ending with the seventh character position of the character variable assigned to LABEL, so C has the value `'VERSUS'`.

Consider the following example:

```
TYPE ORGANIZATION
  INTEGER ID
  CHARACTER*35 NAME
END TYPE ORGANIZATION
TYPE(ORGANIZATION) DIRECTOR
CHARACTER*25 BRANCH, STATE(50)
```

The following are valid substrings based on this example:

```
BRANCH(3:15)           ! parent string is a scalar variable
STATE(20) (1:3)        ! parent string is an array element
DIRECTOR%NAME(:)       ! parent string is a structure component
```

Consider the following example:

```
CHARACTER(*), PARAMETER :: MY_BRANCH = "CHAPTER 204"
CHARACTER(3) BRANCH_CHAP
BRANCH_CHAP = MY_BRANCH(9:11)   ! parent string is a character constant
```

BRANCH_CHAP is a character string of length 3 that has the value `'204'`.

See Also

Arrays
Array Elements
Structure Components

Derived Data Types

You can create derived data types from intrinsic data types or previously defined derived types.

A derived type is resolved into "ultimate" components that are either of intrinsic type or are pointers.

The set of values for a specific derived type consists of all possible sequences of component values permitted by the definition of that derived type. Structure constructors are used to specify values of derived types.

Nonintrinsic assignment for derived-type entities must be defined by a subroutine with an ASSIGNMENT interface. Any operation on derived-type entities must be defined by a function with an OPERATOR interface. Arguments and function values can be of any intrinsic or derived type.

A derived type can be parameterized by type parameters. Each type parameter is defined to be either a kind or a length type parameter, and can have a default value.

See Also

Derived-Type Assignment Statements
Defining Generic Operators for details on OPERATOR interfaces
Defining Generic Assignment for details on ASSIGNMENT interfaces
TYPE Statement (Derived Types)
Parameterized Derived-Type Declarations
Records for details on record structures

Derived-Type Definition Overview

A derived-type definition specifies the name of a user-defined type and the types of its components. For more information on the syntax of a derived-type definition, see TYPE. For more information on derived-type definitions for polymorphic objects, see CLASS.

Default Initialization

Default initialization occurs if initialization appears in a derived-type component definition.

The specified initialization of the component will apply even if the definition is PRIVATE.

Default initialization applies to dummy arguments with INTENT(OUT). It does not imply the derived-type component has the SAVE attribute.

Explicit initialization in a type declaration statement overrides default initialization.

To specify default initialization of an array component, use a constant expression that includes one of the following:

- An array constructor
- A single scalar that becomes the value of each array element

Pointers can have an association status of associated, disassociated, or undefined. If no default initialization status is specified, the status of the pointer is undefined. To specify disassociated status for a pointer component, use =>NULL( ). To default initialize a pointer component as associated with the target T, use => T.

Examples

You do not have to specify initialization for each component of a derived type. For example:

```
TYPE REPORT
  CHARACTER (LEN=20) REPORT_NAME
  INTEGER DAY
  CHARACTER (LEN=3) MONTH
  INTEGER :: YEAR = 1995        ! Only component with default
END TYPE REPORT                 !      initialization
```

Consider the following:

```
TYPE (REPORT), PARAMETER :: NOV_REPORT = REPORT ("Sales", 15, "NOV", 1996)
```

In this case, the explicit initialization in the type declaration statement overrides the YEAR component of NOV_REPORT.

The default initial value of a component can also be overridden by default initialization specified in the type definition. For example:

```
TYPE MGR_REPORT
  TYPE (REPORT) :: STATUS = NOV_REPORT
  INTEGER NUM
END TYPE MGR_REPORT
TYPE (MGR_REPORT) STARTUP
```

In this case, the STATUS component of STARTUP gets its initial value from NOV_REPORT, overriding the initialization for the YEAR component.

See Also

Initialization expressions

Procedure Pointers as Derived-Type Components

A component of derived type can be a procedure pointer. A procedure pointer component definition takes the following form:

`PROCEDURE` ([*proc-interface*]), *proc-attr* [, *proc-attr*]... :: *proc-decl-list*

proc-interface — (Optional) Is the name of an interface or a type specifier.

proc-attr — Is one of the following attributes:

- PUBLIC
- PRIVATE
- POINTER (required)
- NOPASS or PASS [(*arg-name*)]

  where *arg-name* is the name of a dummy argument.

  PASS and NOPASS refer to passed-object dummy arguments. They are mutually exclusive. You can only specify one or the other in a *proc-attr* list, not both.

If you specify NOPASS, procedures will not have have passed-object dummy arguments. NOPASS is required if the interface is implicit.

The PASS attribute can be used to confirm the default (as the first argument), The NOPASS attribute prevents passing the object as an argument.

Each *proc-attr* can only appear once in a given component definition.

proc-decl-list Is one or more of the following:

procedure-name [=> *null-init*]

where *null-init* is a reference to intrinsic function NULL with no arguments.

If => null-init appears, the procedure must have the POINTER attribute.

Examples

The following example defines a type that represents a list of procedures with the same interface, which can be called at some future time:

```
TYPE PROCEDURE_LIST
  PROCEDURE (PROC_INTERFACE), POINTER :: PROC
  TYPE (PROCEDURE_LIST), POINTER :: NEXT => NULL()
END TYPE PROCEDURE_LIST
ABSTRACT INTERFACE
  SUBROUTINE PROC_INTERFACE
  ...
  END SUBROUTINE PROC_INTERFACE
END INTERFACE
```

A procedure pointer can be pointer-assigned to a procedure pointer variable, invoked directly, or passed as an actual argument. For example:

```
TYPE (PROCEDURE_LIST) :: a, b(6)
PROCEDURE (PROC_INTERFACE), POINTER :: R
...
R => a%PROC
CALL SUBROUTINE_NEXT(a%PROC)
CALL b(i)%PROC
```

Type-Bound Procedures

In a derived-type definition, you can optionally specify a type-bound-procedure-part, which consists of a CONTAINS statement, optionally followed by a PRIVATE statement, and one or more procedure binding statements. Specific, GENERIC, and FINAL procedure bindings are defined below.

A *type-bound-procedure-part* in a derived-type definition declares one or more type-bound procedures. It takes the following form:

```
CONTAINS
[PRIVATE]
proc-binding-stmt
[ proc-binding-stmt]...
```

proc-binding-stmt Is one of the following:

- *specific-binding*
- *generic-binding*
- *final-binding*

The default accessibility of type-bound procedures is public even if the components are private. You can change this by including the PRIVATE statement following the CONTAINS statement.

specific-binding
A specific type-bound procedure binds a procedure to a type or it specifies a deferred binding to an abstract type. It takes the following form:

PROCEDURE (*interface-name*), *binding-attr-list* :: *binding-name* [, *binding-name*]...

PROCEDURE [[, *binding-attr-list*] ::] *binding-name* [=> *procedure-name*] [, [*binding-name* [=> *procedure-name*]]…]

interface-name
(Optional) Is the interface for the procedure. It must be the name of an abstract interface or a procedure that has an explicit interface. *interface-name* can only be specified if the binding has the DEFERRED attribute.

binding-attr-list
(Optional) Is one or more of the following. The same attribute can only appear once for the same binding:

- PASS [(*arg-name*)]

  Defines the "passed-object dummy argument" of the procedure. If *arg-name* is specified, the interface of the binding must have a dummy argument named *arg-name*.
- NOPASS

  Indicates that the procedure has no passed-object dummy argument. Use this keyword if the procedure pointer component has an implicit interface or has no arguments. PASS and NOPASS can not both be specified for the same binding.
- *access-spec*

  Is the PUBLIC or PRIVATE attribute.
- NON_OVERRIDABLE

  Determines whether a binding can be overridden in an extended type. You must not specify NON_OVERRIDABLE for a binding with the DEFERRED attribute.
- DEFERRED

  Indicates that the procedure is deferred. Deferred bindings must only be specified for derived-type definitions with the ABSTRACT attribute. A procedure with the DEFERRED binding attribute must specify an *interface-name*. An overriding binding can have the DEFERRED attribute only if the binding it overrides is deferred. The NON_OVERRIDABLE and DEFERRED binding attributes must not both be specified for the same procedure.

binding-name
Is the name of the type-bound procedure. It is referred to in the same way as a component of a type.

procedure-name
(Optional) Is the name of the actual procedure which implements *binding-name*. It is the name of an accessible module procedure or an external procedure that has an explicit interface. It defines the interface for the procedure, and the procedure to be executed when the procedure is referenced.

If neither =>procedure-name nor interface-name appears, the *procedure-name* is the same as the binding-name. If =>procedure-name appears, you must specify the double-colon separator and you must not specify an interface-name.

In general, the invoking variable, the passed-object dummy argument, is passed as an additional argument. By default, this is the first dummy argument of the actual procedure. So, the first argument in the argument list becomes the second argument, etc.

A passed-object dummy argument can be changed by declaring the type-bound procedure with the PASS(arg-name) attribute. In this case, the variable is passed as the named argument. The PASS attribute can also be used to confirm the default as the first argument. The NOPASS attribute prevents passing the object as an argument.

Consider the following:

```
TYPE MY_TYPE
...   ! Component declarations
CONTAINS
  PROCEDURE :: PROC1 => MY_PROC
  PROCEDURE :: PROC2, PROC3 => MY_PROC3
  PROCEDURE :: PROC4 => MY_PROC4, PROC5 => MY_PROC5
END TYPE MY_TYPE
```

This binds MY_PROC with the PROC1.

If B is a scalar variable of type MY_TYPE, an example of a type-bound call is:

```
CALL B%PROC1(C, D)
```

When PASS is in effect, the call passes B to the first argument of the procedure, C to the second argument, and D to the third argument.

generic-binding
A generic type-bound procedure defines a generic interface that is bound to the type. For more information, see the GENERIC statement.

final-binding
An extensible derived type can have one or more "final subroutines" associated with it. A final subroutine can be executed to perform clean-up operations after a data entity of that type is finalized (ceases to exist). For more information, see the FINAL statement.

See Also

TYPE

Passed-Object Dummy Arguments

PROCEDURE statement

Type Extension

Type extension lets you create new derived types by extending pre-existing derived types.

Any derived type can be extended using the EXTENDS keyword, except for types with the SEQUENCE or BIND(C) attributes. SEQUENCE types and BIND(C) types (as well as intrinsic types) are non-extensible.

The extended type either inherits or overrides each type-bound procedure of the parent type. An overriding procedure must be compatible with the parent procedure; in particular, each dummy argument must have the same type except for the passed-object dummy argument which must have the new type. A type-bound procedure that is declared to be NON_OVERRIDABLE cannot be overridden during type extension.

The extended type inherits all the components and parameters of the parent type and they are known by the same name. The extended type can also specify additional components.

For example, consider the following derived-type definition:

```
TYPE PERSON_INFO
  INTEGER :: AGE
  CHARACTER (LEN = 60) :: NAME
END TYPE PERSON_INFO
```

Derived type PERSON_INFO (the parent type) can be extended to create a new type as follows:

```
TYPE, EXTENDS (PERSON_INFO) :: EMPLOYEE_DATA
  INTEGER :: ID
  REAL :: SALARY
END TYPE EMPLOYEE_DATA
```

Extended type EMPLOYEE_DATA inherits all the components in type PERSON_INFO, as well as additional components ID and SALARY.

See Also

TYPE

Passed-Object Dummy Arguments

Parameterized Derived-Type Declarations

A derived type is parameterized if the derived TYPE statement has type parameter names or it inherits any type parameters.

The type parameters must be listed in the type definition. You must specify a type and you must specify whether they are KIND or LEN parameters. A type parameter definition takes the following form:

`INTEGER` [*kind-selector*], *type-param-attr-spec* :: *type-param-decl-list*

kind-selector	(Optional) Is one of the kind type parameter numbers allowed for integer data types. For more information, see Integer Data Types. If you do not specify *kind-selector*, default integer kind is assumed.
type-param-attr-spec	Is KIND or LEN.
type-param-decl-list	Is one or more of the following separated by commas: *type-param-name* [= *init-spec*]
type-param-name	Is the name of the type parameter. Each *type-param-name* must match one of the *type-param-name* parameters listed in the derived TYPE statement.
init-spec	(Optional) Must be a scalar integer constant expression.

If *init-spec* is specified, the type parameter has a default value that is specified by the expression *init-spec*. If necessary, the value of *init-spec* is converted to an integer value of the same kind as the type parameter in accordance with the rules of intrinsic assignment.

Explicit values for the type parameters are normally specified when an object of the parameter type is declared.

Within the type definition, kind type parameters may be used in constant expressions and specification expressions. Length type parameters may be used in specification expressions but not in constant expressions. The type parameters need not be used anywhere in the derived type.

Kind type parameters also participate in generic resolution Unambiguous Generic Procedure References. A single generic can include two specific procedures that have interfaces distinguished only by the value of a kind type parameter of a dummy argument.

An explicit interface is required if a parameterized derived type is used as a dummy argument. In a SELECT TYPE construct, the kind type parameter values of a type guard statement must be the same as those of the dynamic type of the selector. A BIND(C) type and a SEQUENCE type must not have type parameters.

Advantages to Using This Feature

Adding type parameters to a derived type allows you to declare objects of similar types that differ depending on the kind and length parameters used in the object declaration. For example, the type definition `matrix` below has two type parameters, a kind parameter `k` and a length parameter `b`.

```
TYPE matrix ( k, b )
  INTEGER,     KIND :: k = 4
  INTEGER (8), LEN  :: b
  REAL (k)          :: element (b,b)
END TYPE matrix
```

The user can declare a matrix named `square` of 10 by 10 elements of REAL(8) with the following declaration:

```
TYPE (matrix (8, 10)) :: square
```

The user can declare another matrix named `big_square` of 100 by 100 elements of REAL(4) with the following declaration:

```
TYPE (matrix (10)) :: square    ! k defaults to 4
```

See Also

Parameterized TYPE Statements
SELECT TYPE
SEQUENCE

Parameterized TYPE Statements

A derived *type-spec* (see Type Declarations) in a parameterized type declaration has the general form:

type-name [(*type-param-spec-list*)]

type-name	Is an accessible derived type.
type-param-spec-list	Is one or more of the following separate by commas: [*keyword* =] *type-param-value*
keyword	Is the name of a type parameter for the type *type-name*.
type-param-value	Is a scalar integer expression or an asterisk "*" or a colon ":".

A *type-param-spec-list* must appear only if the type is parameterized. There must be at most one *type-param-spec* corresponding to each parameter of the type. If a type parameter does not have a default value, there must be a *type-param-spec* corresponding to that type parameter.

The following topics in this section include cumulative examples demonstrating various kinds of parameterized TYPE statements.

Examples

Consider the following:

```
TYPE matrix (k, d1, d2)
  INTEGER, KIND :: k = kind (0.0)                  ! k has a default value
  INTEGER (selected_int_kind (12)), LEN :: d1, d2  ! Non-default kind for d1
  REAL (k) :: element (d1 ,d2)
END TYPE
! dim is an integer variable
TYPE(matrix(k = KIND(0d0), d1=200+5, d2=dim)) :: my_matrix1
! k has a default value and so the type-param-spec can be omitted
TYPE(matrix(d1=2*dim, d2=dim)) :: my_matrix2
TYPE(matrix(KIND(0d0), :, :)), pointer :: my_deferred_matrix
TYPE(matrix(KIND(0d0), *, *)) :: my_assumed_matrix
```

Each keyword must be the name of one of the parameters of the type. Similar to keyword arguments in procedure calls, "keyword=" can be omitted only if "keyword=" has been omitted for each preceding type parameter. If a keyword appears, the value corresponds to the type parameter named by the keyword. If keywords do not appear, the value corresponds to type parameters in type parameter order. If necessary, the value is converted according to the rules of intrinsic assignment to a value of the same kind as the type parameter.

See Also

Parameterized Derived-Type Declarations

Structure Constructors for Parameterized Derived Types

A *type-param-spec-list* must be given when constructing a parameterized derived type. See Parameterized TYPE Statements.

The syntax takes the following form:

derived-type-spec ([*comp-spec-list*])

derived-type-spec	Is a derived *type-spec*. See Type Declarations for details on derived type specifications.
comp-spec-list	Is a list of one or more component specifications.

Example

```
Matrix (kind (0.0),1,3) :: my_matrix
my_matrix =  matrix (kind (0.0),1,3) ([1.0,2.0,3.0])
```

See Also

Parameterized Derived-Type Declarations
Parameterized TYPE Statements
Structure Constructors

Type Parameter Order for Parameterized Derived Types

The type parameter order of a non-extended type is the order of the type parameter list in the derived type definition. The type parameter order of an extended type consists of the order of its parent type followed by the order of the type parameter list in the derived type definition.

A type parameter declared in an extended type must not have the same name as any accessible component or type parameter of its parent type.

Examples

Consider the following:

```
Type :: t1 (k1,k2)
  Integer, kind :: k1,k2
  Real(k1) ::  a(k2)
End type

Type, extends (t1) :: t2(k3)
  Integer, kind :: k3
  Logical(k3) flag
End type
```

The type parameter order for type `t1` is `k1`, then `k2`.

The type parameter order for type `t2` is `k1`, then `k2`, then `k3`.

See Also

Parameterized Derived-Type Declarations
Parameterized TYPE Statements

Deferred-Length Type Parameters for Parameterized Derived Types

Similar to deferred character lengths and deferred array bounds, the length type parameters of a derived type can be deferred. A deferred-length type parameter is specified when *type-param-value* in the *type-param-spec* is a colon (":").

A derived type object with a deferred-length type parameter must have an ALLOCATABLE or POINTER attribute. The value of a deferred type parameter is deferred until allocation or association.

When the dummy argument is allocatable or a pointer, the actual argument must have deferred the same type parameters as the dummy argument.

```
TYPE(matrix(k=KIND(0.0), d1= :, d2= :)), pointer :: my_mtrx_ptr, my_mtrx_alloc
TYPE(matrix(KIND(0.0), 100, 200)), target :: my_mtrx_tgt
TYPE(matrix(KIND(0.0), 1, 2)) :: my_mtrx_src
```

```
my_mtrx_ptr => my_mtrx_tgt   ! Gets values from target
! my_mtrx_ptr has d1= 100 and d2 = 200.

ALLOCATE(matrix(KIND(0.0), 10, 20) :: my_mtrx_alloc)  ! Gets values from allocation
! my_mtrx_alloc has d1=10 and d2=20
DEALLOCATE(my_mtrx_alloc)

ALLOCATE(my_mtrx_alloc, source=my_mtrx_src)  ! Gets values from allocation
! my_mtrx_alloc has d1=1 and d2=2
```

All *non-deferred* type parameter values of the declared type of the pointer object that correspond to non-deferred type parameters of the pointer target must agree. If a pointer object has non-deferred type parameters that correspond to deferred type parameters of a pointer target, the pointer target must not have undefined association status.

Deferred type parameters of functions, including function procedure pointers, have no values. Instead, they indicate that those type parameters of the function result will be determined by execution of the function, if it returns an allocated allocatable result or an associated pointer result.

If an ALLOCATE statement specifies a derived type object with deferred type parameters, it must either have the derived-type specification or a *source-expr* (a scalar expression). The kind type parameter values and the non-deferred length type parameter values in the derived-type specification or *source-expr* must be the same as the corresponding values of the derived type object.

```
TYPE(matrix(k=KIND(0.0), d1= 100, d2= :)), pointer :: my_mtrx_alloc
TYPE(matrix(KIND(0.0), 1, 2)) :: my_mtrx_src

ALLOCATE(my_mtrx_alloc, source=my_mtrx_src)  ! Illegal - d1 is not deferred
! my_mtrx_src must have the same value for d1
```

See Also

Parameterized Derived-Type Declarations
Parameterized TYPE Statements
ALLOCATE
ALLOCATABLE
POINTER - Fortran

Assumed-Length Type Parameters for Parameterized Derived Types

The length type parameter is assumed when an asterisk is used as a *type-param-value* in a *type-param-spec*. An asterisk may be used only in the declaration of a dummy argument, in the associate name in a SELECT TYPE statement, or in the allocation of a dummy argument. The value is taken from that of the actual argument.

```
Type(matrix(KIND(0d0), 10, :)), pointer :: y(:)
Call print_matrix(y)
...
Subroutine print_matrix(x)

  ! d1 here is assumed and its value '10' is obtained from the actual argument
  Type(matrix(k= KIND(0d0), d1=*, d2=:), pointer :: x(:)
  ALLOCATE(matrix(KIND(0.0), *, 10) :: x(10))
  ...
End Subroutine
```

All length type parameters of the dummy argument must be assumed for a final subroutine. All length type parameters of the first dummy argument to a user-defined I/O subroutine must be assumed. All of the length type parameters of a passed-object dummy argument must be assumed. In a SELECT TYPE construct, each length type parameter in a *type-guard-stmt* must be assumed.

Array Constructors

If *type-spec* in an array constructor specifies a parameterized derived type, all *ac-value* expressions in the array constructor must be of that derived type and must have the same kind type parameter values as specified by *type-spec*. Also, *type-spec* specifies the declared type and type parameters of the array constructor.

Each *ac-value* expression in the array constructor must be compatible with intrinsic assignment to a variable of this type and type parameter. Each value is converted to the type parameters of the array constructor in accordance with the rules of intrinsic assignment.

If *type-spec* is omitted, each *ac-value* expression in the array constructor must have the same length type parameters; in which case, the declared type and type parameters of the array constructor are those of the *ac-value* expressions.

Type Parameter Inquiry

The value of a derived-type parameter can be accessed from outside the type definition, using the same notation as component access.

```
print *,'No: of Rows =',my_matrix%d1
print *,'No: of Columns =',my_matrix%d2
```

Unlike components, a type parameter inquiry cannot be used on the left-hand side of an assignment, and type parameters are effectively always public. The value of a deferred type parameter of an unallocated allocatable, or of a pointer that is not associated with a target, is undefined and must not be inquired about. The length type parameters of an optional dummy argument that is not present must not be the subject of an inquiry.

The type parameters of intrinsic types can also be inquired using this syntax. The result is always scalar, even if the object is an array.

```
REAL(selected_real_kind(10,20)) :: z(100)
..
PRINT *,z%kind ! A single value is printed
```

This is same as calling KIND(z). However, the type parameter inquiry can be used even when the intrinsic function is not available.

```
Subroutine write_centered(ch, len)
  Character(*), intent(inout) :: ch
  Integer, intent(in) :: len
  Integer :: i
  Do i=1, (len - ch%len)/2
  ...
! The inquiry ch%len cannot be replaced with len(ch) since len is the
! name of a dummy argument
```

See Also

Parameterized Derived-Type Declarations
Parameterized TYPE Statements
SELECT TYPE

Structure Components

A reference to a component of a derived-type structure takes the following form:

parent [(*s-list*)] [*image-selector*] [%*component* [(*s-list*)] [*image-selector*]] ... %*component* [(*s-list*)] [*image-selector*]

parent	Is the name of a scalar or array of derived type. The percent sign (%) is called a component selector.
component	Is the name of a component of the immediately preceding parent or component.
s-list	Is a list of one or more subscripts. If the list contains subscript triplets or vector subscripts, the reference is to an array section. Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension. The number of subscripts in any *s-list* must equal the rank of the immediately preceding parent or component.
image-selector	Is an image selector if the parent or component is a coarray. For more information, see Image Selectors.

Description

Each parent or component (except the rightmost) must be of derived type.

The parent or one of the components can have nonzero rank (be an array). Any component to the right of a parent or component of nonzero rank must not have the POINTER attribute.

The rank of the structure component is the rank of the part (parent or component) with nonzero rank (if any); otherwise, the rank is zero. The type and type parameters (if any) of a structure component are those of the rightmost part name.

The structure component must not be referenced or defined before the declaration of the parent object.

If the parent object has the INTENT, TARGET, or PARAMETER attribute, the structure component also has the attribute.

If the rightmost component is of type C_PTR or C_FUNPTR from the intrinsic module ISO_C_BINDING, each parent or component must not have an *image-selector*.

Examples

The following example shows a derived-type definition with two components:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

The following shows how to declare CONTRACT to be of type EMPLOYEE:

```
TYPE(EMPLOYEE) :: CONTRACT
```

Note that both examples started with the keyword TYPE. The first (initial) statement of a derived-type definition is called a derived-type statement, while the statement that declares a derived-type object is called a TYPE statement.

The following example shows how to reference component ID of parent structure CONTRACT:

```
CONTRACT%ID
```

The following example shows a derived type with a component that is a previously defined type:

```
TYPE DOT
  REAL X, Y
END TYPE DOT
....
```

```
TYPE SCREEN
  TYPE(DOT) C, D
END TYPE SCREEN
```

The following declares a variable of type SCREEN:

```
TYPE(SCREEN) M
```

Variable M has components M%C and M%D (both of type DOT); M%C has components M%C%X and M%C %Y of type REAL.

The following example shows a derived type with a component that is an array:

```
TYPE CAR_INFO
  INTEGER YEAR
  CHARACTER(LEN=15), DIMENSION(10) :: MAKER
  CHARACTER(LEN=10) MODEL, BODY_TYPE*8
  REAL PRICE
END TYPE
...
TYPE(CAR_INFO) MY_CAR
```

Note that MODEL has a character length of 10, but BODY_TYPE has a character length of 8. You can assign a value to a component of a structure; for example:

```
MY_CAR%YEAR = 1985
```

The following shows an array structure component:

```
MY_CAR%MAKER
```

In the preceding example, if a subscript list (or substring) was appended to MAKER, the reference would not be to an array structure component, but to an array element or section.

Consider the following:

```
MY_CAR%MAKER(2) (4:10)
```

In this case, the component is substring 4 to 10 of the second element of array MAKER.

Consider the following:

```
TYPE CHARGE
   INTEGER PARTS(40)
   REAL LABOR
   REAL MILEAGE
 END TYPE CHARGE

 TYPE(CHARGE) MONTH
 TYPE(CHARGE) YEAR(12)
```

Some valid array references for this type follow:

```
MONTH%PARTS(I)           ! An array element
MONTH%PARTS(I:K)         ! An array section
YEAR(I)%PARTS            ! An array structure component (a whole array)
YEAR(J)%PARTS(I)         ! An array element
YEAR(J)%PARTS(I:K)       ! An array section
YEAR(J:K)%PARTS(I)       ! An array section
YEAR%PARTS(I)            ! An array section
```

The following example shows a derived type with a pointer component that is of the type being defined:

```
TYPE NUMBER
  INTEGER NUM

  TYPE(NUMBER), POINTER :: START_NUM => NULL( )
  TYPE(NUMBER), POINTER :: NEXT_NUM => NULL( )

END TYPE
```

A type such as this can be used to construct linked lists of objects of type NUMBER. Note that the pointers are given the default initialization status of disassociated.

The following example shows a private type:

```
TYPE, PRIVATE :: SYMBOL
  LOGICAL TEST
  CHARACTER(LEN=50) EXPLANATION
END TYPE SYMBOL
```

This type is private to the module. The module can be used by another scoping unit, but type SYMBOL is not available.

The following example shows how to dereference an array of structures with a coarray structure component that is a real array:

```
TYPE REAL_ARRAY
  REAL,ALLOCATABLE    :: RA(:)
END TYPE

TYPE CO_ARRAY
  TYPE(REAL_ARRAY)    :: CA[*]
END TYPE

TYPE STRUCT
  TYPE(CO_ARRAY)      :: COMP
END TYPE

TYPE(STRUCT)          :: ARRAY(10, 10)

ALLOCATE(ARRAY(5,10)%COMP%CA%RA(10))
ARRAY(5,10)%COMP%CA[5]%RA(:) = 0
```

See Also

Array Elements for details on references to array elements
Array Sections for details on references to array sections
Modules and Module Procedures for examples of derived types in modules

Structure Constructors

A structure constructor lets you specify scalar values of a derived type. It takes the following form:

d-name (*comp-spec*)

d-name	Is the name of the derived type. It cannot be an abstract type.
comp-spec	Is a component specification. Only one *comp-spec* can be specified for a component. However, you can specify more than one *comp-spec* in a constructor. A component specification takes the following form: [*keyword*=] *comp-data-source*

keyword	Is the name of a component of the derived type *d-name*.
comp-data-source	Is one of the following: • *expr* • *data-target* • *proc-target*
expr	Is an expression specifying component values. The values must agree in number and order with the components of the derived type. If necessary, values are converted (according to the rules of assignment), to agree with their corresponding components in type and kind parameters.
data-target	Is a data pointer component. It must have the same rank as its corresponding component.
proc-target	Is a procedure pointer component.

Description

A structure constructor must not appear before its derived type is defined.

If *keyword*= appears, any *expr* is assigned to the component named by the keyword. If *keyword*= is omitted, each *comp-data-source* is assigned to the corresponding component in component order. The *keyword*= can be omitted from a *comp-spec* only if the *keyword*= has been omitted from each preceding *comp-spec* in the constructor, if any.

If a component of the derived type is an array, the shape in the expression list must conform to the shape of the component array.

If a component of the derived type is a pointer, the value in the expression list must evaluate to an object that would be a valid target in a pointer assignment statement. (A constant is not a valid target in a pointer assignment statement.)

If all the values in a structure constructor are constant expressions, the constructor is a derived-type constant expression.

The type name and all components of the type for which a *comp-spec* appears must be accessible in the scoping unit containing the structure constructor.

For a pointer component, the corresponding *comp-data-source* must be an allowable *data-target* or *proc-target* for such a pointer in a pointer assignment statement. If the *comp-data-source* is a pointer, the association of the component is that of the pointer; otherwise, the component is pointer-associated with the *comp-data-source*.

If a component with default initialization has no corresponding *comp-data-source*, then the default initialization is applied to that component.

If a component of a derived type is allocatable, the corresponding constructor expression must either be a reference to the intrinsic function NULL with no arguments or an allocatable entity of the same rank, or it must evaluate to an entity of the same rank.

If the expression is a reference to the intrinsic function NULL, the corresponding component of the constructor has a status of unallocated.

If the expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that allocatable entity. If the entity is allocated, the constructor component has the same dynamic type, bounds, and value. If a length parameter of the component is deferred, its value is the same as the corresponding parameter of the expression. Otherwise the corresponding component of the constructor has an allocation status of allocated and has the same bounds and value as the expression.

Examples

Consider the following derived-type definition:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

This can be used to produce the following structure constructor:

```
EMPLOYEE(3472, "John Doe")
```

The following example shows a type with a component of derived type:

```
TYPE ITEM
  REAL COST
  CHARACTER(LEN=30) SUPPLIER
  CHARACTER(LEN=20) ITEM_NAME
END TYPE ITEM

TYPE PRODUCE
  REAL MARKUP
  TYPE(ITEM) FRUIT
END TYPE PRODUCE
```

In this case, you must use an embedded structure constructor to specify the values of that component; for example:

```
PRODUCE(.70, ITEM (.25, "Daniels", "apple"))
```

See Also

Pointer Assignments
Procedure Pointers

Binary, Octal, Hexadecimal, and Hollerith Constants

Binary, octal, hexadecimal, and Hollerith constants are nondecimal constants. They have no intrinsic data type, but assume a numeric data type depending on their use.

Standard Fortran allows unsigned binary, octal, and hexadecimal constants to be used in DATA statements; the constant must correspond to an integer scalar variable. These constants may also appear as arguments to certain standard intrinsic functions as indicated in their individual descriptions.

Each digit of a binary, octal, or hexadecimal literal constant represents a sequence of bits, according to its numerical interpretation (see Model for Bit Data) with *s* in the model equal to the following:

- one for binary constants
- three for octal constants
- four for hexadecimal constants

A binary, octal, or hexadecimal literal constant represents a sequence of bits that consists of the concatenation of the sequences of bits represented by its digits, in the order that the digits are specified. The positions of bits in the sequence are numbered from right to left, with the position of the rightmost bit being zero.

The length of a sequence of bits is the number of bits in the sequence. The position of the leftmost nonzero bit is interpreted to be at least *m* - 1, where *m* is the maximum value that could result from invoking the intrinsic function STORAGE SIZE with an argument that is a real or an integer scalar of any kind supported by the processor.

In Intel® Fortran, binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

See Also

Determining the Data Type of Nondecimal Constants
Bit Sequence Comparisons

Binary Constants

A *binary constant* is an alternative way to represent a numeric constant. A binary constant takes one of the following forms:

B'*d*[*d*...]'

B"*d*[*d*...]"

d	Is a binary (base 2) digit (0 or 1).

You can specify up to 128 binary digits in a binary constant.

Examples

Valid Binary Constants

```
B'0101110'

B"1"
```

Invalid Binary Constants

`B'0112'`	The character 2 is invalid.
`B10011'`	No apostrophe after the B.
`"1000001"`	No B before the first quotation mark.

See Also

Alternative Syntax for Binary, Octal, and Hexadecimal Constants

Octal Constants

An *octal constant* is an alternative way to represent numeric constants. An octal constant takes one of the following forms:

O'*d*[*d*...]'

O"*d*[*d*...]"

d	Is an octal (base 8) digit (0 through 7).

You can specify up to 128 bits (43 octal digits) in octal constants.

Examples

Valid Octal Constants

```
O'07737'

O"1"
```

Invalid Octal Constants

`O'7782'`	The character 8 is invalid.
`O7772'`	No apostrophe after the O.

"0737"	No O before the first quotation mark.

See Also

Alternative Syntax for Binary, Octal, and Hexadecimal Constants

Hexadecimal Constants

A *hexadecimal constant* is an alternative way to represent numeric constants. A hexadecimal constant takes one of the following forms:

Z'*d*[*d*...]'

Z"*d*[*d*...]"

d	Is a hexadecimal (base 16) digit (0 through 9, or an uppercase or lowercase letter in the range of A to F).

You can specify up to 128 bits (32 hexadecimal digits) in hexadecimal constants.

Examples

Valid Hexadecimal Constants

```
Z'AF9730'

Z"FFABC"

Z'84'
```

Invalid Hexadecimal Constants

Z'999.'	Decimal not allowed.
ZF9"	No quotation mark after the Z.

See Also

Alternative Syntax for Binary, Octal, and Hexadecimal Constants

Hollerith Constants

A *Hollerith constant* is a string of printable ASCII characters preceded by the letter H. Before the H, there must be an unsigned, nonzero default integer constant stating the number of characters in the string (including blanks and tabs).

Hollerith constants are strings of 1 to 2000 characters. They are stored as byte strings, one character per byte.

Examples

Valid Hollerith Constants

```
16HTODAY'S DATE IS:

1HB

4H ABC
```

Invalid Hollerith Constants

3HABCD	Wrong number of characters.

0H	Hollerith constants must contain at least one character.

Determining the Data Type of Nondecimal Constants

Binary, octal, hexadecimal, and Hollerith constants have no intrinsic data type. In most cases, the default integer data type is assumed.

However, these constants can assume a numeric data type depending on their use. When the constant is used with a binary operator (including the assignment operator), the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
INTEGER(2) ICOUNT		
INTEGER(4) JCOUNT		
INTEGER(4) N		
REAL(8) DOUBLE		
REAL(4) RAFFIA, RALPHA		
RAFFIA = B'1001100111111010011'	REAL(4)	4
RAFFIA = Z'99AF2'	REAL(4)	4
RALPHA = 4HABCD	REAL(4)	4
DOUBLE = B'1111111111100110011010'	REAL(8)	8
DOUBLE = Z'FFF99A'	REAL(8)	8
DOUBLE = 8HABCDEFGH	REAL(8)	8
JCOUNT = ICOUNT + B'011101110111'	INTEGER(2)	2
JCOUNT = ICOUNT + O'777'	INTEGER(2)	2
JCOUNT = ICOUNT + 2HXY	INTEGER(2)	2
IF (N .EQ. B'1010100') GO TO 10	INTEGER(4)	4
IF (N .EQ. O'123') GO TO 10	INTEGER(4)	4
IF (N. EQ. 1HZ) GO TO 10	INTEGER(4)	4

When a specific data type (generally integer) is required, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
Y(IX) = Y(O'15') + 3.	INTEGER(4)	4
Y(IX) = Y(1HA) + 3.	INTEGER(4)	4

When a nondecimal constant is used as an actual argument, the following occurs:

- When binary, octal, and hexadecimal constants are specified as arguments to intrinsic functions, and the description of the function specifies the interpretation of such constants, that specified interpretation is used.
- For binary, octal, and hexadecimal constants in other actual argument contexts, if the value fits in a default integer, that integer kind is used. Otherwise, the smallest integer kind large enough to hold the value is used.
- For Hollerith constants, a numeric data type of sufficient size to hold the length of the constant is assumed.

For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`CALL APAC(Z'34BC2')`	INTEGER(4)	4
`CALL APAC(9HABCDEFGHI)`	REAL(16)	9

When a binary, octal, or hexadecimal constant is used in any other context, the default integer data type is assumed. In the following examples, default integer is INTEGER(4):

Statement	Data Type of Constant	Length of Constant (in bytes)
`IF (Z'AF77') 1,2,3`	INTEGER(4)	4
`IF (2HAB) 1,2,3`	INTEGER(4)	4
`I = O'7777' - Z'A39'`[1]	INTEGER(4)	4
`I = 1HC - 1HA`	INTEGER(4)	4
`J = .NOT. O'73777'`	INTEGER(4)	4
`J = .NOT. 1HB`	INTEGER(4)	4

[1] When two typeless constants are used in an operation, they both take default integer type.

When nondecimal constants are not the same length as the length implied by a data type, the following occurs:

- Binary, octal, and hexadecimal constants

  These constants can specify up to 16 bytes of data. When the length of the constant is less than the length implied by the data type, the leftmost digits have a value of zero.

  When the length of the constant is greater than the length implied by the data type, the constant is truncated on the left. An error results if any nonzero digits are truncated.

  The Data Type Storage Requirements table lists the number of bytes that each data type requires.
- Hollerith constants

  When the length of the constant is less than the length implied by the data type, blanks are appended to the constant on the right.

  When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. If any characters other than blank characters are truncated, a warning occurs.

  Each Hollerith character occupies one byte of memory.

Enumerations and Enumerators

An enumeration defines the name of a group of related values and the name of each value within the group. It takes the following form:

ENUM, BIND(C)

  ENUMERATOR [::] *c1* [= *expr*][, *c2* [= *expr*]]...

[ENUMERATOR [::] *c3* [= *expr*][, *c4* [= *expr*]]...]...

END ENUM

c1,*c2*,*c3*,*c4*	Is the name of the enumerator being defined.
expr	Is an optional scalar integer constant expression specifying the value for the enumerator.

If = appears in an enumerator, a double-colon separator must appear before the enumerator or list of enumerators.

The compiler ensures that the integer kind declared for the enumerator is compatible with the integer type used by the corresponding C enumeration. The processor uses the same representation for the types declared by all C enumeration specifiers that specify the same values in the same order.

An enumerator is treated as if it were explicitly declared with the PARAMETER attribute.

The order in which the enumerators appear in an enumerator definition is significant.

If you do not explicitly assign each enumerator a value by specifying an *expr*, the compiler assigns a value according to the following rules:

- If the enumerator is the first enumerator in the enumerator definition, the enumerator has the value 0.
- If the enumerator is not the first enumerator in the enumerator definition, its value is the result of adding one to the value of the immediately preceding enumerator in the enumerator definition.

You can define the enumerators in multiple ENUMERATOR statements or in one ENUMERATOR statement. The order in which the enumerators are declared in an enumeration definition is significant, but the number of ENUMERATOR statements is not.

Examples

The following example shows an enumeration definition:

```
ENUM, BIND(C)
  ENUMERATOR ORANGE
  ENUMERATOR :: RED = 5, BLUE = 7
  ENUMERATOR GREEN
END ENUM
```

The kind type parameter for this enumeration is processor dependent, but the processor must select a kind sufficient to represent the values of the enumerators.

The order of the enumerators is significant if the values are not assigned explicitly. In the above example, the value of ORANGE becomes defined as a named constant with the value zero and the value of GREEN becomes defined as a named constant with the value 8. Note that if RED was the enumerator preceding GREEN, the value of GREEN would be 6 rather than 8.

The following declaration may be considered to be equivalent to the above enumeration definition:

```
INTEGER(SELECTED_INT_KIND(4)), PARAMETER :: ORANGE = 0, RED = 5, BLUE = 7, GREEN = 8
```

An entity of the same kind type parameter value can be declared using the intrinsic function KIND with one of the enumerators as its argument, for example:

```
INTEGER(KIND(BLUE)) :: X
```

Variables

A variable is a data object whose value can be changed (defined or redefined) at any point in a program. A variable can be any of the following:

- A scalar

  A *scalar* is a single object that has a single value; it can be of any intrinsic or derived (user-defined) type.
- An array

An *array* is a collection of scalar elements of any intrinsic or derived type. All elements must have the same type and kind parameters.

- A subobject designator

  A subobject is part of an object. The following are subobjects:

  An array element

  An array section

  A structure component

  A character substring

  For example, B(3) is a subobject (array element) designator for array B. A subobject cannot be a variable if its parent object is a constant.
- A reference to a function that returns a data pointer

  A reference to a function that returns a data pointer is treated as a variable and is permitted in any variable-definition context .

The name of a variable is associated with a single storage location.

A *designator* is a name followed by zero or more component selectors, complex part selectors, array section selectors, array element selectors, image selectors, and substring selectors.

Variables are classified by data type, as constants are. The data type of a variable indicates the type of data it contains, including its precision, and implies its storage requirements. When data of any type is assigned to a variable, it is converted to the data type of the variable (if necessary).

A variable is *defined* when you give it a value. A variable can be defined before program execution by a DATA statement or a type declaration statement. During program execution, variables can be defined or redefined in assignment statements and input statements, or undefined (for example, if an I/O error occurs). When a variable is undefined, its value is unpredictable.

When a variable becomes undefined, all variables associated by storage association also become undefined.

An object with subobjects, such as an array, can only be defined when all of its subobjects are defined. Conversely, when at least one of its subobjects are undefined, the object itself, such as an array or derived type, is undefined.

This section also discusses the Data Types of Scalar Variables and Arrays.

See Also

Type Declarations

DATA statement
Data Type of a Numeric Expressions
Storage Association for details on storage association of variables

Data Types of Scalar Variables

The data type of a scalar variable can be explicitly declared in a type declaration statement. If no type is declared, the variable has an implicit data type based on predefined typing rules or definitions in an IMPLICIT statement.

An explicit declaration of data type takes precedence over any implicit type. Implicit type specified in an IMPLICIT statement takes precedence over predefined typing rules.

See Also

Specification of Data Type
Implicit Typing Rules

Specification of Data Type

Type declaration statements explicitly specify the data type of scalar variables. For example, the following statements associate VAR1 with a 16-byte complex storage location, and VAR2 with an 8-byte double-precision storage location:

```
COMPLEX(8) VAR1
REAL(8) VAR2
```

NOTE
If no kind parameter is specified for a data type, the default kind is used. The default kind can be affected by compiler options that affect the size of variables.

You can explicitly specify the data type of a scalar variable only once.

If no explicit data type specification appears, any variable with a name that begins with the letter in the range specified in the IMPLICIT statement becomes the data type of the variable.

Character type declaration statements specify that given variables represent character values with the length specified. For example, the following statements associate the variable names INLINE, NAME, and NUMBER with storage locations containing character data of lengths 72, 12, and 9, respectively:

```
CHARACTER(72) INLINE
CHARACTER NAME*12, NUMBER*9
```

In single subprograms, assumed-length character arguments can be used to process character strings with different lengths. The assumed-length character argument has its length specified with an asterisk, for example:

```
CHARACTER(*) CHARDUMMY
```

The argument CHARDUMMY assumes the length of the actual argument.

See Also

Type Declarations

Assumed-length character arguments
IMPLICIT statement
Declaration Statements for Character Types

Implicit Typing Rules

By default, all variables, named constants, and functions with names beginning with I, J, K, L, M, or N are assumed to be of default integer type. Variables, named constants, and functions with names beginning with any other letter are assumed to be of default real type. For example:

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM_1
TOTAL_NUM	NTOTAL

Names beginning with a dollar sign ($) are implicitly INTEGER.

You can override the default data type implied in a name by specifying data type in either an IMPLICIT statement or a type declaration statement.

See Also

Type Declarations

IMPLICIT statement

Arrays

An array is a set of scalar elements that have the same type and kind parameters. Any object that is declared with an array specification is an array. Arrays can be declared by using a type declaration statement, or by using a DIMENSION, COMMON, ALLOCATABLE, POINTER, or TARGET statement.

An array can be referenced by element (using subscripts), by section (using a section subscript list), or as a whole. A subscript list (appended to the array name) indicates which array element or array section is being referenced.

A section subscript list consists of subscripts, subscript triplets, or vector subscripts. At least one subscript in the list must be a subscript triplet or vector subscript.

When an array name without any subscripts appears in an intrinsic operation (for example, addition), the operation applies to the whole array (all elements in the array).

An array has the following properties:

- Data type

  An array can have any intrinsic or derived type. The data type of an array (like any other variable) is specified in a type declaration statement or implied by the first letter of its name. All elements of the array have the same type and kind parameters. If a value assigned to an individual array element is not the same as the type of the array, it is converted to the array's type.
- Rank

  The rank of an array is the number of dimensions in the array. An array can have up to 31 dimensions. A rank-one array represents a column of data (a vector), a rank-two array represents a table of data arranged in columns and rows (a matrix), a rank-three array represents a table of data on multiple pages (or planes), and so forth.
- Bounds

  Arrays have a lower and upper bound in each dimension. These bounds determine the range of values that can be used as subscripts for the dimension. The value of either bound can be positive, negative, or zero.

  The bounds of a dimension are defined in an array specification.
- Size

  The size of an array is the total number of elements in the array (the product of the array's extents).

  The *extent* is the total number of elements in a particular dimension. It is determined as follows: upper bound - lower bound + 1. If the value of any of an array's extents is zero, the array has a size of zero.
- Shape

  The shape of an array is determined by its rank and extents, and can be represented as a rank-one array (vector) where each element is the extent of the corresponding dimension.

  Two arrays with the same shape are said to be *conformable*. A scalar is conformable to an array of any shape.

The name and rank of an array must be specified when the array is declared. The extent of each dimension can be constant, but does not need to be. The extents can vary during program execution if the array is a dummy argument array, an automatic array, an array pointer, or an allocatable array.

A whole array is referenced by the array name. Individual elements in a named array are referenced by a scalar subscript or list of scalar subscripts (if there is more than one dimension). A section of a named array is referenced by a section subscript.

This section also discusses:

- Whole Arrays
- Array Elements
- Array Sections
- Array Constructors

Examples

The following are examples of valid array declarations:

```
DIMENSION    A(10, 2, 3)           ! DIMENSION statement
ALLOCATABLE  B(:, :)               ! ALLOCATABLE statement
POINTER      C(:, :, :)            ! POINTER statement
REAL, DIMENSION (2, 5) :: D        ! Type declaration with
                                   !     DIMENSION attribute
```

Consider the following array declaration:

```
INTEGER L(2:11,3)
```

The properties of array L are as follows:

Data type:	INTEGER
Rank:	2 (two dimensions)
Bounds:	First dimension: 2 to 11 Second dimension: 1 to 3
Size:	30; the product of the extents: 10 x 3
Shape:	(/10,3/) (or 10 by 3); a vector of the extents 10 and 3

The following example shows other valid ways to declare this array:

```
DIMENSION L(2:11,3)
INTEGER, DIMENSION(2:11,3) :: L
COMMON L(2:11,3)
```

The following example shows references to array elements, array sections, and a whole array:

```
REAL B(10)            ! Declares a rank-one array with 10 elements

INTEGER C(5,8)        ! Declares a rank-two array with 5 elements in
                      !    dimension one and 8 elements in dimension two
...
B(3) = 5.0            ! Reference to an array element
B(2:5) = 1.0          ! Reference to an array section consisting of
                      !    elements: B(2), B(3), B(4), B(5)
...
C(4,8) = I            ! Reference to an array element
C(1:3,3:4) = J        ! Reference to an array section consisting of
                      !    elements:  C(1,3) C(1,4)
                      !               C(2,3) C(2,4)
                      !               C(3,3) C(3,4)/

B = 99                ! Reference to a whole array consisting of
                      !    elements: B(1), B(2), B(3), B(4), B(5),
                      !    B(6), B(7), B(8), B(9), and B(10)
```

See Also

DIMENSION attribute
Intrinsic data types
Derived data types
Declaration Statements for Arrays for details on array specifications
Categories of Intrinsic Functions for details on intrinsic functions that perform array operations

Whole Arrays

A *whole array* is a named array; it is either a named constant or a variable. It is referenced by using the array name (without any subscripts).

If a whole array appears in a nonexecutable statement, the statement applies to the entire array. For example:

```
INTEGER, DIMENSION(2:11,3) :: L   ! Specifies the type and
                                  !    dimensions of array L
```

If a whole array appears in an executable statement, the statement applies to all of the elements in the array. For example:

```
 L = 10       ! The value 10 is assigned to all the
              !   elements in array L
 WRITE *, L   ! Prints all the elements in array L
```

Array Elements

An *array element* is one of the scalar data items that make up an array. A subscript list (appended to the array or array component) determines which element is being referred to. A reference to an array element takes the following form:

array(*subscript-list*)

array	Is the name of the array.
subscript-list	Is a list of one or more subscripts separated by commas. The number of subscripts must equal the rank of the array. Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension.

Description

Each array element inherits the type, kind type parameter, and certain attributes (INTENT, PARAMETER, and TARGET) of the parent array. An array element cannot inherit the POINTER attribute.

If an array element is of type character, it can be followed by a substring range in parentheses; for example:

```
 ARRAY_D(1,2) (1:3)    ! Elements are substrings of length 3
```

However, by convention, such an object is considered to be a substring rather than an array element.

The following are some valid array element references for an array declared as REAL B(10,20): B(1,3), B(10,10), and B(5,8).

You can use functions and array elements as subscripts. For example:

```
REAL A(3, 3)
REAL B(3, 3), C(89), R
B(2, 2) = 4.5                       ! Assigns the value 4.5 to element B(2, 2)
R = 7.0
C(INT®*2 + 1) = 2.0                 ! Element 15 of C = 2.0
A(1,2) = B(INT(C(15)), INT(SQRT®))  ! Element A(1,2) = element B(2,2) = 4.5
```

For information on forms for array specifications, see Declaration Statements for Arrays.

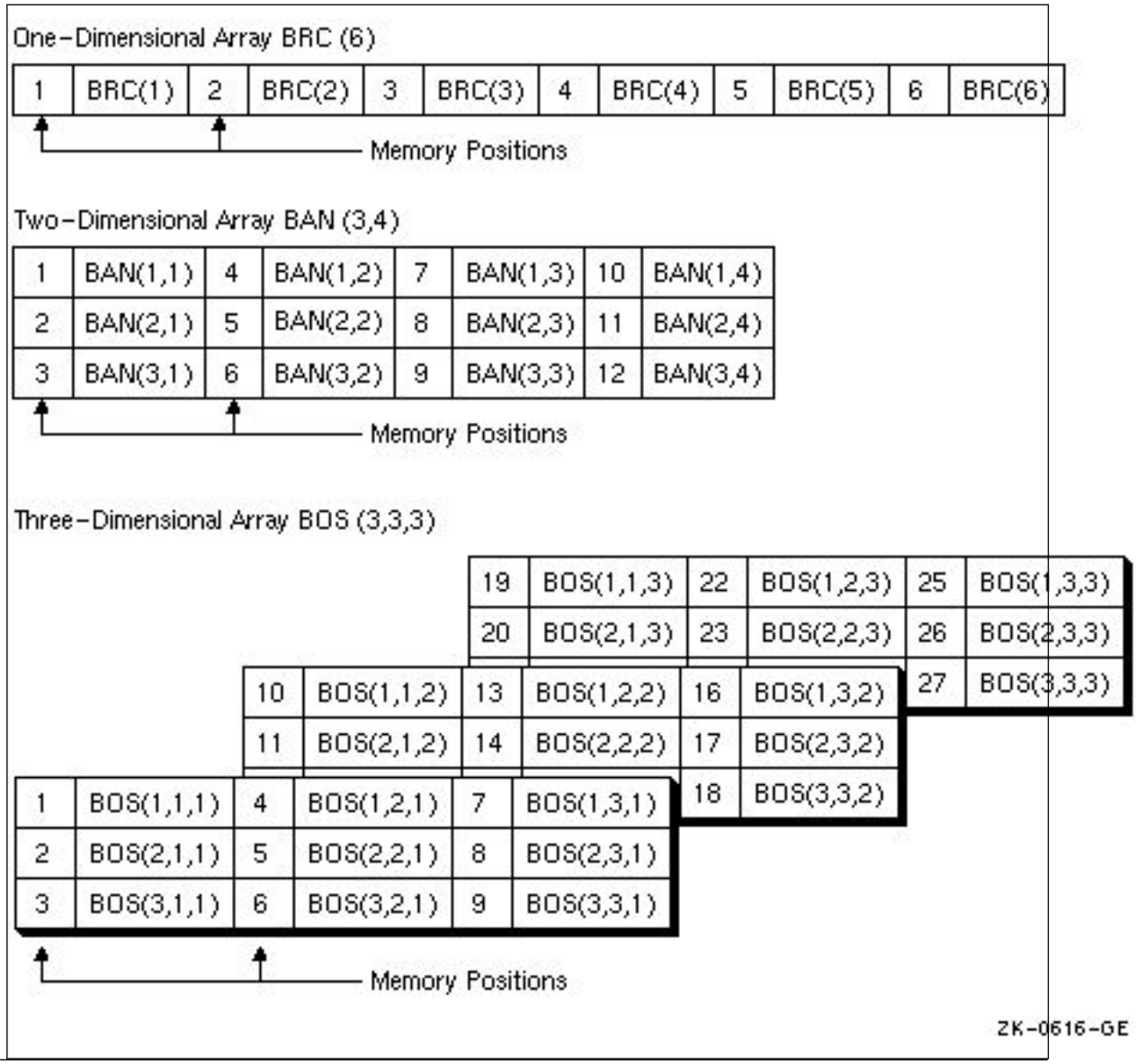
Array Element Order

The elements of an array form a sequence known as array element order. The position of an element in this sequence is its subscript order value.

The elements of an array are stored as a linear sequence of values. A one-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multidimensional array is stored so that the leftmost subscripts vary most rapidly. This is called the order of subscript progression.

The following figure shows array storage in one, two, and three dimensions:

Array Storage



For example, in two-dimensional array BAN, element BAN(1,2) has a subscript order value of 4; in three-dimensional array BOS, element BOS(1,1,1) has a subscript order value of 1.

In an array section, the subscript order of the elements is their order within the section itself. For example, if an array is declared as B(20), the section B(4:19:4) consists of elements B(4), B(8), B(12), and B(16). The subscript order value of B(4) in the array section is 1; the subscript order value of B(12) in the section is 3.

See Also

Array association
Character Constants for details on substrings

Structure Components for details on arrays as structure components
Storage Association for details on storage sequence association

Array Sections

An *array section* is a portion of an array that is an array itself. It is an array subobject. A section subscript list (appended to the array or array component) determines which portion is being referred to. A reference to an array section takes the following form:

array(*sect-subscript-list*)

array	Is the name of the array.
sect-subscript-list	Is a list of one or more section subscripts (subscripts, subscript triplets, or vector subscripts) indicating a set of elements along a particular dimension. At least one of the items in the section subscript list must be a subscript triplet or vector subscript. A subscript triplet specifies array elements in increasing or decreasing order at a given stride. A vector subscript specifies elements in any order. Each subscript and subscript triplet must be a scalar integer (or other numeric) expression. Each vector subscript must be a rank-one integer expression.

Description

If *no* section subscript list is specified, the rank and shape of the array section is the same as the parent array.

Otherwise, the rank of the array section is the number of vector subscripts and subscript triplets that appear in the list. Its shape is a rank-one array where each element is the number of integer values in the sequence indicated by the corresponding subscript triplet or vector subscript.

If any of these sequences is empty, the array section has a size of zero. The subscript order of the elements of an array section is that of the array object that the array section represents.

Each array section inherits the type, kind type parameter, and certain attributes (INTENT, PARAMETER, and TARGET) of the parent array. An array section cannot inherit the POINTER attribute.

If an array (or array component) is of type character, it can be followed by a substring range in parentheses. Consider the following declaration:

```
CHARACTER(LEN=15) C(10,10)
```

In this case, an array section referenced as C(:,:) (1:3) is an array of shape (10,10), whose elements are substrings of length 3 of the corresponding elements of C.

The following shows valid references to array sections. Note that the syntax (/.../) denotes an array constructor.

```
REAL, DIMENSION(20) :: B
...
PRINT *, B(2:20:5)  ! The section consists of elements
                    !    B(2), B(7), B(12), and B(17)

K = (/3, 1, 4/)
B(K) = 0.0    ! Section B(K) is a rank-one array with shape (3) and
              !   size 3. (0.0 is assigned to B(1), B(3), and B(4).)
```

See Also

INTENT attribute
PARAMETER attribute

TARGET attribute
Array constructors
Character Substrings
Structure components for details on array sections as structure components

Subscript Triplets

A *subscript triplet* is a set of three values representing the lower bound of the array section, the upper bound of the array section, and the increment (stride) between them. It takes the following form:

[*first-bound*] : [*last-bound*] [:*stride*]

first-bound	Is a scalar integer (or other numeric) expression representing the first value in the subscript sequence. If omitted, the declared lower bound of the dimension is used.
last-bound	Is a scalar integer (or other numeric) expression representing the last value in the subscript sequence. If omitted, the declared upper bound of the dimension is used. When indicating sections of an assumed-size array, this subscript *must* be specified.
stride	Is a scalar integer (or other numeric) expression representing the increment between successive subscripts in the sequence. It must have a nonzero value. If it is omitted, it is assumed to be 1.

The stride has the following effects:

- If the stride is positive, the subscript range starts with the first subscript and is incremented by the value of the stride, until the largest value less than or equal to the second subscript is attained.

  For example, if an array has been declared as B(6,3,2), the array section specified as B(2:4,1:2,2) is a rank-two array with shape (3,2) and size 6. It consists of the following six elements:

```
B(2,1,2)    B(2,2,2)
B(3,1,2)    B(3,2,2)
B(4,1,2)    B(4,2,2)
```

  If the first subscript is greater than the second subscript, the range is empty.
- If the stride is negative, the subscript range starts with the value of the first subscript and is decremented by the absolute value of the stride, until the smallest value greater than or equal to the second subscript is attained.

  For example, if an array has been declared as A(15), the array section specified as A(10:3:-2) is a rank-one array with shape (4) and size 4. It consists of the following four elements:

```
A(10)
A(8)
A(6)
A(4)
```

  If the second subscript is greater than the first subscript, the range is empty.

If a range specified by the stride is empty, the array section has a size of zero.

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used to select the array elements are within the declared bounds. For example, if an array has been declared as A(15), the array section specified as A(4:16:10) is valid. The section is a rank-one array with shape (2) and size 2. It consists of elements A(4) and A(14).

If the subscript triplet does not specify bounds or stride, but only a colon (:), the entire declared range for the dimension is used.

If you leave out all subscripts, the section defaults to the entire extent in that dimension. For example:

```
REAL A(10)
A(1:5:2) = 3.0   ! Sets elements A(1), A(3), A(5) to 3.0
A(:5:2) = 3.0    ! Same as the previous statement
                 !   because the lower bound defaults to 1
A(2::3) = 3.0    ! Sets elements A(2), A(5), A(8) to 3.0
                 ! The upper bound defaults to 10
A(7:9) = 3.0     ! Sets elements A(7), A(8), A(9) to 3.0
                 ! The stride defaults to 1
A(:) = 3.0       ! Same as A = 3.0; sets all elements of
                 !   A to 3.0
```

See Also

Array Sections

Vector Subscripts

A *vector subscript* is a one-dimensional (rank one) array of integer values (within the declared bounds for the dimension) that selects a section of a whole (parent) array. The elements in the section do not have to be in order and the section can contain duplicate values.

For example, A is a rank-two array of shape (4,6). B and C are rank-one arrays of shape (2) and (3), respectively, with the following values:

```
B = (/1,4/)           ! Syntax (/.../) denotes an array constructor
C = (/2,1,1/)         ! This constructor produces a many-one array section
```

Array section A(3,B) consists of elements A(3,1) and A(3,4). Array section A(C,1) consists of elements A(2,1), A(1,1), and A(1,1). Array section A(B,C) consists of the following elements:

```
A(1,2)    A(1,1)    A(1,1)
A(4,2)    A(4,1)    A(4,1)
```

An array section with a vector subscript that has two or more elements with the same value is called a *many-one array section*. For example:

```
REAL A(3, 3), B(4)
INTEGER K(4)
! Vector K has repeated values
K = (/3, 1, 1, 2/)
! Sets all elements of A to 5.0
A = 5.0
B = A(3, K)
```

The array section A(3,K) consists of the elements:

```
A(3, 3)  A(3, 1)  A(3, 1)  A(3, 2)
```

A many-one section must not appear on the left of the equal sign in an assignment statement, or as an input item in a READ statement.

The following assignments to C also show examples of vector subscripts:

```
INTEGER A(2), B(2), C(2)
...
B    = (/1,2/)
C(B) = A(B)
C    = A((/1,2/))
```

An array section with a vector subscript must not be any of the following:

- An internal file
- An actual argument associated with a dummy array that is defined or redefined (if the INTENT attribute is specified, it must be INTENT(IN))

- The target in a pointer assignment statement

If the sequence specified by the vector subscript is empty, the array section has a size of zero.

See Also

Array sections
Array constructors

Array Constructors

An *array constructor* can be used to create and assign values to rank-one arrays and array constants. An array constructor takes one of the following forms:

(/*ac-spec*/)

[*ac-spec*]

ac-spec	Is one of the following: *type-spec* :: or [*type-spec* ::] *ac-value-list*
type-spec	Is a type specifier.
ac-value-list	Is a list of one or more expressions or implied-DO loops.

An implied-DO loop in an array constructor takes the following form:

(*ac-value-list*, *do-variable* = *expr*1, *expr*2 [,*expr*3])

do-variable	Is the name of a scalar integer variable. Its scope is that of the implied-DO loop.
expr	Is a scalar integer expression. The *expr1* and *expr2* specify a range of values for the loop; *expr3* specifies the stride. The *expr3* must be a nonzero value; if it is omitted, it is assumed to be 1.

Description

If *type-spec* is omitted, each *ac-value-list* expression must have the same type and kind type parameters, and the same length parameters. In this case, the array constructed has the same type as the *ac-value-list* expressions.

If *type-spec* appears, it specifies the type and type parameters of the array constructor. Each *ac-value-list* expression must be compatible with intrinsic assignment to a variable of this type and type parameters. Each value is converted to the type parameters of the array constructor in accordance with the rules of intrinsic assignment.

If *type-spec* specifies an intrinsic type, each *ac-value-list* expression must be of an intrinsic type that conforms with a variable of type type-spec.

If *type-spec* specifies a derived type, all *ac-value-list* expressions must be of that derived type and must have the same kind type parameter values as specified by *type-spec*.

If the sequence of values specified by the array constructor is empty (an empty array expression or the implied-DO loop produces no values), the rank-one array has a size of zero.

An *ac-value* is interpreted as follows:

Form of *ac-value*	Result
A scalar expression	Its value is an element of the new array.

Form of *ac-value*	Result
An array expression	The values of the elements in the expression (in array element order) are the corresponding sequence of elements in the new array.
An implied-DO loop	It is expanded to form a list of array elements under control of the DO variable (like a DO construct).

The following shows the three forms of an *ac-value*:

```
C1 = (/4,8,7,6/)                  ! A scalar expression
C2 = (/B(I, 1:5), B(I:J, 7:9)/)   ! An array expression
C3 = (/(I, I=1, 4)/)              ! An implied-DO loop
```

You can also mix these forms, for example:

```
C4 = (/4, A(1:5), (I, I=1, 4), 7/)
```

If every expression in an array constructor is a constant expression, the array constructor is a constant expression.

If *type-spec* is omitted, Intel® Fortran allows the numeric expressions to be of different kind types, but not different types. The resultant numeric array is the type and kind type of the first expression in the *ac-value-list*. For example:

```
 INTEGER, PARAMETER :: K(2) = [4, 8]
! The following line prints 4.000000       8.00000000000000
PRINT *, REAL (K(1), K(1)), REAL (K(2), K(2))
! The following line prints 4.000000       8.000000
PRINT *, [ (REAL (K(I), K(I)), I=1,20]
```

If *type-spec* is omitted, Intel® Fortran allows the character expressions to be of different character lengths. The length of the resultant character array is the maximum of the lengths of the individual character expressions. For example:

```
print *,len ( (/'a','ab','abc','d'/) )
print *,'++'//(/'a','ab','abc','d'/)//'--'
```

This causes the following to be displayed:

```
           3
++a  --++ab --++abc--++d  --
```

If an implied-DO loop is contained within another implied-DO loop (nested), they cannot have the same DO variable (*do-variable*).

To define arrays of more than one dimension, use the RESHAPE intrinsic function.

The following are alternative forms for array constructors:

- Square brackets (instead of parentheses and slashes) to enclose array constructors; for example, the following two array constructors are equivalent:

```
INTEGER C(4)
C = (/4,8,7,6/)
C = [4,8,7,6]
```

- A colon-separated triplet (instead of an implied-DO loop) to specify a range of values and a stride; for example, the following two array constructors are equivalent:

```
INTEGER D(3)
D = (/1:5:2/)              ! Triplet form - also [1:5:2]
D = (/(I, I=1, 5, 2)/)     ! implied-DO loop form
```

The stride is optional; it defaults to 1. For example, the following two array constructors are equivalent:

```
 INTEGER E(5)
 E = (/1:5/)              ! Triplet form with default stride – also [1:5]
 E = (/(I, I=1, 5)/)      ! implied-DO loop form
```

Examples

The following example shows an array constructor using an implied-DO loop:

```
 INTEGER ARRAY_C(10)
 ARRAY_C = (/(I, I=30, 48, 2)/)
```

The values of ARRAY_C are the even numbers 30 through 48.

Implied-DO expressions and values can be mixed in the value list of an array constructor. For example:

```
INTEGER A(10)
A = (/1, 0, (I, I = -1, -6, -1), -7, -8 /)
! Mixed values and implied-DO in value list.
```

This example sets the elements of A to the values, in order, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8.

The following example shows an array constructor of derived type that uses a structure constructor:

```
TYPE EMPLOYEE
    INTEGER ID
    CHARACTER(LEN=30) NAME
END TYPE EMPLOYEE
TYPE(EMPLOYEE) CC_4T(4)
CC_4T = (/EMPLOYEE(2732,"JONES"), EMPLOYEE(0217,"LEE"),     &
           EMPLOYEE(1889,"RYAN"), EMPLOYEE(4339,"EMERSON")/)
```

The following example shows how the RESHAPE intrinsic function can be used to create a multidimensional array:

```
 E = (/2.3, 4.7, 6.6/)
 D = RESHAPE(SOURCE = (/3.5, (/2.0, 1.0/), E/), SHAPE = (/2,3/))
```

D is a rank-two array with shape (2,3) containing the following elements:

```
  3.5    1.0    4.7
  2.0    2.3    6.6
```

The following shows another example:

```
INTEGER B(2,3), C(8)
! Assign values to a (2,3) array.
B = RESHAPE((/1, 2, 3, 4, 5, 6/),(/2,3/))
! Convert B to a vector before assigning values to
! vector C.
C = (/ 0, RESHAPE(B,(/6/)), 7 /)
```

Consider the following derived-type definition:

```
TYPE EMPLOYEE
 INTEGER AGE
 CHARACTER (LEN = 60) NAME
END TYPE EMPLOYEE
```

The following equivalent lines use the above type to construct a derived-type array value:

```
(/ EMPLOYEE (45, 'OLIVER'), EMPLOYEE (30, 'ONEIL') /)
[EMPLOYEE (45, 'OLIVER'), EMPLOYEE (30, 'ONEIL')]
```

The following example shows an array constructor that specifies a length type parameter:

```
(/ CHARACTER(LEN=8) :: 'Andrews', 'Donahue', 'Dehenney' , 'Washington' /)
```

In this constructor, without the type specification, Intel® Fortran makes all of the array elements length 10, that is, as long as the longest character element.

See Also

Execution Control
Subscript triplets
Derived types
Structure constructors
Array Elements for details on array element order
Array Assignment Statements for details on another way to assign values to arrays
Declaration Statements for Arrays for details on array specifications

Coarrays

Coarrays and synchronization constructs are defined by the Fortran 2008 Standard and extended by Fortran 2018. These constructs support parallel programming using a Single Program Multiple Data (SPMD) model. These features are not available on macOS* systems.

NOTE
32-bit coarrays are deprecated and will be removed in a future release.

You must specify compiler option `[Q]coarray` to enable coarrays in your program.

A Fortran program containing coarrays is interpreted as if it were replicated a fixed number of times and all copies were executed asynchronously.

Each replication is called an "image", and each image has its own set of data objects. The number of images is set at run-time, but it can also be set by a compiler option or an environment variable.

The array syntax of Fortran is extended with additional trailing subscripts in square brackets to provide a clear representation of references to data that is spread across images. References without square brackets are to local data, so source code that can run independently is uncluttered. Any appearance of square brackets indicates communication between images.

Usually, each image resides on one processor. However, several images may share a processor and one image can execute on a cluster.

To reference any array variable that is coindexed, a subscript list must be present. If no subscript list is present, then the coindexed object must be a scalar.

Examples

Consider the following statement:

```
real, dimension(500), codimension[*] :: a,b
```

This statement declares two objects `a` and `b`, each as a coarray. A coarray always has the same shape on each image. In this case, each image has two real coarrays of size 500.

Consider that an image executes the following statement:

```
a(:) = b(:)[q]
```

In this case, the coarray `b` on image `q` is copied into coarray `a` on the executing image.

Consider the coindexed reference `x[k]`. If `x` is a rank 1 array, this reference to `x` on image `k` is incorrect syntax: a subscript list must be present. The correct form is `x(:)[k]` to access the entire array `x` on image `k`. If `x` is a scalar, then the syntax `x[k]` is correct.

See Also

Image Control Statements
Coarrays
Using Coarrays

Image Selectors

An image selector determines the image index for a coindexed object. It takes the following form:

lbracket cosubscript-list [, STAT=*stat-var*] *rbracket*

lbracket	Is a left bracket "[". This is required.
cosubscript	Is a scalar integer expression. Its value must be within the cobounds for its codimension. The number of cosubscripts must be equal to the corank of the object. If the lower bound is not specified, it is assumed to be 1.
stat-var	(Optional) Is a scalar integer variable with an exponent range of at least 4 (KIND=2 or greater). *stat-var* cannot be coindexed.
rbracket	Is a right bracket "]". This is required.

Considering the cobounds and bounds, respectively, the cosubscript list in an image selector determines the image index in the same way that a subscript list in an array element determines the subscript order value.

An image selector must specify an image index value that is not greater than the number of images.

When a statement containing an image selector with STAT= specified is executed, *stat-var* becomes defined with value STAT_FAILED_IMAGE defined in the intrinsic module ISO_FORTRAN_ENV if the object referenced by the image selector is on a failed image. Otherwise, *stat-var* becomes defined with the value zero.

A *stat-var* in an image selector cannot depend on the evaluation of any other entity in the same statement. An expression cannot depend on the value of any *stat-var* that is specified in the same statement. The value of *stat-var* cannot depend on execution of any part of the statement, other than if the object is on a failed image.

Examples

Assume that there are 16 images and the coarray C is declared as follows:

```
REAL :: C(15)[5,*]
```

C(:)[1,4] is valid because it selects image 16, but C(:)[2,4] is invalid because it selects image 17.

C(1)[2, 1, STAT=istatus] will cause istatus to become defined with the value STAT_FAILED_IMAGE if image 6 has failed; otherwise, istatus becomes defined with the value zero.

See Also

Coarrays
ISO_FORTRAN_ENV Module

Deferred-Coshape Coarrays

A deferred-coshape(allocatable) coarray has cobounds that are determined by allocation or argument association.

An allocatable coarray has the ALLOCATABLE attribute and a deferred-coshape specification.

A deferred-coshape specification is indicated by a colon (:).

The corank of an allocatable coarray is equal to the number of colons in its deferred-coshape specification.

The cobounds of an unallocated allocatable coarray are undefined. No part of such a coarray can be referenced or defined; however, the coarray may appear as an argument to an intrinsic inquiry function.

The cobounds of an allocated allocatable coarray are those specified when the coarray is allocated.

The cobounds of an allocatable coarray are unaffected by any subsequent redefinition or undefinition of the variables on which the cobounds' expressions depend.

See Also

Coarrays

Explicit-Coshape Coarrays

An explicit-coshape coarray is a named coarray that has its corank and cobounds declared by an explicit-coshape specification.

An explicit-coshape specification takes the following form:

[[*lower-cobound*:] *upper-cobound*,] ... [*lower-cobound*:] *

lc	Is a specification expression indicating the lower cobound of the coarray. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. If the lower bound is not specified, it is assumed to be 1.
uc	Is a specification expression indicating the upper cobound of the coarray. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. The upper cobound must not be less than the lower cobound (*lc*), if specified.

A nonallocatable coarray must have a coarray specification (coarray-spec) that is an explicit-coshape specification.

The corank is equal to one plus the number of upper cobounds.

A lower cobound or upper cobound that is not a constant expression must appear only in a subprogram or interface body.

If an explicit-coshape coarray is a local variable of a subprogram and has cobounds that are not constant expressions, the cobounds are determined on entry to a procedure defined by the subprogram, by evaluating the cobounds expressions . The cobounds of such a coarray are unaffected by the redefinition or undefinition of any variable during execution of the procedure.

The values of each lower cobound and upper cobound determine the cobounds of the coarray along a particular codimension. The cosubscript range of the coarray in that codimension is the set of integer values between and including the lower and upper cobounds.

Examples

The following are examples of variables with coarray specifications (coarray-specs):

```
A [*]
AA [2:*]
B [2, 3, *]
C [3:5, -2:7, *]
```

See Also

Coarrays

Referencing Coarray Images

Data on other images is normally referenced by cosubscripts enclosed in square brackets.

For coarray images, each set of cosubscripts maps to an image index, which is an integer between one and the number of images, in the same way as a set of array subscripts maps to a position in array element order.

To find the image index for each image, specify intrinsic function THIS_IMAGE() with no arguments.

To find the set of subscript indices that corresponds to the current image for a coarray z, specify THIS_IMAGE(z).

To find the image index that corresponds to a set of cosubscript indices sub for a coarray z, specify intrinsic function IMAGE_INDEX(z,sub).

To find the number of images, specify intrinsic function NUM_IMAGES.

See Also

THIS_IMAGE
IMAGE_INDEX
NUM_IMAGES

Specifying Data Objects for Coarray Images

Each image has its own set of data objects, all of which can be accessed in the normal way.

The rank, bounds, extents, size, and shape of a whole coarray are provided by the data in parentheses in its declaration or allocation. Its corank, cobounds, and coextents are provided by the data in square brackets in its declaration or allocation. Any subobject of the coarray that is a coarray has the same corank, cobounds, and coextents. The cosize of a coarray is always equal to the number of images.

Objects can be declared with codimensions in square brackets immediately following dimensions in parentheses or in place of them, for example:

```
real :: a[*], d[*]                              ! Scalar coarrays
real, dimension(50), codimension[50,*] :: x     ! An array coarray
integer :: n(100)[*]
character :: b(40)[40,0:*]
type(para) :: w[30,*]
```

Unless the coarray is allocatable, the form for the dimensions in square brackets is the same as that for dimensions in parentheses in an assumed-size array.

The total number of subscripts plus cosubscripts is limited to 31. Note that the Fortran 2008 Standard limits the total number of subscripts plus cosubscripts to 15.

You can address a coarray on another image by using subscripts in square brackets following any subscripts in parentheses, for example:

```
b(6)[4,7] = n(6)[4]
e[4] = c
b(:)[3,4] = c[2]
```

Any object whose designator includes square brackets is called a coindexed object. Each subscript in square brackets must be a scalar integer expression.

Subscripts in parentheses must be used whenever the parent array has nonzero rank. For example, for a(:)[2,3], you cannot specify a[2,3].

See Also

NUM_IMAGES

Variable-Definition Context

A variable can appear in different contexts that imply definition or undefinition of the variable. A reference to a function that returns a data pointer is permitted in such variable-definition contexts.

When a function returns a data pointer, that pointer is always associated with a variable that has the TARGET attribute, either by pointer assignment or by allocation. If a reference to a function that returns a data pointer appears in a variable-definition context, the definable target (with which the function result is associated) is the variable that becomes defined or undefined.

This section describes the different variable-definition contexts in which a function reference returning a data pointer can be used. It also describes the contexts in which data pointer function references are not allowed.

Assignment Statement

Function references returning a data pointer can be used on the left-hand side of an assignment statement. References to type bound and generic procedures are also permitted on the left-hand side of an intrinsic assignment. If the variable on the left-hand side is polymorphic, it must be an allocatable. Therefore, a function returning a polymorphic pointer cannot be used on the left-hand side.

In the following example, function STORAGE returns a data pointer to either the module variable OUTSIDE or to an element of VAR into which a value is stored:

```
MODULE TMOD
 PUBLIC

 INTEGER, PARAMETER :: N = 10
 INTEGER, TARGET    :: VAR(N)
 INTEGER, TARGET    :: OUTSIDE
 CONTAINS
 FUNCTION STORAGE(KEY) RESULT(LOC)
   INTEGER, INTENT(IN) :: KEY
   INTEGER, POINTER    :: LOC

   IF( KEY .LT. 1 .OR. KEY .GE. N ) THEN
     LOC=> OUTSIDE
   ELSE
     LOC => VAR(KEY)
   ENDIF
 END FUNCTION
END MODULE

PROGRAM MAIN
  USE TMOD
  OUTSIDE = -1
  STORAGE(1) = 11
  STORAGE(0) = 0
  PRINT *, VAR(1), OUTSIDE    ! prints 11, 0
END
```

The following example shows generic resolution on the left-hand side of an assignment statement:

```
MODULE MYMODULE
    TYPE :: VEC
        INTEGER :: X(3)
    CONTAINS
        GENERIC   :: GET => GETELEMENT, GETARRAY
        PROCEDURE :: GETELEMENT
        PROCEDURE :: GETARRAY
    END TYPE VEC
    CONTAINS
        FUNCTION GETELEMENT( THIS, EL ) RESULT( P )
            IMPLICIT NONE
            CLASS(VEC), TARGET :: THIS
            INTEGER, INTENT(IN) :: EL
            INTEGER, POINTER :: P
            P => THIS%X(EL)
        END FUNCTION GETELEMENT

        FUNCTION GETARRAY( THIS ) RESULT( P )
            IMPLICIT NONE
            CLASS(VEC), TARGET :: THIS
            INTEGER, POINTER :: P(:) ! array pointer
```

```
            P => THIS%X
        END FUNCTION GETARRAY
END MODULE MYMODULE

PROGRAM TEST
   USE MYMODULE
   IMPLICIT NONE
   TYPE(VEC) :: MYVEC
   INTEGER   :: Y(3)
   MYVEC%X = [1,2,3]
   Y = [6,7,8]

   ! expected output 1 2 1 2 3
   WRITE(6,*) MYVEC%GET(1), MYVEC%GET(2), MYVEC%GET()

   ! change any array element
   MYVEC%GET(1) = 4
   MYVEC%GET(2) = 5
   MYVEC%GET(3) = 6

   ! check modified values
   ! expected output 4 5 4 5 6
   WRITE(6,*) MYVEC%GET(1), MYVEC%GET(2), MYVEC%GET()

   MYVEC%GET() = Y          ! array pointer returned

   ! check modified values
   WRITE(6,*)  MYVEC%GET()  ! expected output 6 7 8
END PROGRAM TEST
```

Argument Association

A function reference returning a data pointer can be used as an actual argument in a reference to a procedure with an explicit interface. If the corresponding dummy argument has the INTENT (OUT) or INTENT (INOUT) attribute, then the pointer function is used in a variable definition context.

The following example uses the function STORAGE, which was defined in the above section "Assignment Statement":

```
FUNCTION STORAGE(KEY) RESULT(LOC)
   INTEGER, INTENT(IN) :: KEY
   INTEGER, POINTER    :: LOC
..
END FUNCTION
..
STORAGE(2) = 10
CALL CHANGE_VAL(STORAGE(2)) ! pass storage(2) as actual argument
PRINT *, VAR(2)             ! prints 50
..
SUBROUTINE CHANGE_VAL(X)
  INTEGER, INTENT(OUT) :: X
  X = X*5
END SUBROUTINE CHANGE_VAL
```

The following example shows that the target of the function pointer can get modified inside the subroutine without using the dummy argument corresponding to the function reference:

```
MODULE M200C2
  INTEGER, TARGET :: X = 42
CONTAINS
  FUNCTION FX()
    INTEGER, POINTER :: FX
    FX => X
  END FUNCTION
END MODULE

PROGRAM Q1
  USE M200C2
  CALL TEST(X, FX())
  ! note that corresponding dummy is not INTENT (OUT) or INTENT(INOUT).
  ! FX() is not used in a variable definition context but it still
  !   denotes a variable.
  CONTAINS
  SUBROUTINE TEST(A, B)
  INTEGER, TARGET :: B
    A = A*10
    PRINT *, A, B               !  prints 420 420
  END SUBROUTINE
END PROGRAM
```

SELECT RANK, SELECT TYPE, and ASSOCIATE Construct

A pointer function reference can appear as a variable that is the selector in a SELECT RANK, SELECT TYPE, or ASSOCIATE construct and the associate name of that construct can appear in a variable-definition context. For example:

```
PROGRAM MAIN
    INTEGER, TARGET :: DATA = 123

    ASSOCIATE (ALIAS => FX1())
        ALIAS = 456
        PRINT *, ALIAS, DATA     ! prints 456 456
    END ASSOCIATE

    SELECT TYPE (ALIAS =>FX2())
        TYPE IS (INTEGER)
          ALIAS = 789
          PRINT *, ALIAS, DATA   ! prints 789 789
    END SELECT

    CONTAINS
    FUNCTION FX1()
        INTEGER, POINTER :: FX1
        FX1 => DATA
    END FUNCTION FX1
    FUNCTION FX2()
        CLASS(*),POINTER :: FX2
        FX2 => DATA
    END FUNCTION FX2
END PROGRAM MAIN
```

In the following example, FX() in the ASSOCIATE is a variable and every reference to ALIAS is a reference to the associated variable, so the assignment also changes the value of ALIAS:

```
PROGRAM MAIN
    INTEGER, TARGET :: DATA = 123

    ASSOCIATE (ALIAS => FX1())
        DATA = 0
        PRINT *, ALIAS, DATA     ! prints 0 0
    END ASSOCIATE

    SELECT TYPE (ALIAS => FX2())
        TYPE IS (INTEGER)
          DATA = 1
          PRINT *, ALIAS, DATA   ! prints 1 1
    END SELECT

    CONTAINS

    FUNCTION FX1 ()
        INTEGER, POINTER :: FX1
        FX1 => DATA
    END FUNCTION FX1

    FUNCTION FX2()
        CLASS(*),POINTER :: FX2
        FX2 => DATA
    END FUNCTION FX2

END PROGRAM MAIN
```

Input/Output Statements

A pointer function reference can be used as an input item in a READ statement.

A function reference returning a character pointer can be used as an internal file variable in a WRITE statement.

A scalar integer pointer function reference can be an IOSTAT= or a SIZE= specifier in an input/output statement. A scalar character pointer function reference can be an IOMSG= specifier in an input/output statement.

A function returning a scalar pointer, whose datatype matches the specifier, can be specified in an INQUIRE statement except for the three specifiers FILE=, ID=, and UNIT=.

A function returning a scalar integer pointer can be a NEWUNIT= specifier in an OPEN statement.

Consider the following example:

```
...
    CHARACTER(50), TARGET :: V(33)
    INTEGER, TARGET :: I
    ..
    FUNCTION RET_CHAR(INDEX) RESULT (DCV)
      CHARACTER(50), POINTER :: DCV
      INTEGER :: INDEX
      DCV => V(INDEX)
    END FUNCTION
    FUNCTION RET_INT() RESULT (P)
      INTEGER, POINTER :: P
      P => I
```

```
    END FUNCTION
  ...
! an input item in a read stmt
    READ (6, *) RET_INT()
    READ  10, RET_INT()

! an internal file variable in a write stmt
    WRITE  (RET_CHAR(10), FMT=*) 666

! an IOSTAT=, SIZE=, or IOMSG= specifier in an I/O statement
    READ (10, FMT=*, IOSTAT=RET_INT(), SIZE=RET_INT(), &
          IOMSG=RET_CHAR(6) ) STR

! a specifier in an inquire statement except FILE=, ID=, and UNIT=
    OPEN(NEWUNIT = NUM, FILE = 'A.TXT', ACTION = 'READ')
    INQUIRE(NUM,                &
    ACCESS = RET_CHAR(2),       &
    EXIST = RET_CHAR(10),       &
    ID = 13,                    &
    IOSTAT = RET_CHAR(14),      &
    SIZE = RET_CHAR(30))
    CLOSE(NUM, STATUS = 'DELETE')

! a NEWUNIT= SPECIFIER in an OPEN statement
    OPEN(NEWUNIT = RET_INT(1), STATUS = 'SCRATCH')
    CLOSE(RET_INT(1), STATUS = 'DELETE') ! allowed on CLOSE
```

STAT=, ERRMSG=, and ACQUIRED_LOCK= Specifiers

A scalar integer pointer function reference can be used as a STAT= variable. A scalar character pointer function reference can be used as an ERRMSG= variable. STAT= and ERRMSG= are allowed in EVENT POST, EVENT WAIT, SYNC ALL, SYNC IMAGES, SYNC MEMORY, LOCK, UNLOCK, ALLOCATE, and DEALLOCATE statements. A scalar logical pointer function reference can be an ACQUIRED_LOCK= specifier in a LOCK statement.

The following example uses RET_CHAR and RET_INT, which were defined in the above section "Input/Output Statements":

```
    TYPE(EVENT_TYPE) :: ET[*]
    TYPE(LOCK_TYPE) :: LT[*]
    INTEGER, POINTER     :: AR(:)
    ALLOCATE(AR(2), STAT=RET_INT())
    DEALLOCATE(AR, STAT=RET_INT())

    EVENT POST (EV[THIS_IMAGE() + 1], STAT=RET_INT(), ERRMSG=RET_CHAR())
    EVENT WAIT(EV, STAT=RET_INT(), ERRMSG=RET_CHAR())
    LOCK(LT, ACQUIRED_LOCK=GET_LOGICAL(), STAT=RET_INT(), &
    ERRMSG=RET_CHAR())
    UNLOCK(LT, STAT=RET_INT(), ERRMSG=RET_CHAR())
    SYNC IMAGES(*,STAT=RET_INT(), ERRMSG=RET_CHAR())
    SYNC ALL(STAT=RET_INT(), ERRMSG=RET_CHAR())
    SYNC MEMORY(STAT=RET_INT(), ERRMSG=RET_CHAR())
```

Execution of EVENT POST and EVENT WAIT statements

An event variable of type EVENT_TYPE from the ISO_FORTRAN_ENV module becomes defined by the successful execution of an EVENT POST or an EVENT WAIT statement.

Execution of a LOCK or UNLOCK statement

A lock variable of LOCK_TYPE from the ISO_FORTRAN_ENV module becomes defined by the successful execution of an UNLOCK statement, or a LOCK statement without an ACQUIRED_LOCK= specifier. Successful execution of a LOCK statement with an ACQUIRED_LOCK= specifier causes the specified logical variable to become defined. If it is defined with the value true, the lock variable in the LOCK statement also becomes defined.

Disallowed Contexts

The Fortran Standard defines both a "variable" and a "variable name". For function F, F is a variable name; F(7) is a function. If F returns a data pointer, F(7) is a variable and can be used in a variable-definition context.

For the following variable-definition contexts, the Fortran Standard specifies that a "variable name" must be used and not a "variable":

- The pointer object in a NULLIFY statement
- A data pointer object or procedure pointer object in a pointer assignment statement
- The DO variable in a DO statement or an implied DO construct
- A variable name in a NAMELIST statement if the NAMELIST group name appears in a NML= specifier in a READ statement
- The object in an ALLOCATE or DEALLOCATE statement
- An event variable in an EVENT POST or EVENT WAIT statement
- The lock variable in a LOCK or UNLOCK statement

A function reference can return a pointer to any data object, even one that cannot be stored into, for example, a USE associated PROTECTed object or a constant. This will not be caught at compile time. It is possible that the target of the pointer function is a local variable from a different subprogram or a private USE associated variable, in which case the pointer returned has an undefined association status.

A More Complex Example

The following example has pointer functions that return data pointers to parameterized derived type objects. The pointer function results are automatic objects whose length type parameters depend on the dummy arguments:

```
MODULE TMOD
  PUBLIC
  TYPE PDT(K, L)
    INTEGER, KIND :: K
    INTEGER, LEN :: L
    INTEGER :: FIELD(L)
  END TYPE PDT
END MODULE
PROGRAM MAIN
    USE TMOD
    IMPLICIT NONE
    TYPE(PDT(4,2)), TARGET :: PDTOBJ1, OBJ
    TYPE(PDT(2,2)), POINTER :: ACTARG
    CHARACTER(10), TARGET :: C1

    BAR() = PDT(4,2)((/5,3/))
    PRINT *, PDTOBJ1%FIELD            ! prints 5 3

    AUTO_RES(ACTARG) = PDT(4,2)((/6,4/))
    PRINT *, PDTOBJ1%FIELD            ! prints 6 4

    AUTO_CHAR(10) = "TEST"
```

```
    PRINT *, C1                       ! prints TEST
    CONTAINS
    FUNCTION BAR() RESULT(LOC)
        TYPE(PDT(4,2)), POINTER :: LOC
        LOC => PDTOBJ1
    END FUNCTION
    FUNCTION AUTO_CHAR(DUM1) RESULT(LOC)
         INTEGER, INTENT(IN) :: DUM1
         CHARACTER(DUM1), POINTER :: LOC
         LOC => C1
    END FUNCTION

    FUNCTION AUTO_RES(DUM1) RESULT(LOC)
         TYPE(PDT(4,:)), POINTER, INTENT(IN) :: DUM1
         TYPE(PDT(4,DUM1%L)), POINTER :: LOC
         LOC => PDTOBJ1
    END FUNCTION
END PROGRAM
```

Expressions and Assignment Statements

An expression represents either a data reference or a computation, and is formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.

An assignment causes variables to be defined or redefined.

For more information, see the individual topics in this section.

Expressions

An expression represents either a data reference or a computation, and is formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.

If the value of an expression is of intrinsic type, it has a kind type parameter. (If the value is of intrinsic type CHARACTER, it also has a length parameter.) If the value of an expression is of derived type, it has no kind type parameter.

An operand is a scalar or array. An operator can be either intrinsic or defined. An intrinsic operator is known to the compiler and is always available to any program unit. A defined operator is described explicitly by a user in a function subprogram and is available to each program unit that uses the subprogram.

The simplest form of an expression (a primary) can be any of the following:

- A constant; for example, 4.2
- A subobject of a constant; for example, 'LMNOP' (2:4)
- A variable; for example, VAR_1
- A structure constructor; for example, EMPLOYEE(3472, "JOHN DOE")
- An array constructor; for example, (/12.0,16.0/)
- A function reference; for example, COS(X)
- Another expression in parentheses; for example, (I+5)

Any variable or function reference used as an operand in an expression must be defined at the time the reference is executed. If the operand is a pointer, it must be associated with a target object that is defined. An integer operand must be defined with an integer value rather than a statement label value. All of the characters in a character data object reference must be defined.

When a reference to an array or an array section is made, all of the selected elements must be defined. When a structure is referenced, all of the components must be defined.

In an expression that has intrinsic operators with an array as an operand, the operation is performed on each element of the array. In expressions with more than one array operand, the arrays must be conformable (they must have the same shape). The operation is applied to corresponding elements of the arrays, and the result is an array of the same shape (the same rank and extents) as the operands.

In an expression that has intrinsic operators with a pointer as an operand, the operation is performed on the value of the target associated with the pointer.

For defined operators, operations on arrays and pointers are determined by the procedure defining the operation.

A scalar is conformable with any array. If one operand of an expression is an array and another operand is a scalar, it is as if the value of the scalar were replicated to form an array of the same shape as the array operand. The result is an array of the same shape as the array operand.

The following sections describe numeric, character, relational, and logical expressions; defined operations; a summary of operator precedence; and initialization and specification expressions.

See Also

Arrays
Derived data types
Defining Generic Operators for details on function subprograms that define operators
POINTER statement for details on pointers

Numeric Expressions

Numeric expressions express numeric computations, and are formed with numeric operands and numeric operators. The evaluation of a numeric operation yields a single numeric value.

The term *numeric* includes logical data, because logical data is treated as integer data when used in a numeric context. The default for .TRUE. is -1; .FALSE. is 0. Note that the default can change if compiler option fpscomp logicals is used.

Numeric operators specify computations to be performed on the values of numeric operands. The result is a scalar numeric value or an array whose elements are scalar numeric values. The following are numeric operators:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus (identity)
-	Subtraction or unary minus (negation)

Unary operators operate on a single operand. *Binary operators* operate on a pair of operands. The plus and minus operators can be unary or binary. When they are unary operators, the plus or minus operators precede a single operand and denote a positive (identity) or negative (negation) value, respectively. The exponentiation, multiplication, and division operators are binary operators.

Valid numeric operations must have results that are defined by the arithmetic used by the processor. For example, raising a negative-valued base to a real power is invalid.

Numeric expressions are evaluated in an order determined by a precedence associated with each operator, as follows (see also Summary of Operator Precedence):

Operator	Precedence
**	Highest
* and /	.
Unary + and -	.
Binary + and -	Lowest

Operators with equal precedence are evaluated in left-to-right order. However, exponentiation is evaluated from right to left. For example, A**B**C is evaluated as A**(B**C). B**C is evaluated first, then A is raised to the resulting power.

Normally, two operators cannot appear together. However, Intel® Fortran allows two consecutive operators if the second operator is a plus or minus.

Examples

In the following example, the exponentiation operator is evaluated first because it takes precedence over the multiplication operator:

`A**B*C` is evaluated as `(A**B)*C`

Ordinarily, the exponentiation operator would be evaluated first in the following example. However, because Intel Fortran allows the combination of the exponentiation and minus operators, the exponentiation operator is not evaluated until the minus operator is evaluated:

`A**-B*C` is evaluated as `A**(-(B*C))`

Note that the multiplication operator is evaluated first, since it takes precedence over the minus operator.

When consecutive operators are used with constants, the unary plus or minus before the constant is treated the same as any other operator. This can produce unexpected results. In the following example, the multiplication operator is evaluated first, since it takes precedence over the minus operator:

`X/-15.0*Y` is evaluated as `X/-(15.0*Y)`

See Also

fpscomp compiler option

Using Parentheses in Numeric Expressions

You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, that part is evaluated first. The resulting value is used in the evaluation of the remainder of the expression.

In the following examples, the numbers below the operators indicate a possible order of evaluation. Alternative evaluation orders are possible in the first three examples because they contain operators of equal precedence that are not enclosed in parentheses. In these cases, the compiler is free to evaluate operators of equal precedence in any order, as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation.

```
    4 + 3 * 2 - 6/2   = 7
      ^   ^   ^  ^
      2   1   4  3
    (4 + 3) * 2 - 6/2   = 11
       ^    ^   ^  ^
       1    2   4  3
    (4 + 3 * 2 - 6)/2   = 2
       ^   ^   ^   ^
       2   1   3   4
```

```
((4 + 3) * 2 - 6)/2   = 4
    ^    ^   ^   ^
    1    2   3   4
```

Expressions within parentheses are evaluated according to the normal order of precedence. In expressions containing nested parentheses, the innermost parentheses are evaluated first.

Nonessential parentheses do not affect expression evaluation, as shown in the following example:

```
4 + (3 * 2) - (6/2)
```

However, using parentheses to specify the evaluation order is often important in high-accuracy numerical computations. In such computations, evaluation orders that are algebraically equivalent may not be computationally equivalent when processed by a computer (because of the way intermediate results are rounded off).

Parentheses can be used in argument lists to force a given argument to be treated as an expression, rather than as the address of a memory item.

Data Type of Numeric Expressions

If every operand in a numeric expression is of the same data type, the result is also of that type.

If operands of different data types are combined in an expression, the evaluation of that expression and the data type of the resulting value depend on the ranking associated with each data type. The following table shows the ranking assigned to each data type:

Data Type	Ranking
LOGICAL(1) and BYTE	Lowest
LOGICAL(2)	.
LOGICAL(4)	.
LOGICAL(8)	.
INTEGER(1)	.
INTEGER(2)	.
INTEGER(4)	.
INTEGER(8)	.
REAL(4)	.
REAL(8)[1]	.
REAL(16)	.
COMPLEX(4)	.
COMPLEX(8)[2]	.
COMPLEX(16)	Highest

[1] DOUBLE PRECISION

[2] DOUBLE COMPLEX

The data type of the value produced by an operation on two numeric operands of different data types is the data type of the highest-ranking operand in the operation. For example, the value resulting from an operation on an integer and a real operand is of real type. However, an operation involving a COMPLEX(4) or COMPLEX(8) data type and a DOUBLE PRECISION data type produces a COMPLEX(8) result.

The data type of an expression is the data type of the result of the last operation in that expression, and is determined according to the following conventions:

- Integer operations: Integer operations are performed only on integer operands. (Logical entities used in a numeric context are treated as integers.) In integer arithmetic, any fraction resulting from division is truncated, not rounded. For example, the result of `9/10` is 0, not 1.
- Real operations: Real operations are performed only on real operands or combinations of real, integer, and logical operands. Any integer operands present are converted to real data type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. However, in the statement `Y = (I /J)*X`, an integer division operation is performed on I and J, and a real multiplication is performed on that result and X.

  If one operand is a higher-precision real (REAL(8) or REAL(16)) type, the other operand is converted to that higher-precision real type before the expression is evaluated.

  When a single-precision real operand is converted to a double-precision real operand, low-order binary digits are set to zero. This conversion does not increase accuracy; conversion of a decimal number does not produce a succession of decimal zeros. For example, a REAL variable having the value `0.3333333` is converted to approximately `0.3333331346511184D0`. It is not converted to either `0.3333330000000000D0` or `0.3333333333333333D0`.
- Complex operations: In operations that contain any complex operands, integer operands are converted to real type, as previously described. The resulting single-precision or double-precision operand is designated as the real part of a complex number and the imaginary part is assigned a value of zero. The expression is then evaluated using complex arithmetic and the resulting value is of complex type. Operations involving a COMPLEX(4) or COMPLEX(8) operand and a DOUBLE PRECISION operand are performed as COMPLEX(8) operations; the DOUBLE PRECISION operand is not rounded.

These rules also generally apply to numeric operations in which one of the operands is a constant. However, if a real or complex constant is used in a higher-precision expression, additional precision will be retained for the constant. The effect is as if a DOUBLE PRECISION (REAL(8)) or REAL(16) representation of the constant were given. For example, the expression `1.0D0 + 0.3333333` is treated as if it is `1.0D0 + dble(0.3333333)`.

Character Expressions

A character expression consists of a character operator (//) that concatenates two operands of type character. The evaluation of a character expression produces a single value of that type.

The result of a character expression is a character string whose value is the value of the left character operand concatenated to the value of the right operand. The length of a character expression is the sum of the lengths of the values of the operands. For example, the value of the character expression `'AB'//'CDE'` is `'ABCDE'`, which has a length of five.

Parentheses do not affect the evaluation of a character expression; for example, the following character expressions are equivalent:

```
('ABC'//'DE')//'F'
'ABC'//('DE'//'F')
'ABC'//'DE'//'F'
```

Each of these expressions has the value `' ABCDEF'`.

If a character operand in a character expression contains blanks, the blanks are included in the value of the character expression. For example, `'ABC '//'D E'//'F '` has a value of `'ABC D EF '`.

Relational Expressions

A *relational expression* consists of two or more expressions whose values are compared to determine whether the relationship stated by the relational operator is satisfied. The following are relational operators:

Operator	Relationship
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or ==	Equal to
.NE. or /=	Not equal to
.GT. or >	Greater than
.GE. or >=	Greater than or equal to

The result of the relational expression is .TRUE. if the relation specified by the operator is satisfied; the result is .FALSE. if the relation specified by the operator is not satisfied.

Relational operators are of equal precedence. Numeric operators and the character operator // have a higher precedence than relational operators.

In a numeric relational expression, the operands are numeric expressions. Consider the following example:

```
APPLE+PEACH > PEAR+ORANGE
```

This expression states that the sum of APPLE and PEACH is greater than the sum of PEAR and ORANGE. If this relationship is valid, the value of the expression is .TRUE.; if not, the value is .FALSE.

Operands of type complex can only be compared using the equal operator (== or .EQ.) or the not equal operator (/= or .NE.). Complex entities are equal if their corresponding real and imaginary parts are both equal.

In a character relational expression, the operands are character expressions. In character relational expressions, less than (< or .LT.) means the character value precedes in the ASCII collating sequence, and greater than (> or .GT.) means the character value follows in the ASCII collating sequence. For example:

```
'AB'//'ZZZ' .LT. 'CCCCC'
```

This expression states that `'ABZZZ'` is less than `'CCCCC'`. In this case, the relation specified by the operator is satisfied, so the result is .TRUE..

Character operands are compared one character at a time, in order, starting with the first character of each operand. If the two character operands are not the same length, the shorter one is padded on the right with blanks until the lengths are equal; for example:

```
'ABC' .EQ. 'ABC  '
'AB' .LT. 'C'
```

The first relational expression has the value .TRUE. even though the lengths of the expressions are not equal, and the second has the value .TRUE. even though `'AB'` is longer than `'C'`.

A relational expression can compare two numeric expressions of different data types. In this case, the value of the expression with the lower-ranking data type is converted to the higher-ranking data type before the comparison is made.

See Also

Data Type of Numeric Expressions

Logical Expressions

A logical expression consists of one or more logical operators and logical, numeric, or relational operands. The following are logical operators:

Operator	Example	Meaning
.AND.	`A .AND. B`	Logical conjunction: the expression is true if both A and B are true.
.OR.	`A .OR. B`	Logical disjunction (inclusive OR): the expression is true if either A, B, or both, are true.
.NEQV.	`A .NEQV. B`	Logical inequivalence (exclusive OR): the expression is true if either A or B is true, but false if both are true.
.XOR.	`A .XOR. B`	Same as .NEQV.
.EQV.	`A .EQV. B`	Logical equivalence: the expression is true if both A and B are true, or both are false.
.NOT.[1]	`.NOT. A`	Logical negation: the expression is true if A is false and false if A is true.

[1] .NOT. is a unary operator.

Periods cannot appear consecutively except when the second operator is .NOT. For example, the following logical expression is valid:

```
A+B/(A-1) .AND. .NOT. D+B/(D-1)
```

Data Types Resulting from Logical Operations

Logical operations on logical operands produce single logical values (.TRUE. or .FALSE.) of logical type.

Logical operations on integers produce single values of integer type. The operation is carried out bit-by-bit on corresponding bits of the internal (binary) representation of the integer operands.

Logical operations on a combination of integer and logical values also produce single values of integer type. The operation first converts logical values to integers, then operates as it does with integers.

Logical operations cannot be performed on other data types.

Evaluation of Logical Expressions

Logical expressions are evaluated according to the precedence of their operators. Consider the following expression:

```
A*B+C*ABC == X*Y+DM/ZZ .AND. .NOT. K*B > TT
```

This expression is evaluated in the following sequence:

```
(((A*B)+(C*ABC)) == ((X*Y)+(DM/ZZ))) .AND. (.NOT. ((K*B) > TT))
```

As with numeric expressions, you can use parentheses to alter the sequence of evaluation.

When operators have equal precedence, the compiler can evaluate them in any order, as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation (except for exponentiation, which is evaluated from right to left).

You should not write logical expressions whose results might depend on the evaluation order of subexpressions. The compiler is free to evaluate subexpressions in any order. In the following example, either (A(I)+1.0) or B(I)*2.0 could be evaluated first:

```
(A(I)+1.0) .GT. B(I)*2.0
```

Some subexpressions might not be evaluated if the compiler can determine the result by testing other subexpressions in the logical expression. Consider the following expression:

```
A .AND. (F(X,Y) .GT. 2.0) .AND. B
```

If the compiler evaluates A first, and A is false, the compiler might determine that the expression is false and might not call the subprogram F(X,Y).

See Also

Summary of Operator Precedence

Defined Operations

When operators are defined for functions, the functions can then be referenced as defined operations.

The operators are defined by using a generic interface block specifying OPERATOR, followed by the defined operator (in parentheses).

A defined operation is not an intrinsic operation. However, you can use a defined operation to extend the meaning of an intrinsic operator.

For defined unary operations, the function must contain one argument. For defined binary operations, the function must contain two arguments.

Interpretation of the operation is provided by the function that defines the operation.

A Standard Fortran defined operator can contain up to 31 letters, and is enclosed in periods (.). Its name cannot be the same name as any of the following:

- The intrinsic operators (.NOT., .AND., .OR., .XOR., .EQV., .NEQV., .EQ., .NE., .GT., .GE., .LT., and .LE.)
- The logical literal constants (.TRUE. or .FALSE.)

An intrinsic operator can be followed by a defined unary operator.

The result of a defined operation can have any type. The type of the result (and its value) must be specified by the defining function.

Examples

The following examples show expressions containing defined operators:

```
.COMPLEMENT. A
X .PLUS. Y .PLUS. Z
M * .MINUS. N
```

See Also

Defining Generic Operators
Summary of Operator Precedence

Summary of Operator Precedence

The following table shows the precedence of all intrinsic and defined operators:

Precedence of Expression Operators

Category	Operator	Precedence
	Defined Unary Operators	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	Unary + or -	.
Numeric	Binary + or -	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., ==, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.XOR., .EQV., .NEQV.	.
	Defined Binary Operators	Lowest

Constant and Specification Expressions

A constant expression is an expression that is evaluated when a program is compiled. It can be used as a kind type parameter, a named constant, or to specify an initial value for an entity.

A specification expression is a scalar, integer expression that is restricted specifications such as length type parameters and array bounds.

Constant and specification expressions can appear in specification statements, with some restrictions.

Constant Expressions

A constant expression is an expression that you can use as a kind type parameter, a named constant, or to specify an initial value for an entity. It is evaluated when a program is compiled.

In a constant expression, each operation is intrinsic and each primary is one of the following:

- A constant or subobject of a constant
- A specification inquiry where each designator or function argument is one of the following:
  - A constant expression
  - A variable whose properties inquired about are not assumed, deferred, or defined by an expression that is not a constant expression
- A reference to the transformational function IEEE_SELECTED_REAL_KIND from the intrinsic module IEEE_ARITHMETIC, where each argument is a constant expression
- A kind type parameter of the derived type being defined
- A DO variable within an array constructor where each scalar integer expression of the corresponding DO loop is an constant expression
- Another constant expression enclosed in parentheses, where each subscript, section subscript, substring starting and ending point, and type parameter value is a constant expression

If a constant expression invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement (or to the left of the inquiry function in the same statement). The previous specification cannot be in the same entity declaration.

If a reference to a generic entity is included in a constant expression that is in the specification part of a module or submodule, that generic entity shall have no specific procedures defined subsequent to the constant expression in the module or submodule.

Examples

Valid constant Expressions

`-1 + 3`	
`SIZE(B)`	! B is a named constant
`7_2`	
`INT(J, 4)`	! J is a named constant
`SELECTED_INT_KIND (2)`	

Invalid constant Expressions

`SUM(A)`	Not an allowed function.
`A/4.1 - K**1.2`	Exponential does not have integer power (A and K are named constants).
`HUGE(4.0)`	Argument is not an integer.

See Also

Array constructors
Specification Expressions
Structure constructors
Intrinsic procedures

Specification Expressions

A specification expression is a restricted scalar integer expression that you can use in specifications such as length type parameters and array bounds. Unless a specification expression is in an interface body, the specification part of a subprogram or BLOCK construct, a derived type definition, or the declaration type spec of a FUNCTION statement, it must be a constant expression.

In a restricted expression, each operation is intrinsic or defined by a specification function and each primary is one of the following:

- A constant or subobject of a constant
- An object designator with a base object that is one of the following:
  - A dummy argument that does not have the OPTIONAL or INTENT (OUT) attribute
  - In a common block
  - Made accessible by use or host association
  - A local variable of the procedure containing the BLOCK construct in which the restricted expression appears
  - A local variable of an outer BLOCK construct containing the BLOCK construct in which the restricted expression appears
  - An array constructor where each element and each scalar integer expression of each DO loop is a restricted expression
- A structure constructor whose components are restricted expression
- A specification inquiry where each designator or function argument is a restricted expression or a variable whose properties inquired about are not one of the following:
  - Dependent on the upper bound of the last dimension of an assumed-size array
  - Deferred
  - Defined by an expression that is not a restricted expression

- A specification inquiry that is a constant expression
- A reference to the PRESENT intrinsic function
- A reference to any other intrinsic function where each argument is a restricted expression
- A reference to an intrinsic transformational function from the intrinsic module ISO_C_BINDING where each argument is a restricted expression
- A reference to a specification function where each argument is a restricted expression
- A type parameter of the derived type being defined
- A DO variable within an array constructor, where each scalar integer expression of the corresponding implied-DO is a restricted expression
- A restricted expression enclosed in parentheses, where each subscript, section subscript, substring starting and ending point, and type parameter value is a restricted expression, and where any final subroutine that is invoked is pure

A specification inquiry is a reference to one of the following:

- An intrinsic inquiry function other than PRESENT
- A type parameter inquiry
- An intrinsic inquiry function from the modules IEEE_ARITHMETIC or IEEE_EXCEPTIONS
- The function C_SIZEOF from the intrinsic module ISO_C_BINDING
- The COMPILER_VERSION or COMPILER_OPTIONS inquiry function from the intrinsic module ISO_FORTRAN_ENV

Specification functions can be used in specification expressions to determine the attributes of data objects.

A function is a specification function if it is a pure function, does not have a dummy procedure argument, and is *not* one of the following:

- An standard intrinsic function
- An internal function
- A statement function
- A function with a dummy procedure argument

Evaluation of a specification expression must not directly or indirectly cause invocation of a procedure defined by the subprogram in which it appears.

NOTE
The requirement that specification functions be pure ensures that they cannot have side effects that could affect other objects being declared in the same specification.

The restriction that specification functions cannot be internal ensures that they cannot use host association to inquire about other objects being declared in the same specification. The restriction against recursion prevents the creation of a new instance of a procedure during construction of that procedure.

A variable in a specification expression must have its type and type parameters (if any) specified in one of the following ways:

- By a previous declaration in the same scoping unit
- By the implicit typing rules currently in effect for the scoping unit
- By host or use association

If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implied type and type parameters.

If a specification expression includes a specification inquiry that depends on a type parameter or an array bound or cobound of an entity specified in the same specification statement, the type parameter or array bound or cobound must be specified in a previous specification statement (or to the left of the inquiry function in the same statement). The previous specification cannot be in the same entity declaration. If a specification expression includes a reference to the value of an element of an array specified in the same specification statement, the array must be completely specified in previous declarations.

In the specification part of a module or submodule, if a specification expression includes a reference to a generic entity, that generic entity must have no specific procedures defined in the module or submodule subsequent to the specification expression.

In a specification expression, the number of arguments for a function reference is limited to 255.

Examples

The following shows valid specification expressions:

```
POPCNT(I) + J          ! I and J are scalar integer variables
UBOUND(ARRAY_B,20)  ! ARRAY_B is an assumed-shape dummy array
```

See Also

Array constructors
Structure constructors
Constant Expressions
Intrinsic procedures
Implicit typing rules
Use and host association
PURE procedures

Assignments

An assignment causes variables to be defined or redefined.

Assignment is specified by the following:

- An intrinsic assignment statement

  This lets you assign a value to a nonpointer variable.
- A defined assignment statement

  This lets you specify an assignment operation.
- pointer assignment

  This lets you associate a pointer with a target.
- masked array assignment

  This kind of assignment is denoted by a WHERE construct or statement. It lets you perform an array operation on selected elements in an array.
- element array assignment

  This kind of assignment is denoted by a FORALL construct or statement. It is a generalization of masked array assignment. It allows more general array shapes to be assigned, especially if it is used in construct form.

Note that the ASSIGN statement assigns a label to an integer variable. It is discussed elsewhere.

Intrinsic Assignment Statements

Intrinsic assignment is used to assign a value to a nonpointer variable. In the case of pointers, intrinsic assignment is used to assign a value to the target associated with the pointer variable. The value assigned to the variable (or target) is determined by evaluation of the expression to the right of the equal sign.

An intrinsic assignment statement takes the following form:

variable = expression

variable	Is the name of a scalar or array of intrinsic or derived type (with no defined assignment). The array cannot be an assumed-size array, and neither the scalar nor the array can be declared with the PARAMETER or INTENT(IN) attribute.

expression	Is of intrinsic type or the same derived type as *variable*. Its shape must conform with *variable*. If necessary, it is converted to the same type and kind as *variable*.

Description

Before a value is assigned to the variable, the expression part of the assignment statement and any expressions within the variable are evaluated. No definition of expressions in the variable can affect or be affected by the evaluation of the expression part of the assignment statement.

> **NOTE**
> When the run-time system assigns a value to a scalar integer or character variable and the variable is shorter than the value being assigned, the assigned value may be truncated and significant bits (or characters) lost. This truncation can occur without warning, and can cause the run-time system to pass incorrect information back to the program.

The following rules apply to intrinsic assignment statements:

- The variable and expression must be conformable unless the variable is an allocatable array that has the same rank as the expression, and is neither a coarray nor a coindexed object.
- If the expression is an array, then the variable must also be an array.
- If the variable is polymorphic, it must be allocatable and it must not be a coarray. The polymorphic variable must be type compatible with the expression and of the same rank. If it is allocated but the dynamic type differs from that of the expression, it is deallocated. If it is not allocated or becomes deallocated, it is allocated with the dynamic type of the expression.
- If the variable is a pointer, it must be associated with a definable target. The shape of the target and expression must conform and their type and kind parameters must match.
- If the variable is of type character, the expression must have the same kind type parameter.
- If the variable is of derived type:
  - Each kind type parameter of the variable must have the same value as the corresponding kind type parameter of the expression.
  - Each length type parameter of the variable must have the same value as the corresponding type parameter of the expression unless the variable is an allocatable array and its corresponding type parameter is deferred.
- If the variable is a coindexed object, each deferred-length type parameter must have the same value as the corresponding type parameter of the expression. Also, the variable must not be polymorphic, and it must not have an allocatable ultimate component.

The following sections discuss numeric, logical, character, derived- type, and array intrinsic assignment.

See Also

Arrays
Pointers
Derived data types
Defining Generic Assignment for details on subroutine subprograms that define assignment
Examples of Intrinsic Assignment to Polymorphic Variables

Numeric Assignment Statements

For numeric assignment statements, the variable and expression must be numeric type. The expression may also be of logical type.

The expression must yield a value that conforms to the range requirements of the variable. For example, a real expression that produces a value greater than 32767 is invalid if the entity on the left of the equal sign is an INTEGER(2) variable.

Significance can be lost if an INTEGER(4) value, which can exactly represent values of approximately the range -2*10**9 to +2*10**9, is converted to REAL(4) (including the real part of a complex constant), which is accurate to only about seven digits.

If the variable has the same data type as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the variable before it is assigned.

The following table summarizes the data conversion rules for numeric assignment statements.

Conversion Rules for Numeric Assignment Statements

Scalar Memory Reference (V)	Expression (E)	
	Integer or Real	**Complex**
Integer	V=INT(E)	V=INT(REAL(E)) Imaginary part of E is not used.
REAL (KIND=4)	V=REAL(E)	V=REAL(REAL(E)) Imaginary part of E is not used.
REAL (KIND=8)	V=DBLE(E)	V=DBLE(REAL(E)) Imaginary part of E is not used.
REAL (KIND=16)	V=QEXT(E)	V=QEXT(REAL(E)) Imaginary part of E is not used.
COMPLEX (KIND=4)	V=CMPLX(REAL(E), 0.0)	V=CMPLX(REAL(REAL(E)), REAL(AIMAG(E)))
COMPLEX (KIND=8)	V=CMPLX(DBLE(E), 0.0)	V=CMPLX(DBLE(REAL(E)), DBLE(AIMAG(E)))
COMPLEX (KIND=16)	V=CMPLX(QEXT(E), 0.0)	V=CMPLX(QEXT(REAL(E)), QEXT(AIMAG(E)))

If the expression (E) is of type logical, it is first converted to type integer as follows:

- If E evaluates to .TRUE. the result is -1 or 1 depending on the setting of the compiler option `fpscomp logicals`
- Otherwise the result is 0

The result of this conversion is then interpreted according to the above table.

Examples

Valid Numeric Assignment Statements

```
BETA = -1./(2.*X)+A*A /(4.*(X*X))

PI = 3.14159

SUM = SUM + 1.

ARRAY_A = ARRAY_B + ARRAY_C + SCALAR_I        ! Valid if all arrays conform in shape
```

Invalid Numeric Assignment Statements

```
3.14 = A - B                                  Entity on the left must be a variable.

ICOUNT = A//B(3:7)                            Implicitly typed data types do not match.
```

SCALAR_I = ARRAY_A(:)	Shapes do not match.

See Also

INT
REAL
DBLE
QEXT
CMPLX
AIMAG

Logical Assignment Statements

For logical assignment statements, the variable must be of logical type and the expression can be of logical or numeric type.

If the expression is of numeric type, it is converted to integer if necessary, then the value is interpreted as .TRUE. or .FALSE. according to the setting of the compiler option `fpscomp logicals`.

Examples

The following examples demonstrate valid logical assignment statements:

```
PAGEND = .FALSE.
PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND
ABIG = A.GT.B .AND. A.GT.C .AND. A.GT.D
LOGICAL_VAR = 123 ! Assigns .TRUE. to LOGICAL_VAR
```

Character Assignment Statements

For character assignment statements, the variable and expression must be of character type and have the same kind parameter.

The variable and expression can have different lengths. If the length of the expression is greater than the length of the variable, the character expression is truncated on the right. If the length of the expression is less than the length of the variable, the character expression is filled on the right with blank characters.

If you assign a value to a character substring, you do not affect character positions in any part of the character scalar variable not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged. If the character position is undefined, it remains undefined.

Examples

Valid Character Assignment Statements. (All variables are of type character.)

```
FILE = 'PROG2'

REVOL(1) = 'MAR'//'CIA'

LOCA(3:8) = 'PLANT5'

TEXT(I,J+1)(2:N-1) = NAME//X
```

Invalid Character Assignment Statements

`'ABC' = CHARS`	Left element must be a character variable, array element, or substring reference.
`CHARS = 25`	Expression does not have a character data type.

`STRING=5HBEGIN`	Expression does not have a character data type. (Hollerith constants are numeric, not character.)

Derived-Type Assignment Statements

In derived-type assignment statements, the variable and expression must be of the same derived type. There must be no accessible interface block with defined assignment for objects of this derived type.

The derived-type assignment is performed as if each component of the expression is assigned to the corresponding component of the variable. Pointer assignment is performed for pointer components, and intrinsic assignment is performed for nonpointer components.

Examples

The following example shows derived-type assignment:

```
TYPE DATE
  LOGICAL(1) DAY, MONTH
  INTEGER(2) YEAR
END TYPE DATE

TYPE(DATE) TODAY, THIS_WEEK(7)
TYPE APPOINTMENT
...
  TYPE(DATE) APP_DATE
END TYPE

TYPE(APPOINTMENT) MEETING

DO I = 1,7
  CALL GET_DATE(TODAY)
  THIS_WEEK(I) = TODAY
END DO
MEETING%APP_DATE = TODAY
```

See Also

Derived types
Pointer assignments

Array Assignment Statements

Array assignment is permitted when the array expression on the right has the same shape as the array variable on the left, or the expression on the right is a scalar.

If the expression is a scalar, and the variable is an array, the scalar value is assigned to every element of the array.

If the expression is an array, the variable must also be an array. The array element values of the expression are assigned (element by element) to corresponding elements of the array variable.

A *many-one array section* is a vector-valued subscript that has two or more elements with the same value. In intrinsic assignment, the variable cannot be a many-one array section because the result of the assignment is undefined.

Examples

In the following example, X and Y are arrays of the same shape:

```
  X = Y
```

The corresponding elements of Y are assigned to those of X element by element; the first element of Y is assigned to the first element of X, and so forth. The processor can perform the element-by-element assignment in any order.

The following example shows a scalar assigned to an array:

```
B(C+1:N, C) = 0
```

This sets the elements B (C+1,C), B (C+2,C),...B (N,C) to zero.

The following example causes the values of the elements of array A to be reversed:

```
REAL A(20)
...
A(1:20) = A(20:1:-1)
```

See Also

Arrays
Array constructors
WHERE for details on masked array assignment
FORALL for details on element array assignment

Examples of Intrinsic Assignment to Polymorphic Variables

Intrinsic assignment to an allocatable polymorphic variable is allowed. The variable must be type compatible with the expression and of the same rank. If it is allocated but the dynamic type differs from that of the expression, it is deallocated. If it is not allocated or becomes deallocated, it is allocated with the dynamic type of the expression.

Syntax:

assignment-stmt is *variable = expr*

variable is an allocatable polymorphic and *expr* is not necessarily an allocatable, but *variable* and *expr* must be type compatible.

A polymorphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same declared type or any of its extensions. Even though an unlimited polymorphic entity is not considered to have a declared type, it is type compatible with all entities.

variable and *expr* must be of the same rank.

If *expr* is a scalar and *variable* is allocated, then *expr* is treated as an array with the same bounds as *variable*, so the bounds of *variable* would remain unchanged. It is an error if *variable* is unallocated when *expr* is a scalar.

In the examples below, we have the following types:

```
Type type1
End type

Type type2
End type

Type, extends(type1) :: type3
End type
```

Example of same type and same size:

If *variable* is the same type and size as *expr* then just do the assignment.

```
Class(type1), allocatable :: var
Class(type1), allocatable :: expr
Allocate(expr, source=type1())
Allocate(var, source=type1())
var = expr             ! No deallocation of var, simple assignment
```

Example of same type and different size:

If *variable* is the same type as *expr* but their sizes are not the same then deallocate `var`, reallocate it, and then do the assignment.

```
Class(type1), allocatable :: var(:)
Class(type1), allocatable :: expr(:)
Allocate(var(5), source=type1())
Allocate(expr(6), source=type1())
var = expr              ! var deallocated and then
                        ! allocated to the size of expr - then
                        ! the usual assignment is performed
```

Example of different types or shape and same size:

If *variable* and *expr* are of different types or shapes but of the same size, then do the assignment and update the type/bounds of `var` to be same as that of `expr`, only if type1 and type3 are type compatible.

```
Class(type1), allocatable :: var(:,:)
Class(type1), allocatable :: expr(:,:)
Allocate(var(2,3), source=type1())
Allocate(expr(3,2), source=type3())
var = expr              ! No deallocation
                        ! simple assignment with var's
                        ! bounds updated
```

Example of different types or shape and different size:

If *variable* and *expr* are of different types or shapes and of different sizes, then deallocate `var` and allocate it with the same type and shape as `expr`, only if type1 and type3 are type compatible.

```
Class(type1), allocatable :: var(:,:)
Class(type3), allocatable :: expr(:,:)
Allocate(var(2,4), source=type1())
Allocate(expr(3,3), source=type3())
var = expr              ! var deallocated and then
                        ! allocated to the size of expr and
                        ! then the usual assignment var
                        ! has dynamic type set to type3
```

Example of incompatible types on Left-Hand Side (LHS) and Right-Hand Side (RHS):

If LHS and RHS are type incompatible then you get an error.

```
Class(type1), allocatable :: var
Class(type2), allocatable :: expr
Allocate(var,source=type1())
Allocate(expr,source=type2())
var = expr              ! This is an error
```

Example of unallocated allocatable polymorphic on the LHS:

If `var` is unallocated then `expr` must be a scalar or it is a shape mismatch error.

```
Class(*), allocatable :: var(:)
var = 5                 ! This is an error
```

Otherwise, allocate `var` with same dynamic type, shape, and size as `expr`.

```
Class(*), allocatable :: var
var = 5                 ! This is valid
```

Example of unlimited polymorphic on the LHS:

Unlimited polymorphic allocatable on the LHS is type compatible with any type. If `var` is of a different size than `expr`, `var` is deallocated and then allocated with the type and shape of `expr`.

```
Class(*), allocatable :: var
Class(type2), allocatable :: expr
Allocate(var, source=type1())
Allocate(expr, source=type2())
var = expr              ! var is an unlimited polymorphic
                        ! so it is type compatible with
                        ! any type expr
```

Defined Assignment Statements

Defined assignment specifies an assignment operation. It is defined by a subroutine subprogram containing a generic interface block that specifies ASSIGNMENT(=). The subroutine is specified in a SUBROUTINE or ENTRY statement that has two nonoptional dummy arguments.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

The dummy arguments represent the variable and expression, in that order. The rank (and shape, if either or both are arrays), type, and kind parameters of the variable and expression in the assignment statement must match those of the corresponding dummy arguments.

The dummy arguments must not both be numeric, or of type logical or character with the same kind parameter.

If the variable in an elemental assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of the variable and expression. If the expression is scalar, it is treated as if it were an array of the same shape as the variable with every element of the array equal to the scalar value of the expression.

See Also

Subroutines
Derived data types
Defining Generic Assignment for details on subroutine subprograms that define assignment
Numeric Expressions for details on intrinsic operations
Character Expressions for details on intrinsic operations

Pointer Assignments

In ordinary assignment involving pointers, the pointer is an alias for its target. In pointer assignment, the pointer is associated with a target. If the target is undefined or disassociated, the pointer acquires the same status as the target. Pointer assignment has the following form:

pointer-object [(*s-spec*)] => *target*

pointer-object	Is a variable name or structure component declared with the POINTER attribute.
s-spec	Is a shape specification consisting of bounds information in the form "[*lower-bound*]:" or "[*lower-bound*] :*upper-bound*".
target	Is a variable or expression. Its type and kind parameters, and rank must be the same as *pointer-object* unless bounds remapping is specified. It cannot be an array section with a vector subscript.

Description

If the target is a variable, it must have the POINTER or TARGET attribute, or be a subobject whose parent object has the TARGET attribute.

If the target is an expression, the result must be a pointer.

If the target is not a pointer (it has the TARGET attribute), the pointer object is associated with the target.

If the target is a pointer (it has the POINTER attribute), its status determines the status of the pointer object, as follows:

- If the pointer is associated, the pointer object is associated with the same object as the target
- If the pointer is disassociated, the pointer object becomes disassociated
- If the pointer is undefined, the pointer object becomes undefined

A pointer must not be referenced or defined unless it is associated with a target that can be referenced or defined.

When pointer assignment occurs, any previous association between the pointer object and a target is terminated.

Pointers can also be assigned for a pointer structure component by execution of a derived-type intrinsic assignment statement or a defined assignment statement.

Pointers can also become associated by using the ALLOCATE statement to allocate the pointer.

Pointers can become disassociated by deallocation, nullification of the pointer (using the DEALLOCATE or NULLIFY statements), or by reference to the NULL intrinsic function.

Pointer assignment for arrays allows lower bounds to be specified. The specified lower bounds can be any scalar integer expressions.

Remapping of the elements of a rank-one array is permitted. The mapping is in array-element order and the target array must be large enough. The specified bounds may be any scalar integer expressions.

Examples

The following are examples of pointer assignments:

```
HOUR => MINUTES(1:60)            ! target is an array
M_YEAR => MY_CAR%YEAR          ! target is a structure component
NEW_ROW%RIGHT => CURRENT_ROW     ! pointer object is a structure component
PTR => M                       ! target is a variable
POINTER_C => NULL ()             ! reference to NULL intrinsic
```

The following example shows a target as a pointer:

```
INTEGER, POINTER :: P, N
INTEGER, TARGET :: M
INTEGER S
M = 14
N => M                          ! N is associated with M
P => N                          ! P is associated with M through N
S = P + 5
```

The value assigned to S is 19 (14 + 5).

You can use the intrinsic function ASSOCIATED to find out if a pointer is associated with a target or if two pointers are associated with the same target. For example:

```
REAL C (:), D(:), E(5)
POINTER C, D
TARGET E
LOGICAL STATUS
! Pointer assignment.
C => E
! Pointer assignment.
D => E
! Returns TRUE; C is associated.
STATUS = ASSOCIATED (C)
! Returns TRUE; C is associated with E.
```

```
  STATUS = ASSOCIATED (C, E)
  ! Returns TRUE; C and D are associated with the
  ! same target.
  STATUS = ASSOCIATED (C, D)
```

The following example shows how to specify lower bounds on a pointer:

```
REAL, TARGET :: A(4,4)
POINTER P
P (0:,0:) => A            ! LBOUND (P) == [0,0]
                          ! UBOUND (P) == [3,3]
```

The following example shows pointer remapping of the elements of a rank-one array:

```
REAL, TARGET :: V(100)
POINTER P
INTEGER N
P(1:N,1:2*N) => V(1:2*N*N)
```

See Also

Arrays
ALLOCATE statement
DEALLOCATE statement
NULLIFY statement
NULL intrinsic function
POINTER attribute
TARGET attribute
Defined assignments
Intrinsic Assignments for details on derived-type intrinsic assignments

Specification Statements

A *specification statement* is a nonexecutable statement that declares the attributes of data objects. In Standard Fortran, many of the attributes that can be defined in specification statements can also be optionally specified in type declaration statements.

The following are specification statements:

- Type Declarations

  Explicitly specify the properties (for example: data type, rank, and extent) for data objects or functions.
- ALLOCATABLE attribute and statement

  Specifies that an array is an allocatable array with a deferred shape. The shape of an allocatable array is determined when an ALLOCATE statement is executed, dynamically allocating space for the array.
- ASYNCHRONOUS attribute and statement

  Specifies that a variable can be used for asynchronous input and output.
- AUTOMATIC and STATIC attributes and statements

  Control the storage allocation of variables in subprograms.
- BIND attribute and statement

  Specifies that an object is interoperable with C and has external linkage.
- CODIMENSION attribute and statement

  Specifies that an entity is a coarray.
- COMMON statement

Defines one or more contiguous areas, or blocks, of physical storage (called common blocks) that can be accessed by any of the scoping units in an executable program. COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items.

- CONTIGUOUS attribute and statement

  Specifies that the target of a pointer or an assumed-sized array is contiguous.
- DATA statement

  Assigns initial values to variables before program execution.
- DIMENSION attribute and statement

  Specifies that an object is an array, and defines the shape of the array.
- EQUIVALENCE statement

  Specifies that a storage area is shared by two or more objects in a program unit. This causes total or partial storage association of the objects that share the storage area.
- EXTERNAL attribute and statement

  Allows an external or dummy procedure to be used as an actual argument.
- IMPLICIT statement

  Overrides the default implicit typing rules for names.
- INTENT attribute and statement

  Specifies the intended use of one or more dummy arguments.
- INTRINSIC attribute and statement

  Allows the specific name of an intrinsic procedure to be used as an actual argument. Certain specific function names cannot be used. For more information, see Intrinsic Functions Not Allowed as Actual Arguments.
- NAMELIST statement

  Associates a name with a list of variables. This group name can be referenced in some input/output operations.
- OPTIONAL attribute and statement

  Allows dummy arguments to be omitted in a procedure reference.
- PARAMETER attribute and statement

  Defines a named constant.
- POINTER attribute and statement

  Specifies that an object or a procedure is a pointer (a dynamic variable).
- PRIVATE and PUBLIC and attributes and statements

  Specifies the accessibility of entities in a module. (These attributes are also called accessibility attributes.)
- PROTECTED attribute and statement

  Specifies limitations on the use of module entities.
- SAVE attribute and statement

  Causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram.
- TARGET attribute and statement

  Specifies that an object can become the target of a pointer.
- VALUE attribute and statement

  Specifies a type of argument association for a dummy argument.
- VOLATILE attribute and statement

  Specifies that the value of an object is entirely unpredictable, based on information local to the current program unit.

See Also

BLOCK DATA statement
PROGRAM statement
Derived data types

DATA
Initialization expressions
Intrinsic Data Types
Implicit Typing Rules
Specification of Data Type

Type Declarations

A type declaration is a nonexecutable statement specifying the data type of one or more variables. The declaration can be specified in an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, CHARACTER, LOGICAL, or TYPE statement. A type declaration statement may also specify attributes for the variables.

Declarations for Noncharacter Types

The following table shows the data types that can appear in noncharacter type declarations.

Noncharacter Data Types
BYTE[1]
LOGICAL[2]
LOGICAL([KIND=]1) (or LOGICAL*1)
LOGICAL([KIND=]2) (or LOGICAL*2)
LOGICAL([KIND=]4) (or LOGICAL*4)
LOGICAL([KIND=]8) (or LOGICAL*8)
INTEGER[3]
INTEGER([KIND=]1) (or INTEGER*1)
INTEGER([KIND=]2) (or INTEGER*2)
INTEGER([KIND=]4) (or INTEGER*4)
INTEGER([KIND=]8) (or INTEGER*8)
REAL[4]
REAL([KIND=]4) (or REAL*4)
DOUBLE PRECISION (REAL([KIND=]8) or REAL*8)
REAL([KIND=]16) (or REAL*16)
COMPLEX[5]
COMPLEX([KIND=]4) (or COMPLEX*8)
DOUBLE COMPLEX (COMPLEX([KIND=]8) or COMPLEX*16)
COMPLEX([KIND=]16) (or COMPLEX*32)

[1] Same as INTEGER(1).

[2] This is treated as default logical.

[3] This is treated as default integer.

[4] This is treated as default real.

Noncharacter Data Types

[5] This is treated as default complex.

In noncharacter type declarations, you can optionally specify the name of the data object or function as v*n, where *n* is the length (in bytes) of *v*. The length specified overrides the length implied by the data type.

The value for *n* must be a valid length for the type of *v*. The type specifiers BYTE, DOUBLE PRECISION, and DOUBLE COMPLEX have one valid length, so the *n* specifier is invalid for them.

For an array specification, the *n* must be placed immediately following the array name; for example, in an INTEGER declaration, IVEC*2(10) is an INTEGER(2) array of 10 elements.

Note that certain compiler options can affect the defaults for numeric and logical data types.

Examples

In a noncharacter type declaration, a subsequent kind parameter overrides any initial kind parameter. For example, consider the following statements:

```
INTEGER(KIND=2) I, J, K, M12*4, Q, IVEC*4(10)
REAL(KIND=8) WX1, WXZ, WX3*4, WX5, WX6*4
REAL(KIND=8) PI/3.14159E0/, E/2.72E0/, QARRAY(10)/5*0.0,5*1.0/
```

In the first statement, M12*4 and IVEC*4 override the KIND=2 specification. In the second statement, WX3*4 and WX6*4 override the KIND=8 specification. In the third statement, QARRAY is initialized with implicit conversion of the REAL(4) constants to a REAL(8) data type.

See Also

Type Declarations for details on the general form and rules for type declarations

Declarations for Character Types

A CHARACTER type specifier can be immediately followed by an optional KIND type parameter and the length of the character object or function. It takes one of the following forms:

Keyword Forms

CHARACTER ([LEN=] *len*)

CHARACTER [([LEN=] *len* [, [KIND=] *n*])]

CHARACTER [(KIND= *n* [, LEN= *len*])]

Obsolete Form

CHARACTER* *len*[,]

len

Is a length type parameter. It can have one of the following values:

- A scalar integer expression
- *
- :

The following rules also apply:

- In keyword forms

  The *len* is a specification expression, an asterisk (*), or a colon (:). If no length is specified, the default length is 1.

  If the length evaluates to a negative value, the length of the character entity is zero.
- In the obsolete form

The *len* is a specification expression or an asterisk enclosed in parentheses, or a scalar integer literal constant (with no kind parameter). The comma is permitted only if no double colon (::) appears in the type declaration statement.

This form can also (optionally) be specified following the name of the data object or function (*v***len*). In this case, the length specified overrides any length following the CHARACTER type specifier.

The largest valid value for *len* in both forms is 2**31-1 on IA-32 architecture; 2**63-1 on Intel® 64 architecture. Negative values are treated as zero.

n

Is a scalar integer constant expression specifying a valid kind parameter. Currently the only kind available is 1.

Description

An automatic object can appear in a character declaration. The object cannot be a dummy argument, and its length must be declared with a specification expression that is not a constant expression.

The length specified for a character-valued statement function or statement function dummy argument of type character must be an integer constant expression.

When an asterisk length specification *(*) is used for a function name or dummy argument, it assumes the length of the corresponding function reference or actual argument. Similarly, when an asterisk length specification is used for a named constant, the name assumes the length of the actual constant it represents. For example, STRING assumes a 9-byte length in the following statements:

```
CHARACTER*(*) STRING
PARAMETER (STRING = 'VALUE IS:')
```

A function name must not be declared with a * length unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.

A function name declared with a * length must not be an array, a pointer, recursive, elemental, or pure.

If the CHARACTER type declaration statement specifies a colon (:) length, the length type parameter is a deferred type parameter. An entity or component with a deferred type parameter must specify the ALLOCATABLE or POINTER attribute. A deferred type parameter is a length type parameter whose value can change during execution of the program.

The obsolete form is an obsolescent feature in the Fortran standard.

Examples

In the following example, the character string last_name is given a length of 20:

```
CHARACTER (LEN=20) last_name
```

In the following example, stri is given a length of 12, while the other two variables retain a length of 8.

```
CHARACTER *8 strg, strh, stri*12
```

In the following example, a dummy argument strh will assume the length of the associated actual argument, while the other two variables retain a length of 8:

```
CHARACTER *8 strg, strh(*), stri
```

The following examples show ways to specify strings of known length:

```
CHARACTER*32 string1
CHARACTER string2*32
```

The following examples show ways to specify strings of unknown length:

```
CHARACTER string3*(*)
CHARACTER*(*) string4
```

The following example declares an array NAMES containing 100 32-character elements, an array SOCSEC containing 100 9-character elements, and a variable NAMETY that is 10 characters long and has an initial value of 'ABCDEFGHIJ'.

```
CHARACTER*32 NAMES(100),SOCSEC(100)*9,NAMETY*10 /'ABCDEFGHIJ'/
```

The following example includes a CHARACTER statement declaring two 8-character variables, LAST and FIRST.

```
INTEGER, PARAMETER :: LENGTH=4
CHARACTER*(4+LENGTH) LAST, FIRST
```

The following example shows a CHARACTER statement declaring an array LETTER containing 26 one-character elements. It also declares a dummy argument BUBBLE that has a passed length defined by the calling program.

```
CHARACTER LETTER(26), BUBBLE*(*)
```

In the following example, NAME2 is an automatic object:

```
SUBROUTINE AUTO_NAME(NAME1)
  CHARACTER(LEN = *) NAME1
  CHARACTER(LEN = LEN(NAME1)) NAME2
```

The following example shows the handling of a deferred-length CHARACTER variables:

```
CHARACTER(:), ALLOCATABLE :: namea
namea = 'XYZ'                   ! Allocates namea as length 3 with value 'XYZ'
DEALLOCATE (namea)
ALLOCATE (CHARACTER(10)::namea) ! Allocates namea as length 10, no value
namea = 'ABCDEF'                ! Reallocates namea to length 6 with value 'ABCDEF'
```

See Also

Obsolescent Language Features in the Fortran Standard
Data Types of Scalar Variables
Assumed-Length Character Arguments for details on asterisk length specifications
Type Declarations for details on the general form and rules for type declaration statements

Declarations for Derived Types

A derived-type declaration specifies the properties of objects and functions of derived (user-defined) type.

The derived type must be defined before you can specify objects of that type in a TYPE type declaration.

An object of derived type must not have the PUBLIC attribute if its type is PRIVATE.

A structure constructor specifies values for derived-type objects.

Examples

The following are examples of derived-type declarations:

```
TYPE(EMPLOYEE) CONTRACT
...
TYPE(SETS), DIMENSION(:,:), ALLOCATABLE :: SUBSET_1
```

The following example shows a public type with private components:

```
TYPE LIST_ITEMS
  PRIVATE
  ...
  TYPE(LIST_ITEMS), POINTER :: NEXT, PREVIOUS
END TYPE LIST_ITEMS
```

See Also

TYPE
Use and host association
PUBLIC
PRIVATE
Structure constructors
Type Declarations for details on the general form and rules for type declarations

Declarations for Arrays

An array declaration (or array declarator) declares the shape of an array. It takes the following form:

(*a-spec*)

a-spec — Is one of the following array specifications:

- Explicit-shape
- Assumed-shape
- Assumed-size
- Deferred-shape
- Assumed-rank
- Implied-shape

The array specification can be appended to the name of the array when the array is declared.

Examples

The following examples show array declarations:

```
SUBROUTINE SUB(N, C, D, Z)
  REAL, DIMENSION(N, 15) :: IARRY       ! An explicit-shape array
  REAL C(:), D(0:)                      ! An assumed-shape array
  REAL, POINTER :: B(:,:)               ! A deferred-shape array pointer
  REAL, ALLOCATABLE, DIMENSION(:) :: K  ! A deferred-shape allocatable array
  REAL :: Z(N,*)                        ! An assumed-size array
  INTEGER, PARAMETER :: R(*) = [1,2,3]  ! An implied-shape constant array
```

See Also

Type Declarations for details on the general form and rules for type declaration statements

Explicit-Shape Specifications

An *explicit-shape array* is declared with explicit values for the bounds in each dimension of the array. An explicit-shape specification takes the following form:

([*dl*:] *du*[, [*dl*:] *du*] ...)

dl — Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

If the lower bound is not specified, it is assumed to be 1.

du	Is a specification expression indicating the upper bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.

The bounds can be specified as constant or nonconstant expressions, as follows:

- If the bounds are constant expressions, the subscript range of the array in a dimension is the set of integer values between and including the lower and upper bounds. If the lower bound is greater than the upper bound, the range is empty, the extent in that dimension is zero, and the array has a size of zero.
- If the bounds are nonconstant expressions, the array must be declared in a procedure. The bounds can have different values each time the procedure is executed, since they are determined when the procedure is entered.

  The bounds are not affected by any redefinition or undefinition of the variables in the specification expression that occurs while the procedure is executing.

  The following explicit-shape arrays can specify nonconstant bounds:

  - An automatic array (the array is a local variable)
  - An adjustable array (the array is a dummy argument to a subprogram)

The following are examples of explicit-shape specifications:

```
INTEGER I(3:8, -2:5)          ! Rank-two array; range of dimension one is
...                           ! 3 to 8, range of dimension two is -2 to 5
SUBROUTINE SUB(A, B, C)
  INTEGER :: B, C
  REAL, DIMENSION(B:C) :: A   ! Rank-one array; range is B to C
```

Consider the following:

```
 INTEGER M(10, 10, 10)
 INTEGER K(-3:6, 4:13, 0:9)
```

M and K are both explicit-shape arrays with a rank of 3, a size of 1000, and the same shape (10,10,10). Array M uses the default lower bound of 1 for each of its dimensions. So, when it is declared only the upper bound needs to be specified. Each of the dimensions of array K has a lower bound other than the default, and the lower bounds as well as the upper bounds are declared.

Automatic Arrays

An *automatic array* is an explicit-shape array that is a local variable. Automatic arrays are only allowed in function and subroutine subprograms, and are declared in the specification part of the subprogram. At least one bound of an automatic array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The following example shows automatic arrays:

```
SUBROUTINE SUB1 (A, B)
  INTEGER A, B, LOWER
  COMMON /BOUND/ LOWER
  ...
  INTEGER AUTO_ARRAY1(B)
  ...
  INTEGER AUTO_ARRAY2(LOWER:B)
  ...
  INTEGER AUTO_ARRAY3(20, B*A/2)
END SUBROUTINE
```

Consider the following:

```
 SUBROUTINE EXAMPLE (N, R1, R2)
   DIMENSION A (N, 5), B(10*N)
   ...
   N = IFIX(R1) + IFIX(R2)
```

When the subroutine is called, the arrays A and B are dimensioned on entry into the subroutine with the value of the passed variable N. Later changes to the value of N have no effect on the dimensions of array A or B.

Adjustable Arrays

An *adjustable array* is an explicit-shape array that is a dummy argument to a subprogram. At least one bound of an adjustable array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The array specification can contain integer variables that are either dummy arguments or variables in a common block.

When the subprogram is entered, each dummy argument specified in the bounds must be associated with an actual argument. If the specification includes a variable in a common block, the variable must have a defined value. The array specification is evaluated using the values of the actual arguments, as well as any constants or common block variables that appear in the specification.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

To avoid possible errors in subscript evaluation, make sure that the bounds expressions used to declare multidimensional adjustable arrays match the bounds as declared by the caller.

In the following example, the function computes the sum of the elements of a rank-two array. Notice how the dummy arguments M and N control the iteration:

```
  FUNCTION THE_SUM(A, M, N)
    DIMENSION A(M, N)
    SUMX = 0.0
    DO J = 1, N
      DO I = 1, M
        SUMX = SUMX + A(I, J)
      END DO
    END DO
    THE_SUM = SUMX
  END FUNCTION
```

The following are examples of calls on THE_SUM:

```
DIMENSION A1(10,35), A2(3,56)
SUM1 = THE_SUM(A1,10,35)
SUM2 = THE_SUM(A2,3,56)
```

The following example shows how the array bounds determined when the procedure is entered do not change during execution:

```
DIMENSION ARRAY(9,5)
L = 9
M = 5
CALL SUB(ARRAY,L,M)
END

SUBROUTINE SUB(X,I,J)
  DIMENSION X(-I/2:I/2,J)
  X(I/2,J) = 999
  J = 1
  I = 2
END
```

The assignments to I and J do not affect the declaration of adjustable array X as X(-4:4,5) on entry to subroutine SUB.

See Also

Specification expressions

Assumed-Shape Specifications

An *assumed-shape array* is a dummy argument array that assumes the shape of its associated actual argument array. An assumed-shape specification takes the following form:

([*dl*]:[, [*dl*]:] ...)

dl	Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. If the lower bound is not specified, it is assumed to be 1.

The rank of the array is the number of colons (:) specified.

The value of the upper bound is the extent of the corresponding dimension of the associated actual argument array + *lower-bound* - 1.

Examples

The following is an example of an assumed-shape specification:

```
INTERFACE
  SUBROUTINE SUB(M)
    INTEGER M(:, 1:, 5:)
  END SUBROUTINE
END INTERFACE
INTEGER L(20, 5:25, 10)

CALL SUB(L)
SUBROUTINE SUB(M)
  INTEGER M(:, 1:, 5:)
END SUBROUTINE
```

Array M has the same extents as array L, but array M has bounds (1:20, 1:21, 5:14).

Note that an explicit interface is *required* when calling a routine that expects an assumed-shape or pointer array.

Consider the following:

```
SUBROUTINE ASSUMED(A)
  REAL A(:, :, :)
```

Array A has rank 3, indicated by the three colons (:) separated by commas (,). However, the extent of each dimension is unspecified. When the subroutine is called, A takes its shape from the array passed to it. For example, consider the following:

```
REAL X (4, 7, 9)
...
CALL ASSUMED(X)
```

This gives A the dimensions (4, 7, 9). The actual array and the assumed-shape array must have the same rank.

Consider the following:

```
SUBROUTINE ASSUMED(A)
  REAL A(3:, 0:, -2:)
...
```

If the subroutine is called with the same actual array X(4, 7, 9), as in the previous example, the lower and upper bounds of A would be:

```
A(3:6, 0:6, -2:6)
```

Assumed-Size Specifications

An *assumed-size array* is a dummy argument array that assumes the size (only) of its associated actual argument array, or the associate name of a RANK (*) block in a SELECT RANK construct. The rank and extents can differ for the actual and dummy arrays. An assumed-size specification takes the following form:

([*expli-shape-spec*,] [*expli-shape-spec*,] ... [*dl*:] *)

expli-shape-spec	Is an explicit-shape specification.
dl	Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. If the lower bound is not specified, it is assumed to be 1.
*	Is the upper bound of the last dimension.

The rank of the array is the number of explicit-shape specifications plus 1.

The size of the array is assumed from the actual argument associated with the assumed-size dummy array as follows:

- If the actual argument is an array of type other than default character, the size of the dummy array is the size of the actual array.
- If the actual argument is an array element of type other than default character, the size of the dummy array is `a + 1 - s`, where *s* is the subscript order value and *a* is the size of the actual array.
- If the actual argument is a default character array, array element, or array element substring, and it begins at character storage unit *b* of an array with *n* character storage units, the size of the dummy array is as follows:

```
MAX(INT((n + 1 - b)/y), 0)
```

  The *y* is the length of an element of the dummy array.

An assumed-size array can only be used as a whole array reference in the following cases:

- When it is an actual argument in a procedure reference that does not require the shape
- In the intrinsic function LBOUND

Because the actual size of an assumed-size array is unknown, an assumed-size array cannot be used as any of the following in an I/O statement:

- An array name in the I/O list
- A unit identifier for an internal file
- A run-time format specifier

Examples

The following is an example of an assumed-size specification:

```
SUBROUTINE SUB(A, N)
  REAL A, N
  DIMENSION A(1:N, *)
  ...
```

The following example shows that you can specify lower bounds for any of the dimensions of an assumed-size array, including the last:

```
SUBROUTINE ASSUME(A)
  REAL A(-4:-2, 4:6, 3:*)
```

See Also

Array Elements

Assumed-Rank Specifications

An assumed-rank array is a dummy argument whose rank is inherited from the actual argument associated with it, or it is the associate name of a RANK DEFAULT block of a SELECT RANK construct. It can have zero rank.

You declare an assumed-rank object (a dummy variable) by using DIMENSION(..) or (..)*array bounds* in its declaration.

Its rank is assumed from its effective argument, which means it is passed by descriptor.

An assumed-rank entity must not have the CODIMENSION or VALUE attribute. It can have the CONTIGUOUS attribute.

An assumed-rank variable name must not appear in a designator or expression except as one of the following:

- An actual argument corresponding to a dummy argument that is assumed-rank
- The argument of the C_LOC function in the ISO_C_BINDING intrinsic module
- The first argument in a reference to an intrinsic inquiry function
- The selector of a SELECT RANK construct

If an assumed-size or nonallocatable, nonpointer, assumed-rank array is an actual argument corresponding to a dummy argument that is an INTENT(OUT) assumed-rank array, it must not be polymorphic, finalizable, of a type with an allocatable ultimate component, or of a type for which default initialization is specified.

You can find the rank of an assumed-rank object by using the RANK intrinsic.

If a procedure has an assumed-rank argument, the procedure must have an explicit interface.

When an assumed-rank object is passed from Fortran to a BIND(C) routine, it is passed by C descriptor. A Fortran procedure that has the BIND(C) *language-binding-spec* attribute will also receive an assumed-rank object by C descriptor.

Examples

The following shows an assumed-rank object:

```
SUBROUTINE sub (foo, bar)
! As sub does not have BIND, foo and bar are passed by "normal" descriptor
REAL, DIMENSION(..) :: foo
INTEGER :: bar(..)

INTERFACE
  SUBROUTINE csub (baz) BIND(C)
  REAL, DIMENSION(..) :: baz
  END SUBROUTINE
END INTERFACE

CALL baz(foo)   ! Passed by C descriptor
```

See Also

Argument Association
RANK

Deferred-Shape Specifications

A *deferred-shape array* is an array pointer or an allocatable array.

The array specification contains a colon (:) for each dimension of the array. No bounds are specified. The bounds (and shape) of allocatable arrays and array pointers are determined when space is allocated for the array during program execution.

An *array pointer* is an array declared with the POINTER attribute. Its bounds and shape are determined when it is associated with a target by pointer assignment, or when the pointer is allocated by execution of an ALLOCATE statement.

In pointer assignment, the lower bound of each dimension of the array pointer is the result of the LBOUND intrinsic function applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the UBOUND intrinsic function applied to the corresponding dimension of the target.

An actual argument that is a pointer can be associated with a nonpointer dummy argument. Normally, a pointer dummy argument can be associated only with a pointer actual argument. However, a pointer dummy argument with INTENT(IN) can be argument associated with a non-pointer actual argument with the TARGET attribute. During the execution of the procedure, it is pointer associated with the actual argument.

A function result can be declared to have the pointer attribute.

An *allocatable array* is declared with the ALLOCATABLE attribute. Its bounds and shape are determined when the array is allocated by execution of an ALLOCATE statement.

Examples

The following are examples of deferred-shape specifications:

```
REAL, ALLOCATABLE :: A(:,:)       ! Allocatable array
REAL, POINTER :: C(:), D (:,:,:)  ! Array pointers
```

If a deferred-shape array is declared in a DIMENSION or TARGET statement, it must be given the ALLOCATABLE or POINTER attribute in another statement. For example:

```
 DIMENSION P(:, :, :)
 POINTER P

 TARGET B(:,:)
 ALLOCATABLE B
```

If the deferred-shape array is an array of pointers, its size, shape, and bounds are set in an ALLOCATE statement or in the pointer assignment statement when the pointer is associated with an allocated target. A pointer and its target must have the same rank.

For example:

```
 REAL, POINTER :: A(:,:), B(:), C(:,:)
 INTEGER, ALLOCATABLE :: I(:)
 REAL, ALLOCATABLE, TARGET :: D(:, :), E(:)
 ...
 ALLOCATE (A(2, 3), I(5), D(SIZE(I), 12), E(98) )
 C => D              ! Pointer assignment statement
 B => E(25:56)       ! Pointer assignment to a section
                     !   of a target
```

A pointer dummy argument with INTENT(IN) can be argument associated with a non-pointer actual argument with the TARGET attribute. It is pointer associated with the actual argument, so the following example prints "17".

```
program test
   integer, target :: j = 17
   call f (j)
contains
   subroutine f (i)
   integer, intent (in), pointer :: i
   print *,i
   end subroutine f
end program test
```

See Also

POINTER attribute

ALLOCATABLE attribute
ALLOCATE statement
Pointer assignment
LBOUND intrinsic function
UBOUND intrinsic function

Implied-Shape Specifications

Short Description

An *implied-shape array* is a named constant that takes its shape from the constant expression in its declaration. An implied-shape specification takes the following form:

([*dl*:] *du* [, [*dl*:] *du*] ...)

dl	Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. If the lower bound is not specified, it is assumed to be 1.
du	Is the upper bound of the dimension or *.

The constant expression in the implied-shape declaration must be an array.

The rank of the array must be the same as the rank of the constant expression in its declaration.

The extent of each dimension of an implied-shape array is the same as the extent of the corresponding dimension of the constant expression.

The value of the upper bound is 1 less than the sum of the lower bound and the extent.

Examples

The following examples show implied-shape specifications:

```
INTEGER, PARAMETER :: R(*) = [1,2,3]
! means SHAPE (R) is [3]

integer, parameter :: x (0:*,1:*) = reshape ( [1,2,3,4], [2,2] )
! means dimensions of X are 0 to 1 and 1 to 2

REAL :: M (2:*, -1:*)
PARAMETER (M = RESHAPE ([R,R], [3,2]))
! means dimensions of M are 2 to 4 and -1 to 0

integer, parameter :: Y (0:*,1:2) = reshape ( [1,2,3,4], [2,2] )
! means dimensions of Y are 0 to 1 and 1 to 2
```

Effects of Equivalency and Interaction with COMMON Statements

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays.

When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets associations between the other corresponding characters in the character entities.

A common block can extend beyond its original boundaries if variables or arrays are associated with entities stored in the common block. However, a common block can only extend beyond its last element; the extended portion cannot precede the first element in the block.

COMMON and EQUIVALENCE are obsolescent language features in Standard Fortran.

For more information, see the topics in this section.

Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage. If the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

Two or more elements of the same array should not be associated with each other in one or more EQUIVALENCE statements. For example, you cannot use an EQUIVALENCE statement to associate the first element of one array with the first element of another array, and then attempt to associate the fourth element of the first array with the seventh element of the other array.

Consider the following example:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE(TABLE(2,2), TRIPLE(1,2,2))
```

These statements cause the entire array TABLE to share part of the storage allocated to TRIPLE. The following table shows how these statements align the arrays:

Equivalence of Array Storage

Array TRIPLE		Array TABLE	
Array Element	**Element Number**	**Array Element**	**Element Number**
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Each of the following statements also aligns the two arrays as shown in the above table:

```
EQUIVALENCE(TABLE, TRIPLE(2,2,1))
EQUIVALENCE(TRIPLE(1,1,2), TABLE(2,1))
```

You can also make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the following statement:

```
EQUIVALENCE(A(3,4), B(2,4))
```

The entire array A shares part of the storage allocated to array B. The following table shows how these statements align the arrays. The arrays can also be aligned by the following statements:

```
EQUIVALENCE(A, B(4,1))
EQUIVALENCE(B(3,2), A(2,2))
```

Equivalence of Arrays with Nonunity Lower Bounds

Array B		Array A	
Array Element	**Element Number**	**Array Element**	**Element Number**
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

Only in the EQUIVALENCE statement can you identify an array element with a single subscript (the linear element number), even though the array was defined as multidimensional. For example, the following statements align the two arrays as shown in the above table:

```
DIMENSION B(2:4,1:4), A(2:3,1:4)
EQUIVALENCE(B(6), A(4))
```

Making Substrings Equivalent

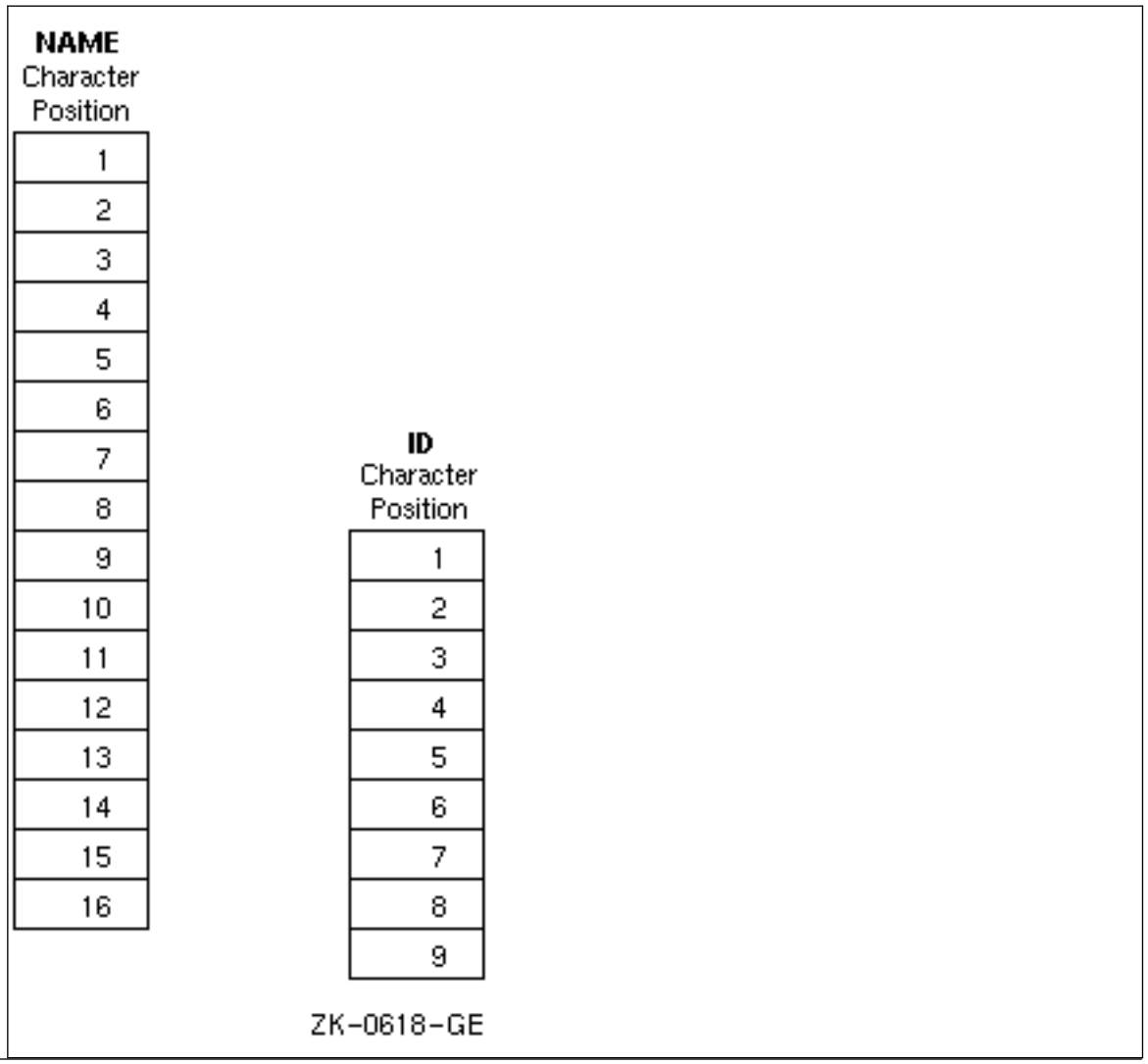
When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets associations between the other corresponding characters in the character entities; for example:

```
CHARACTER NAME*16, ID*9
EQUIVALENCE(NAME(10:13), ID(2:5))
```

These statements cause character variables NAME and ID to share space (see the following figure). The arrays can also be aligned by the following statement:

```
EQUIVALENCE(NAME(9:9), ID(1:1))
```

Equivalence of Substrings

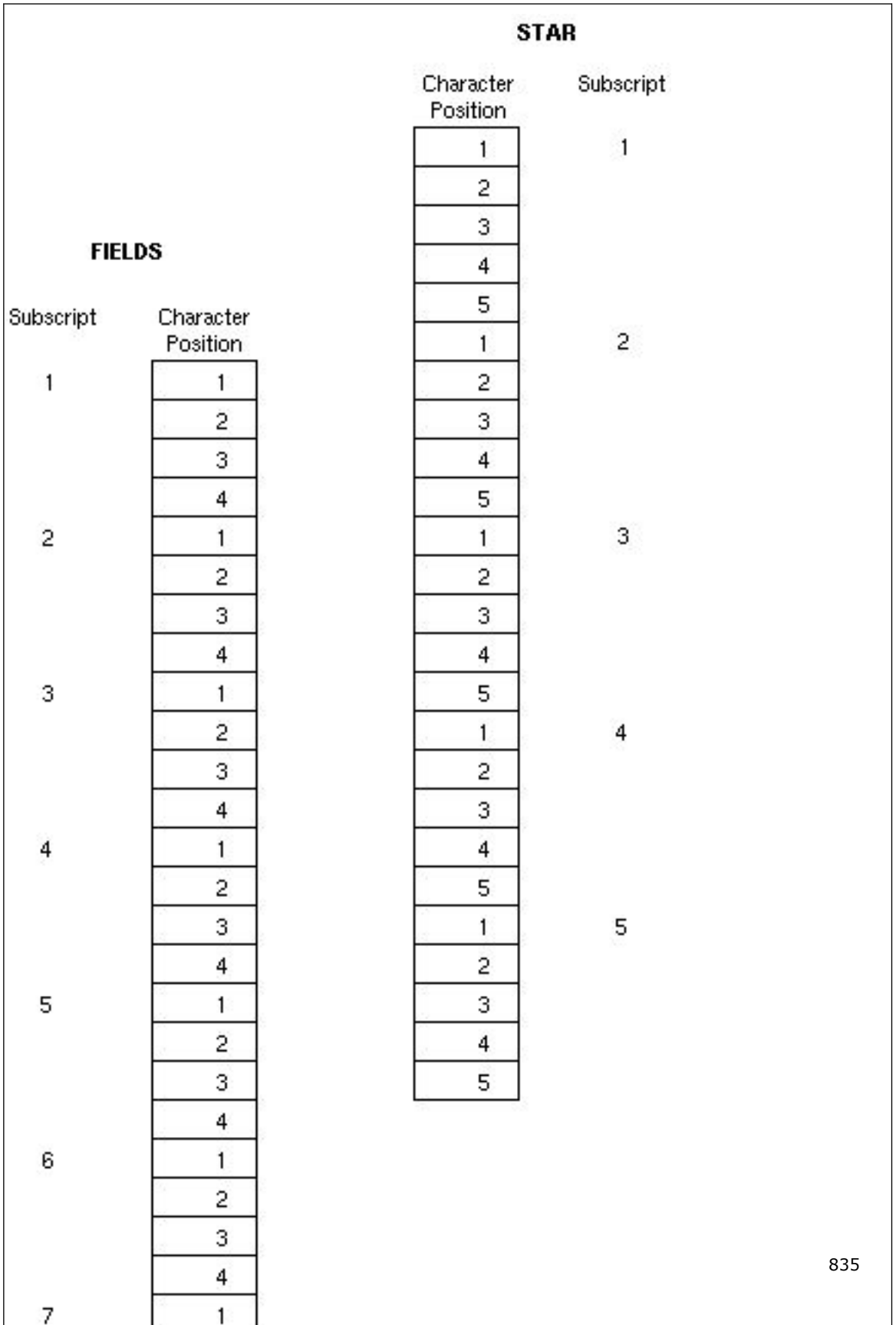


If the character substring references are array elements, the EQUIVALENCE statement sets associations between the other corresponding characters in the complete arrays.

Character elements of arrays can overlap at any character position. For example, the following statements cause character arrays FIELDS and STAR to share storage (see the following figure).

```
CHARACTER FIELDS(100)*4, STAR(5)*5
EQUIVALENCE(FIELDS(1)(2:4), STAR(2)(3:5))
```

Equivalence of Character Arrays



The EQUIVALENCE statement cannot assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array. The EQUIVALENCE statement also cannot assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

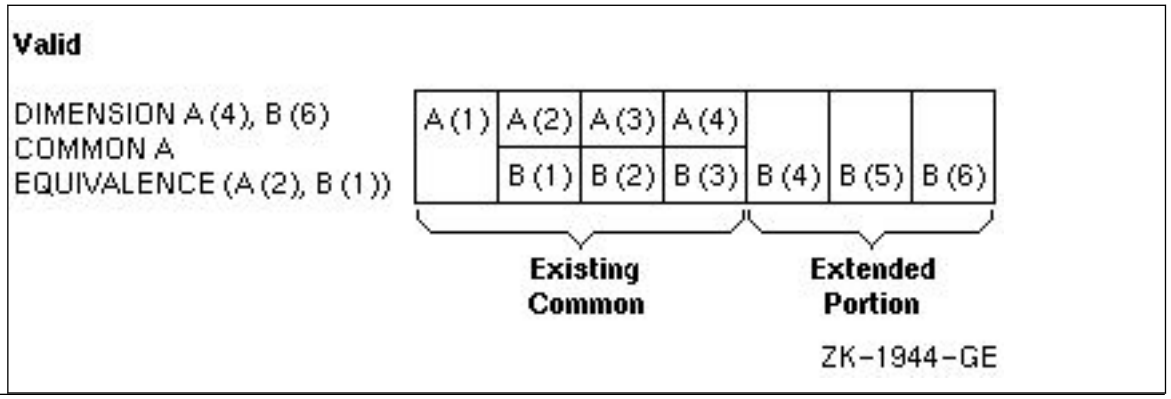
EQUIVALENCE and COMMON Interaction

A common block can extend beyond its original boundaries if variables or arrays are associated with entities stored in the common block. However, a common block can only extend beyond its last element; the extended portion cannot precede the first element in the block.

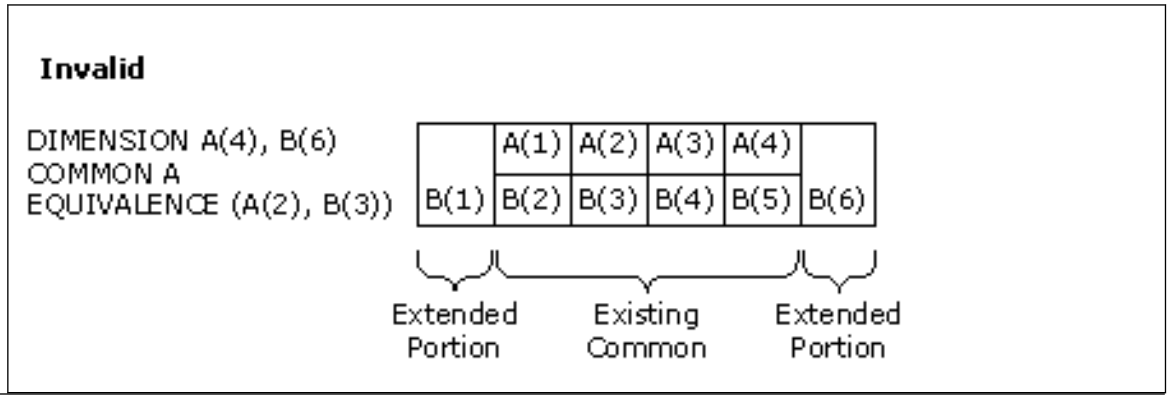
Examples

The following two figures demonstrate valid and invalid extensions of the common block, respectively.

A Valid Extension of a Common Block



An Invalid Extension of a Common Block



The second example is invalid because the extended portion, B(1), precedes the first element of the common block.

The following example shows a valid EQUIVALENCE statement and an invalid EQUIVALENCE statement in the context of a common block.

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE(B, D(1))      ! Valid, because common block is extended
                          !   from the end.

COMMON A, B, C
```

```
DIMENSION D(3)
EQUIVALENCE(B, D(3))      ! Invalid, because D(1) would extend common
                          !   block to precede A's location.
```

Dynamic Allocation

Data objects can be static or dynamic. If a data object is static, a fixed amount of memory storage is created for it at compile time and is not freed until the program exits. If a data object is dynamic, memory storage for the object can be created (allocated), altered, or freed (deallocated) as a program executes.

Pointers, classes, deferred length character, allocatable scalars and arrays, and automatic arrays are dynamic data objects.

No storage space is created for a pointer until it is allocated with an ALLOCATE statement or until it is assigned to an allocated target. The storage space allocated is uninitialized.

An ALLOCATE statement can also be used to create storage for an allocatable array. A DEALLOCATE statement can be used to free the storage space reserved in a previous ALLOCATE statement. It also causes any pointers to become disassociated.

A pointer can be dynamically disassociated from a target by using a NULLIFY statement.

Automatic arrays differ from allocatable arrays in that they are automatically allocated and deallocated whenever you enter or leave a procedure, respectively.

Dynamic allocation occurs at run time and is handled by the Fortran Run-Time Library. There are several restrictions on allocation and deallocation that must be observed when these operations on a specific object are performed in program units that are separately compiled. When allocation and deallocation of an object are split between procedures in static code and dynamic shared libraries (.so files on Linux*) or dynamic-link libraries (DLLs on Windows*), the following applies:

- If the dynamic library is compiled with the `[q or Q]openmp` compiler option, then the main program must be compiled and linked with `[q or Q]openmp` to include the OpenMP memory handling routines in the program.
- If the dynamic library allocates data in High bandwidth (HBW) memory on Linux*, then the program must be linked with the `libmemkind` library to include the HBW memory handling routines in the program.

> **NOTE**
> Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. Dynamic allocations that are too large or otherwise attempt to use the protected memory of other applications result in General Protection Fault errors. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Control Panel or redefine the swap file size.
>
> Some programming techniques can help minimize memory requirements, such as using one large array instead of two or more individual arrays. Allocated arrays that are no longer needed should be deallocated.

See Also

`qopenmp, Qopenmp` compiler option
Pointer Assignments
Automatic arrays
NULL intrinsic function, which can also be used to disassociate a pointer
TARGET

Effects of Allocation

When you allocate allocatable variables, allocatable arrays, and pointer targets, it can have various effects on your program.

For more information, see the topics in this section.

Allocation of Allocatable Variables

The status of an allocatable variable becomes "allocated" under the following conditions:

- If it is allocated by an ALLOCATE statement
- If it is allocated during assignment
- If it is allocated by the intrinsic subroutine MOVE_ALLOC

An allocatable variable with allocated status can be referenced, defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement. The intrinsic function ALLOCATED returns TRUE for such a variable.

The status of an allocatable variable becomes "unallocated" under the following conditions:

- If it is not allocated
- If it is deallocated
- If it is unallocated by the intrinsic subroutine MOVE_ALLOC

An allocatable variable with unallocated status must not be referenced or defined. It must not be supplied as an actual argument corresponding to a nonallocatable dummy argument, except to certain intrinsic inquiry functions. It can be allocated with the ALLOCATE statement. Deallocating it causes an error condition in the DEALLOCATE statement. The intrinsic function ALLOCATED returns FALSE for such a variable.

At the beginning of execution of a program, allocatable variables are unallocated.

When the allocation status of an allocatable variable changes, the allocation status of any associated allocatable variable also changes accordingly. Allocation of an allocatable variable establishes values for the deferred type parameters of all associated allocatable variables.

An unsaved allocatable local variable of a procedure has a status of unallocated at the beginning of each invocation of the procedure. An unsaved local variable of a construct has a status of unallocated at the beginning of each execution of the construct.

When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate components have an allocation status of unallocated unless the SOURCE= specifier appears and the corresponding component of the SOURCE= expression is allocated.

The intrinsic function ALLOCATED can be used to determine whether a variable is allocated or unallocated.

Allocation of Allocatable Arrays

The bounds (and shape) of an allocatable array are determined when it is allocated. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array specification.

If the lower bound is greater than the upper bound, that dimension has an extent of zero, and the array has a size of zero. If the lower bound is omitted, it is assumed to be 1.

When an array is allocated, it is definable. If you try to allocate a currently allocated allocatable array, an error occurs.

If an allocatable variable is a coarray, the corank is declared, but the cobounds are determined when it is allocated.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
REAL, ALLOCATABLE :: E(:,:)
...
IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7))
```

Allocation Status

During program execution, the allocation status of an allocatable array is one of the following:

- Not currently allocated

  The array was never allocated or the last operation on it was a deallocation. Such an array must not be referenced or defined.
- Currently allocated

  The array was allocated by an ALLOCATE statement. Such an array can be referenced, defined, or deallocated.

If an allocatable array has the SAVE attribute, it has an initial status of "not currently allocated". If the array is then allocated, its status changes to "currently allocated". It keeps that status until the array is deallocated.

If an allocatable array *does not* have the SAVE attribute, it has the status of "not currently allocated" at the beginning of each invocation of the procedure. If the array's status changes to "currently allocated", it is deallocated if the procedure is terminated by execution of a RETURN or END statement.

Example: Allocating Virtual Memory

The following example shows a program that performs virtual memory allocation. This program uses Fortran standard-conforming statements instead of calling an operating system memory allocation routine.

```
! Program accepts an integer and displays square root values

  INTEGER(4) :: N
  READ (5,*) N                          ! Reads an integer value
  CALL MAT(N)
  END

! Subroutine MAT uses the typed integer value to display the square
! root values of numbers from 1 to N (the number read)

  SUBROUTINE MAT(N)
  REAL(4), ALLOCATABLE :: SQR(:)        ! Declares SQR as a one-dimensional
                                        !          allocatable array
  ALLOCATE (SQR(N))                     ! Allocates array SQR

  DO J=1,N
     SQR(J) = SQRT(FLOATJ(J))           ! FLOATJ converts integer to REAL
  ENDDO

  WRITE (6,*) SQR                       ! Displays calculated values
  DEALLOCATE (SQR)                      ! Deallocates array SQR
  END SUBROUTINE MAT
```

See Also

ALLOCATED intrinsic function
ALLOCATE statement

Allocation of Pointer Targets

When a pointer is allocated, the pointer is associated with a target and can be used to reference or define the target. The target can be an array or a scalar, depending on how the pointer was declared.

Other pointers can become associated with the pointer target (or part of the pointer target) by pointer assignment.

In contrast to allocatable arrays, a pointer can be allocated a new target even if it is currently associated with a target. The previous association is broken and the pointer is then associated with the new target.

If the previous target was created by allocation, it becomes inaccessible unless it can still be referred to by other pointers that are currently associated with it.

The intrinsic function ASSOCIATED can be used to determine whether a pointer is currently associated with a target. The association status of the pointer must be *defined*. For example:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR(:)
PTR => TAR
...
IF (ASSOCIATED(PTR,TAR))...
```

See Also

POINTER statement and attribute
Pointer assignments
ASSOCIATED intrinsic function

Effects of Deallocation

When you deallocate allocatable variables, allocatable arrays, and pointer targets, it can have various effects on your program.

For more information, see the topics in this section.

Deallocation of Allocatable Variables

Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE statement.

Deallocating an allocatable variable with the TARGET attribute causes the pointer association status of any pointer associated with it to become undefined.

When the execution of a procedure is terminated by execution of a RETURN or END statement, an unsaved allocatable local variable of the procedure retains its allocation and definition status if it is a function result variable or a subobject of one; otherwise, it is deallocated.

If an executable construct references a function whose result is either allocatable or a structure with a subobject that is allocatable, and the function reference is executed, an allocatable result and any subobject that is an allocated allocatable entity in the result returned by the function is deallocated after execution of the innermost executable construct containing the reference.

If a function whose result is either allocatable or a structure with an allocatable subobject is referenced in the specification part of a scoping unit, and the function reference is executed, an allocatable result and any subobject that is an allocated allocatable entity in the result returned by the function is deallocated before execution of the executable constructs of the scoping unit.

When a procedure is invoked, any allocated allocatable object that is an actual argument corresponding to an INTENT (OUT) allocatable dummy argument is deallocated. Any allocated allocatable object that is a subobject of an actual argument corresponding to an INTENT (OUT) dummy argument is deallocated.

When an intrinsic assignment statement is executed, any noncoarray allocated allocatable subobject of the variable is deallocated before the assignment takes place.

When a variable of derived type is deallocated, any allocated allocatable subobject is deallocated.

If an allocatable component is a subobject of a finalizable object, that object is finalized before the component is automatically deallocated.

The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a *dealloc-opt* argument.

When a DEALLOCATE statement is executed for which an object is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment following the statement is delayed until all other images have executed the same statement the same number of times. If the coarray is a dummy argument, its ultimate argument must be the same coarray on every image.

There is also an implicit synchronization of all images in association with the deallocation of a coarray or coarray subcomponent caused by the execution of a RETURN or END statement.

The intrinsic function ALLOCATED can be used to determine whether a variable is allocated or unallocated.

Consider the following example:

```
SUBROUTINE PROCESS
   REAL, ALLOCATABLE :: TEMP(:)
   REAL, ALLOCATABLE, SAVE :: X(:)
   ...
END SUBROUTINE PROCESS
```

Upon return from subroutine PROCESS, the allocation status of X is preserved because X has the SAVE attribute. TEMP does not have the SAVE attribute, so it will be deallocated if it was allocated. On the next invocation of PROCESS, TEMP will have an allocation status of unallocated.

Deallocation of Allocatable Arrays

If the DEALLOCATE statement specifies an array that is not currently allocated, an error occurs.

If an allocatable array with the TARGET attribute is deallocated, the association status of any pointer associated with it becomes undefined.

If a RETURN or END statement terminates a procedure, an allocatable array has one of the following allocation statuses:

- It keeps its previous allocation and association status if the following is true:
  - It has the SAVE attribute.
  - It is in the scoping unit of a module that is accessed by another scoping unit that is currently executing.
  - It is accessible by host association.
- It remains allocated if it is accessed by use association.
- Otherwise, its allocation status is deallocated.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
SUBROUTINE TEST
  REAL, ALLOCATABLE, SAVE :: F(:,:)

  REAL, ALLOCATABLE :: E(:,:,:)
  ...
  IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7,14))
END SUBROUTINE TEST
```

Note that when subroutine TEST is exited, the allocation status of F is maintained because F has the SAVE attribute. Since E does not have the SAVE attribute, it is deallocated. On the next invocation of TEST, E will have the status of "not currently allocated".

See Also

Host association
TARGET statement and attribute
RETURN statement
END statement
SAVE statement

Deallocation of Pointer Targets

A pointer must not be deallocated unless it has a defined association status. If the DEALLOCATE statement specifies a pointer that has undefined association status, or a pointer whose target was not created by allocation, an error occurs.

A pointer must not be deallocated if it is associated with an allocatable array, or it is associated with a portion of an object (such as an array element or an array section).

If a pointer is deallocated, the association status of any other pointer associated with the target (or portion of the target) becomes undefined.

Execution of a RETURN or END statement in a subprogram causes the pointer association status of any pointer declared or accessed in the procedure to become undefined, unless any of the following applies to the pointer:

- It has the SAVE attribute.
- It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
- It is accessible by host association.
- It is in blank common.
- It is in a named common block that appears in another scoping unit that is currently executing.
- It is the return value of a function declared with the POINTER attribute.

If the association status of a pointer becomes undefined, it cannot subsequently be referenced or defined.

Examples

The following example shows deallocation of a pointer:

```
INTEGER ERR
REAL, POINTER :: PTR_A(:)
...
ALLOCATE (PTR_A(10), STAT=ERR)
...
DEALLOCATE(PTR_A)
```

See Also

POINTER statement and attribute
COMMON statement
NULL intrinsic function
Host association
TARGET statement and attribute
RETURN statement
END statement
SAVE statement

Execution Control

Execution of a program consists of the asynchronous execution of the program in a fixed number of one or more of its images. Each image has its own execution environment, including floating-point status, a set of data objects, input/output units, and procedure pointers.

A program normally executes statements in the order in which they are written. Executable control constructs and statements, and procedure invocations, modify this normal execution by transferring control to another statement in the program, or by selecting blocks (groups) of constructs and statements for execution or repetition.

Procedures may be invoked by the CALL statement (subroutine), during expression evaluation (function), or as part of data definition and handling (user-defined operators and FINAL procedures). There are many ways to define a procedure: for example, external, internal, contained, type-bound, defined operator or assignment, and module. All procedures have one entry point; procedures usually return to their caller.

The control constructs ASSOCIATE, CASE, DO, IF, SELECT RANK, and SELECT TYPE contain blocks and can be named. The name must be a unique identifier in the scoping unit, and must appear on the initial line and terminal line of the construct. On the initial line, the name is separated from the statement keyword by a colon (:).

A block can contain any executable Fortran statement except an END statement. You can transfer control out of a block, but you cannot transfer control into another block.

DO loops cannot partially overlap blocks. The DO statement and its terminal statement must appear together in a statement block.

The following are execution control statements or constructs:

- ASSOCIATE construct

  Creates a temporary association between a named entity and a variable or the value of an expression. The association lasts for the duration of the block.
- BLOCK construct

  Executes a block of statements or constructs that can contain declarations.
- CALL statement

  Transfers control to a subroutine subprogram.
- CASE construct

  Conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement.
- CONTINUE statement

  Primarily used to terminate a labeled DO construct when the construct would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement.
- CRITICAL construct

  Limits execution of a block to one image at a time.
- DO construct

  Controls the repeated execution of a block of statements or constructs. The following statements are used in DO constructs:

  - DO CONCURRENT statement

    Specifies that there are no data dependencies between the iterations of a DO loop.
  - DO WHILE statement

    Executes the range of a DO construct while a specified condition remains true.
  - CYCLE statement

    Interrupts the current execution cycle of the innermost (or named) DO construct.
  - EXIT statement

    Terminates execution of a DO construct
- END statement

  Marks the end of a program unit.
- IF construct and IF statement

  The IF construct conditionally executes one block of statements or constructs. The IF statement conditionally executes one statement. The decision to transfer control or to execute the statement or block is based on the evaluation of a logical expression within the IF statement or construct.
- PAUSE statement

  Temporarily suspends program execution until the user or system resumes execution.

  These statements are deleted features in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.
- RETURN statement

  Transfers control from a subprogram to the calling program unit.
- SELECT RANK construct

  Selects for execution at most one of its constituent blocks based on the rank of an assumed-rank variable.

- SELECT TYPE construct

  Selects for execution at most one of its constituent blocks based on the dynamic type of an expression specified.
- STOP and ERROR STOP statement

  The STOP statement initiates normal termination of an image before the execution of an END statement of the main program. The ERROR STOP statement initiates error termination.

Program Termination

Program termination may involve flushing I/O buffers, closing open I/O files, writing of a STOP code or an ERROR STOP code, or reporting an error status on one or more images. There are two types of image termination, normal termination and error termination.

Normal termination occurs when an image executes a STOP statement or an END [PROGRAM] statement. If there are multiple images running, execution of STOP or END [PROGRAM] statement effects only the image that executes the statement; it has no effect on other images. When an image initiates normal termination, its image status becomes STOPPED, and it waits until all other active images initiate normal termination at which time all images terminate execution. While an image has the status STOPPED, its coarrays are still accessible for reference or definition by other active images.

Error termination of one image causes termination of all other images. Error termination is not initiated if an error condition occurs during the execution of an I/O statement which specifies either an IOSTAT= or ERR= specifier, during the execution of an image control statement that specifies a STAT= specifier, or during a reference to an intrinsic function with a present STAT argument. Otherwise, if an error condition occurs, error termination is initiated.

A program terminates execution when all images that have not failed terminate execution.

Branch Statements Overview

Branching affects the normal execution sequence by transferring control to a labeled statement in the same scoping unit. The transfer statement is called the *branch statement*, while the statement to which the transfer is made is called the *branch target statement*. A branch target statement inside a construct may only be branched to from within the same block of the construct that contains the branch target statement.

Any executable statement can be a branch target statement, except for the following:

- CASE statement
- ELSE statement
- ELSE IF statement
- END FORALL statement
- END WHERE statement
- RANK case statement
- A statement in a FORALL or WHERE construct
- A *type-guard* statement (TYPE IS, CLASS IS, or CLASS DEFAULT)

Certain restrictions apply to the following statements:

Statement	Restriction
DO terminal statement	The branch must be taken from within its nonblock DO construct[1].
END ASSOCIATE	The branch must be taken from within its ASSOCIATE construct.
END BLOCK	The branch must be taken from within its BLOCK construct.
END DO	The branch must be taken from within its block DO construct.

Statement	Restriction
END IF	The branch should be taken from within its IF construct[2].
END SELECT	The branch must be taken from within its SELECT CASE, SELECT RANK, or SELECT TYPE construct.

[1]If the terminal statement is shared by more than one nonblock DO construct, the branch can only be taken from within the innermost DO construct.

[2]You can branch to an END IF statement from outside the IF construct; this is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.

The following are branch statements:

- GOTO - Unconditional statement

  Transfers control to the same branch target statement every time it executes.
- GOTO - COMPUTED statement

  Transfers control to one of a set of labeled branch target statements based on the value of an expression.
- CALL statement with an alternate return specified

  Transfers control to one of the alternate return branch target statements based on the value of the expression on the RETURN statement executed in the called subroutine to return control to the caller.
- An input/output statement with an END=, EOR=, or ERR= specifier

  Transfers control to the specified labeled branch target statement if an end of file condition (END=), an end of record condition (EOR=), or an error condition (ERR=) occurs during execution of the input/output statement.
- ASSIGN and assigned GO TO statements

  Assigns a label to an integer variable. Subsequently, this variable can be used as a branch target statement by an assigned GO TO statement or as a format specifier in a formatted input/output statement.

  These statements are deleted features in Fortran 95. Intel® Fortran fully supports features deleted in Fortran 95.
- IF - Arithmetic statement

  Conditionally transfers control to one of three statements, based on the value of an arithmetic expression.

See Also

ASSOCIATE
BLOCK
DO Statement
FORALL
IF constructs
SELECT CASE
SELECT RANK
SELECT TYPE
WHERE

Effects of DO Constructs

This section discusses ways you can use DO loops and their effects on your program.

For more information, see the topics in this section.

Iteration Loop Control

DO iteration loop control takes the following form:

do-var = *expr1*, *expr2* [, *expr3*]

do-var	Is the name of a scalar variable of type integer or real. It cannot be the name of an array element or structure component.
expr	Is a scalar numeric expression of type integer, logical, or real. If it is not the same type as *do-var*, it is converted to that type.

Description

A DO variable or expression of type real is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.

The following steps are performed in iteration loop control:

1. The expressions *expr1*, *expr2*, and *expr3* are evaluated to respectively determine the initial, terminal, and increment parameters.

   The increment parameter (*expr3*) is optional and must not be zero. If an increment parameter is not specified, it is assumed to be of type default integer with a value of 1.
2. The DO variable (*do-var*) becomes defined with the value of the initial parameter (*expr1*).
3. The iteration count is determined as follows:

```
MAX(INT((expr2 - expr1 + expr3)/expr3), 0)
```

   The iteration count is zero if either of the following is true:

```
expr1 > expr2 and expr3 > 0
expr1 < expr2 and expr3 < 0
```

4. The iteration count is tested. If the iteration count is zero, the loop terminates and the DO construct becomes inactive. (Compiler option f66 can affect this.) If the iteration count is nonzero, the range of the loop is executed.
5. The iteration count is decremented by one, and the DO variable is incremented by the value of the increment parameter, if any.

After termination, the DO variable retains its last value (the one it had when the iteration count was tested and found to be zero).

The DO variable must not be redefined or become undefined during execution of the DO range.

If you change variables in the initial, terminal, or increment expressions during execution of the DO construct, it does not affect the iteration count. The iteration count is fixed each time the DO construct is entered.

Examples

The following example specifies 25 iterations:

```
DO 100 K=1,50,2
```

K=49 during the final iteration, K=51 after the loop.

The following example specifies 27 iterations:

```
DO 350 J=50,-2,-2
```

J=-2 during the final iteration, J=-4 after the loop.

The following example specifies 9 iterations:

```
DO NUMBER=5,40,4
```

NUMBER=37 during the final iteration, NUMBER=41 after the loop. The terminating statement of this DO loop must be END DO.

See Also

f66

Deleted and Obsolescent Language Features for details on obsolescent features in Standard Fortran, as well as features deleted in Standard Fortran

Nested DO Constructs

A DO construct can contain one or more complete DO constructs (loops). The range of an inner nested DO construct must lie completely within the range of the next outer DO construct. Nested nonblock DO constructs can share a labeled terminal statement.

The following figure shows correctly and incorrectly nested DO constructs:

In a nested DO construct, you can transfer control from an inner construct to an outer construct. However, you cannot transfer control from an outer construct to an inner construct.

If two or more nested DO constructs share the same terminal statement, you can transfer control to that statement only from within the range of the innermost construct. Any other transfer to that statement constitutes a transfer from an outer construct to an inner construct, because the shared statement is part of the range of the innermost construct.

When the nested DO constructs contain no statements between the DO statements in the nest of DO constructs, the nest is called "perfectly nested". When perfectly nested DO constructs are modified by a COLLAPSE clause in any of these OpenMP* directives:

- !$OMP DISTRIBUTE
- !$OMP DO
- !$OMP SIMD

There are restrictions on which general compiler directives (see General Compiler Directives) and OpenMP Fortran compiler directives (see OpenMP Fortran Compiler Directives) can appear before the DO statements in the nested DO construct:

- Any OpenMP or general directives that are allowed to affect DO loops are allowed prior to the first DO loop of the "perfectly nested" DO construct.
- It is an error if any of these directives are between any of the perfectly nested DO loop statements for the loops affected by the COLLAPSE clause.

Examples

In the following example, COLLAPSE (1) on affects the DO I loop. Therefore general directives before the DO J loop, which is at level 2, are allowed:

```
!$OMP SIMD collapse (1)
!dir$ prefetch  …           ! this is allowed since it is before the start
                            !    of the perfectly nested DO construct

 do i = …
     !dir$ loop count …     ! this is allowed since collapse only applies
                            !    to the i-loop, not the j-loop
      do j = …
      enddo                 ! end for j-loop
 enddo                      ! end for i-loop
```

In the following example, COLLAPSE (2) affects the DO I loop and the DO J loop but not the DO k loop.

```
!$OMP SIMD collapse (2)
!dir$ prefetch  …           ! this is allowed since it is before the start
                            !    of the perfectly nested DO construct
 do i = …
    do j = …
       !dir$ loop count …   ! this is allowed since collapse only applies to
                            !   the i-loop and the j-loop, not the k-loop
        do k= ….
        enddo               ! end for k-loop
     enddo                  ! end for j-loop
 enddo                      ! end for i-loop
```

In the following example, COLLAPSE (2) affects the DO I loop and the DO J loop so there can be no directives before the DO J loop.

```
!$OMP SIMD collapse (2)
!dir$ prefetch  …           ! this is allowed since it is before the start
                            !    of the perfectly nested DO construct
 do i = …
    !dir$ loop count …      ! this is not allowed: it is breaks the perfectness
                            !    of the  i-loop and the nj-loop collapsing
    do j = …
    enddo                   ! end for j-loop
 enddo                      ! end for i-loop
```

See Also

!$OMP DISTRIBUTE
!$OMP DO
!$OMP SIMD
Rules for General Directives that Affect DO loops

Extended Range

A DO construct has an extended range if both of the following are true:

- The DO construct contains a control statement that transfers control out of the construct.
- Another control statement returns control back into the construct after execution of one or more statements.

The range of the construct is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the construct.

The following rules apply to a DO construct with extended range:

- A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.
- The extended range of a DO statement must not change the control variable of the DO statement.

The following figure shows valid and invalid extended range control transfers:

Control Transfers and Extended Range

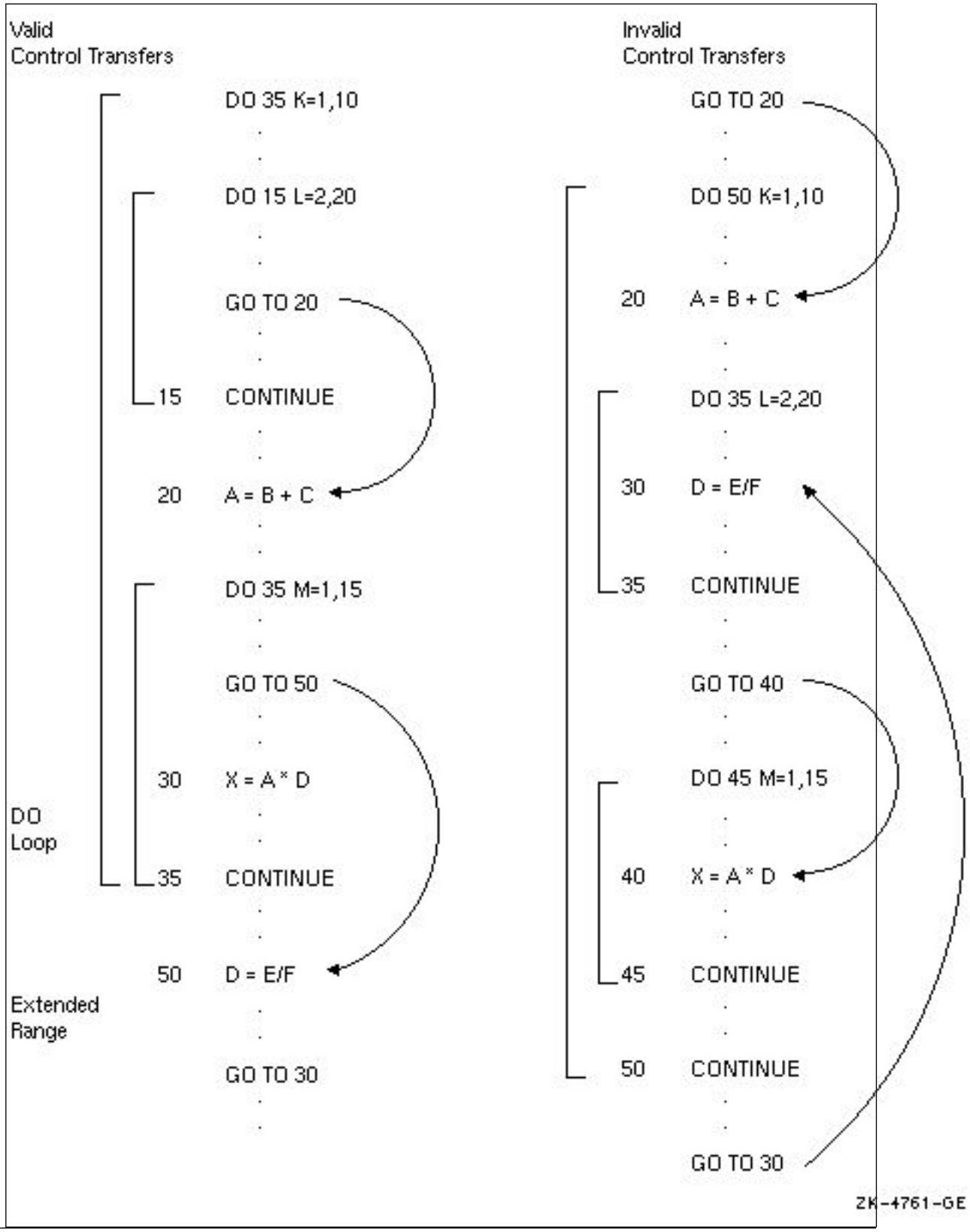


Image Control Statements

Execution of an image control statement divides the execution sequence on an image into segments. The following are image control statements:

- The SYNC ALL statement
- The SYNC IMAGES statement
- The SYNC MEMORY statement
- An ALLOCATE or DEALLOCATE statement that has a coarray allocatable object
- The CRITICAL or END CRITICAL statement
- The EVENT POST or EVENT WAIT statement
- The LOCK or UNLOCK statement
- Any statement that completes execution of a block or procedure and which results in the implicit deallocation of a coarray
- A CALL statement that references the intrinsic subroutine MOVE_ALLOC with coarray arguments
- The STOP statement
- The END statement of a main program

A SYNC MEMORY statement is executed for all image control statements except CRITICAL, END CRITICAL, EVENT POST, EVENT WAIT, LOCK, and UNLOCK.

During an execution of a statement that invokes more than one procedure, at most one invocation can cause execution of an image control statement other than CRITICAL or END CRITICAL.

NOTE
Collective actions may hang if images have stopped or failed and the other images have not all detected the stop or fail; for more information, see FAILED_IMAGES.

See Also

Execution Segments for Images
FAILED_IMAGES

STAT= and ERRMSG= Specifiers in Image Control Statements

If the STAT= specifier appears in an image control statement, successful execution of the statement causes the specified variable to become defined with the value zero. In a given image control statement, the *stat-var* in a STAT= specifier, the *err-var* in an ERRMSG= specifier, the *log-var* in an ACQUIRED_LOCK= specifier, and an event variable or lock variable must not depend on one another.

If the STAT= specifier appears in an EVENT WAIT or a SYNC MEMORY statement and an error occurs, *stat-var* is defined with a processor-dependent positive value that is different from the value of STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

The images involved in the execution of a SYNC ALL statement are the images of the current team. The images involved in the execution of a SYNC IMAGES statement are the specified images and the image executing the SYNC IMAGES statement. The images involved in the execution of an EVENT POST statement are the image executing the statement and the image on which the event variable is located.

If the STAT= specifier appears in an EVENT POST, SYNC ALL, or SYNC IMAGES statement and execution of one of these statements involves synchronization with an image that has initiated normal termination, the variable becomes defined with the value of the constant STAT_STOPPED_IMAGE in the intrinsic module ISO_FORTRAN_ENV. Otherwise, if no other error condition occurs and one of the involved images has failed, the STAT= specifier becomes defined with the value STAT_FAILED_IMAGE in the intrinsic module ISO_FORTRAN_ENV. If any other error condition occurs during execution of one of these statements, the variable becomes defined with a processor-dependent positive integer value that is different from the value of STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

If the STAT= specifier appears in a SYNC ALL or SYNC IMAGES statement and the error STAT_STOPPED_IMAGES occurs, the effect of executing the statement is the same as that of executing the SYNC_MEMORY statement.

If the STAT= specifier appears in a LOCK statement and the lock variable is located on an image that has failed, the specified variable becomes defined with the value STAT_FAILED_IMAGE. If the lock variable is locked by the executing image, the specified variable becomes defined with the value of STAT_LOCKED. Otherwise, if the lock variable is unlocked because the image that locked it has failed, the specified STAT= variable becomes defined with the value STAT_UNLOCKED_FAILED_IMAGE defined in the intrinsic module ISO_FORTRAN_ENV.

If the STAT= specifier appears in an UNLOCK statement and the lock variable is located on an image that has failed, the specifier becomes defined with the value STAT_FAILED_IMAGE. Otherwise, if the lock variable has the value unlocked, the variable specified by the STAT= specifier becomes defined with the value of STAT_UNLOCKED. If the STAT= specifier appears in an UNLOCK statement and the lock variable is locked by a different image, the specified variable becomes defined with the value STAT_LOCKED_OTHER_IMAGE. The named constants STAT_LOCKED, STAT_UNLOCKED, and STAT_LOCKED_OTHER_IMAGE are defined in the intrinsic module ISO_FORTRAN_ENV.

If any other error condition occurs during execution of a LOCK or UNLOCK statement, the specified variable becomes defined with a positive integer value that is different from STAT_LOCKED, STAT_UNLOCKED, STAT_UNLOCKED_FAILED_IMAGE, and STAT_LOCKED_OTHER_IMAGE.

If an image completes execution of a CRITICAL statement that has a STAT= specifier and the previous image that entered the CRITICAL construct failed during execution of the construct, the specifier becomes defined with the value STAT_FAILED_IMAGE and the execution of the construct executes normally. If any other error occurs during execution of the construct, the specifier becomes defined with a positive integer value other different from STAT_FAILED_IMAGE.

If an error condition occurs during execution of an image control statement that does not contain the STAT= specifier, error termination is initiated.

If an ERRMSG= specifier appears in an image control statement, and an error condition occurs during execution of that statement, the processor will assign an explanatory message to the specified variable. If no such condition occurs, the processor will not change the value or definition status of the variable.

The set of error conditions that can occur during execution of an image control statement is processor dependent.

See Also

ISO_FORTRAN_ENV Module

Execution Segments for Images

On each image, a segment is the sequence of statements executed before the first execution of an image control statement, between the execution of two image control statements, or after the last execution of an image control statement.

The segment executed immediately before the execution of an image control statement includes the evaluation of all expressions within that image control statement.

A coarray can be referenced or defined by execution of an atomic subroutine during the execution of a segment that is unordered, relative to the execution of a segment in which the coarray is referenced or defined by execution of an atomic subroutine. An event variable can be referenced or defined during the execution of a segment that is unordered relative to the execution of another segment in which that event variable is defined. Otherwise, the following rules apply:

- If a variable is defined on an image in a segment, it must not be referenced, defined, or become undefined in a segment on another image unless the segments are ordered.
- If the allocation of an allocatable subobject of a coarray or the pointer association of a pointer subobject of a coarray is changed on an image in a segment, that subobject must not be referenced or defined in a segment on another image unless the segments are ordered.
- If a procedure invocation on image P is in execution in segments Pi, Pi+1, ..., Pk and defines a noncoarray dummy argument, the effective argument must not be referenced, defined, or become undefined on another image Q in a segment Qj unless Qj precedes Pi or succeeds Pk (because a copy of the actual argument may be passed to the procedure)

Incorrect sequencing of image control statements can suspend execution indefinitely. For example, one image might be executing a SYNC ALL statement while another is executing an ALLOCATE statement for a coarray.

See Also

Image Control Statements

Program Units and Procedures

A Fortran program consists of one or more program units. There are four types of program units:

- Main program

  The program unit that denotes the beginning of execution. It may or may not have a PROGRAM statement as its first statement.
- External procedures

  Program units that are either user-written functions or subroutines.
- Modules and submodules

  Program units that contain declarations, type definitions, procedures, or interfaces that can be shared by other program units. A module can be extended by one or more program units called submodules. A submodule can in turn be extended by one or more submodules.
- Block data program units

  Program units that provide initial values for variables in named common blocks.

A program unit does not have to contain executable statements; for example, it can be a module containing interface blocks for subroutines.

A procedure can be invoked during program execution to perform a specific task. It specifies the EXTERNAL attribute for all procedure entities in the procedure declaration list. A procedure declaration is denoted by a PROCEDURE statement.

There are several kinds of procedures, as follows:

Kind of Procedure	Description
External Procedure	A procedure that is not part of any other program unit.
Module Procedure	A procedure defined within a module.
Internal Procedure[1]	A procedure (other than a statement function) contained within a main program, function, or subroutine.
Intrinsic Procedure	A procedure defined by the Fortran language.
Dummy Procedure	A dummy argument specified as a procedure or appearing in a procedure reference. A dummy procedure with the POINTER attribute is a dummy procedure pointer.
Procedure Pointer	A procedure that has the EXTERNAL and POINTER attributes. It may be pointer associated with an external procedure, a module procedure, an intrinsic procedure, or a dummy procedure that is not a procedure pointer.
Statement function	A computing procedure defined by a single statement.

Kind of Procedure	Description

[1] The program unit that contains an internal procedure is called its *host*.

A *function* is invoked in an expression using the name of the function or a defined operator. It returns a a single value (function result) that is used to evaluate the expression.

A *subroutine* is invoked in a CALL statement or by a defined assignment statement. It does not directly return a value, but values can be passed back to the calling program unit through arguments (or variables) known to the calling program.

Recursion (direct or indirect) is permitted for functions and subroutines.

A procedure interface refers to the properties of a procedure that interact with or are of concern to the calling program. A procedure interface can be explicitly defined in interface blocks. All program units, except block data program units, can contain interface blocks.

See Also

Program structure
Intrinsic procedures
Scope
RECURSIVE keyword

Main Program

A main program is a program unit whose first statement is not a SUBROUTINE, FUNCTION, MODULE, SUBMODULE, or BLOCK DATA statement. Program execution always begins with the first executable statement in the main program, so there must be exactly one main program unit in every executable program. For more information, see PROGRAM.

Procedure Characteristics

The characteristics of a procedure are as follows:

- Whether it is classified as a function or subroutine
- The characteristics of its result value if it is a function
- The characteristics of its dummy arguments
- Whether it is pure
- Whether it is impure
- Whether it is elemental
- Whether it has the BIND attribute

Characteristics of Dummy Arguments

Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, or an asterisk indicating an alternate return indicator.

The characteristics of a dummy data object are as follows:

- Its type and type parameters (if any)
- Its shape
- Its intent
- Its corank
- Its codimensions
- Whether it is optional
- Whether it is allocatable
- Whether it has the ASYNCHRONOUS, CONTIGUOUS, VALUE, or VOLATILE attribute

- Whether it is a pointer or a target
- Whether it is polymorphic
- Whether or not it is assumed rank or assumed type

If a type parameter of an object or a bound of an array is not a constant expression, a characteristic is the exact dependence on the entities in the expression. Another characteristic is whether a shape, size, or type parameter is assumed or deferred.

The characteristics of a dummy procedure are as follows:

- Whether it is optional
- The explicitness of its interface
- Its characteristics as a procedure if the interface is explicit
- Whether it is a pointer

Characteristics of Function Results

The characteristics of a function result are as follows:

- Its type, type parameters (if any), and rank
- Whether it is allocatable
- Whether it has the CONTIGUOUS attribute
- Whether it is a pointer
- Whether it is a procedure pointer
- Whether it is polymorphic

If a function result is an array that is not allocatable or a pointer, a characteristic is its shape.

If a type parameter of a function result or a bound of a function result array is not a constant expression, a characteristic is the exact dependence on the entities in the expression.

If type parameters of a function result are deferred, a characteristic is which parameters are deferred. Another characteristic is whether the length of a character function result is assumed.

Modules and Module Procedures

A module program unit contains specifications and definitions that can be made accessible to other program units. There are two types of modules, intrinsic and nonintrinsic. Intrinsic modules are included in the Fortran library; nonintrinsic modules are user-defined.

For the module to be accessible, the other program units must reference its name in a USE statement, and the module entities must be public. This module reference lets the program unit access the public definitions, specifications, and procedures in the module. Entities in a module are public by default, unless the USE statement specifies otherwise or the PRIVATE attribute is specified for the module entities.

A module reference causes use association between the using program unit and the entities in the module.

A submodule extends a module or another submodule. It provides additional structuring facilities for modules. Data objects and procedure pointers declared in a module implicitly have the SAVE attribute.

For more information on module program units, see MODULE. For more information about submodule program units, see SUBMODULE.

A module procedure is a procedure declared and defined in a module, between its CONTAINS and END statements. For more information, see MODULE PROCEDURE.

See Also

PRIVATE attribute
PUBLIC attribute
Use association

Separate Module Procedures

A *separate module procedure* is a module procedure that is declared in a *separate interface body*. To denote separate module procedures, you must specify the keyword MODULE as a prefix in the initial statement of both of the following:

- A separate module procedure body
- A separate interface body

The interface block that contains the separate interface body must be nonabstract.

A separate interface body can be declared in a module or a submodule. The corresponding separate module procedure may be defined (implemented) in the same module or submodule or a descendent of the module or submodule. A separate module procedure can only be defined once.

Usually, the separate interface body is specified in a module and the separate module procedure is defined in a descendent submodule.

In the following example, FOO is a separate module procedure whose interface is specified in module M while the procedure body is defined in submodule A:

```
module M
  type tt
    real r
  end type tt

  interface
    real module function FOO (arg)
      type(tt), intent(in) :: arg
    end function FOO
  end interface
end module M

submodule (M) A
contains
  real module function FOO (arg) result(res)
     type(tt), intent(in) :: arg
     res = arg%r
  end function FOO
end submodule A
```

A separate module procedure is accessible by use association only if its interface body is declared in a module and it is public (for example, FOO in the above example). If a separate module interface is declared in a submodule, the module procedure is only accessible in that submodule or its descendent submodules.

A separate module procedure interface body (either in a module or submodule) has access to entities in its host through host association.

> **NOTE**
> For an interface body that is *not* a separate interface body, IMPORT statements are required to make entities accessible by host association. However, IMPORT statements are not permitted in a separate interface body.

The initial statement of a separate module procedure body can take one of the two following forms:

- MODULE appears as a prefix for a FUNCTION or SUBROUTINE statement

  The following shows an example using this form:

```
submodule (M) A
contains
  real module function foo (arg) result(res)
```

```
    type(tt), intent(in) :: arg
    res = arg%r
  end function foo
end submodule A
```

With this form, a separate module procedure must specify the same characteristics and dummy argument names as its corresponding separate interface body.

They must both be functions or subroutines; they must both be pure or not; they must both be elemental or not. The characteristics of its dummy arguments and the characteristics of the function result must also be the same.

NON_RECURSIVE or RECURSIVE can appear only if NON_RECURSIVE or RECURSIVE appear respectively as a prefix in the corresponding separate interface body.

Note that the restrictions of matching dummy argument names and matching PURE, NON_RECURSIVE, and RECURSIVE specifications only apply to separate module procedures. For an external procedure, the procedure definition and its interface body can differ with regard to dummy argument names, to whether it is pure, and to whether or not it is recursive.

A procedure defined in a submodule with the BIND attribute cannot have a binding label (that is, BIND(C, NAME="a-binding-label") unless it is a separate module procedure and its interface is declared in the ancestor module. The binding label specified in a separate module procedure definition must match the binding label specified in the separate interface body.

For this form, the result variable name for a function is determined by the FUNCTION statement in the module subprogram. The result variable name in the interface is ignored.

- MODULE PROCEDURE statement

  This has the following form:

```
MODULE PROCEDURE procedure-name
  [specification-part]
  [execution-part]
  [internal-subprogram-part]
END [PROCEDURE [procedure-name]]
```

  The following shows an example using this form:

```
submodule (M) A
contains
  module procedure foo
     foo = arg%r
  end procedure foo
end submodule A
```

  This syntax avoids the redeclaration of the function or subroutine in the separate module procedure definition and just takes the characteristics, dummy argument names, and the function result variable name from the separate interface body.

A separate module procedure does not have to be defined. The separate module procedure interface can be used to specify an explicit interface; however, the procedure must not be called.

> **NOTE**
> Two modules cannot have USE statements that reference each other because circular reference is not allowed. However, you can solve that problem by putting at least one side of the USEs in submodules. This is because submodules cannot be referenced by use association and the USE statements in submodules are effectively hidden.

Examples

See the examples in SUBMODULE.

See Also
MODULE
SUBMODULE

Intrinsic Modules

Intrinsic modules, like other module program units, contain specifications and definitions that can be made accessible to other program units. The intrinsic modules are part of the Fortran library.

An intrinsic module is specified in a USE statement, as follows:

`USE, INTRINSIC :: mod-name [, rename-list] ...`

`USE, INTRINSIC :: mod-name, ONLY : [, only-list]`

mod-name	Is the name of the intrinsic module.
rename-list	See the description in USE.
only-list	See the description in USE.

Procedures and types defined in an intrinsic module are not themselves intrinsic.

An intrinsic module can have the same name as other global entities, such as program units, common blocks, or external procedures. A scoping unit must not access both an intrinsic module and a non-intrinsic module with the same name.

When INTRINSIC is used, *mod-name* must be the name of an intrinsic module. If NON_INTRINSIC is used, *mod-name* must be the name of an nonintrinsic module. If neither is specified, *mod-name* must be the name of an intrinsic or nonintrinsic module. If both are provided, the nonintrinsic module is used.

The following intrinsic modules are included in the Fortran library: ISO_C_BINDING, ISO_FORTRAN_ENV, and IEEE Intrinsic Modules.

ISO_C_BINDING Module

The ISO_C_BINDING intrinsic module provides access to data entities that are useful in mixed-language programming. It takes the following form:

`USE, INTRINSIC :: ISO_C_BINDING`

This intrinsic module provides access to the following data entities:

- Named Constants
- Derived Types

  Derived type C_PTR is interoperable with any C object pointer type. Derived type C_FUNPTR is interoperable with any C function pointer type.
- Intrinsic Module Procedures

See Also
Standard Tools for Interoperabillity

Named Constants in the ISO_C_BINDING Module

The ISO_C_BINDING named constants represent kind type parameters of data representations compatible with C types.

Intrinsic-Type Constants

The following table shows interoperable Fortran types and C Types.

Fortran Type	Named Constant for the KIND	C Type
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char, unsigned char
	C_SIZE_T	size_t
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intptr_t
	C_PTRDIFF_T	ptrdiff_t
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL[1]	C_BOOL	_Bool
CHARACTER[2]	C_CHAR	char

[1] Use compiler option `fpscomp logicals` so that .TRUE. is 1 and .FALSE. is 0 as defined for C's _Bool.

Fortran Type	Named Constant for the KIND	C Type
[2]For character type, the length type parameter must be omitted or it must be specified by a constant expression whose value is one.		

For example, an integer type with the kind type parameter C_LONG is interoperable with the C integer type "long" or any C type derived from "long".

The value of C_INT will be a valid value for an integer kind type parameter on the processor. The values for the other integer named constants (C_INT*) will be a valid value for an integer kind type parameter on the processor, if any, or one of the following:

- -1 if the C processor defines the corresponding C type and there is no interoperating Fortran processor kind
- -2 if the C processor does not define the corresponding C type

The values of C_FLOAT, C_DOUBLE, and C_LONGDOUBLE will be a valid value for a real kind type parameter on the processor, if any, or one of the following:

- -1 if the C processor's type does not have a precision equal to the precision of any of the Fortran processor's real kinds
- -2 if the C processor's type does not have a range equal to the range of any of the Fortran processor's real kinds
- -3 if the C processor's type has neither the precision or range equal to the precision or range of any of the Fortran processor's real kinds
- -4 if there is no interoperating Fortran processor or kind for other reasons

The values of C_FLOAT_COMPLEX, C_DOUBLE_COMPLEX, and C_LONG_DOUBLE_COMPLEX will be the same as those of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE, respectively.

The value of C_BOOL will be a valid value for a logical kind parameter on the processor, if any, or -1.

The value of C_CHAR is the character kind.

Character Constants

The following table shows interoperable named constants and C characters:

Fortran Named Constant	Definition	C Character
C_NULL_CHAR	null character	'\0'
C_ALERT	alert	'\a'
C_BACKSPACE	backspace	'\b'
C_FORM_FEED	form feed	'\f'
C_NEW_LINE	new line	'\n'
C_CARRIAGE_RETURN	carriage return	'\r'
C_HORIZONTAL_TAB	horizontal tab	'\t'
C_VERTICAL_TAB	vertical tab	'\v'

Derived-Type Constants

The constant C_NULL_PTR is of type C_PTR; it has the value of a C null data pointer. The constant C_NULL_FUNPTR is of type C_FUNPTR; it has the value of a C null function pointer.

Intrinsic Module Procedures - ISO_C_BINDING

The following procedures are provided with the ISO_C_BINDING intrinsic module:

- C_ASSOCIATED
- C_F_POINTER
- C_F_PROCPOINTER
- C_FUNLOC
- C_LOC
- C_SIZEOF

C_F_POINTER and C_F_PROCPOINTER are impure, the other procedures are pure.

ISO_FORTRAN_ENV Module

The ISO_FORTRAN_ENV intrinsic module provides information about the Fortran run-time environment. It takes the following form:

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
```

This intrinsic module provides access to the following data entities:

- Named Constants
- Derived Types
- Intrinsic Module Procedures

Named Constants in the ISO_FORTRAN_ENV Module

The ISO_FORTRAN_ENV intrinsic module provides the following named constants that you can use to get information on the Fortran environment. They are all scalars of type default integer.

Named Constant	Definition
ATOMIC_INT_KIND	Is the kind type parameter used when defining integer variables used in atomic operations.
ATOMIC_LOGICAL_KIND	Is the kind type parameter used when defining logical variables used in atomic operations.
CHARACTER_KINDS	Is the kind type parameter supported by the processor that is used when defining variables of type character. This is a default integer array constant. The rank of the array is one, its lower bound is one, and its size is the number of character kinds supported. In Intel® Fortran, its value is [1].
CHARACTER_STORAGE_SIZE	Is the size of the character storage unit expressed in bits.
ERROR_UNIT	Identifies the preconnected external unit used for error reporting.
FILE_STORAGE_SIZE	Is the size of the file storage unit expressed in bits. To use this constant, compiler option assume byterecl must be enabled.
INPUT_UNIT	Identifies the preconnected external unit as the one specified by an asterisk in a READ statement. To use this constant, compiler option assume noold_unit_star must be enabled.

Named Constant	Definition
INT8 INT16 INT32 INT64	Are the kind type parameters that specify an INTEGER type whose storage size is 8 bits, 16 bits, 32 bits, and 64 bits, respectively. If, for any of these constants, the processor supports more than one kind of that size, the kind value is determined by the processor. If the processor supports no kind of a particular size, that constant is equal to -2 if the processor supports a kind with larger size; otherwise, -1. In Intel Fortran, their respective values are 1, 2, 4, and 8.
INTEGER_KINDS	Is the kind type parameter supported by the processor that is used when defining variables of type integer. This is a default integer array constant. The rank of the array is one, its lower bound is one, and its size is the number of integer kinds supported. In Intel Fortran its value is [1, 2, 4, 8].
IOSTAT_END	Is the value assigned to the variable specified in an IOSTAT= specifier if an end-of-file condition occurs during execution of an input/output statement and no error condition occurs.
IOSTAT_EOR	Is the value assigned to the variable specified in an IOSTAT= specifier if an end-of-record condition occurs during execution of an input/output statement and no error condition occurs.
IOSTAT_INQUIRE_INTERNAL_UNIT	Is the value assigned to the variable specified in an IOSTAT= specifier in an INQUIRE statement if the unit number identifies an internal unit. This is a negative value, indicating an error condition.
LOGICAL_KINDS	Is the kind type parameter supported by the processor that is used when defining variables of type logical. This is a default integer array constant. The rank of the array is one, its lower bound is one, and its size is the number of logical kinds supported. In Intel® Fortran its value is [1, 2, 4, 8].
NUMERIC_STORAGE_SIZE	Is the size of the numeric storage unit expressed in bits.
OUTPUT_UNIT	Identifies the preconnected external unit as the one specified by an asterisk in a WRITE statement. To use this constant, compiler option assume noold_unit_star must be enabled.
REAL_KINDS	Is the kind type parameter supported by the processor that is used when defining variables of type real. This is a default integer array constant. The rank of the array is one, its lower bound is one, and its size is the number of real kinds supported. In Intel Fortran its value is [4, 8, 16].
REAL32 REAL64 REAL128	Are the kind type parameters that specify a real type whose storage size is 32 bits, 64 bits, and 128 bits, respectively. If, for any of these constants, the processor supports more than one kind of that size,

Named Constant	Definition
	the kind value is determined by the processor. If the processor supports no kind of a particular size, that constant is equal to -2 if the processor supports kinds of a larger size; otherwise, -1. In Intel Fortran, their respective values are 4, 8, and 16.
STAT_FAILED_IMAGE	The value assigned to the variable specified in a STAT= specifier of an image control statement or coindexed object reference, or the STAT argument of a collective or atomic subroutine if an image involved in the execution of that statement, reference, or subroutine has failed.
STAT_LOCKED	The value assigned to the variable specified in a STAT= specifier of a LOCK statement if the lock variable is locked by the executing image.
STAT_LOCKED_OTHER_IMAGE	The value assigned to the variable specified in a STAT= specifier of an UNLOCK statement if the lock variable is locked by another image.
STAT_STOPPED_IMAGE	The value assigned to the variable specified in a STAT= specifier of a statement if execution of the statement requires synchronization with an image that has initiated normal termination (an image control statement). It is the value assigned to the STAT variable of a collective subroutine if the current team contains images that inititiated normal termination.
STAT_UNLOCKED	The value assigned to the variable specified in a STAT= specifier of an UNLOCK statement if the lock variable is unlocked.
STAT_UNLOCKED_FAILED_IMAGE	The value assigned to the STAT= specifier of a LOCK statement if the lock variable is unlocked because the image that locked it has failed.

Derived Types in the ISO_FORTRAN_ENV Module

The ISO_FORTRAN_ENV intrinsic module provides the following predefined derived types.

EVENT_TYPE	This is a derived type with private components. It is an extensible type with no type parameters. Each component that is nonallocatable is initialized by default. A scalar variable of type EVENT_TYPE is an event variable. The value of an event variable contains its event count, which is modified by a sequence of EVENT POST and EVENT WAIT statements. A modification to the event count is as if the intrinsic ATOMIC_ADD were executed with a variable that stores the count as its ATOM argument. A coarray that is of type EVENT_TYPE can be referenced or defined during execution of a segment that is unordered relative to the execution of another segment in which that coarray is

	defined. The event count is an integer of ATOMIC_INT_KIND. The initial value of the event count of a variable of type EVENT_TYPE is zero. A named entity with declared type EVENT_TYPE, or which has a noncoarray potential subobject component with declared type EVENT_TYPE, must be a variable. A component with type EVENT_TYPE must be a data component. A named variable with declared type EVENT_TYPE must be a coarray. A named variable with a noncoarray potential subobject component of type EVENT TYPE must be a coarray. An event variable must not appear in a variable-definition context except as the *event-var* argument in an EVENT POST or EVENT WAIT statement, as an allocatable object, or as an actual argument in a reference to a procedure with an explicit interface if the corresponding dummy argument has INTENT(INOUT). A variable with a nonpointer subobject of type EVENT_TYPE must not appear in a variable-definition context except as an allocatable object in an ALLOCATE statement without a SOURCE= specifier, as an allocatable object in a DEALLOCATE statement, or as an actual argument in a reference to a procedure with an explicit interface if the corresponding dummy argument has INTENT(INOUT). If EXTENDS appears in a TYPE statement and the type being defined has a potential subobject component of type EVENT_TYPE, its parent type must be EVENT_TYPE or LOCK_TYPE, or have a potential subobject component of type EVENT_TYPE or LOCK_TYPE.
LOCK_TYPE	This is a derived type with private components; none of the components can be allocatable or a pointer. It is an extensible type with no type parameters. It does not have the BIND (C) attribute or type parameters, and is not a sequence type. All components have default initialization. A scalar variable of type LOCK_TYPE is a lock variable. A lock variable can have one of two states: locked or unlocked. The unlocked state is represented by the one value that is the initial value of a LOCK_TYPE variable. The locked state is represented by all other values. The value of a lock variable can be changed with the LOCK and UNLOCK statements. A named variable of type LOCK_TYPE must be a coarray. A named variable with a noncoarray sub-component of type LOCK_TYPE must also be a coarray. A named constant can not be of type LOCK_TYPE, nor can it have a noncoarrray potential subobject component with a declared LOCK_TYPE.

	If EXTENDS appears in TYPE statement and the type being defined has a potential subobject component of type LOCK_TYPE, its parent type must be EVENT_TYPE or LOCK_TYPE, or have a potential subobject component of type EVENT_TYPE or LOCK_TYPE. A lock variable must not appear in a variable definition context except as the lock-variable in a LOCK or UNLOCK statement, as an allocatable object, or as an actual argument in a reference to a procedure with an explicit interface where the corresponding dummy argument has INTENT (INOUT). A variable with a subobject of type LOCK_TYPE must not appear in a variable definition context except as an allocatable object or as an actual argument in a reference to a procedure with an explicit interface where the corresponding dummy argument has INTENT (INOUT).

Intrinsic Module Procedures - ISO_FORTRAN_ENV

The following procedures are provided with the ISO_FORTRAN_ENV intrinsic module:

- COMPILER_OPTIONS
- COMPILER_VERSION

IEEE Intrinsic Modules and Procedures

Intel® Fortran includes IEEE intrinsic modules that support IEEE arithmetic and exception handling. The modules contain derived data types that include named constants for controlling the level of support, and intrinsic module procedures.

To include an IEEE module in your program, specify the intrinsic module name in a USE statement; for example:

```
USE,INTRINSIC :: IEEE_ARITHMETIC
```

You must include the INTRINSIC attribute or the processor will look for a non-intrinsic module. Once you include a module, all related intrinsic procedures are defined.

The following three IEEE intrinsic modules are described in this section:

- IEEE_ARITHMETIC
- IEEE_EXCEPTIONS
- IEEE_FEATURES

Determining Availability of IEEE Features

Before using a particular IEEE feature, you can determine whether your processor supports it by using the IEEE inquiry functions (listed in below section Restrictions for IEEE Intrinsic Procedures).

For example:

- To determine whether IEEE arithmetic is available for a particular kind of real, use intrinsic module function IEEE_SUPPORT_DATATYPE.
- To determine whether you can change a rounding mode, use intrinsic module function IEEE_SUPPORT_ROUNDING.
- To determine whether a divide operation will be supported with the accuracy specified by the IEEE standard, use intrinsic module function IEEE_SUPPORT_DIVIDE.

- To determine whether you can control halting after an exception has occurred, use intrinsic module function IEEE_SUPPORT_HALTING.
- To determine which exceptions are supported in a scoping unit, use intrinsic module function IEEE_SUPPORT_FLAG.
- To determine whether all IEEE features are supported, use intrinsic module function IEEE_SUPPORT_STANDARD.

The compiler establishes the initial IEEE floating-point environment. The user can affect this initial environment with several different command-line options. For the IEEE intrinsic module procedures to work as defined by the Fortran Standard, the following command lines options must be set as follows:

- Option `/fpe:3` (Windows*) or `-fpe3` (Linux* and macOS*) must be set to disable all floating-point exceptions.
- Option `/Qftz-` (Windows*) or `-no-ftz` (Linux* and macOS*) must be set to disable flushing subnormal results to zero (notice that all optimization levels, except `O0`, set `ftz` so the user has to explicitly set "no ftz").
- Option `/fp:precise` (Windows*) or option `-fp-model precise` (Linux* and macOS*) must be set to disable floating-point exception semantics.

Restrictions for IEEE Intrinsic Procedures

The following intrinsic procedures can only be invoked if IEEE_SUPPORT_DATATYPE is true for their arguments of type REAL:

IEEE_CLASS	IEEE_REM
IEEE_COPY_SIGN	IEEE_RINT
IEEE_FMA	IEEE_SCALB
IEEE_IS_FINITE	IEEE_SET_ROUNDING_MODE [3]
IEEE_NEGATIVE	IEEE_SIGNALING_EQ
IEEE_INT	IEEE_SIGNALING_GE
IEEE_IS_NORMAL	IEEE_SIGNALING_GT
IEEE_LOGB	IEEE_SIGNALING_LE
IEEE_MAX_NUM	IEEE_SIGNALING_LT
IEEE_MAX_NUM_MAG	IEEE_SIGNALING_NE
IEEE_MIN_NUM	IEEE_SIGNBIT
IEEE_MIN_NUM_MAG	IEEE_SUPPORT_DENORMAL
IEEE_NEXT_AFTER	IEEE_SUPPORT_DIVIDE
IEEE_NEXT_DOWN [1]	IEEE_SUPPORT_INF
IEEE_NEXT_UP [1]	IEEE_SUPPORT_IO
IEEE_QUIET_EQ	IEEE_SUPPORT_NAN
IEEE_QUIET_GE	IEEE_SUPPORT_ROUNDING
IEEE_QUIET_GT	IEEE_SUPPORT_SQRT
IEEE_QUIET_LE	IEEE_SUPPORT_SUBNORMAL

IEEE_QUIET_LT	IEEE_SUPPORT_UNORDERED
IEEE_QUIET_NE	IEEE_SUPPORT_VALUE
IEEE_REAL [2]	IEEE_VALUE

[1] IEEE_SUPPORT_INF() must be true if IEEE_NEXT_DOWN is called with the argument -HUGE (X) or if IEEE_NEXT_UP is called with the argument HUGE (X).

[2] IEEE_SUPPORT_DATATYPE (IEEE_REAL (A, KIND)) must also be true.

[3] IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X) must also be true.

For example, the IEEE_IS_NORMAL(X) function can only be invoked if IEEE_SUPPORT_DATATYPE(X) has the value true. Consider the following:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
...
  IF IEEE_SUPPORT_DATATYPE(X) THEN
    IF IEEE_IS_NORMAL(X) THEN
      PRINT *, ' X is a "normal" '
    ELSE
      PRINT *, ' X is not "normal" '
    ENDIF
  ELSE
    PRINT *, ' X is not a supported IEEE type '
  ENDIF
...
```

Certain other IEEE intrinsic module procedures have similar restrictions:

- IEEE_IS_NAN(X) can only be invoked if IEEE_SUPPORT_NAN(X) has the value true.
- IEEE_SET_HALTING_MODE(FLAG, HALTING) can only be invoked if IEEE_SUPPORT_HALTING(FLAG) has the value true.
- IEEE_GET_UNDERFLOW_MODE(GRADUAL) can only be invoked if IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is true for some X.

For intrinsic module function IEEE_CLASS(X), some of the possible return values also have restrictions. These restrictions are also true for argument CLASS in intrinsic module function IEEE_VALUE(X, CLASS):

- IEEE_POSITIVE_INF and IEEE_NEGATIVE_INF can only be returned if IEEE_SUPPORT_INF(X) has the value true.
- IEEE_POSITIVE_DENORMAL, IEEE_POSITIVE_SUBNORMAL, IEEE_NEGATIVE_SUBNORMAL, and IEEE_NEGATIVE_DENORMAL can only be returned if IEEE_SUPPORT_DENORMAL(X) and IEEE_SUPPORT_SUBNORMAL(X) have the value true.
- IEEE_SIGNALING_NAN and IEEE_QUIET_NAN can only be returned if IEEE_SUPPORT_NAN(X) has the value true.

IEEE_ARITHMETIC Intrinsic Module

The IEEE_ARITHMETIC module contains derived data types that include named constants for controlling the level of support, and intrinsic module procedures.

The derived types in the intrinsic modules have components that are private. The IEEE_ARITHMETIC intrinsic module supports IEEE arithmetic and features. It defines the following derived types:

- IEEE_CLASS_TYPE: Identifies a class of floating-point values. Its values are the following named constants:

IEEE_SIGNALING_NAN	IEEE_NEGATIVE_NORMAL
IEEE_QUIET_NAN	IEEE_POSITIVE_DENORMAL
IEEE_POSITIVE_INF	IEEE_NEGATIVE_DENORMAL

IEEE_NEGATIVE_INF	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_ZERO
IEEE_OTHER_VALUE	

- IEEE_ROUND_TYPE: Identifies a rounding mode. Its values are the following named constants:

IEEE_AWAY [1]	IEEE_OTHER [4]
IEEE_DOWN [2]	IEEE_TO_ZERO [5]
IEEE_NEAREST [3]	IEEE_UP [6]

[1] Corresponds to ISO/IEC/IEEE 60559:2011 rounding attribute `roundTiesToAway`. Intel hardware does not support this mode.

[2] Corresponds to ISO/IEC/IEEE 60559:2011 rounding attribute `roundTowardNegative`.

[3] Corresponds to ISO/IEC/IEEE 60559:2011 rounding attribute `roundTiesToEven`.

[4] Specifies the rounding mode does not conform to the IEEE standard.

[5] Corresponds to ISO/IEC/IEEE 60559:2011 rounding attribute `roundTowardZero`.

[6] Corresponds to ISO/IEC/IEEE 60559:2011 rounding attribute `roundTowardPositive`.

The IEEE_ARITHMETIC intrinsic module also defines the following operators:

- Elemental operator = = for two values of one of the above types to return true if the values are the same; otherwise, false.
- Elemental operator /= for two values of one of the above types to return true if the values differ; otherwise, false.

The IEEE_ARITHMETIC module includes support for IEEE_EXCEPTIONS module, and public entities in IEEE_EXCEPTIONS module are also public in the IEEE_ARITHMETIC module.

To see a summary of all the IEEE_ARITHMETIC intrinsic procedures, see IEEE Intrinsic Modules Quick Reference Tables

IEEE_EXCEPTIONS Intrinsic Module

The IEEE_EXCEPTIONS module contains derived data types that include named constants for controlling the level of support, and intrinsic module procedures.

The derived types in the intrinsic modules have components that are private. The IEEE_EXCEPTIONS intrinsic module supports the setting, clearing, saving, restoring, or testing of exception flags. It defines the following derived types:

- IEEE_FLAG_TYPE: Identifies an exception flag for errors that occur during an IEEE arithmetic operation or assignment. Its values are the following named constants:

IEEE_INVALID	IEEE_DIVIDE_BY_ZERO
IEEE_OVERFLOW	IEEE_INEXACT
IEEE_UNDERFLOW	

  Each of the above exceptions has a flag whose value is either quiet or signaling. The initial value is quiet and it signals when the associated exception occurs. To determine the value of a flag, use intrinsic module subroutine IEEE_GET_FLAG. To change the status for a flag, use intrinsic module subroutine IEEE_SET_FLAG or IEEE_SET_STATUS.

  If a flag is signaling on entry to a procedure, the processor sets it to quiet on entry and restores it to signaling on return.

  If a flag is quiet on entry to a procedure with access to modules IEEE_ARITHMETIC or IEEE_EXCEPTIONS, and is signaling on return, the processor will not restore it to quiet.

  The IEEE_FLAG_TYPE module also defines the following named array constants:

  - IEEE_USUAL=(/IEEE_OVERFLOW,IEEE_DIVIDE_BY_ZERO, IEEE_INVALID/)
  - IEEE_ALL=(/IEEE_USUAL,IEEE_UNDERFLOW,IEEE_INEXACT/)
- IEEE_MODES_TYPE: The floating-point modes are the values of the rounding modes, underflow mode, and halting mode. They can be saved in a variable of type IEEE_MODES_TYPE by calling the subroutine IEEE_GET_MODES and restored by calling the subroutine IEEE_SET_MODES.
- IEEE_STATUS_TYPE: The floating-point status can be saved in a variable of type IEEE_STATUS_TYPE by calling the subroutine IEEE_GET_STATUS and restored by calling the subroutine IEEE_SET_STATUS.

The IEEE_ARITHMETIC module includes support for IEEE_EXCEPTIONS module, and public entities in IEEE_EXCEPTIONS module are also public in the IEEE_ARITHMETIC module.

To see a summary of all the IEEE_EXCEPTIONS intrinsic procedures, see IEEE Intrinsic Modules Quick Reference Tables

IEEE_FEATURES Intrinsic Module

The IEEE_FEATURES module contains derived data types that include named constants for controlling the level of support, and intrinsic module procedures.

The derived types in the intrinsic modules have components that are private. The IEEE_FEATURES intrinsic module supports specification of essential IEEE features. It defines the following derived type:

- IEEE_FEATURES_TYPE: Specifies IEEE features. Its values are the following named constants:

IEEE_DATATYPE	IEEE_INF
IEEE_DIVIDE	IEEE_NAN
IEEE_ROUNDING	IEEE_INEXACT_FLAG
IEEE_SQRT	IEEE_INVALID_FLAG
IEEE_SUBNORMAL	IEEE_UNDERFLOW_FLAG
IEEE_HALTING	

IEEE Intrinsic Modules Quick Reference Tables

This topic contains quick reference tables showing categories of IEEE intrinsic modules, a summary of the IEEE_ARITHMETIC intrinsic procedures, and a summary of the IEEE_EXCEPTIONS intrinsic procedures.

Categories of Intrinsic Module Functions

Category	Sub-category	Description
IEEE	Arithmetic	Test IEEE values or provide features: IEEE_CLASS, IEEE_COPY_SIGN, IEEE_FMA, IEEE_INT, IEEE_IS_FINITE, IEEE_IS_NAN, IEEE_IS_NORMAL, IEEE_IS_NEGATIVE, IEEE_LOGB, IEEE_MAX_NUM, IEEE_MAX_NUM_MAG, IEEE_MIN_NUM, IEEE_MIN_NUM_MAG, IEEE_NEXT_AFTER, IEEE_QUIET_EQ, IEEE_QUIET_GE, IEEE_QUIET_GT, IEEE_QUIET_LE, IEEE_QUIET_LT, IEEE_QUIET_NE, IEEE_REAL, IEEE_REM, IEEE_RINT, IEEE_SCALB, IEEE_SIGNALING_EQ, IEEE_SIGNALING_GE, IEEE_SIGNALING_GT, IEEE_SIGNALING LE, IEEE_SIGNALING _LT, IEEE_SIGNALING_NE, IEEE_UNORDERED, IEEE_VALUE, IEEE_NEXT_DOWN, IEEE_NEXT_UP, IEEE_SIGNBIT
	Inquiry	Returns whether the processor supports certain exceptions or IEEE features: IEEE_SUPPORT_DATATYPE, IEEE_SUPPORT_DENORMAL, IEEE_SUPPORT_DIVIDE, IEEE_SUPPORT_INF, IEEE_SUPPORT_IO, IEEE_SUPPORT_NAN,

Category	Sub-category	Description
		IEEE_SUPPORT_SQRT, IEEE_SUPPORT_STANDARD, IEEE_SUPPORT_SUBNORMAL, IEEE_SUPPORT_UNDERFLOW_CONTROL
	Transformational	Returns the kind type parameter of an IEEE value, or whether the processor supports certain IEEE features: IEEE_SELECTED_REAL_KIND, IEEE_SUPPORT_FLAG, IEEE_SUPPORT_HALTING, IEEE_SUPPORT_ROUNDING

Summary of IEEE_ARITHMETIC Procedures

Procedure	Class	Value Returned or Result
IEEE_CLASS (X)	E	The IEEE class
IEEE_COPY_SIGN (X, Y)	E	An argument with a copied sign; the IEEE copysign function
IEEE_FMA (A, B, C)	E	Fused multiply-add
IEEE_GET_ROUNDING_MODE (ROUND_VALUE [, RADIX])	SI	The current IEEE rounding mode
IEEE_GET_UNDERFLOW_MODE (GRADUAL)	SI	The current underflow mode
IEEE_INT (A, ROUND [, KIND])	E	Conversion to INTEGER data type
IEEE_IS_FINITE (X)	E	Whether a value is finite
IEEE_IS_NAN (X)	E	Whether a value is NaN
IEEE_IS_NEGATIVE (X)	E	Whether a value is negative
IEEE_IS_NORMAL (X)	E	Whether a value is normal
IEEE_LOGB (X)	E	An exponent in IEEE floating-point format; the IEEE logb function
IEEE_MAX_NUM (X, Y)	E	The maximum numeric value
IEEE_MAX_NUM_MAG (X, Y)	E	The maximum magnitude numeric value
IEEE_MIN_NUM (X, Y)	E	The minimum numeric value
IEEE_MIN_NUM_MAG (X, Y)	E	The minimum magnitude numeric value
IEEE_NEXT_AFTER (X, Y)	E	The next representable value after X toward Y; the IEEE nextafter function
IEEE_NEXT_DOWN (X)	E	The next lower adjacent machine number
IEEE_NEXT_UP (X)	E	The next higher adjacent machine number
IEEE_QUIET_EQ (A, B)	E	Quiet compare for equality

Procedure	Class	Value Returned or Result
IEEE_QUIET_GE (A, B)	E	Quiet compare for greater than or equal
IEEE_QUIET_GT (A, B)	E	Quiet compare for greater than
IEEE_QUIET_LE (A, B)	E	Quiet compare for less than or equal
IEEE_QUIET_LT (A, B)	E	Quiet compare for less than
IEEE_QUIET_NE (A, B)	E	Quiet compare for inequality
IEEE_REAL (A [, KIND])	E	Conversion to REAL data type
IEEE_REM (X, Y)	E	The result of a remainder operation; the IEEE rem function
IEEE_RINT (X [, ROUND])	E	An integer value rounded according to the current or specified rounding mode
IEEE_SCALB (X, I)	E	The value of X multiplied by 2**I; the IEEE scalb function
IEEE_SELECTED_REAL_KIND ([P] [, R])	T	The kind type parameter for an IEEE real
IEEE_SET_ROUNDING_MODE (ROUND_VALUE [, RADIX])	SI	Sets the IEEE rounding mode
IEEE_SET_UNDERFLOW_MODE (GRADUAL)	SI	Sets the current underflow mode
IEEE_SIGNALING_EQ (A, B)	E	Signaling compare for equality
IEEE_SIGNALING_GE (A, B)	E	Signaling compare for greater than or equal
IEEE_SIGNALING_GT (A, B)	E	Signaling compare for greater than
IEEE_SIGNALING_LE (A, B)	E	Signaling compare for less than or equal
IEEE_SIGNALING_LT (A, B)	E	Signaling compare for less than
IEEE_SIGNALING_NE (A, B)	E	Signaling compare for inequality
IEEE_SIGNBIT (X)	E	Tests the sign bit of X
IEEE_SUPPORT_DATATYPE ([X])	I	Whether IEEE arithmetic is supported
IEEE_SUPPORT_DENORMAL ([X])	I	Whether subnormal numbers are supported
IEEE_SUPPORT_DIVIDE ([X])	I	Whether divide accuracy compares to IEEE standard
IEEE_SUPPORT_INF ([X])	I	Whether IEEE infinities are supported

Procedure	Class	Value Returned or Result
IEEE_SUPPORT_IO ([X])	I	Whether IEEE base conversion rounding is supported during formatted I/O
IEEE_SUPPORT_NAN ([X])	I	Whether IEEE Not-A-Number is supported
IEEE_SUPPORT_ROUNDING (ROUND_VALUE [, X])	T	Whether a particular rounding mode is supported
IEEE_SUPPORT_SQRT ([X])	I	Whether IEEE square root is supported
IEEE_SUPPORT_STANDARD ([X])	I	Whether all IEEE capabilities are supported
IEEE_SUPPORT_SUBNORMAL ([X])	I	Whether subnormal numbers are supported
IEEE_SUPPORT_UNDERFLOW_CONTROL(X)	I	Whether control of underflow mode is supported
IEEE_UNORDERED (X, Y)	E	Whether one or both arguments are NaN; the IEEE unordered function
IEEE_VALUE (X, CLASS)	E	An IEEE value
Key to Classes		
E-Elemental function		
I-Inquiry		
SI-Impure Subroutine		
T-Transformational		

Summary of IEEE_EXCEPTIONS Procedures

Procedure	Class	Value Returned or Result
IEEE_GET_FLAG (FLAG, FLAG_VALUE)	ES	Whether an exception flag is signaling
IEEE_GET_HALTING_MODE (FLAG, HALTING)	ES	The current halting mode for an exception
IEEE_GET_MODES (MODES)	SI	The current IEEE floating-point modes
IEEE_GET_STATUS (STATUS_VALUE)	SI	The current state of the floating-point environment
IEEE_SET_FLAG (FLAG, FLAG_VALUE)	SP	Assigns a value to an exception flag
IEEE_SET_HALTING_MODE (FLAG, HALTING)	SP	Controls the halting mode after an exception
IEEE_SET_MODES (MODES)	SI	Restores the current IEEE floating-point modes

Procedure	Class	Value Returned or Result
IEEE_SET_STATUS (STATUS_VALUE)	SI	Restores the state of the floating-point environment
IEEE_SUPPORT_FLAG (FLAG [, X])	T	Whether an exception is supported
IEEE_SUPPORT_HALTING (FLAG)	T	Whether halting after and exception is supported

Key to Classes

ES-Elemental subroutine

SI-Impure Subroutine

SP-Pure Subroutine

T-Transformational

Block Data Program Units Overview

A block data program unit provides initial values for nonpointer variables in named common blocks. For more information, see BLOCK DATA.

Examples

An example of a block data program unit follows:

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10,10)
   DATA A /1.0/, B /2.0/, C /100*0.0/
END BLOCK DATA WORK
```

Functions, Subroutines, and Statement Functions

Functions, subroutines, and statement functions are user-written subprograms that perform computing procedures. The computing procedure can be either a series of arithmetic operations or a series of Fortran statements. A single subprogram can perform a computing procedure in several places in a program, to avoid duplicating a series of operations or statements in each place.

The following table shows the statements that define these subprograms, and how control is transferred to the subprogram:

Subprogram	Defining Statements	Control Transfer Method
Function	FUNCTION or ENTRY	Function reference[1]
Subroutine	SUBROUTINE or ENTRY	CALL statement[2]
Statement function	Statement function definition	Function reference

[1] A function can also be invoked by a defined operation (see Defining Generic Operators).

[2] A subroutine can also be invoked by a defined assignment (see Defining Generic Assignment).

A *function reference* is used in an expression to invoke a function; it consists of the function name and its actual arguments. The function reference returns a value to the calling expression that is used to evaluate the expression.

See Also

ENTRY statement

CALL statement

General Rules for Function and Subroutine Subprograms

A subprogram can be an external, module, or internal subprogram. The END statement for an internal or module subprogram must be END SUBROUTINE [name] for a subroutine, or END FUNCTION [name] for a function. In an external subprogram, the SUBROUTINE and FUNCTION keywords are optional.

If a subprogram name appears after the END statement, it must be the same as the name specified in the SUBROUTINE or FUNCTION statement.

Function and subroutine subprograms can change the values of their arguments, and the calling program can use the changed values.

A SUBROUTINE or FUNCTION statement can be optionally preceded by an OPTIONS statement.

Dummy arguments (except for dummy pointers or dummy procedures) can be specified with an intent and can be made optional.

Subroutines and functions are by default assumed to be non-recursive. This can be changed by declaring the function RECURSIVE, either with the RECURSIVE keyword on the SUBROUTINE or FUNCTION statement, or by specifying an option on the command line or in an OPTIONS statement. The Fortran 2018 Standard specifies that the default mode of compilation is recursion; prior standards specified non-recursion. The default mode of compilation will change to recursion in a future release.

See Also

RECURSIVE procedures
PURE procedures
User-defined ELEMENTAL procedures
Module procedures
Internal procedures
External procedures
Optional arguments
INTENT attribute

Recursive Procedures

A recursive procedure is a function or subroutine that references itself, either directly or indirectly. For more information, see RECURSIVE.

Pure Procedures

A pure procedure is a procedure that has no side effects. For more information, see PURE.

Impure Procedures

An impure procedure is a user-defined procedure that has side effects. For more information, see IMPURE

Elemental Procedures

An elemental procedure is a user-defined procedure defined on scalar arguments that may be called with array arguments. An elemental procedure is pure unless you specify that it is impure. For more information, see PURE, ELEMENTAL, and IMPURE.

Functions Overview

A *function* subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression. For more information, see FUNCTION.

RESULT Keyword Overview

If you use the RESULT keyword in a FUNCTION statement, you can specify a local variable name for the function result. For more information, see RESULT.

Function References

Functions are invoked by a function reference in an expression or by a defined operation.

A function reference takes the following form:

fun ([*a-arg* [, *a-arg*] ...])

fun	Is the name of the function subprogram or a procedure pointer.
a-arg	Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the function. The keyword is assigned a value when the procedure is invoked. Each actual argument must be a variable, an expression, or the name of a procedure. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

Description

When a function is referenced, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

Execution of the function produces a result that is assigned to the function name or to the result name, depending on whether the RESULT keyword was specified.

The program unit uses the result value to complete the evaluation of the expression containing the function reference.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

If a dummy argument is specified with the INTENT attribute, its use may be limited. A dummy argument whose intent is not specified is subject to the limitations of its associated actual argument.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see table Specific Functions Not Allowed as Actual Arguments in Intrinsic Procedures).

Examples

Consider the following example:

```
X = 2.0
NEW_COS = COS(X)        ! A function reference
```

Intrinsic function COS calculates the cosine of 2.0. The value -0.4161468 is returned (in place of COS(X)) and assigned to NEW_COS.

See Also

INTENT attribute
Defining Generic Operators
Dummy Procedure Arguments
Intrinsic Procedures

Optional arguments
RESULT keyword
FUNCTION statement
Argument Association for details on procedure arguments

Subroutines Overview

A *subroutine* subprogram is invoked in a CALL statement or by a defined assignment statement, and does not return a particular value. For more information, see SUBROUTINE.

Statement Functions Overview

A statement function is a procedure defined by a single statement in the same program unit in which the procedure is referenced. For more information, see Statement Function.

Entry Points in Subprograms

The ENTRY statement provides multiple entry points within a subprogram. It is not executable and must precede any CONTAINS statement (if any) within the subprogram. For more information on the ENTRY statement, see ENTRY.

Entry Points in Function Subprograms

If the ENTRY statement is contained in a function subprogram, it defines an additional function. The name of the function is the name specified in the ENTRY statement, and its result variable is the entry name or the name specified by RESULT (if any).

If the entry result variable has the same characteristics as the FUNCTION statement's result variable, their result variables identify the same variable, even if they have different names. Otherwise, the result variables are storage associated and must all be nonpointer scalars of intrinsic type, in one of the following groups:

Group 1	Type default integer, default real, double precision real, default complex, double complex, or default logical
Group 2	Type REAL(16) and COMPLEX(16)
Group 3	Type default character (with identical lengths)

All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names with the same data type. All associated names with different data types become undefined.

If RECURSIVE is specified in the FUNCTION statement, all entry points in the FUNCTION are recursive. The interface of the function defined by the ENTRY statement is explicit within the function subprogram.

Examples

The following example shows a function subprogram that computes the hyperbolic functions SINH, COSH, and TANH:

```
REAL FUNCTION TANH(X)
  TSINH(Y) = EXP(Y) - EXP(-Y)
  TCOSH(Y) = EXP(Y) + EXP(-Y)

  TANH = TSINH(X)/TCOSH(X)
  RETURN

  ENTRY SINH(X)
```

```
  SINH = TSINH(X)/2.0
  RETURN

  ENTRY COSH(X)
  COSH = TCOSH(X)/2.0
  RETURN
END
```

See Also

RESULT keyword

Entry Points in Subroutine Subprograms

If the ENTRY statement is contained in a subroutine subprogram, it defines an additional subroutine. The name of the subroutine is the name specified in the ENTRY statement.

If RECURSIVE is specified on the SUBROUTINE statement, all entry points in the subroutine are RECURSIVE. The interface of the subroutine defined by the ENTRY statement is explicit within the subroutine subprogram.

Examples

The following example shows a main program calling a subroutine containing an ENTRY statement:

```
PROGRAM TEST
  ...
  CALL SUBA(A, B, C)       ! A, B, and C are actual arguments
  ...                      !    passed to entry point SUBA
END
SUBROUTINE SUB(X, Y, Z)
  ...
  ENTRY SUBA(Q, R, S)      ! Q, R, and S are dummy arguments
  ...                      ! Execution starts with this statement
END SUBROUTINE
```

The following example shows an ENTRY statement specifying alternate returns:

```
CALL SUBC(M, N, *100, *200, P)
...
SUBROUTINE SUB(K, *, *)
  ...
  ENTRY SUBC(J, K, *, *, X)
  ...
  RETURN 1
  RETURN 2
END
```

Note that the CALL statement for entry point SUBC includes actual alternate return arguments. The RETURN 1 statement transfers control to statement label 100 and the RETURN 2 statement transfers control to statement label 200 in the calling program.

External Procedures

External procedures are user-written functions or subroutines. They are located outside of the main program and can't be part of any other program unit.

External procedures can be invoked by the main program or any procedure of an executable program.

External procedures can include internal subprograms (defining internal procedures). Internal subprograms are placed after a CONTAINS statement.

An external procedure can reference itself (directly or indirectly).

The interface of an external procedure is implicit unless an interface block is supplied for the procedure.

See Also

Functions, Subroutines, and Statement Functions
Procedure Interfaces

Internal Procedures

Internal procedures are functions or subroutines that follow a CONTAINS statement in a program unit. The program unit in which the internal procedure appears is called its *host*.

Internal procedures can appear in the main program, in an external subprogram, or in a module subprogram.

An internal procedure takes the following form:

CONTAINS

internal-subprogram

[*internal-subprogram*] ...

internal-subprogram	Is a function or subroutine subprogram that defines the procedure. An internal subprogram must not contain any other internal subprograms.

Description

Internal procedures are the same as external procedures, except for the following:

- Only the host program unit can use an internal procedure.
- An internal procedure has access to host entities by host association; that is, names declared in the host program unit are useable within the internal procedure.
- An internal procedure must not contain an ENTRY statement.

An internal procedure can reference itself (directly or indirectly); it can be referenced in the execution part of its host and in the execution part of any internal procedure contained in the same host (including itself).

The interface of an internal procedure is always explicit.

Examples

The following example shows an internal procedure:

```
PROGRAM COLOR_GUIDE
...
 CONTAINS
  FUNCTION HUE(BLUE)   ! An internal procedure
  ...
  END FUNCTION HUE
END PROGRAM
```

The following example program contains an internal subroutine `find`, which performs calculations that the main program then prints. The variables `a`, `b`, and `c` declared in the host program are also known to the internal subroutine.

```
program INTERNAL
! shows use of internal subroutine and CONTAINS statement
   real a,b,c
   call find
   print *, c
contains
   subroutine find
    read *, a,b
```

```
    c = sqrt(a**2 + b**2)
   end subroutine find
end
```

See Also

Functions, Subroutines, and Statement Functions
Host association
Procedure Interfaces
CONTAINS statement

Argument Association in Procedures

Procedure arguments provide a way for different program units to access the same data.

When a procedure is referenced in an executable program, the program unit invoking the procedure can use one or more *actual* arguments to pass values to the procedure's *dummy* arguments. The dummy arguments are associated with their corresponding actual arguments when control passes to the subprogram.

In general, when control is returned to the calling program unit, the last value assigned to a dummy argument is assigned to the corresponding actual argument.

An actual argument can be a variable, expression, or procedure name. The type and kind parameters, and rank of the actual argument must match those of its associated dummy argument.

A dummy argument is either a dummy data object, a dummy procedure, or an alternate return specifier (*). Except for alternate return specifiers, dummy arguments can be optional.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments.

A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

A scalar dummy argument can be associated with only a scalar actual argument.

If a dummy argument is an array, it must be no larger than the array that is the actual argument. You can use adjustable arrays to process arrays of different sizes in a single subprogram.

An actual argument associated with a dummy argument that is allocatable or a pointer must have the same type parameters as the dummy argument.

A dummy argument referenced as a subprogram must be associated with an actual argument that has been declared EXTERNAL or INTRINSIC in the calling routine.

If a scalar dummy argument is of type character, its length must not be greater than the length of its associated actual argument.

If the character dummy argument's length is specified as *(*) (assumed length), it uses the length of the associated actual argument.

Once an actual argument has been associated with a dummy argument, no action can be taken that affects the value or availability of the actual argument, except indirectly through the dummy argument. For example, if the following statement is specified:

```
CALL SUB_A (B(2:6), B(4:10))
```

B(4:6) must not be defined, redefined, or become undefined through either dummy argument, since it is associated with both arguments. However, B(2:3) is definable through the first argument, and B(7:10) is definable through the second argument.

Similarly, if any part of the actual argument is defined through a dummy argument, the actual argument can only be referenced through that dummy argument during execution of the procedure. For example, if the following statements are specified:

```
MODULE MOD_A
  REAL :: A, B, C, D
END MODULE MOD_A

PROGRAM TEST
  USE MOD_A
  CALL SUB_1 (B)
  ...
END PROGRAM TEST

SUBROUTINE SUB_1 (F)
  USE MOD_A
  ...
  WRITE (*,*) F
END SUBROUTINE SUB_1
```

Variable B must not be directly referenced during the execution of SUB_1 because it is being defined through dummy argument F. However, B can be indirectly referenced through F (and directly referenced when SUB_1 completes execution).

The ultimate argument is the effective argument if the effective argument is not a dummy argument or a subobject of a dummy argument. If the effective argument is a dummy argument, the ultimate argument is the ultimate argument of that dummy argument. If the effective argument is a subobject of a dummy argument, the ultimate argument is the corresponding subobject of the ultimate argument of that dummy argument.

Consider the following sequence of subroutine calls:

```
INTEGER :: X(100)
CALL SUBA (X)
...
SUBROUTINE SUBA(A)
INTEGER :: A(:)
CALL SUBB (A(1:5), A(5:1:-1))
...
SUBROUTINE SUBB(B, C)
INTEGER :: B(:), C(:)
```

The ultimate argument of B is X(1:5). The ultimate argument of C is X(5:1:-1), which is not the same object as the ultimate argument of B.

The following sections provide more details on arguments:

- Optional arguments
- The different kinds of arguments:
  - Array arguments
  - Pointer arguments
  - Passed-Object Dummy Arguments
  - Assumed-length character arguments
  - Character constant and Hollerith arguments
  - Alternate return arguments
  - Dummy procedure arguments
  - Coarray Dummy Arguments
- References to generic procedures
- References to non-Fortran procedures (%REF, %VAL, and %LOC)

See Also

CALL for details on argument keywords in subroutine references

Function References for details on argument keywords in function references

Optional Arguments

Dummy arguments can be made optional if they are declared with the OPTIONAL attribute. In this case, an actual argument does not have to be supplied for it in a procedure reference.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments. A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

Positional arguments (if any) must appear first in an actual argument list, followed by keyword arguments (if any). If an optional argument is the last positional argument, it can simply be omitted if desired.

However, if the optional argument is to be omitted but it is not the last positional argument, keyword arguments must be used for any subsequent arguments in the list.

Optional arguments must have explicit procedure interfaces so that appropriate argument associations can be made.

The PRESENT intrinsic function can be used to determine if an actual argument is associated with an optional dummy argument in a particular reference.

A dummy argument or an entity that is host associated with a dummy argument is not present if any of the following are true for the dummy argument:

- It does not correspond to an actual argument.
- It corresponds to an actual argument that is not present.
- It does not have the ALLOCATABLE or POINTER attribute, and corresponds to one of the following:
  - An actual argument that has the ALLOCATABLE attribute and is not allocated
  - An actual argument that has the POINTER attribute and is disassociated

The following example shows optional arguments:

```
PROGRAM RESULT
TEST_RESULT = LGFUNC(A, B=D)
...
CONTAINS
  FUNCTION LGFUNC(G, H, B)
    OPTIONAL H, B
    ...
  END FUNCTION
END
```

In the function reference, A is a positional argument associated with required dummy argument G. The second actual argument D is associated with optional dummy argument B by its keyword name (B). No actual argument is associated with optional argument H.

The following shows another example:

```
 ! Arguments can be passed out of order, but must be
 !  associated with the correct dummy argument.
 CALL EXT1 (Z=C, X=A, Y=B)
 . . .
 END

 SUBROUTINE EXT1(X,Y,Z)
    REAL X, Y
    REAL, OPTIONAL :: Z
    . . .
 END SUBROUTINE
```

In this case, argument A is associated with dummy argument X by explicit assignment. Once EXT1 executes and returns, A is no longer associated with X, B is no longer associated with Y, and C is no longer associated with Z.

If you pass an omitted dummy argument as the actual argument to a procedure, the corresponding dummy argument is considered to be omitted as well. This applies to both intrinsic and non-intrinsic procedures. For example:

```
CALL SUB1()
CONTAINS
  SUBROUTINE SUB1(B)
  LOGICAL, OPTIONAL :: B
  PRINT *, INDEX('Fortran','r',BACK=B) ! Prints 3
  CALL SUB2(B) ! Same as CALL SUB2()
  END SUBROUTINE SUB1

  SUBROUTINE SUB2(C)
  LOGICAL, OPTIONAL :: C
  PRINT *, PRESENT(C) ! Prints F
  END SUBROUTINE SUB2
END
```

See Also

OPTIONAL attribute
PRESENT intrinsic function
Argument association for details on general rules for procedure argument association
CALL for details on argument keywords in subroutine references
Function References for details on argument keywords in function references

Array Arguments

Arrays are sequences of elements. Each element of an actual array is associated with the element of the dummy array that has the same position in array element order.

If the dummy argument is an explicit-shape or assumed-size array, the size of the dummy argument array must not exceed the size of the actual argument array.

The type and kind parameters of an explicit-shape or assumed-size dummy argument must match the type and kind parameters of the actual argument, but their ranks need not match.

If the dummy argument is an assumed-shape array, the size of the dummy argument array is equal to the size of the actual argument array. The associated actual argument must not be an assumed-size array or a scalar (including a designator for an array element or an array element substring).

If the actual argument is an array section with a vector subscript, the associated dummy argument must not be defined and it must not have the INTENT (OUT), INTENT (INOUT), VOLATILE, or ASYNCHRONOUS attribute.

If an actual argument is an array section or an assumed-shape array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument must be an assumed-shape array.

If an actual argument is a pointer array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument must be an assumed-shape array or a pointer array.

The declaration of an array used as a dummy argument can specify the lower bound of the array.

If a dummy argument is allocatable, the actual argument must be allocatable and the type parameters and ranks must agree. An example of an allocatable function with allocatable arrays appears in FUNCTION.

Dummy argument arrays declared as assumed-shape, deferred-shape, or pointer arrays require an explicit interface visible to the caller.

See Also

Arrays
Array association
Procedure Interfaces
Argument association for details on general rules for procedure argument association
Array Elements for details on array element order
Explicit-Shape Specifications for details on explicit-shape arrays
Assumed-Shape Specifications for details on assumed-shape arrays
Assumed-Size Specifications for details on assumed-size arrays

Pointer Arguments

An argument is a pointer if it is declared with the POINTER attribute.

When a procedure is invoked, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target.

The pointer association status of the dummy argument can change during the execution of the procedure, and any such changes are reflected in the actual argument.

If both the dummy and actual arguments are pointers, an explicit interface is required.

A dummy argument that is a pointer can be associated only with an actual argument that is a pointer. However, an actual argument that is a pointer can be associated with a nonpointer dummy argument. In this case, the actual argument is associated with a target and the dummy argument, through argument association, also becomes associated with that target.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument when the procedure is invoked.

If the dummy argument has the TARGET attribute, and is either a scalar or assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, the following occurs:

- Any pointer associated with the actual argument becomes associated with the corresponding dummy argument when the procedure is invoked.
- Any pointers associated with the dummy argument remain associated with the actual argument when execution of the procedure completes.

If the dummy argument has the TARGET attribute, and is an explicit-shape or assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, association of actual and corresponding dummy arguments when the procedure is invoked or when execution is completed is processor dependent.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have that attribute or is an array section with a vector subscript, any pointer associated with the dummy argument becomes undefined when execution of the procedure completes.

See Also

POINTER statement and attribute
Pointer assignments
TARGET statement and attribute
Argument association for details on general rules for procedure argument association

Passed-Object Dummy Arguments

A procedure component or a binding procedure (type-bound procedure) can be declared to have a passed-object dummy argument. This kind of argument is associated with a special actual argument, which is not explicitly written in the actual argument list. The appropriate actual argument is then added to the argument list.

A passed-object dummy argument must be a scalar. It must not be a pointer, must not be allocatable, and all its length type parameters must be assumed. Its declared type must be the type in which the component or binding procedure appears.

The passed-object dummy argument must be a scalar, nonpointer, nonallocatable dummy data object. Its declared type must be the type in which the component or binding appears. All of its length type paramaters must be assumed.

The determination of the passed-object dummy argument depends on the following:

- The PASS and NOPASS attributes specified or in effect
- The interface of the procedure component or binding procedure

The following rules apply to PASS and NOPASS:

- PASS and NOPASS are mutually exclusive. You can only specify one of these attributes for the same procedure component or binding.
- If you specify PASS (*arg-name*), dummy argument *arg-name* is the passed-object dummy argument. The interface of the procedure pointer component or binding procedure must have a dummy argument named *arg-name*.
- If NOPASS is specified, there is no passed-object dummy argument.
- NOPASS must be specified if the procedure component or binding procedure has an implicit interface.
- If you do not specify PASS or NOPASS, or you specify PASS without *arg-name*, the first dummy argument of a procedure pointer component or binding procedure is the passed-object dummy argument. In this case, there must be at least one dummy argument.

See Also

TYPE

Passed-Object Dummy Arguments

Assumed-Length Character Arguments

An assumed-length character argument is a dummy argument that assumes the length attribute of its corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.

A character array dummy argument can also have an assumed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The assumed length and the array declarator together determine the size of the assumed-length character array.

The following example shows an assumed-length character argument:

```
INTEGER FUNCTION ICMAX(CVAR)
  CHARACTER*(*) CVAR
  ICMAX = 1
  DO I=2,LEN(CVAR)
    IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX=I
  END DO
  RETURN
END
```

The function ICMAX finds the position of the character with the highest ASCII code value. It uses the length of the assumed-length character argument to control the iteration. Intrinsic function LEN determines the length of the argument.

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression. Each of the following function references specifies a different length for the dummy argument:

```
CHARACTER VAR*10, CARRAY(3,5)*20
...
I1 = ICMAX(VAR)
I2 = ICMAX(CARRAY(2,2))
I3 = ICMAX(VAR(3:8))
I4 = ICMAX(CARRAY(1,3)(5:15))
I5 = ICMAX(VAR(3:4)//CARRAY(3,5))
```

See Also

LEN intrinsic function

Argument association for details on general rules for procedure argument association

Character Constant and Hollerith Arguments

If an actual argument is a character constant (for example, 'ABCD'), the corresponding dummy argument must be of type character. If an actual argument is a Hollerith constant (for example, 4HABCD), the corresponding dummy argument must have a numeric data type.

The following example shows character and Hollerith constants being used as actual arguments:

```
SUBROUTINE S(CHARSUB, HOLLSUB, A, B)
EXTERNAL CHARSUB, HOLLSUB
...
CALL CHARSUB(A, 'STRING')
CALL HOLLSUB(B, 6HSTRING)
```

The subroutines CHARSUB and HOLLSUB are themselves dummy arguments of the subroutine S. Therefore, the actual argument 'STRING' in the call to CHARSUB must correspond to a character dummy argument, and the actual argument 6HSTRING in the call to HOLLSUB must correspond to a numeric dummy argument.

See Also

Argument association for details on general rules for procedure argument association

Alternate Return Arguments

Alternate return (dummy) arguments can appear in a subroutine argument list. They cause execution to transfer to a labeled statement rather than to the statement immediately following the statement that called the routine. The alternate return is indicated by an asterisk (*). (An alternate return is an obsolescent feature in Standard Fortran.)

There can be any number of alternate returns in a subroutine argument list, and they can be in any position in the list.

An actual argument associated with an alternate return dummy argument is called an alternate return specifier; it is indicated by an asterisk (*) or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement.

Alternate returns cannot be declared optional.

You can also use the RETURN statement to specify alternate returns.

The following example shows alternate return actual and dummy arguments:

```
CALL MINN(X, Y, *300, *250, Z)
...
SUBROUTINE MINN(A, B, *, *, C)
```

See Also

Argument association for details on general rules for procedure argument association
SUBROUTINE statement
CALL statement
RETURN statement
Deleted and Obsolescent Language Features for details on obsolescent features in Standard Fortran

Dummy Procedure Arguments

If an actual argument is a procedure, its corresponding dummy argument is a dummy procedure. Dummy procedures can appear in function or subroutine subprograms.

The actual argument must be the specific name of an external, module, intrinsic, or another dummy procedure. If the specific name is also a generic name, only the specific name is associated with the dummy argument. Not all specific intrinsic procedures can appear as actual arguments. (For more information, see table Specific Functions Not Allowed as Actual Arguments in Intrinsic Procedures.)

The actual argument and corresponding dummy procedure must both be subroutines or both be functions.

If the interface of the dummy procedure is explicit, the type and kind parameters, and rank of the associated actual procedure must be the same as that of the dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a subroutine, the actual argument must be a subroutine or a dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a function or is explicitly typed, the actual argument must be a function or a dummy procedure.

Dummy procedures can be declared optional, but they must not be declared with an intent.

The following is an example of a procedure used as an argument:

```
REAL FUNCTION LGFUNC(BAR)
  INTERFACE
    REAL FUNCTION BAR(Y)
      REAL, INTENT(IN) :: Y
    END
  END INTERFACE
  ...
  LGFUNC = BAR(2.0)
  ...
END FUNCTION LGFUNC
```

See Also

Argument association for details on general rules for procedure argument association

Coarray Dummy Arguments

If a dummy argument is a coarray, the corresponding actual argument must be a coarray and must have the VOLATILE attribute if and only if the dummy argument has the VOLATILE attribute.

If a dummy argument is an array coarray that has the CONTIGUOUS attribute or is not of assumed shape, the corresponding actual argument must be simply contiguous.

Examples

When a procedure is invoked on a particular image, each dummy coarray is associated with its ultimate argument on the image. During the execution of the procedure, this image can access the coarray corresponding to the ultimate argument on any other image. For example, consider the following:

```
INTERFACE
   SUBROUTINE MY_SUB(Y)
      REAL :: Y[*]
   END SUBROUTINE MY_SUB
END INTERFACE
...
REAL :: B(700)[:]
...
CALL MY_SUB(B(10))
```

When subroutine MY_SUB is invoked, the executing image has access through the syntax Y[P] to B(10) on image P.

Each invocation of a procedure with a nonallocatable coarray dummy argument establishes a dummy coarray for the image with its own bounds and cobounds. During the execution of the procedure, this image may use its own bounds and cobounds to access the coarray corresponding to the ultimate argument on any other image. For example, consider the following:

```
INTERFACE
   SUBROUTINE MY_SUB(Y,I)
      INTEGER :: I
      REAL :: Y(I,I)[I,*]
   END SUBROUTINE MY_SUB
END INTERFACE
...
REAL :: B(1000)[:]
...
CALL MY_SUB(B,10)
```

When subroutine MY_SUB is invoked, the executing image has access through the syntax Y(1,2)[3,4] to B(11) on the image with image index 33.

See Also

Image Control Statements

References to Generic Procedures

Generic procedures are procedures with different specific names that can be accessed under one generic (common) name. In FORTRAN 77, generic procedures were limited to intrinsic procedures. In the current Fortran standard, you can use generic interface blocks to specify generic properties for intrinsic and user-defined procedures.

If you refer to a procedure by using its generic name, the selection of the specific routine is based on the number of arguments and the type and kind parameters, and rank of each argument.

All procedures given the same generic name must be subroutines, or all must be functions. Any two must differ enough so that any invocation of the procedure is unambiguous.

The following sections describe references to generic intrinsic functions and show an example of using intrinsic function names.

See Also

Unambiguous Generic Procedure References

Intrinsic procedures

Resolving Procedure References

Defining Generic Names for Procedures for details on user-defined generic procedures

References to Generic Intrinsic Functions

The generic intrinsic function name COS lists six specific intrinsic functions that calculate cosines: COS, DCOS, QCOS, CCOS, CDCOS, and CQCOS. These functions return different values: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), and COMPLEX(16) respectively.

If you invoke the cosine function by using the generic name COS, the compiler selects the appropriate routine based on the arguments that you specify. For example, if the argument is REAL(4), COS is selected; if it is REAL(8), DCOS is selected; and if it is COMPLEX(4), CCOS is selected.

You can also explicitly refer to a particular routine. For example, you can invoke the double-precision cosine function by specifying DCOS.

Procedure selection occurs independently for each generic reference, so you can use a generic reference repeatedly in the same program unit to access different intrinsic procedures.

You cannot use generic function names to select intrinsic procedures if you use them as follows:

- The name of a statement function
- A dummy argument name, a common block name, or a variable or array name

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Not all specific intrinsic functions can appear as actual arguments. (For more information, see table Specific Functions Not Allowed as Actual Arguments in Intrinsic Procedures.)

A reference to a generic intrinsic procedure name in a program unit does not prevent use of the name for other purposes elsewhere in the program.

Normally, an intrinsic procedure name refers to the Fortran library procedure with that name. However, the name can refer to a user-defined procedure when the name appears in an EXTERNAL statement.

NOTE
If you call an intrinsic procedure by using the wrong number of arguments or an incorrect argument type, the compiler assumes you are referring to an external procedure. For example, intrinsic procedure SIN requires one argument; if you specify two arguments, such as SIN(10,4), the compiler assumes SIN is external and not intrinsic.

The data type of an intrinsic procedure does not change if you use an IMPLICIT statement to change the implied data type rules.

Intrinsic and user-defined procedures cannot have the same name if they appear in the same program unit.

Examples

The following example shows the local and global properties of an intrinsic function name. It uses the name SIN in different procedures as follows:

- The name of a statement function
- The generic name of an intrinsic function
- The specific name of an intrinsic function
- The name of a user-defined function

Using and Redefining an Intrinsic Function Name

```
!    Compare ways of computing sine
     PROGRAM SINES
       DOUBLE PRECISION X, PI
       PARAMETER (PI=3.14159265358979323 8D0)
       COMMON V(3)
```

```
!    Define SIN as a statement function  1
       SIN(X) = COS(PI/2-X)
       print *
       print *, "                    Way of computing SIN(X)"
       print *
       print *, "      X       Statement   Intrinsic   Intrinsic     User's "
       print *, "              function      DSIN       SIN as arg     SIN  "
       print *
       DO X = -PI, PI, PI/2
                  CALL COMPUT(X)
!    References the statement function SIN  2
                  WRITE (6,100) X, SIN(X), V
        END DO
100     FORMAT (5F12.7)
      END

      SUBROUTINE COMPUT(Y)
        DOUBLE PRECISION Y

!    Use intrinsic function SIN - double-precision DSIN will be passed
!       as an actual argument  3
        INTRINSIC SIN
        COMMON V(3)

!    Makes the generic name SIN reference the double-precision sine DSIN  4
        V(1) = SIN(Y)

!    Use intrinsic function SIN as an actual argument - will pass DSIN 5
         CALL SUB(REAL(Y),SIN)
      END

      SUBROUTINE SUB(A,S)

!    Declare SIN as name of a user function  6
         EXTERNAL SIN
!    Declare SIN as type DOUBLE PRECISION  7
         DOUBLE PRECISION SIN
         COMMON V(3)

!    Evaluate intrinsic function SIN passed as the dummy argument 8
         V(2) = S(A)

!    Evaluate user-defined SIN function  9
         V(3) = SIN(A)
      END

!    Define the user SIN function  10
      DOUBLE PRECISION FUNCTION SIN(X)
         INTEGER FACTOR
         SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)     &
             - X**7/FACTOR(7)
      END
```

```
!    Compute the factorial of N
       INTEGER FUNCTION FACTOR(N)
          FACTOR = 1
          DO I=N,1,-1
             FACTOR = FACTOR * I
          END DO
       END
```

1 The statement function named SIN is defined in terms of the generic function name COS. Because the argument of COS is double precision, the double-precision cosine function is evaluated. The statement function SIN is itself single precision.

2 The statement function SIN is called.

3 The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at 5.

4 The generic function name SIN is used to refer to the double-precision sine function.

5 The single-precision intrinsic sine function is used as an actual argument.

6 The name SIN is declared a user-defined function name.

7 The type of SIN is declared double precision.

8 The single-precision sine function passed at 5 is evaluated.

9 The user-defined SIN function is evaluated.

10 The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

See Also

EXTERNAL attribute
INTRINSIC attribute
Intrinsic procedures
Names for details on the scope of names

References to Elemental Intrinsic Procedures

An *elemental intrinsic procedure* has scalar dummy arguments that can be called with scalar or array actual arguments. If actual arguments are array-valued, they must have the same shape. There are many elemental intrinsic functions, but only one elemental intrinsic subroutine (MVBITS).

If the actual arguments are scalar, the result is scalar. If the actual arguments are array-valued, the scalar-valued procedure is applied element-by-element to the actual argument, resulting in an array that has the same shape as the actual argument.

The values of the elements of the resulting array are the same as if the scalar-valued procedure had been applied separately to the corresponding elements of each argument.

For example, if A and B are arrays of shape (5,6), MAX(A, 0.0, B) is an array expression of shape (5,6) whose elements have the value MAX(A (i, j), 0.0, B (i, j)), where i = 1, 2,..., 5, and j = 1, 2,..., 6.

A reference to an elemental intrinsic procedure is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.

Examples

Consider the following:

```
REAL, DIMENSION (2) :: a, b
a(1) = 4; a(2) = 9
b = SQRT(a)                ! sets b(1) = SQRT(a(1)), and b(2) = SQRT(a(2))
```

See Also

Arrays

Intrinsic procedures for details on elemental procedures

References to Non-Fortran Procedures

When a procedure is called, Fortran (by default) passes the address of the actual argument, and its length if it is of type character. To call non-Fortran procedures, you may need to pass the actual arguments in a form different from that used by Fortran.

The built-in functions %REF and %VAL let you change the form of an actual argument. You must specify these functions in the actual argument list of a CALL statement or function reference. You cannot use them in any other context.

%LOC computes the internal address of a storage item.

Procedure Interfaces

Every procedure has an interface, which consists of the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units.

If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration). The following table shows which procedures have implicit or explicit interfaces:

Kind of Procedure	Interface
External procedure	Implicit [1]
Module procedure	Explicit
Internal procedure	Explicit
Intrinsic procedure	Explicit
Dummy procedure	Implicit [1]
Statement function	Implicit

[1] This kind of procedure is explicit in a scoping unit other than its own if an interface body for the procedure is supplied or is accessible.

The interface of a recursive subroutine or function is explicit within the subprogram that defines it.

An explicit interface can come from any of the following:

- An interface block
- The procedure's definition in a module
- An internal procedure

A procedure must not access through use association its own interface.

An abstract interface lets you give a name to a set of characteristics and argument keyword names that create an explicit interface to a procedure. It does not declare any actual procedure to have those characteristics.

Depending on the characteristics of the procedure and its dummy arguments, an explicit interface may be required to be visible to its caller. For more information see Procedures that Require Explicit Interfaces.

You can use a procedure declaration statement to declare procedure pointers, dummy procedures, and external procedures. It specifies the EXTERNAL attribute for all procedure entities in the procedure declaration list.

You can use the IMPORT statement to make host entities accessible in the interface body of an interface block.

You can specify the ALLOCATABLE, OPTIONAL, or POINTER attributes for a dummy argument in a procedure interface that has the BIND attribute.

Examples

An example of an interface block follows:

```
INTERFACE
  SUBROUTINE Ext1 (x, y, z)
  REAL, DIMENSION (100,100) :: x, y, z
  END SUBROUTINE Ext1

  SUBROUTINE Ext2 (x, z)
  REAL x
  COMPLEX (KIND = 4) z (2000)
  END SUBROUTINE Ext2

  FUNCTION Ext3 (p, q)
  LOGICAL Ext3
  INTEGER p (1000)
  LOGICAL q (1000)
  END FUNCTION Ext3
END INTERFACE
```

See Also

INTERFACE
ABSTRACT INTERFACE
PROCEDURE
IMPORT

Procedures that Require Explicit Interfaces

When a procedure is referenced, it must have an explicit interface in the following cases:

- If a reference to the procedure appears in one of the following:
  - An actual argument that is specified with a keyword
  - In a context that requires it to be PURE
- If the procedure has a dummy argument that is one of the following:
  - An object that has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET, VALUE, or VOLATILE attribute
  - An assumed-shape array
  - A polymorphic object (an object declared with a CLASS statement)
  - A coarray (an object declared with a CODIMENSION attribute or statement)
  - An object of a parameterized derived type
  - An object of assumed-rank or assumed-type
- If the procedure has any of the following:

  - A result that is an array, or a pointer, or is allocatable (functions only)
  - A result whose length is neither assumed nor a constant (character functions only)
- If a reference to the procedure appears as follows:
  - With an argument keyword
  - As a reference by its generic name
  - As a defined assignment (subroutines only)
  - In an expression as a defined operator (functions only)
  - In a context that requires it to be pure
- If the procedure is elemental
- If the procedure has the BIND attribute

Statement functions do not require an explicit interface.

See Also

Optional arguments
Array arguments
Pointer arguments
CALL for details on argument keywords in subroutine references
Function references for details on argument keywords in function references
Pure procedures
Elemental procedures
Procedure Interfaces
Defining Generic Names for Procedures for details on user-defined generic procedures
Defining Generic Operators for details on defined operators
Defining Generic Assignment for details on defined assignment
Parameterized Derived-Type Declarations

Explicit and Abstract Interfaces

An explicit interface defines characteristics for external or dummy procedures. It can also be used to define a generic name for procedures, a new operator for functions, and a new form of assignment for subroutines. For more information, see INTERFACE.

An abstract interface defines a subprogram whose name can be used in a PROCEDURE declaration statement to declare subprograms with identical arguments and characteristics. For more information, see ABSTRACT INTERFACE and PROCEDURE.

Defining Generic Names for Procedures

An interface block or a GENERIC statement can be used to specify a generic name to reference all of the procedures within the interface block.

The initial line for such an interface block takes the following form:

INTERFACE *generic-name*

generic-name	Is the generic name. It can be the same as any of the procedure names in the interface block, or the same as any accessible generic name (including a generic intrinsic name). A generic name can be the same as a derived-type name. In this case, all of the procedures in the interface block must be functions.

A generic interface can be used to extend or redefine a generic intrinsic procedure.

The procedures that are given the generic name must be the same kind of subprogram: all must be functions, or all must be subroutines.

Any procedure reference involving a generic procedure name must be resolvable to one specific procedure; it must be unambiguous. For more information, see Unambiguous Generic Procedure References.

The following is an example of a procedure interface block defining a generic name:

```
INTERFACE GROUP_SUBS
  SUBROUTINE INTEGER_SUB (A, B)
    INTEGER, INTENT(INOUT) :: A, B
  END SUBROUTINE INTEGER_SUB

  SUBROUTINE REAL_SUB (A, B)
    REAL, INTENT(INOUT) :: A, B
  END SUBROUTINE REAL_SUB

  SUBROUTINE COMPLEX_SUB (A, B)
    COMPLEX, INTENT(INOUT) :: A, B
  END SUBROUTINE COMPLEX_SUB
END INTERFACE
```

The three subroutines can be referenced by their individual specific names or by the group name GROUP_SUBS.

The following example shows a reference to INTEGER_SUB:

```
INTEGER V1, V2
CALL GROUP_SUBS (V1, V2)
```

Consider the following:

```
 INTERFACE LINE_EQUATION

   SUBROUTINE REAL_LINE_EQ(X1,Y1,X2,Y2,M,B)
     REAL,INTENT(IN) :: X1,Y1,X2,Y2
     REAL,INTENT(OUT) :: M,B
   END SUBROUTINE REAL_LINE_EQ

   SUBROUTINE INT_LINE_EQ(X1,Y1,X2,Y2,M,B)
     INTEGER,INTENT(IN) :: X1,Y1,X2,Y2
     INTEGER,INTENT(OUT) :: M,B
   END SUBROUTINE INT_LINE_EQ

 END INTERFACE
```

In this example, LINE_EQUATION is the generic name which can be used for either REAL_LINE_EQ or INT_LINE_EQ. Fortran selects the appropriate subroutine according to the nature of the arguments passed to LINE_EQUATION. Even when a generic name exists, you can always invoke a procedure by its specific name. In the previous example, you can call REAL_LINE_EQ by its specific name (REAL_LINE_EQ), or its generic name LINE_EQUATION.

See Also

INTERFACE statement for details on interface blocks

GENERIC statement for an alternate to interface blocks for declaring generic procedures

Defining Generic Operators

An interface block can be used to define a generic operator. The only procedures allowed in the interface block are functions that can be referenced as defined operations.

The initial line for such an interface block takes the following form:

INTERFACE OPERATOR (*op*)

op	Is one of the following:

- A defined unary operator (one argument)
- A defined binary operator (two arguments)
- An extended intrinsic operator (number of arguments must be consistent with the intrinsic uses of that operator)

The functions within the interface block must have one or two nonoptional arguments with intent IN, and the function result must not be of type character with assumed length. A defined operation is treated as a reference to the function.

The following shows the form (and an example) of a defined unary and defined binary operation:

Operation	Form	Example
Defined Unary	.defined-operator. operand[1]	.MINUS. C
Defined Binary	operand[2] .defined-operator. operand[3]	B .MINUS. C

[1] The operand corresponds to the function's dummy argument.

[2] The left operand corresponds to the first dummy argument of the function.

[3] The right operand corresponds to the second argument.

For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Both forms of each relational operator have the same interpretation, so extending one form (such as >=) defines both forms (>= and .GE.).

The following is an example of a procedure interface block defining a new operator:

```
INTERFACE OPERATOR(.BAR.)
  FUNCTION BAR(A_1)
    INTEGER, INTENT(IN) :: A_1
    INTEGER :: BAR
  END FUNCTION BAR
END INTERFACE
```

The following example shows a way to reference function BAR by using the new operator:

```
INTEGER B
I = 4 + (.BAR. B)
```

The following is an example of a procedure interface block with a defined operator extending an existing operator:

```
INTERFACE OPERATOR(+)
  FUNCTION LGFUNC (A, B)
    LOGICAL, INTENT(IN) :: A(:), B(SIZE(A))
    LOGICAL :: LGFUNC(SIZE(A))
  END FUNCTION LGFUNC
END INTERFACE
```

The following example shows two equivalent ways to reference function LGFUNC:

```
LOGICAL, DIMENSION(1:10) :: C, D, E
N = 10
E = LGFUNC(C(1:N), D(1:N))
E = C(1:N) + D(1:N)
```

See Also

INTENT attribute

INTERFACE for details on interface blocks

Expressions for details on intrinsic operators

Defined Operations for details on defined operators and operations

Defining Generic Assignment

An interface block can be used to define generic assignment. The only procedures allowed in the interface block are subroutines that can be referenced as defined assignments.

The initial line for such an interface block takes the following form:

INTERFACE ASSIGNMENT (=)

The subroutines within the interface block must have two nonoptional arguments, the first with intent OUT or INOUT, and the second with intent IN and/or attribute VALUE.

A defined assignment is treated as a reference to a subroutine. The left side of the assignment corresponds to the first dummy argument of the subroutine; the right side of the assignment corresponds to the second argument.

The ASSIGNMENT keyword extends or redefines an assignment operation if both sides of the equal sign are of the same derived type.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

Any procedure reference involving generic assignment must be resolvable to one specific procedure; it must be unambiguous. For more information, see Unambiguous Generic Procedure References.

The following is an example of a procedure interface block defining assignment:

```
INTERFACE ASSIGNMENT (=)
  SUBROUTINE BIT_TO_NUMERIC (NUM, BIT)
    INTEGER, INTENT(OUT) :: NUM
    LOGICAL, INTENT(IN)  :: BIT(:)
  END SUBROUTINE BIT_TO_NUMERIC

  SUBROUTINE CHAR_TO_STRING (STR, CHAR)
    USE STRING_MODULE                   ! Contains definition of type STRING
    TYPE(STRING), INTENT(OUT) :: STR    ! A variable-length string
    CHARACTER(*), INTENT(IN)  :: CHAR
  END SUBROUTINE  CHAR_TO_STRING
END  INTERFACE
```

The following example shows two equivalent ways to reference subroutine BIT_TO_NUMERIC:

```
CALL BIT_TO_NUMERIC(X, (NUM(I:J)))
X = NUM(I:J)
```

The following example shows two equivalent ways to reference subroutine CHAR_TO_STRING:

```
CALL CHAR_TO_STRING(CH, '432C')
CH = '432C'
```

See Also

Defined Assignments
INTENT attribute
INTERFACE statement for details on interface blocks

Interoperability of Procedures and Procedure Interfaces

A Fortran procedure is interoperable if it has the BIND attribute.

A Fortran procedure interface is interoperable with a C function prototype if the following is true:

- The interface has the BIND attribute
- One of the following is true:
  - The interface describes a function whose result variable is a scalar that is interoperable with the result of the prototype.

  - The interface describes a subroutine and the prototype has a result type of void.
- The number of dummy arguments of the interface is equal to the number of formal parameters of the prototype.
- Any dummy argument with the VALUE attribute is interoperable with the corresponding formal parameter of the prototype.
- Any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of a pointer type, and one of the following is true:
  - The dummy argument is interoperable with an entity of the referenced type of the formal parameter.
  - The dummy argument is a nonallocatable nonpointer variable of type CHARACTER with assumed character length and the formal parameter is a pointer to the C descriptior descriptor CFI_cdesc_t.
  - The dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer without the CONTIGUOUS attribute, and the formal parameter is a pointer to the C descriptior descriptor CFI_cdesc_t.
  - The dummy argument is assumed-type and not allocatable, assumed-shape, assumed-rank, or a pointer, and the formal parameter is a pointer to void.
- Each allocatable or pointer dummy argument of type CHARACTER has deferred character length.
- The prototype does not have variable arguments as denoted by an ellipsis (...).

In an invocation of an interoperable procedure whose Fortran interface has an assumed-shape or assumed-rank dummy argument with the CONTIGUOUS attribute, the associated effective argument can be an array that is not contiguous or it can be the address of a C descriptor for such an array.

If the procedure is invoked from Fortran or the procedure is a Fortran procedure, the Fortran processor will handle the difference in contiguity.

If the procedure is invoked from C or the procedure is a C procedure, the C code within the procedure must be able to handle the situation of receiving a discontiguous argument.

An actual argument that is absent in a reference to an interoperable procedure is indicated by a corresponding formal parameter with the value of a null pointer. An optional dummy argument that is absent in a reference to an interoperable procedure from a C function is indicated by a corresponding argument with the value of a null pointer.

The following rules also apply:

- C functions must not invoke a function pointer whose value is the result of a reference to C_FUNLOC with a noninteroperable argument.
- When passing an argument to a C procedure where the corresponding C formal parameter is a C descriptor, Fortran must pass a C descriptor and, on return, ensure that any updates to the C descriptor are reflected in Fortran.
- If the interface specifies that the dummy argument is CONTIGUOUS, the passed argument (and the C descriptor's description of that argument, if relevant) must be contiguous.
- A Fortran procedure with BIND(C) that has a dummy argument that is assumed-length CHARACTER or is allocatable, assumed-shape, assumed-rank, or a pointer without CONTIGUOUS must accept that argument as a C descriptor and make sure that on return, the C descriptor reflects any changes made to the argument during execution of the Fortran procedure.
- A Fortran procedure with one of the following arguments must accept that argument as a C descriptor and make sure that on return, the C descriptor reflects any changes made to the argument during execution of the Fortran procedure:
  - A dummy argument that is assumed-length CHARACTER
  - A dummy argument that is allocatable, assumed-shape, assumed-rank, or a pointer without CONTIGUOUS

Procedure Pointers

A procedure pointer has the POINTER attribute and points to a procedure instead of a data object. It can be associated with an external procedure, a module procedure, an intrinsic procedure, or a dummy procedure that is not a procedure pointer. It can have an implicit or explicit interface, but the interface cannot be generic or elemental.

A procedure pointer can be one of the following:

- A named pointer (described below)

- A derived-type component (See Procedure Pointers as Derived-Type Components.)

Procedure Pointers as Named Pointers

You can declare a procedure pointer in a procedure declaration statement by including the POINTER attribute. For example:

```
PROCEDURE(QUARK), POINTER :: Q => NULL()
```

The above declares Q to be a procedure pointer with interface QUARK; it also initializes Q to be a disassociated pointer.

A named procedure pointer can also be declared by specifying the POINTER attribute in addition to the normal procedure declaration.

The following example uses a type declaration statement to declare a procedure pointer:

```
POINTER :: MyP
INTERFACE
  SUBROUTINE MyP(c,d)
    REAL, INTENT(INOUT) :: c
    REAL, INTENT(IN) :: d
  END SUBROUTINE MyP
END INTERFACE
REAL, EXTERNAL, POINTER :: MyR
```

The above specifies that MyP is a pointer to a subroutine with an explicit interface. It also specifies that MyR is a pointer to a scalar REAL function with an implicit interface.

Note that in a type declaration statement, you must specify the EXTERNAL attribute as well as the POINTER attribute when declaring the procedure pointer.

See Also

INTERFACE
ABSTRACT INTERFACE
PROCEDURE

Intrinsic Procedures

Intrinsic procedures are functions and subroutines that are included in the Fortran library. The following are classes of intrinsic procedures:

- Elemental procedures

  These procedures have scalar dummy arguments that can be called with scalar or array actual arguments. There are many elemental intrinsic functions and one elemental intrinsic subroutine (MVBITS). All elemental intrinsic procedures are pure.

  If the arguments are all scalar, the result is scalar. If an actual argument is array-valued, the intrinsic procedure is applied to each element of the actual argument, resulting in an array that has the same shape as the actual argument.

  If there is more than one array-valued argument, they must all have the same shape.

  Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays. For example, consider the following:

```
a = b + c
...               ! a, b, c, and s are all arrays of similar shape
s = sum(a)
```

  The above statements can replace entire DO loops.

Consider the following:

```
real, dimension (5,5) :: x,y
. . .      ! Assign values to x
y = sin(x) ! Pass the entire array as an argument
```

In this example, since the SIN(X) function is an elemental procedure, it operates element-by-element on the array x when you pass it the name of the whole array.

- Inquiry functions

  These functions have results that depend on the properties of their principal argument, not the value of the argument (the argument value can be undefined).
- Transformational functions

  These functions have one or more array-valued dummy or actual arguments, an array result, or both. The intrinsic function is not applied elementally to an array-valued actual argument; instead it changes (transforms) the argument array into another array.
- Nonelemental procedures

  These procedures must be called with only scalar arguments; they return scalar results. All subroutines (except MVBITS) are nonelemental.
- Atomic subroutines

  These subroutines perform an action on a variable (its *atom* argument) atomically. When an atomic subroutine is executed, it is as if the subroutine were executed instantaneously without overlapping other atomic actions that might occur asynchronously. For information on the semantics of atomic subroutines, see Overview of Atomic Subroutines.
- Collective subroutines

  These subroutines perform a cooperative calculation on a team of images and require no synchronization. For information on the semantics of collective subroutines, see Overview of Collective Subroutines.

The intrinsic subroutine MVBITS, and the subroutine MOVE_ALLOC with a noncoarray argument FROM, are pure. All other intrinsic subroutines are impure.

Intrinsic procedures are invoked the same way as other procedures, and follow the same rules of argument association.

The intrinsic procedures have generic (or common) names, and many of the intrinsic functions have specific names. (Some intrinsic functions are both generic and specific.)

In general, generic functions accept arguments of more than one data type; the data type of the result is the same as that of the arguments in the function reference. For elemental functions with more than one argument, all arguments must be of the same type (except for the function MERGE).

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Some specific intrinsic functions are not allowed as actual arguments in all circumstances. The following table lists specific functions that cannot be passed as actual arguments or as targets in procedure pointer assignment statements.

Starting with Fortran 2018, specific names of intrinsic functions that also have generic names are obsolescent.

Specific Intrinsic Functions Not Allowed as Actual Arguments

AIMAX0	FLOATJ	JFIX	MAX0
AIMIN0	FLOATK	JIDINT	MAX1
AJMAX0	FP_CLASS	JIFIX	MIN0
AJMIN0	HFIX	JINT	MIN1
AKMAX0	IADDR	JIQINT	NARGS
AKMIN0	IARGC	JMAX0	QCMPLX
AMAX0	ICHAR	JMAX1	QEXT

AMAX1	IDINT	JMIN0	QEXTD
AMIN0	IFIX	JMIN1	QMAX1
AMIN1	IIDINT	JNUM	QMIN1
CHAR	IIFIX	JZEXT	QNUM
CMPLX	IINT	KIDINT	QREAL
DBLE	IIQINT	KIFIX	RAN
DBLEQ	IJINT	KINT	RANF
DCMPLX	IMAX0	KIQINT	REAL
DFLOTI	IMAX1	KMAX0	RNUM
DFLOTJ	IMIN0	KMAX1	SECNDS
DFLOTK	IMIN1	KMIN0	SHIFTL
DMAX1	INT	KMIN1	SHIFTR
DMIN1	INT1	KNUM	SNGL
DNUM	INT2	KZEXT	SNGLQ
DREAL	INT4	LGE	ZEXT
DSHIFTL	INT8	LGT	
DSHIFTR	INUM	LLE	
FLOAT	IQINT	LLT	
FLOATI	IZEXT	LOC	

Note that none of the intrinsic subroutines can be passed as actual arguments or as targets in procedure pointer assignment statements.

The A to Z Reference contains the descriptions of all intrinsics listed in alphabetical order. Each reference entry indicates whether the procedure is inquiry, elemental, transformational, or nonelemental, and whether it is a function or a subroutine.

See Also

Argument association
MERGE
Optional arguments
Data representation models
References to Generic Intrinsic Functions
References to Elemental Intrinsic Procedures

Argument Keywords in Intrinsic Procedures

For all intrinsic procedures, the arguments shown are the names you must use as keywords when using the keyword form for actual arguments. For example, a reference to function CMPLX(X, Y, KIND) can be written as follows:

Using positional arguments:	CMPLX(F, G, L)
Using argument keywords: [1]	CMPLX(KIND=L, Y=G, X=F)

[1] Note that argument keywords can be written in any order.

Some argument keywords are optional (denoted by square brackets). The following describes some of the most commonly used optional arguments:

BACK	Specifies that a string scan is to be in reverse order (right to left).
DIM	Specifies a selected dimension of an array argument.
KIND	Specifies the kind type parameter of the function result.
MASK	Specifies that a mask can be applied to the elements of the argument array to exclude the elements that are not to be involved in an operation.

Examples

The syntax for the DATE_AND_TIME intrinsic subroutine shows four optional positional arguments: DATE, TIME, ZONE, and VALUES. The following shows some valid ways to specify these arguments:

```
! Keyword example
CALL DATE_AND_TIME (ZONE=Z)

! Positional example
CALL DATE_AND_TIME (DATE, TIME, ZONE)
```

See Also

CALL for details on argument keywords in subroutine references

Function references for details on argument keywords in function references

Argument Association

Overview of Atomic Subroutines

Atomic subroutines are impure intrinsic procedures that perform an action on their *atom* argument atomically. If a reference to an atomic subroutine has an *old* argument, the value to be assigned to that argument is also determined atomically with the action performed on the *atom* argument. The evaluation or definition of any other argument is not performed atomically.

The *stat* argument, if present, has a value of zero if no error condition occurs during the subroutine reference.

If *stat* is present and an error condition occurs, any INTENT(INOUT) or INTENT(OUT) argument becomes undefined. If the *atom* argument is on a failed image, *stat*, if present, becomes defined with the value STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV and an error condition occurs. If any other error condition occurs, *stat* becomes defined with a processor-dependent positive integer value other than that of STAT_FAILED_IMAGE.

If an error condition occurs and *stat* is not present, error termination is initiated.

For a list of all atomic intrinsic subroutines, including links to the subroutine's full description, see Atomic Intrinsic Subroutines.

Overview of Collective Subroutines

Collective subroutines are impure intrinsic subroutines that perform a calculation on a team of images, assigning the result to one of the images or all of the images on the current team. Synchronization is not required. When the collective subroutine is invoked, it is invoked by the same statement on all active images of the current team. Corresponding references to the subroutine participate in the same collective operation.

The sequence of invocations of collective subroutines must be the same across all active images of the current team. A collective subroutine cannot be invoked anywhere an image control statement is not permitted. For example, a pure procedure or a critical construct cannot contain a reference to a collective subroutine.

If argument *a* in an invocation of a collective subroutine is a coarray, it must ultimately be the same coarray on each active image of the current team.

If argument *stat* is present in the reference to a collective subroutine on one image, it must be present in corresponding references on all images of the current team.

Successful execution of a collective subroutine causes the value of *stat*, if present, to become defined with the value 0.

If an error condition occurs during the reference to the collective subroutine and *stat* is present, *stat* is assigned a positive value and argument *a* becomes undefined. If *stat* is present and the current team contains a stopped image, an error condition occurs and *stat* becomes defined with the value STAT_STOPPED_IMAGE defined in the intrinsic module ISO_FORTRAN_ENV. Otherwise, if the current team contains a failed image, an error condition occurs and *stat* becomes defined with the value STAT_FAILED_IMAGE from ISO_FORTRAN_ENV. If any other error condition occurs, *stat* becomes defined with a positive integer value other than STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

If *stat* is not present in a reference to a collective subroutine and an error condition occurs during the reference, error termination is initiated.

If argument *errmsg* is present when an error condition occurs, it becomes defined with an explanatory message, padded with blanks or truncated as necessary. If no error condition occurs, the value and definition status of *errmsg* is not changed.

Internal synchronization occurs during a reference to a collective subroutine, but a statement containing a reference to a collective subroutine is not an image control statement.

For a list of all collective intrinsic subroutines, including links to the subroutine's full description, see Collective Intrinsic Subroutines.

Overview of Bit Functions

Integer data types are represented internally in binary two's complement notation. Bit positions in the binary representation are numbered from right (least significant bit) to left (most significant bit); the rightmost bit position is numbered 0.

The intrinsic functions IAND, IOR, IEOR, and NOT operate on all of the bits of their argument (or arguments). Bit 0 of the result comes from applying the specified logical operation to bit 0 of the argument. Bit 1 of the result comes from applying the specified logical operation to bit 1 of the argument, and so on for all of the bits of the result.

The functions ISHFT and ISHFTC shift binary patterns.

The functions IBSET, IBCLR, BTEST, and IBITS and the subroutine MVBITS operate on bit fields.

A *bit field* is a contiguous group of bits within a binary pattern. Bit fields are specified by a starting bit position and a length. A bit field must be entirely contained in its source operand.

For example, the integer 47 is represented by the following:

Binary pattern:	0...0101111
Bit position:	n...6543210 Where *n* is the number of bit positions in the numeric storage unit.

You can refer to the bit field contained in bits 3 through 6 by specifying a starting position of 3 and a length of 4.

Negative integers are represented in two's complement notation. For example, the integer -47 is represented by the following:

Binary pattern:	1...1010001
Bit position:	n...6543210 Where n is the number of bit positions in the numeric storage unit.

The value of bit position n is as follows:

```
  1 for a negative number
  0 for a non-negative number
```

All the high-order bits in the pattern from the last significant bit of the value up to bit n are the same as bit n.

IBITS and MVBITS operate on general bit fields. Both the starting position of a bit field and its length are arguments to these intrinsics. IBSET, IBCLR, and BTEST operate on 1-bit fields. They do not require a length argument.

For IBSET, IBCLR, and BTEST, the bit position range is as follows:

- 0 to 63 for INTEGER(8) and LOGICAL(8)
- 0 to 31 for INTEGER(4) and LOGICAL(4)
- 0 to 15 for INTEGER(2) and LOGICAL(2)
- 0 to 7 for BYTE, INTEGER(1), and LOGICAL(1)

For IBITS, the bit position can be any number. The length range is 0 to 63 on Intel® 64 architecture; 0 to 31 on IA-32 architecture.

The following example shows IBSET, IBCLR, and BTEST:

```
I = 4
J = IBSET (I,5)
PRINT *, 'J = ',J
K = IBCLR (J,2)
PRINT *, 'K = ',K
PRINT *, 'Bit 2 of K is ',BTEST(K,2)
END
```

The results are: J = 36, K = 32, and Bit 2 of K is F.

For optimum selection of performance and memory requirements, Intel® Fortran provides the following integer data types:

Data Type	Storage Required (in bytes)
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8

The bit manipulation functions each have a generic form that operates on all of these integer types and a specific form for each type.

When you specify the intrinsic functions that refer to bit positions or that shift binary patterns within a storage unit, be careful that you do not create a value that is outside the range of integers representable by the data type. If you shift by an amount greater than or equal to the size of the object you're shifting, the result is 0.

Consider the following:

```
INTEGER(2) I,J
I = 1
J = 17
I = ISHFT(I,J)
```

The variables I and J have INTEGER(2) type. Therefore, the generic function ISHFT maps to the specific function IISHFT, which returns an INTEGER(2) result. INTEGER(2) results must be in the range -32768 to 32767, but the value 1, shifted left 17 positions, yields the binary pattern 1 followed by 17 zeros, which represents the integer 131072. In this case, the result in I is 0.

The previous example would be valid if I was INTEGER(4), because ISHFT would then map to the specific function JISHFT, which returns an INTEGER(4) value.

If ISHFT is called with a constant first argument, the result will either be the default integer size or the smallest integer size that can contain the first argument, whichever is larger.

Categories and Lists of Intrinsic Procedures

This section describes the categories of generic intrinsic functions (including a summarizing table) and lists the intrinsic subroutines.

Intrinsic procedures are fully described (in alphabetical order) in the A to Z Reference.

Categories of Intrinsic Functions

Generic intrinsic functions can be divided into categories, as shown in the following table:

Categories of Intrinsic Functions

Category	Subcategory	Description
Numeric	Computation	Elemental functions that perform type conversions or simple numeric operations: ABS, AIMAG, AINT, AMAX0, AMIN0, ANINT, CEILING, CMPLX, CONJG, DBLE, DCMPLX, DFLOAT, DIM, DNUM, DPROD, DREAL, FLOAT, FLOOR, IFIX, IMAG, INT, INUM, JNUM, KNUM MAX, MAX1, MIN, MIN1, MOD, MODULO, NINT, QCMPLX, QEXT, QFLOAT, QNUM, QREAL, REAL, RNUM, SIGN, SNGL, ZEXT
		Nonelemental function that provides a pseudorandom number: RAN
		Elemental function that generates a random number: RANF
	Manipulation[1]	Elemental functions that return values related to the components of the model values associated with the actual value of the argument: EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING
	Inquiry[1]	Functions that return scalar values from the models associated with the type and kind parameters of their arguments[2]: DIGITS, EPSILON, HUGE, ILEN,

Category	Subcategory	Description
		MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE, SIZEOF, TINY
	Transformational	Functions that perform vector and matrix multiplication: DOT_PRODUCT, MATMUL
	System	Functions that return information about a process or processor: MCLOCK, SECNDS
Kind type		Functions that return kind type parameters: KIND, SELECTED_CHAR_KIND, SELECTED_INT_KIND, SELECTED_REAL_KIND
Mathematical		Elemental functions that perform mathematical operations: ACOS, ACOSD, ACOSH, ASIN, ASIND, ASINH, ATAN, ATAN2, ATAN2D,ATAND, ATANH, BESSEL_J0, BESSEL_J1, BESSEL_JN, BESSEL_Y0, BESSEL_Y1, BESSEL_YN, COS, COSD, COSH,COTAN, COTAND, EXP, EXP10, GAMMA, HYPOT, LOG, LOG10, LOG_GAMMA, SIN, SIND, SINH, SQRT, TAN, TAND, TANH
Bit	Manipulation	Elemental functions that perform bit operations, such as single-bit processing, logical and shift operations, and allowing bit subfields to be referenced: AND, BGE, BGT, BLE, BLT, BTEST, DSHIFTL, DSHIFTR, IAND, IBCHNG, IBCLR, IBITS, IBSET, IEOR, IOR, ISHA, ISHC, ISHFT, ISHFTC, ISHL, IXOR,LSHIFT, MASKL, MASKR, MERGE_BITS, NOT, OR, RSHIFT, SHIFTA, SHIFTL, SHIFTR, XOR
	Inquiry	Function that lets you determine bit size and storage size: BIT_SIZE, STORAGE_SIZE
	Representation	Elemental functions that return information on bit representation of integers: LEADZ, POPCNT, POPPAR, TRAILZ
Character	Comparison	Elemental functions that make a lexical comparison of the character-string arguments and return a default logical result: LGE, LGT, LLE, LLT

Category	Subcategory	Description
	Conversion	Elemental functions that take character arguments and return integer, ASCII, or character values[4]: ACHAR, CHAR, IACHAR, ICHAR
	String handling	Functions that perform operations on character strings, return lengths of arguments, and search for certain arguments: Elemental: ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, SCAN, VERIFY; Nonelemental: REPEAT, TRIM
	Inquiry	Functions that return the length of an argument or information about command-line arguments: COMMAND_ARGUMENT_COUNT, IARG, IARGC, LEN, NARGS, NUMARG
Array	Construction	Functions that construct new arrays from the elements of existing arrays: Elemental: MERGE; Nonelemental: PACK, SPREAD, UNPACK
	Inquiry	Functions that let you determine if an array argument is allocated, and return the size or shape of an array, and the lower and upper bounds of subscripts along each dimension: ALLOCATED, IS_CONTIGUOUS, LBOUND, RANK, SHAPE, SIZE, UBOUND
	Location	Transformational functions that find the geometric locations of the maximum and minimum values of an array, and find the location of a specified value in an array: MAXLOC, MINLOC, FINDLOC
	Manipulation	Transformational functions that shift an array, transpose an array, or change the shape of an array: CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE
	Reduction	Transformational functions that perform operations on arrays. The functions "reduce" elements of a whole array to produce a scalar result, or they can be applied to a specific dimension of an array to produce a result array with a rank reduced by one: ALL,

Category	Subcategory	Description
		ANY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, NORM2, PARITY, PRODUCT, SUM
Coarray	Inquiry	Functions that return execution status of an image, convert cosubscripts, or return sizes of codimensions, or lower or upper cobounds: Elemental: IMAGE_STATUS Nonelemental: COSHAPE, IMAGE_INDEX, LCOBOUND, UCOBOUND
	Transformational	Functions that return the number of images or cosubscripts, or return image indices of failed or stopped images: NUM_IMAGES, THIS_IMAGE, FAILED_IMAGES, STOPPED_IMAGES
Polymorphic	Inquiry	Functions that let you determine the dynamic type of an object: EXTENDS_TYPE_OF, SAME_TYPE_AS
Miscellaneous		Functions that do the following: • Check for pointer association (ASSOCIATED) • Return an address (BADDRESS or IADDR) • Return the size of a level of the memory cache (CACHESIZE) • Check for end-of-file (EOF) • Return error functions (ERF, ERFC,and ERFC_SCALED) • Return the class of a floating-point argument (FP_CLASS) • Return the INTEGER KIND that will hold an address (INT_PTR_KIND) • Test for Not-a-Number values (ISNAN) • Return the internal address of a storage item (LOC) • Return a logical value of an argument (LOGICAL) • Allocate memory (MALLOC) • Return a new line character (NEW_LINE) • Return a disassociated pointer (NULL) • Check for argument presence (PRESENT) • Convert a bit pattern (TRANSFER)

Category	Subcategory	Description
		• Check for end-of-file condition (IS_IOSTAT_END) • Check for end-of-record condition (IS_IOSTAT_EOR)

[1] All of the numeric manipulation, and many of the numeric inquiry functions are defined by the model sets for integers and reals.

[2] The value of the argument does not have to be defined.

[3] For more information on bit functions, see Bit functions.

[4] The Intel® Fortran processor character set is ASCII, so ACHAR = CHAR and IACHAR = ICHAR.

The following table summarizes the generic intrinsic functions and indicates whether they are elemental, inquiry, or transformational functions. Optional arguments are shown within square brackets.

Some intrinsic functions are specific with no generic association. These functions are listed below.

Summary of Generic Intrinsic Functions

Generic Function	Class	Value Returned
ABS (A)	E	The absolute value of an argument
ACHAR (I [,KIND])	E	The character in the specified position of the ASCII character set
ACOS (X)	E	The arccosine (in radians) of the argument
ACOSD (X)	E	The arccosine (in degrees) of the argument
ACOSH (X)	E	The hyperbolic arccosine of the argument
ADJUSTL (STRING)	E	The specified string with leading blanks removed and placed at the end of the string
ADJUSTR (STRING)	E	The specified string with trailing blanks removed and placed at the beginning of the string
AIMAG (Z)	E	The imaginary part of a complex argument
AINT (A [,KIND])	E	A real value truncated to a whole number
ALL (MASK [,DIM])	T	.TRUE. if all elements of the masked array are true
ALLOCATED ([ARRAY=]array) or ALLOCATED ([SCALAR=]scalar)	I	The allocation status of the argument array or scalar
AMAX0 (A1, A2 [, A3,...])	E	The maximum value in a list of integers (returned as a real value)

Generic Function	Class	Value Returned
AMIN0 (A1, A2 [, A3,...])	E	The minimum value in a list of integers (returned as a real value)
AND (I, J)	E	See IAND
ANINT (A [, KIND])	E	A real value rounded to a whole number
ANY (MASK [, DIM])	T	.TRUE. if any elements of the masked array are true
ASIN (X)	E	The arcsine (in radians) of the argument
ASIND (X)	E	The arcsine (in degrees) of the argument
ASINH (X)	E	The hyperbolic arcsine of the argument
ASSOCIATED (POINTER [,TARGET])	I	.TRUE. if the pointer argument is associated or the pointer is associated with the specified target
ATAN (X)	E	The arctangent (in radians) of the argument
ATAN2 (Y, X)	E	The arctangent (in radians) of the arguments
ATAN2D (Y, X)	E	The arctangent (in degrees) of the arguments
ATAND (X)	E	The arctangent (in degrees) of the argument
ATANH (X)	E	The hyperbolic arctangent of the argument
BADDRESS (X)	I	The address of the argument
BESSEL_J0 (X)	E	A Bessel function of the first kind, order 0
BESSEL_J1 (X)	E	A Bessel function of the first kind, order 1
BESSEL_JN (N, X)	E	A Bessel function of the first kind, order N
BESSEL_JN (N1, N2, X)	T	A Bessel function of the first kind
BESSEL_Y0 (X)	E	A Bessel function of the second kind, order 0
BESSEL_Y1 (X)	E	A Bessel function of the second kind, order 1
BESSEL_YN (N, X)	E	A Bessel function of the second kind, order N

Generic Function	Class	Value Returned
BESSEL_YN (N1, N2, X)	T	A Bessel function of the second kind
BGE (I, J)	E	Bitwise greater than or equal to
BGT (I, J)	E	Bitwise greater than
BIT_SIZE (I)	I	The number of bits (*s*) in the bit model
BLE (I, J)	E	Bitwise less than or equal to
BLT (I, J)	E	Bitwise less than
BTEST (I, POS)	E	.TRUE. if the specified position of argument I is one
CEILING (A [,KIND])	E	The smallest integer greater than or equal to the argument value
CHAR (I [,KIND])	E	The character in the specified position of the processor character set
COMMAND_ARGUMENT_COUNT ()	I	The number of command arguments
CONJG (Z)	E	The conjugate of a complex number
COS (X)	E	The cosine of the argument, which is in radians
COSD (X)	E	The cosine of the argument, which is in degrees
COSH (X)	E	The hyperbolic cosine of the argument
COSHAPE (COARRAY [,KIND])	I	The sizes of codimensions of a coarray.
COTAN (X)	E	The cotangent of the argument, which is in radians
COTAND (X)	E	The cotangent of the argument, which is in degrees
COUNT (MASK [, DIM, KIND])	T	The number of .TRUE. elements in the argument array
CSHIFT (ARRAY, SHIFT [,DIM])	T	An array that has the elements of the argument array circularly shifted
DBLE (A)	E	The corresponding double precision value of the argument
DFLOAT (A)	E	The corresponding double precision value of the integer argument

Generic Function	Class	Value Returned
DIGITS (X)	I	The number of significant digits in the model for the argument
DIM (X, Y)	E	The positive difference between the two arguments
DOT_PRODUCT (VECTOR_A, VECTOR_B)	T	The dot product of two rank-one arrays (also called a vector multiply function)
DREAL (A)	E	The corresponding double-precision value of the double complex argument
DSHIFTL (ILEFT, IRIGHT, ISHIFT)	E	The upper (leftmost) 64 bits of a left-shifted 128-bit integer
DSHIFTR (ILEFT, IRIGHT, ISHIFT)	E	The lower (rightmost) 64 bits of a right-shifted 128-bit integer
EOF (A)	I	.TRUE. or .FALSE. depending on whether a file is beyond the end-of-file record
EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])	T	An array that has the elements of the argument array end-off shifted
EPSILON (X)	I	The number that is almost negligible when compared to one
ERF (X)	E	The error function of an argument
ERFC (X)	E	The complementary error function of an argument
ERFC_SCALED (X)	E	The scaled complementary error function of an argument
EXP (X)	E	The exponential e^x for the argument x
EXPONENT (X)	E	The value of the exponent part of a real argument
EXTENDS_TYPE_OF (A, MOLD)	I	Whether one dynamic type is an extension of another dynamic type
FAILED_IMAGES ([KIND])	T	Image indices of images known to have failed on the specified or current team
FINDLOC (ARRAY, VALUE, DIM [, MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE [, MASK, KIND, BACK])	T	Location of a specified value in an array.
FLOAT (X)	E	The corresponding real value of the integer argument

Generic Function	Class	Value Returned
FLOOR (A [,KIND])	E	The largest integer less than or equal to the argument value
FP_CLASS (X)	E	The class of the IEEE floating-point argument
FRACTION (X)	E	The fractional part of a real argument
GAMMA (X)	E	A gamma function
HUGE (X)	I	The largest number in the model for the argument
HYPOT (X, Y)	E	A Euclidean distance function
IACHAR (C [,KIND])	E	The position of the specified character in the ASCII character set
IADDR (X)	E	See BADDRESS
IAND (I, J)	E	The logical AND of the two arguments
IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])	T	The result of a bitwise AND operation
IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY [, MASK])	T	The result of a bitwise OR operation
IBCLR (I, POS)	E	The specified position of argument I cleared (set to zero)
IBCHNG (I, POS)	E	The reversed value of a specified bit
IBITS (I, POS, LEN)	E	The specified substring of bits of argument I
IBSET (I, POS)	E	The specified bit in argument I set to one
ICHAR (C [, KIND])	E	The position of the specified character in the processor character set
IEOR (I, J)	E	The logical exclusive OR of the corresponding bit arguments
IFIX (X)	E	The corresponding integer value of the real argument rounded as if it were an implied conversion in an assignment
ILEN (I)	I	The length (in bits) in the two's complement representation of an integer
IMAG (Z)	E	See AIMAG
IMAGE_INDEX (COARRAY, SUB)	T	The index of the corresponding image

Generic Function	Class	Value Returned
IMAGE_STATUS (IMAGE)	E	Execution status of the specified image number on the specified or current team
INDEX (STRING, SUBSTRING [, BACK, KIND])	E	The position of the specified substring in a character expression
INT (A [, KIND])	E	The corresponding integer value (truncated) of the argument
IOR (I, J)	E	The logical inclusive OR of the corresponding bit arguments
IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY [, MASK])	T	The result of a bitwise exclusive OR operation
IS_CONTIGUOUS (ARRAY)	I	The contiguity of an array
IS_IOSTAT_END (I)	E	.TRUE. for an end-of-file condition
IS_IOSTAT_EOR (I)	E	.TRUE. for an end-of-record condition
ISHA (I, SHIFT)	E	Argument I shifted left or right by a specified number of bits
ISHC (I, SHIFT)	E	Argument I rotated left or right by a specified number of bits
ISHFT (I, SHIFT)	E	The logical end-off shift of the bits in argument I
ISHFTC (I, SHIFT [,SIZE])	E	The logical circular shift of the bits in argument I
ISHL (I, SHIFT)	E	Argument I logically shifted left or right by a specified number of bits
ISNAN (X)	E	Tests for Not-a-Number (NaN) values
IXOR (I, J)	E	See IEOR
KIND (X)	I	The kind type parameter of the argument
LBOUND (ARRAY [, DIM, KIND])	I	The lower bounds of an array (or one of its dimensions)
LEADZ (I)	E	The number of leading zero bits in an integer
LEN (STRING [,KIND])	I	The length (number of characters) of the argument character string
LEN_TRIM (STRING [,KIND])	E	The length of the specified string without trailing blanks

Generic Function	Class	Value Returned
LGE (STRING_A, STRING_B)	E	A logical value determined by a > or = comparison of the arguments
LGT (STRING_A, STRING_ B)	E	A logical value determined by a > comparison of the arguments
LLE (STRING_A, STRING_B)	E	A logical value determined by a < or = comparison of the arguments
LLT (STRING_A, STRING_B)	E	A logical value determined by a < comparison of the arguments
LOC (A)	I	The internal address of the argument.
LOG (X)	E	The natural logarithm of the argument
LOG10 (X)	E	The common logarithm (base 10) of the argument
LOG_GAMMA (X)	E	The logarithm of the absolute value of the gamma function
LOGICAL (L [,KIND])	E	The logical value of the argument converted to a logical of type KIND
LSHIFT (I, POSITIVE_SHIFT)	E	See ISHFT
LSHFT (I, POSITIVE_SHIFT)	E	Same as LSHIFT; see ISHFT
MALLOC (I)	E	The starting address for the block of memory allocated
MASKL (I [,KIND])	E	A left-justified mask
MASKR (I [,KIND])	E	A right-justified mask
MATMUL (MATRIX_A, MATRIX_B)	T	The result of matrix multiplication (also called a matrix multiply function)
MAX (A1, A2 [, A3,...])	E	The maximum value in the set of arguments
MAX1 (A1, A2 [, A3,...])	E	The maximum value in the set of real arguments (returned as an integer)
MAXEXPONENT (X)	I	The maximum exponent in the model for the argument
MAXLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MAXLOC (ARRAY [, MASK, KIND, BACK])	T	The rank-one array that has the location of the maximum element in the argument array
MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])	T	The maximum value of the elements in the argument array

Generic Function	Class	Value Returned
MERGE (TSOURCE, FSOURCE, MASK)	E	An array that is the combination of two conformable arrays (under a mask)
MERGE_BITS (I, J, MASK)	E	The merge of bits under a mask
MIN (A1, A2 [, A3,...])	E	The minimum value in the set of arguments
MIN1 (A1, A2 [, A3,...])	E	The minimum value in the set of real arguments (returned as an integer)
MINEXPONENT (X)	I	The minimum exponent in the model for the argument
MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])	T	The rank-one array that has the location of the minimum element in the argument array
MINVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])	T	The minimum value of the elements in the argument array
MOD (A, P)	E	The remainder of the arguments (has the sign of the first argument)
MODULO (A, P)	E	The modulo of the arguments (has the sign of the second argument)
NEAREST (X, S)	E	The nearest different machine-representable number in a given direction
NEW_LINE (A)	I	A new line character
NINT (A [,KIND])	E	A real value rounded to the nearest integer
NORM2 (X [,DIM])	T	The L2 norm of an array
NOT (I)	E	The logical complement of the argument
NULL ([MOLD])	T	A disassociated pointer
NUM_IMAGES ()	T	The number of images
OR (I, J)	E	See IOR
PACK (ARRAY, MASK [,VECTOR])	T	A packed array of rank one (under a mask)
PARITY (MASK [, DIM])	T	The result of an exclusive OR operation
POPCNT (I)	E	The number of 1 bits in the integer argument
POPPAR (I)	E	The parity of the integer argument

Generic Function	Class	Value Returned
PRECISION (X)	I	The decimal precision (real or complex) of the argument
PRESENT (A)	I	.TRUE. if an actual argument has been provided for an optional dummy argument
PRODUCT (ARRAY, DIM [,MASK]) or PRODUCT (ARRAY [, MASK])	T	The product of the elements of the argument array
QEXT (A)	E	The corresponding REAL(16) precision value of the argument
QFLOAT (A)	E	The corresponding REAL(16) precision value of the integer argument
RADIX (X)	I	The base of the model for the argument
RANGE (X)	I	The decimal exponent range of the model for the argument
RANF ()	E	A random number between 0.0 and RAND_MAX
RANK (A)	I	The rank of a data object
REAL (A [, KIND])	E	The corresponding real value of the argument
REPEAT (STRING, NCOPIES)	T	The concatenation of zero or more copies of the specified string
RESHAPE (SOURCE, SHAPE [, PAD,ORDER])	T	An array that has a different shape than the argument array, but the same elements
RRSPACING (X)	E	The reciprocal of the relative spacing near the argument
RSHIFT (I, NEGATIVE_SHIFT)	E	See ISHFT
RSHFT (I, N EGATIVE_SHIFT)	E	Same as RSHIFT; see ISHFT
SAME_TYPE_AS (A, B)	I	Whether two dynamic types are the same.
SCALE (X, I)	E	The value of the exponent part (of the model for the argument) changed by a specified value
SCAN (STRING, SET [, BACK, KIND])	E	The position of the specified character (or set of characters) within a string
SELECTED_CHAR_KIND (NAME)	T	The value of the kind type parameter of the character set named by the argument

Generic Function	Class	Value Returned
SELECTED_INT_KIND (R)	T	The integer kind parameter of the argument
SELECTED_REAL_KIND ([P, R, RADIX])	T	The real kind parameter of the argument; one of the optional arguments must be specified
SET_EXPONENT (X, I)	E	The value of the exponent part (of the model for the argument) set to a specified value
SHAPE (SOURCE [,KIND])	I	The shape (rank and extents) of an array or scalar
SHIFTA (I, SHIFT)	E	A right shift with fill
SHIFTL (I, SHIFT)	E	Argument IVALUE shifted left by a specified number of bits
SHIFTR (I, SHIFT)	E	Argument IVALUE shifted right by a specified number of bits
SIGN (A, B)	E	A value with the sign transferred from its second argument
SIN (X)	E	The sine of the argument, which is in radians
SIND (X)	E	The sine of the argument, which is in degreess
SINH (X)	E	The hyperbolic sine of the argument
SIZE (ARRAY [, DIM, KIND])	I	The size (total number of elements) of the argument array (or one of its dimensions)
SIZEOF (X)	I	The bytes of storage used by the argument
SNGL (X)	E	The corresponding real value of the argument
SPACING (X)	E	The value of the absolute spacing of model numbers near the argument
SPREAD (SOURCE, DIM, NCOPIES)	T	A replicated array that has an added dimension
SQRT (X)	E	The square root of the argument
STOPPED_IMAGES ([KIND])	T	Image indices of stopped images on the specified or current team
STORAGE_SIZE (A [,KIND])	I	The storage size in bits
SUM (ARRAY, DIM] [, MASK]) or SUM (ARRAY [, MASK])	T	The sum of the elements of the argument array
TAN (X)	E	The tangent of the argument, which is in radians

Generic Function	Class	Value Returned
TAND (X)	E	The tangent of the argument, which is in degrees
TANH (X)	E	The hyperbolic tangent of the argument
THIS_IMAGE () or THIS_IMAGE (COARRAY [, DIM])	T	The index of the invoking image or the cosubscripts for the image
TINY (X)	I	The smallest positive number in the model for the argument
TRAILZ (I)	E	The number of trailing zero bits in an integer
TRANSFER (SOURCE, MOLD [,SIZE])	T	The bit pattern of SOURCE converted to the type and kind parameters of MOLD
TRANSPOSE (MATRIX)	T	The matrix transpose for the rank-two argument array
TRIM (STRING)	T	The argument with trailing blanks removed
UBOUND (ARRAY [, DIM, KIND])	I	The upper bounds of an array (or one of its dimensions)
UNPACK (VECTOR, MASK, FIELD)	T	An array (under a mask) unpacked from a rank-one array
VERIFY (STRING, SET [, BACK, KIND])	E	The position of the first character in a string that does not appear in the given set of characters
XOR (I, J)	E	See IEOR
ZEXT (X [,KIND])	E	A zero-extended value of the argument

Key to Classes

E-Elemental

I-Inquiry

T-Transformational

The following table lists specific functions that have no generic function associated with them and indicates whether they are elemental, nonelemental, or inquiry functions. Optional arguments are shown within square brackets.

Specific Functions with No Generic Association

Generic Function	Class	Value Returned
CACHESIZE (N)	I	The size of a level of the memory cache
CMPLX (X [,KIND]) or CMPLX (X [,Y, KIND])	E	The corresponding complex value of the argument
DCMPLX (X, Y)	E	The corresponding double complex value of the argument

Generic Function	Class	Value Returned
DNUM (I)	E	The corresponding REAL(8) value of a character string
DPROD (X, Y)	E	The double-precision product of two real arguments
DREAL (A)	E	The corresponding double-precision value of the double-complex argument
IARG ( )	I	See IARGC
IARGC ( )	I	The index of the last command-line argument
INT_PTR_KIND ( )	I	The INTEGER kind that will hold an address
INUM (I)	E	The corresponding INTEGER(2) value of a character string
JNUM (I)	E	The corresponding INTEGER(4) value of a character string
KNUM (I)	E	The corresponding INTEGER(8) value of a character string
MCLOCK ( )	I	The sum of the current process's user time and the user and system time of all its child processes
NARGS ( )	I	The total number of command-line arguments, including the command
NUMARG ( )	I	See IARGC
QCMPLX (X, Y)	E	The corresponding COMPLEX(16) value of the argument
QNUM (I)	E	The corresponding REAL(16) value of a character string
QREAL (A)	E	The corresponding REAL(16) value of the real part of a COMPLEX(16) argument
RAN (I)	N	The next number from a sequence of pseudorandom numbers (uniformly distributed in the range 0 to 1)
RNUM (I)	E	The corresponding REAL(4) value of a character string
SECNDS (X)	E	The system time of day (or elapsed time) as a floating-point value in seconds

Key to Classes

E-Elemental

Generic Function	Class	Value Returned
I-Inquiry		
N-Nonelemental		

Intrinsic Subroutines

The following table lists the intrinsic subroutines. Optional arguments are shown within square brackets. All these subroutines are nonelemental except for MVBITS. All of these subroutines, with the exception of MVBITS, and MOVE_ALLOC with a noncoarray FROM argument, are impure. None of the intrinsic subroutines can be passed as actual arguments.

Intrinsic Subroutines

Subroutine	Value Returned or Result
ATOMIC_ADD (atom, value [, stat])	Performs atomic addition.
ATOMIC_AND (atom, value [, stat])	Performs atomic bitwise AND.
ATOMIC_CAS (atom, old, compare, new [,stat])	Performs atomic compare and swap.
ATOMIC_DEFINE (atom, value [, stat])	Defines a variable atomically.
ATOMIC_FETCH_ADD (atom, value, old [, stat])	Performs atomic fetch and addition.
ATOMIC_FETCH_AND (atom, value, old [, stat])	Performs atomic fetch and bitwise AND.
ATOMIC_FETCH_OR (atom, value, old [, stat])	Performs atomic fetch and bitwise OR.
ATOMIC_FETCH_XOR (atom, value, old [, stat])	Performs atomic fetch and bitwise exclusive OR.
ATOMIC_OR (atom, value [, stat])	Performs atomic bitwise OR.
ATOMIC_REF (value, atom [, stat])	References a variable atomically.
ATOMIC_XOR (atom, value [, stat])	Performs atomic bitwise exclusive OR.
CO_BROADCAST (a, source_image [, stat, errmsg])	Broadcasts a value to other images.
CO_MAX (a, result_image [, stat, errmsg])	Computes maximum value across images.
CO_MIN (a, result_image [, stat, errmsg])	Computes minimum value across images.
CO_REDUCE (a, operation [, result_image, stat, errmsg])	Performs a generalized reduction across images.
CO_SUM (a, result_image [, stat, errmsg])	Performs a sum reduction across images.
CPU_TIME (time)	Returns the processor time in seconds.
DATE (buf)	Returns the ASCII representation of the current date (in dd-mmm-yy form).
DATE_AND_TIME ([date] [,time] [,zone] [,values])	Returns the date and time information from the real-time clock.
ERRSNS ([io_err] [,sys_err] [,stat] [,unit] [,cond])	Returns information about the most recently detected error condition.
EVENT_QUERY (event, count [, stat])	Queries an event count.
EXECUTE_COMMAND_LINE (command [, wait, exitstat, cmdstat, cmdmsg])	Executes the command line.

Subroutine	Value Returned or Result
EXIT ([status])	Image exit status is optionally returned; the program is terminated, all files closed, and control is returned to the operating system.
FREE (a)	Frees memory that is currently allocated.
GETARG (n, buffer [,status])	Returns the specified command line argument (where the command itself is argument number zero).
GET_COMMAND ([command, length, status])	Returns the entire command that was used to invoke the program.
GET_COMMAND_ARGUMENT (n [, value, length, status])	Returns a command line argument of the command that invoked the program.
GET_ENVIRONMENT_VARIABLE (name [, value, length, status, trim_name])	Returns the value of an environment variable.
IDATE (i, j, k)	Returns three integer values representing the current month, day, and year.
MM_PREFETCH (address [,hint] [,fault] [,exclusive])	Returns data from the specified address on one memory cache line.
MOVE_ALLOC (from, to [, stat, errmsg])	Causes an allocation to be moved from one allocatable object to another.
MVBITS (from, frompos, len, to, topos)[1]	Causes a sequence of bits (bit field) to be copied from one location to another.
RANDOM_NUMBER (harvest)	Returns a pseudorandom number taken from a sequence of pseudorandom numbers uniformly distributed within the range 0.0 to 1.0.
RANDOM_SEED ([size] [,put] [,get])	Causes the initialization or retrieval of the pseudorandom number generator seed value.
RANDU (i1, i2, x)	Returns a pseudorandom number as a single-precision value (within the range 0.0 to 1.0).
SYSTEM_CLOCK ([count] [,count_rate] [,count_max])	Returns data from the processors real-time clock.
TIME (buf)	Returns the ASCII representation of the current time (in hh:mm:ss form).

[1] An elemental subroutine

Data Transfer I/O Statements

Input/Output (I/O) statements can be used for data transfer, file connection, file inquiry, and file positioning.

This section discusses data transfer and contains information on the following topics:

- An overview of records and files
- Components of data transfer statements
- Data transfer input statements:
  - READ

Data can be input from external sequential or direct-access records, or from internal records.
- ACCEPT

  This statement is the same as a formatted, sequential READ statement, except that an ACCEPT statement must never be connected to user-specified I/O units.
- Data transfer output statements:
  - WRITE

    Data can be output to external sequential or direct-access records, or to internal records.
  - PRINT and TYPE

    The PRINT statement is the same as a formatted, sequential WRITE statement, except that the PRINT statement must never transfer data to user-specified I/O units.

    TYPE is a synonym for PRINT. All forms and rules for the PRINT statement also apply to the TYPE statement.
  - REWRITE

    It rewrites the current record and it can be formatted or unformatted.

File connection, file inquiry, and file positioning I/O statements are discussed in File Operation I O Statements.

Records and Files

A record is a sequence of values or a sequence of characters. There are three kinds of Fortran records, as follows:

- Formatted

  A record containing formatted data that requires translation from internal to external form. Formatted I/O statements have explicit format specifiers (which can specify list-directed formatting) or namelist specifiers (for namelist formatting). Only formatted I/O statements can read formatted data.
- Unformatted

  A record containing unformatted data that is not translated from internal form. An unformatted record can also contain no data. The internal representation of unformatted data is processor- dependent. Only unformatted I/O statements can read unformatted data.
- Endfile

  The last record of a file. An endfile record can be explicitly written to a sequential file by an ENDFILE statement.

A file is a sequence of records. There are two types of Fortran files, as follows:

- External

  A file that exists in a medium (such as computer disks or terminals) external to the executable program.

  Records in an external file must be either all formatted or all unformatted. There are two ways to access records in external files: sequential and direct access.

  In sequential access, records are processed in the order in which they appear in the file. In direct access, records are selected by record number, so they can be processed in any order.
- Internal

  Memory (internal storage) that behaves like a file. This type of file provides a way to transfer and convert data in memory from one format to another. The contents of these files are stored as scalar character variables.

See Also

Unit Specifier (UNIT=)

Components of Data Transfer Statements

Data transfer statements take one of the following forms:

io-keyword (*io-control-list*) [*io-list*]

io-keywordformat [, *io-list*]

io-keyword

Is one of the following: ACCEPT, PRINT (or TYPE), READ, REWRITE, or WRITE.

io-control-list

Is one or more of the following input/output (I/O) control specifiers:

[UNIT=]io-unit	ASYNCHRONOUS	IOMSG	SIZE
[FMT=]format	END	IOSTAT	
[NML=]group	EOR	POS	
ADVANCE	ERR	REC	

io-list

Is an I/O list, which can contain variables (except for assumed-size arrays) or implied-DO lists. Output statements can contain constants or expressions.

format

Is the nonkeyword form of a control-list format specifier (no FMT=).

If a format specifier ([FMT=]format) or namelist specifier ([NML=]group) is present, the data transfer statement is called a formatted I/O statement; otherwise, it is an unformatted I/O statement.

If a record specifier (REC=) is present, the data transfer statement is a direct-access I/O statement; otherwise, it is a sequential-access I/O statement.

If an error, end-of-record, or end-of-file condition occurs during data transfer, file positioning and execution are affected, and certain control-list specifiers (if present) become defined. (For more information, see Branch Specifiers.)

Following sections describe the I/O control list and I/O lists.

I/O Control List

The I/O control list specifies one or more of the following:

- The I/O unit to act upon ([UNIT=]io-unit)

  This specifier must be present; the rest are optional.
- The format (explicit or list-directed) to use for data editing; if explicit, the keyword form must appear ([FMT=]
- The namelist group name to act upon ([NML=]group)
- The number of a record to access (REC=)
- The name of a variable that contains the completion status of an I/O operation (IOSTAT=)
- The label of the statement that receives control if an error (ERR=), end-of-file (END=), or end-of-record (EOR=) condition occurs
- Whether you want to use advancing or nonadvancing I/O (ADVANCE=)
- The number of characters read from a record (SIZE=)
- Whether you want to use asynchronous or synchronous I/O (ASYNCHRONOUS=)
- The identifier for a pending data transfer operation (ID=)
- The identifier for the file position in file storage units in a stream file (POS=)
- The name of a variable that contains an error message (IOMSG=)

No control specifier can appear more than once, and the list must not contain both a format specifier and namelist group name specifier.

Control specifiers can take any of the following forms:

- Keyword form

  When the keyword form (for example, UNIT=io-unit) is used for all control-list specifiers in an I/O statement, the specifiers can appear in any order.

- Nonkeyword form

  When the nonkeyword form (for example, io-unit) is used for all control-list specifiers in an I/O statement, the io-unit specifier must be the first item in the control list. If a format specifier or namelist group name specifier is used, it must immediately follow the io-unit specifier.
- Mixed form

  When a mix of keyword and nonkeyword forms is used for control-list specifiers in an I/O statement, the nonkeyword values must appear first. Once a keyword form of a specifier is used, all specifiers to the right must also be keyword forms.

Unit Specifier (UNIT=)

The unit specifier identifies the I/O unit to be accessed. It takes the following form:

[UNIT=]*io-unit*

io-unit

For external files, it identifies a logical unit and is one of the following:

- A scalar integer expression that refers to a specific file, I/O device, or pipe. If necessary, the value is converted to integer data type before use. The integer is in the range 0 through 2,147,483,643, equal to the value of one of the constants INPUT_UNIT, OUTPUT_UNIT or ERROR_UNIT from intrinsic module ISO_FORTRAN_ENV, or a value returned by a NEWUNIT= specifier from an OPEN statement.

  Units 5, 6, and 0 are associated with preconnected units.
- An asterisk (*). This is the default (or implicit) external unit, which is preconnected for formatted sequential access. You can also preconnect files by using an environment variable.

For internal files, it identifies a scalar or array character variable that is an internal file. An internal file is designated internal storage space (a variable buffer) that is used with formatted (including list-directed) sequential READ and WRITE statements.

The *io-unit* must be specified in a control list. If the keyword UNIT is omitted, the *io-unit* must be first in the control list.

A unit number is assigned either explicitly through an OPEN statement or implicitly by the system. If a READ statement implicitly opens a file, the file's status is STATUS='OLD'. If a WRITE statement implicitly opens a file, the file's status is STATUS='UNKNOWN'.

If the internal file is a *scalar* character variable, the file has only one record; its length is equal to that of the variable.

If the internal file is an *array* character variable, the file has a record for each element in the array; each record's length is equal to one array element.

An internal file can be read only if the variable has been defined and a value assigned to each record in the file. If the variable representing the internal file is a pointer, it must be associated; if the variable is an allocatable array, it must be currently allocated.

Before data transfer, an internal file is always positioned at the beginning of the first character of the first record.

See Also

OPEN statement

Format Specifier (FMT=)

The format specifier indicates the format to use for data editing. It takes the following form:

[FMT=]*format*

format

Is one of the following:

- The statement label of a FORMAT statement

  The FORMAT statement must be in the same scoping unit as the data transfer statement.
- An asterisk (*), indicating list-directed formatting
- A scalar default integer variable that has been assigned the label of a FORMAT statement (through an ASSIGN statement)

  The FORMAT statement must be in the same scoping unit as the data transfer statement.
- A character expression (which can be an array or character constant) containing the run-time format

  A default character expression must evaluate to a valid format specification. If the expression is an array, it is treated as if all the elements of the array were specified in array element order and were concatenated.
- The name of a numeric array (or array element) containing the format

If the keyword FMT is omitted, the format specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a format specifier appears in a control list, a namelist group specifier must not appear.

See Also

FORMAT statement
Interaction between FORMAT statements and I/O lists
Rules for List-Directed Sequential READ Statements for details on list-directed input
Rules for List-Directed Sequential WRITE Statements for details on list-directed output

Namelist Specifier (NML=)

The namelist specifier indicates namelist formatting and identifies the namelist group for data transfer. It takes the following form:

[NML=]*group*

group

Is the name of a namelist group previously declared in a NAMELIST statement.

If the keyword NML is omitted, the namelist specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a namelist specifier appears in a control list, a format specifier must *not* appear.

See Also

Rules for Namelist Sequential READ Statements for details on namelist input
Rules for Namelist Sequential WRITE Statements for details on namelist output
READ
WRITE

Record Specifier (REC=)

The record specifier identifies the number of the record for data transfer in a file connected for direct access. It takes the following form:

REC=*r*

r — Is a scalar numeric expression indicating the record number. The value of the expression must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file.

If necessary, the value is converted to integer data type before use.

If REC is present, no END specifier, * format specifier, or namelist group name can appear in the same control list.

See Also

Alternative Syntax for a Record Specifier

I/O Status Specifier (IOSTAT=)

The I/O status specifier designates a variable to store a value indicating the status of a data transfer operation. It takes the following form:

IOSTAT=*i-var*

i-var — Is a scalar integer variable. When a data transfer statement is executed, *i-var* is set to one of the following values:

A positive integer	Indicating an error condition occurred.
A negative integer	Indicating an end-of-file or end-of-record condition occurred. The negative integers differ depending on which condition occurred.
Zero	Indicating no error, end-of-file, or end-of-record condition occurred.

Execution continues with the statement following the data transfer statement, or the statement identified by a branch specifier (if any).

An end-of-file condition occurs only during execution of a sequential READ statement; an end-of-record condition occurs only during execution of a nonadvancing READ statement.

See Also

List of Run-Time Error Messages
CLOSE
READ
WRITE

Branch Specifiers (END=, EOR=, ERR=)

A branch specifier identifies a branch target statement that receives control if an error, end-of-file, or end-of-record condition occurs. There are three branch specifiers, taking the following forms:

ERR=*label*

END=*label*

EOR=*label*

label — Is the label of the branch target statement that receives control when the specified condition occurs.

The branch target statement must be in the same scoping unit as the data transfer statement.

The following rules apply to these specifiers:

- ERR

  The error specifier can appear in a sequential access READ or WRITE statement, a direct-access READ statement, or a REWRITE statement.

  If an error condition occurs, the position of the file is indeterminate, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a positive integer value. If SIZE was specified (in a nonadvancing READ statement), the SIZE variable becomes defined as an integer value. If a *label* was specified, execution continues with the labeled statement.
- END

  The end-of-file specifier can appear only in a sequential access READ statement.

  An end-of-file condition occurs when no more records exist in a file during a sequential read, or when an end-of-file record produced by the ENDFILE statement is encountered. End-of-file conditions do not occur in direct-access READ statements.

  If an end-of-file condition occurs, the file is positioned after the end-of-file record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If a *label* was specified, execution continues with the labeled statement.
- EOR

  The end-of-record specifier can appear only in a formatted, sequential access READ statement that has the specifier ADVANCE='NO'(nonadvancing input).

  An end-of-record condition occurs when a nonadvancing READ statement tries to transfer data from a position after the end of a record.

  If an end-of-record condition occurs, the file is positioned after the current record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If PAD='YES' was specified for file connection, the record is padded with blanks (as necessary) to satisfy the input item list and the corresponding data edit descriptor. If SIZE was specified, the SIZE variable becomes defined as an integer value. If a *label* was specified, execution continues with the labeled statement.

If one of the conditions occurs, no branch specifier appears in the control list, but an IOSTAT specifier appears, execution continues with the statement following the I/O statement. If neither a branch specifier nor an IOSTAT specifier appears, the program terminates.

See Also

I/O Status Specifier
Branch Statements

Compiler Reference section: *Error Handling* for details on error processing

Advance Specifier (ADVANCE=)

The advance specifier determines whether nonadvancing I/O occurs for a data transfer statement. It takes the following form:

ADVANCE=*c-expr*

c-expr	Is a scalar character expression that evaluates to 'YES' for advancing I/O or 'NO' for nonadvancing I/O. The default value is 'YES'. Trailing blanks in the expression are ignored. The values specified are without regard to case.

The ADVANCE specifier can appear only in a formatted, sequential data transfer statement that specifies an external unit. It must not be specified for list-directed or namelist data transfer, for a data transfer statement within a DO CONCURRENT block, nor for a data transfer statement within a DO CONCURRENT block.

Advancing I/O always positions a file at the end of a record, unless an error condition occurs. Nonadvancing I/O can position a file at a character position within the current record.

See Also

Compiler Reference: Data and I/O: Fortran I/O: Advancing and Nonadvancing Record I/O

Asynchronous Specifier (ASYNCHRONOUS=)

The asynchronous specifier determines whether asynchronous I/O occurs for a data transfer statement. It takes the following form:

ASYNCHRONOUS=*i-expr*

i-expr
: Is a scalar character constant expression that evaluates to 'YES' for asynchronous I/O or 'NO' for synchronous I/O. The value 'YES' should not appear unless the data transfer statement specifies a file unit number for *io-unit*. The default value is 'NO'.

  Trailing blanks in the expression are ignored. The values specified are without regard to case.

Asynchronous I/O is permitted only for external files opened with an OPEN statement that specifies ASYNCHRONOUS='YES'.

When an asynchronous I/O statement is executed, the pending I/O storage sequence for the data transfer operation is defined to be:

- The set of storage units specified by the I/O item list or by the NML= specifier
- The storage units specified by the SIZE= specifier

Character Count Specifier (SIZE=)

The character count specifier defines a variable to contain the count of the characters transferred by data edit descriptors during execution of the current input statement. It takes the following form:

SIZE=*i-var*

i-var
: Is a scalar integer variable.

If PAD='YES' was specified for file connection, blanks inserted as padding are not counted.

For input statements, the SIZE= specifier can appear only in a formatted, sequential READ statement that has the specifier ADVANCE='NO' (nonadvancing input). It must not be specified for list-directed or namelist data transfer.

For asynchronous nonadvancing input, the storage units specified in the SIZE= specifier become defined with the count of the characters transferred when the corresponding wait operation is executed.

ID Specifier (ID=)

The ID specifier identifies a pending data transfer operation for a specified unit. It takes the following form:

ID=*id-var*

id-var
: Is a scalar integer variable to be used as an identifier.

  This specifier can only be used if the value of ASYNCHRONOUS=*i-expr* is 'YES'.

If an ID specifier is used in a data transfer statement, a wait operation is performed for the operation. If it is omitted, wait operations are performed for all pending data transfers for the specified unit.

If an error occurs during the execution of a data transfer statement containing an ID specifier, the variable specified becomes undefined.

In an INQUIRE statement, the ID= specifier identifies a pending asynchronous data transfer. It is is used with the PENDING specifier to determine whether a specific asynchronous pending data transfer is completed.

POS Specifier (POS=)

The POS specifier identifies the file position in file storage units in a stream file (ACCESS='STREAM'). It takes the following form:

POS=*p*

p	Is a scalar integer expression that specifies the file position. It can only be specified on a file opened for stream access. If omitted, the stream I/O occurs starting at the next file position after the current file position.

Each file storage unit has a unique file position, represented by a positive integer. The first file storage unit is a file is at file position 1. The position of each subsequent file storage unit is one greater than that of its preceding file storage unit.

For a formatted file, the file storage unit is an eight-bit byte. For an unformatted file, the file storage unit is an eight-bit byte (if option assume byterecl is specified) or a 32-bit word (if option assume nobyterecl, the default, is specified).

I/O Message Specifier (IOMSG=)

The I/O message specifier designates a variable to contain the message to be returned when an I/O error occurs. It takes the following form:

IOMSG=*msg-var*

msg-var	Is a scalar default character variable.

If an error (ERR=), end-of-file (END=), or end-of-record (EOR=) condition occurs during execution of an I/O statement, *msg-var* is assigned an explanatory message.

If no error occurs, the value of the variable remains unchanged.

I/O Lists

In a data transfer statement, the I/O list specifies the entities whose values will be transferred. An input list is made up of implied-do lists and simple lists of variables (except for assumed-size arrays). An output list is made up of implied-do lists, expressions, and simple lists of variables (except for assumed-size arrays).

In input statements, the I/O list cannot contain constants and expressions because these do not specify named memory locations that can be referenced later in the program.

However, constants and expressions can appear in the I/O lists for output statements because the compiler can use temporary memory locations to hold these values during the execution of the I/O statement.

If an input item is a pointer, it must be currently associated with a definable target; data is transferred from the file to the associated target. If an output item is a pointer, it must be currently associated with a target; data is transferred from the target to the file.

If an input or output item is an array, it is treated as if the elements (if any) were specified in array element order. For example, if ARRAY_A is an array of shape (2,1), the following input statements are equivalent:

```
READ *, ARRAY_A
READ *, ARRAY_A(1,1), ARRAY_A(2,1)
```

However, no element of that array can affect the value of any expression in the input list, nor can any element appear more than once in an input list. For example, the following input statements are invalid:

```
INTEGER  B(50)
...
READ *, B(B)
READ *, B(B(1):B(10))
```

If an input or output item is an allocatable array, it must be currently allocated.

If an input or output item is a derived type, the following rules apply:

- Any derived-type component must be in the scoping unit containing the I/O statement.
- The derived type must not have a pointer component.
- In a formatted I/O statement, a derived type is treated as if all of the components of the structure were specified in the same order as in the derived-type definition.
- In an unformatted I/O statement, a derived type is treated as a single object.

See Also

Simple List Items in I/O Lists
Implied-DO Lists in I/O Lists

Simple List Items in I/O Lists

In a data transfer statement, a simple list of items takes the following form:

item [, *item*] ...

item	Is one of the following: • For input statements: a variable name The variable must not be an assumed-size array, unless one of the following appears in the last dimension: a subscript, a vector subscript, or a section subscript specifying an upper bound. • For output statements: a variable name, expression, or constant Any expression must not attempt further I/O operations on the same logical unit. For example, it must not refer to a function subprogram that performs I/O on the same logical unit.

The data transfer statement assigns values to (or transfers values from) the list items in the order in which the items appear, from left to right.

When multiple array names are used in the I/O list of an unformatted input or output statement, only one record is read or written, regardless of how many array name references appear in the list.

Examples

The following example shows a simple I/O list:

```
WRITE (6,10) J, K(3), 4, (L+4)/2, N
```

When you use an array name reference in an I/O list, an input statement reads enough data to fill every item of the array. An output statement writes all of the values in the array.

Data transfer begins with the initial item of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. The following statement defines a two-dimensional array:

```
DIMENSION ARRAY(3,3)
```

If the name ARRAY appears with no subscripts in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

An input record contains the following values:

```
1,3,721.73
```

The following example shows how variables in the I/O list can be used in array subscripts later in the list:

```
DIMENSION ARRAY(3,3)
...
READ (1,30) J, K, ARRAY(J,K)
```

When the READ statement is executed, the first input value is assigned to J and the second to K, establishing the subscript values for ARRAY(J,K). The value 721.73 is then assigned to ARRAY(1,3). Note that the variables must appear before their use as array subscripts.

Consider the following derived-type definition and structure declaration:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
...
TYPE(EMPLOYEE) :: CONTRACT   ! A structure of type EMPLOYEE
```

The following statements are equivalent:

```
READ *, CONTRACT

READ *, CONTRACT%ID, CONTRACT%NAME
```

The following shows more examples:

```
!  A variable and array element in iolist:
      REAL b(99)
      READ (*, 300) n, b(n) ! n and b(n) are the iolist
300   FORMAT (I2, F10.5) ! FORMAT statement telling what form the input data has

!  A derived type and type element in iolist:
      TYPE YOUR_DATA
        REAL a
        CHARACTER(30) info
        COMPLEX cx
      END TYPE YOUR_DATA
      TYPE (YOUR_DATA) yd1, yd2
      yd1.a = 2.3
      yd1.info = "This is a type demo."
      yd1.cx = (3.0, 4.0)
      yd2.cx = (4.5, 6.7)
!  The iolist follows the WRITE (*,500).
      WRITE (*, 500) yd1, yd2.cx
!  The format statement tells how the iolist will be output.
500   FORMAT (F5.3, A21, F5.2, ',', F5.2, ' yd2.cx = (', F5.2,
       ',',F5.2, ' )')
!  The output looks like:
!  2.300This is a type demo 3.00, 4.00 yd2.cx = ( 4.50, 6.70 )
```

The following example uses an array and an array section:

```
!  An array in the iolist:
      INTEGER handle(5)
      DATA handle / 5*0 /
      WRITE (*, 99) handle
99    FORMAT (5I5)
!  An array section in the iolist.
      WRITE (*, 100) handle(2:3)
100   FORMAT (2I5)
```

The following shows another example:

```
PRINT *,'(I5)', 2*3   ! The iolist is the expression 2*3.
```

The following example uses a namelist:

```
! Namelist I/O:
     INTEGER int1
     LOGICAL log1
     REAL r1
     CHARACTER (20) char20
     NAMELIST /mylist/ int1, log1, r1, char20
     int1 = 1
     log1 = .TRUE.
     r1 = 1.0
     char20 = 'NAMELIST demo'
     OPEN (UNIT = 4, FILE = 'MYFILE.DAT', DELIM = 'APOSTROPHE')
     WRITE (UNIT = 4, NML = mylist)
! Writes the following:
! &MYLIST
! INT1 = 1,
! LOG1 = T,
! R1 = 1.000000,
! CHAR20 = 'NAMELIST demo '
! /
     REWIND(4)
     READ (4, mylist)
```

See Also

I/O Lists for details on the general rules for I/O lists

Implied-DO Lists in I/O Lists

In a data transfer statement, an implied-DO list acts as though it were a part of an I/O statement within a DO loop. It takes the following form:

(*list*, *do-var* = *expr* 1, *expr* 2 [, *expr* 3])

list	Is a list of variables, expressions, or constants (see Simple List Items in I/O Lists).
do-var	Is the name of a scalar integer or real variable. The variable must not be one of the input items in *list*.
expr	Are scalar numeric expressions of type integer or real. They do not all have to be the same type, or the same type as the DO variable.

The implied-DO loop is initiated, executed, and terminated in the same way as a DO construct.

The *list* is the range of the implied-DO loop. Items in that list can refer to *do-var*, but they must not change the value of *do-var*.

Two nested implied-DO lists must not have the same (or an associated) DO variable.

Use an implied-DO list to do the following:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array items in a sequence different from the order of subscript progression

If the I/O statement containing an implied-DO list terminates abnormally (with an END, EOR, or ERR branch or with an IOSTAT value other than zero), the DO variable becomes undefined.

Examples

The following two output statements are equivalent:

```
WRITE (3,200) (A,B,C, I=1,3)            ! An implied-DO list

WRITE (3,200) A,B,C,A,B,C,A,B,C         ! A simple item list
```

The following example shows nested implied-DO lists. Execution of the innermost list is repeated most often:

```
WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
```

The inner DO loop is executed 10 times for each iteration of the outer loop; the second subscript (L) advances from 1 through 10 for each increment of the first subscript (K). This is the reverse of the normal array element order. Note that K is incremented by 2, so only the odd-numbered rows of the array are output.

In the following example, the entire list of the implied-DO list (P(1), Q(1,1), Q(1,2)...,Q(1,10)) are read before I is incremented to 2:

```
READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

The following example uses fixed subscripts and subscripts that vary according to the implied-DO list:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

Input values are assigned to BOX(1,1) through BOX(1,10), but other elements of the array are not affected.

The following example shows how a DO variable can be output directly:

```
WRITE (6,1111) (I, I=1,20)
```

Integers 1 through 20 are written.

Consider the following:

```
      INTEGER mydata(25)
      READ (10, 9000) (mydata(I), I=6,10,1)
9000  FORMAT (5I3)
```

In this example, the *iolist* specifies to put the input data into elements 6 through 10 of the array called mydata. The third value in the implied-DO loop, the increment, is optional. If you leave it out, the increment value defaults to 1.

See Also

Execution Control

I/O Lists for details on the general rules for I/O lists

Forms for READ Statements

This section discusses the various forms you can specify for READ statements.

Forms for Sequential READ Statements

Sequential READ statements transfer input data from external sequential-access records. The statements can be formatted with format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

A sequential READ statement takes one of the following forms:

Formatted:

READ (*eunit*, *format* [, *advance*] [, *asynchronous*] [, *blank*] [, *decimal*] [, *id*] [, *pad*] [, *pos*] [, *round*] [, *size*] [, *iostat*] [, *err*] [, *end*] [, *eor*] [, *iomsg*]) [*io-list*]

READ *form* [, *io-list*]

Formatted - List-Directed:

READ (*eunit*, * [, *asynchronous*] [, *blank*] [, *decimal*] [, *id*] [, *pad*] [, *pos*] [, *round*] [, *iostat*] [, *err*] [, *end*] [, *iomsg*]) [*io-list*]

READ * [, *io-list*]

Formatted - Namelist:

READ (*eunit*, *nml-group* [, *asynchronous*] [, *blank*] [, *decimal*] [, *id*] [, *pad*] [, *pos*] [, *round*] [, *iostat*] [, *err*] [, *end*] [, *iomsg*])

READ *nml*

Unformatted:

READ (*eunit* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *end*] [, *iomsg*]) [*io-list*]

See Also

READ
I/O control-list specifiers
I/O lists

Rules for Formatted Sequential READ Statements

Formatted, sequential READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

The following example shows formatted, sequential READ statements:

```
READ (*, '(B)', ADVANCE='NO') C

READ (FMT="(E2.4)", UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

See Also

READ statement
Forms for Sequential READ Statements

Rules for List-Directed Sequential READ Statements

List-directed, sequential READ statements translate data from character to binary form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then assigned to the entities in the I/O list in the order in which they appear, from left to right.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

List-Directed Records

A list-directed external record consists of a sequence of values and value separators. A value can be any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  - A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see the table in Numeric Assignment Statements). Conversion is not performed between numeric and logical types unless compiler option `assume old_logical_ldio` is in effect. The decimal point in a numeric constant can either be a period if DECIMAL='POINT' or a comma if DECIMAL='COMMA'.
  - A complex constant has the form of a pair of real or integer constants separated by a comma if DECIMAL='POINT' or a semicolon if DECIMAL='COMMA' and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma or semicolon, and between the second constant and the closing parenthesis.
  - A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

  A character string does not need delimiting apostrophes or quotation marks if the corresponding I/O list item is of type default character, and the following is true:

  - The character string does not contain a blank, comma (,), or slash (/).
  - The character string is not continued across a record boundary.
  - The first nonblank character in the string is not an apostrophe or a quotation mark.
  - The leading character is not a string of digits followed by an asterisk.

  A nondelimited character string is terminated by the first blank, comma, slash, or end-of-record encountered. Apostrophes and quotation marks within nondelimited character strings are transferred as is.
- A null value

  A null value is specified by two consecutive value separators (such as,,) or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value.)

  A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.
- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

A value separator is any number of blanks, a slash, or a comma if DECIMAL='POINT' or a semicolon if DECIMAL='COMMA', preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the character constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

Examples

Suppose the following statements are specified:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
```

Then suppose the following external record is read:

```
4 6.3 (3.4,4.2), (3, 2 ), T,F,,3*14.6,'ABC,DEF/GHI''JK'/
```

The following values are assigned to the I/O list items when DECIMAL='POINT':

I/O List Item	Value Assigned
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
J	Unchanged
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI' JK
A	Unchanged
B	Unchanged

With DECIMAL='COMMA', the following external record produces the same values as in the table above:

```
4 6,3 (3,4;4,2); (3; 2 ); T;F;;3*14,6,'ABC,DEF/GHI''JK'/
```

The following example shows list-directed input and output:

```
REAL    a
INTEGER i
COMPLEX c
LOGICAL up, down
DATA a /2358.2E-8/, i /91585/, c /(705.60,819.60)/
DATA up /.TRUE./, down /.FALSE./
OPEN (UNIT = 9, FILE = 'listout', STATUS = 'NEW')
WRITE  (9, *) a, i
WRITE  (9, *) c, up, down
REWIND (9)
READ   (9, *) a, i
READ   (9, *) c, up, down
WRITE  (*, *) a, i
WRITE  (*, *) c, up, down
END
```

The preceding program produces the following output:

```
  2.3582001E-05     91585
 (705.6000,819.6000) T F
```

See Also

READ

Forms for Sequential READ Statements

Intrinsic Data Types for details on the literal constant forms of intrinsic data types

Rules for List-Directed Sequential WRITE Statements for details on list-directed output

Rules for Namelist Sequential READ Statements

Namelist, sequential READ statements translate data from external to internal form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is assigned to objects in the namelist group by specifying the name of the object to which the data is to be assigned.

The order of the object name and data pairs in the input records need not match the order of the objects in the namelist *var-list*. The input need not specify all objects in the namelist *var-list*. They may specify a part of an object more than once. Namelist group names and object names are case insensitive.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged. An ampersand (&) encountered during input is treated the same as a slash (/).

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Namelist Records

A namelist external record takes the following form:

&*group-nameobject* = *value* [{, | ;} *object* = *value*] .../

group-name	Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit. The name cannot contain embedded blanks and must be contained within a single record.

object

Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks except within the parentheses of a subscript or substring specifier. Each object must be contained in a single record.

value

Is any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a delimited or nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  - A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see the table in Numeric Assignment Statements). Logical list items and logical constants are not considered numeric, unless the compiler option `assume old_logical_ldio` is specified. The decimal point in a numeric constant can either be a period if DECIMAL='POINT' or a comma if DECIMAL='COMMA'.
  - A complex constant has the form of a pair of real or integer constants separated by a comma if DECIMAL='POINT' or a semicolon if DECIMAL='COMMA' and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma or semicolon, and between the second constant and the closing parenthesis.
  - A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

  Normally, a character string in a NAMELIST statement must be delimited to be read. A delimited string is denoted by apostrophes (DELIM=APOSTROPHE), or quotes (DELIM=QUOTE).

  Intel® Fortran also allows setting DELIM=NONE, which is the default for both input and output. In this case, non-delimited strings are allowed under certain circumstances.

  A character string does not need delimiting apostrophes or quotation marks if the corresponding NAMELIST item is of type default character and it complies with the following rules:

  - The character string cannot contain a blank, tab, equal sign (=), dollar sign ($), slash (/), new line, ampersand (&), or exclamation point(!). A nondelimited character string is terminated if one of these is encountered.

  - Normally, a comma will also cause termination of the nondelimited character string. However, if DECIMAL=COMMA is specified, then the character string cannot contain semicolons but it can contain commas. In this case, if a semicolon is encountered the nondelimited character string is terminated.
- A null value

  A null value is specified by two consecutive value separators (such as two adjacent commas ",,") or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value.)

  A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.
- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

Blanks can precede or follow the beginning ampersand (&), follow the group name, precede or follow the equal sign, or precede the terminating slash.

Comments (beginning with ! only) can appear anywhere in namelist input. The comment extends to the end of the source line.

If an entity appears more than once within the input record for a namelist data transfer, the last value is the one that is used.

If there is more than one *object = value* pair, they must be separated by value separators.

A value separator is any number of blanks, a slash, or a comma if DECIMAL='POINT' or a semicolon if DECIMAL='COMMA', preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the character constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

When the name in the input record is an array variable or a variable of derived type, the effect is as if the variable represented were expanded into a sequence of scalar list items of intrinsic data types. Each input *value* following the equal sign must comply with format specifications for the intrinsic type of the list item in the corresponding position in the expanded sequence.

The number of values following the equal sign must be equal to or less than the number of list items in the expanded sequence. In the latter case (less than), the effect is as if null values have been added to match any remaining list items in the expanded sequence.

The string length in the NAMELIST statement is not checked against the size of the CHARACTER variable to which it will be assigned. This means that an array of *n* elements written in a NAMELIST statement with DELIM=NONE may not be read back as *n* values. For example, consider a three-element array ARR of three-character elements with values "ABC","DEF","GHI". In DELIM=NONE form, it prints to the data file as follows:

```
ARR = ABCDEFGHI
```

If your program reads that data file, the value will be interpreted as one non-delimited string with the value "ABCDEFGHI" because termination is caused by the trailing blank, tab, or new line.

In some cases, values can be read as more than one string; for example, if the values themselves have final or internal blanks, tabs, or new lines. Consider that ARR contains strings "WX ","Y\tR", "S\nQ", where "\t" is a tab character and "\n" is a newline character. In this case, the data file contains:

```
ARR = WX Y\tRS\nQ
```

The NAMELIST processing will interpret this as four non-delimited strings: "WX", "Y", "RS" and "Q".

NOTE
In NAMELIST declarations, you may get unexpected results if all of the following are true:

1. DELIM=NONE is in effect on input.
2. A character variable is followed by another variable.
3. The other variable is either an array variable that is subscripted or a string variable that is a substring.
4. That subscript or substring expression contains blanks.

In Intel® Fortran, a list of values may follow the equal sign when the object is a single array element. In this case, values are assigned to the specified array element and subsequent elements, in element sequence order. For example, suppose the following input is read:

```
&ELEM
ARRAY_A(3)=34.54, 45.34, 87.63, 3*20.00
/
```

New values are assigned only to array `ARRAY_A` elements 3 through 8. The other element values are unchanged.

Prompting for Namelist Group Information

During execution of a program containing a namelist READ statement, you can specify a question mark character (?) or a question mark character preceded by an equal sign (=?) to get information about the namelist group. The ? or =? must follow one or more blanks.

If specified for a unit capable of both input and output, the ? causes display of the group name and the objects in that group. The =? causes display of the group name, objects within that group, and the current values for those objects (in namelist output form). If specified for another type of unit, the symbols are ignored.

For example, consider the following statements:

```
NAMELIST /NLIST/ A,B,C
REAL A /1.5/
INTEGER B /2/
CHARACTER*5 C /'ABCDE'/
READ (5,NML=NLIST)
WRITE (6,NML=NLIST)
END
```

During execution, if a blank followed by ? is entered on a terminal device, the following values are displayed:

```
&NLIST
   A
   B
   C
/
```

If a blank followed by =? is entered, the following values are displayed:

```
&NLIST
    A  =   1.500000,
    B  =           2,
    C  = ABCDE
/
```

Examples

Suppose the following statements are specified:

```
NAMELIST /CONTROL/ TITLE, RESET, START, STOP, INTERVAL
CHARACTER*10 TITLE
REAL(KIND=8) START, STOP
LOGICAL(KIND=4) RESET
INTEGER(KIND=4) INTERVAL
READ (UNIT=1, NML=CONTROL)
```

The NAMELIST statement associates the group name CONTROL with a list of five objects. The corresponding READ statement reads the following input data from unit 1:

```
 &CONTROL
       TITLE='TESTT002AA',
       INTERVAL=1,
       RESET=.TRUE.,
       START=10.2,
       STOP =14.5
/
```

The following values are assigned to objects in group CONTROL:

Namelist Object	Value Assigned
TITLE	TESTT002AA
RESET	T
START	10.2
STOP	14.5
INTERVAL	1

It is not necessary to assign values to all of the objects declared in the corresponding NAMELIST group. If a namelist object does not appear in the input statement, its value (if any) is unchanged.

Similarly, when character substrings and array elements are specified, only the values of the specified variable substrings and array elements are changed. For example, suppose the following input is read:

```
&CONTROL TITLE(9:10)='BB' /
```

The new value for TITLE is TESTT002BB; only the last two characters in the variable change.

The following example shows an array as an object:

```
DIMENSION ARRAY_A(20)
NAMELIST /ELEM/ ARRAY_A
READ (UNIT=1,NML=ELEM)
```

Suppose the following input is read:

```
&ELEM
ARRAY_A=1.1, 1.2,, 1.4
/
```

The following values are assigned to the ARRAY_A elements:

Array Element	Value Assigned
ARRAY_A(1)	1.1
ARRAY_A(2)	1.2
ARRAY_A(3)	Unchanged
ARRAY_A(4)	1.4
ARRAY_A(5)...ARRAY(20)	Unchanged

Nondelimited character strings that are written out by using a NAMELIST write may not be read in as expected by a corresponding NAMELIST read. Consider the following:

```
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/'AAA', 'BBB', 'CCC', 'DDD'/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
WRITE (1, NML=TEST)
END
```

The output file NMLTEST.DAT will contain:

```
&TEST CHARR   = AAABBBCCCDDD/
```

If an attempt is then made to read the data in NMLTEST.DAT with a NAMELIST read using nondelimited character strings, as follows:

```
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/4*'   '/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
READ (1, NML=TEST)
PRINT *, 'CHARR read in >', CHARR(1),'<  >',CHARR(2),'<  >',
1       CHARR(3), '<  >', CHARR(4), '<'
END
```

The result is the following:

```
CHARR read in >AAA<  >   <  >   <  >   <
```

The 12 characters of input were read, truncated to 3 characters, and stored in CHARR(1). The other 3 elements of CHARR were unchanged.

See Also

NAMELIST
Alternative Form for Namelist External Records
Rules for Formatted Sequential READ Statements
Rules for Namelist Sequential WRITE Statements

Rules for Unformatted Sequential READ Statements

Unformatted, sequential READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. The value transferred from the file is called a field. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, sequential READ statement reads a single record. Each field value in the record must be of the same type as the corresponding entity in the input list, unless the field value is real or complex.

If the field value is real or complex, one complex field value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is *less than or equal to* the number of fields in an input record, the READ statement ignores the excess fields. If the number of I/O list items is *greater than* the number of fields in an input record, an error occurs.

If a READ statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

You have previously been able to buffer the output (WRITEs) of variable length, unformatted, sequential files, by specifying certain values for an OPEN statement, environment variable, or compiler option. You can now do the same buffering for input (READs) of records. To enable buffering for the input of records, you can specify any of the following:

- BUFFERED=YES in the file's OPEN statement
- Value YES (Y or y), or TRUE (T or t), or a number > 0 for the environment variable FORT_BUFFERED
- Setting `buffered_io` for the `assume` option

When any of the above are specified, the Fortran Runtime Library buffers all input records from variable length, unformatted, sequential files, regardless of the size of the records in the file. In addition, if the environment variable FORT_ BUFFERING_THRESHOLD has a positive value *n*, the following occurs:

- I/O list items with a size <= *n* are buffered and are moved one at a time from the runtime buffer to the I/O list item
- I/O list items with a size > *n* are not buffered and are moved one at a time from the file to the I/O list item

Determining the Size of I/O Buffers

If both the block size and the buffer count have been specified with positive values, their product determines the size in bytes of the buffer for that I/O unit. If neither is specified, the default size of the buffer is 8KB (8192 bytes). This is the initial size of the I/O buffer; the buffer may be expanded to hold a larger record.

If block size is not specified, the following occurs:

- The block size defaults to 128KB
- If the buffer count is not specified, it defaults to 1
- The initial default buffer size is 8KB
- If buffer count is specified, then the initial default buffer size is (8KB * buffercount)

If a block size is specified, the following occurs:

- The block size is the specified value, rounded up to 512-byte boundary
- If the buffer count is not specified, it defaults to 1
- The initial default buffer size is the block size, rounded up to 512-byte boundary
- If buffer count is specified, then the initial default buffer size is (block size * buffer count)

Optimizing for Time or Space

When reading variable length, unformatted sequential records, the runtime system may optimize for time or for space.

This optimization decision is made during runtime on a record-by-record basis using specifications made by the program and the length of a given record. That is, one record may be optimized for time while another record from the same file may be optimized for space.

The default behavior when reading records of this type whose length exceeds the specified block size, is to *not* buffer the input records. You can override this default behavior by requesting that the input be buffered.

The following table shows the relationship between a file's specified block size and the length of its variable length records. Note that the length of the individual record is the key:

	record length >= block size	record length < block size
buffering unspecified	Optimizes for space	Optimizes for time
OPEN (BUFFERED='YES')	Optimizes for time	Optimizes for time
OPEN (BUFFERED='NO')	Optimizes for space	Optimizes for time

If an input record's length is *less than or equal to* the specified block size, then by default, the runtime system always optimizes for time rather than for space and the input is buffered.

If an input record’s length is *greater than* the specified block size, then the following occurs:

- By default, the runtime system always optimizes for space rather than time and the input is not buffered.
- If you request buffering of input, then the runtime system optimizes for time and the input is buffered.
- If you request no buffering of input, then the runtime system optimizes for space rather than time and the input is not buffered.
- If you request dynamic buffering of input, the runtime system optimizes based on the size of the I/O list item and some items are buffered and some are not.

Optimizing for time:

Traditionally, optimizing for time comes at the expense of using more memory.

When the runtime system optimizes for time, it buffers input. It reads as much data as possible during one disk access into the runtime's internal buffer, extending it if necessary to hold the file's largest record. Fields within the record are then moved to the user space in response to READs from the file. Typically, minimizing file accesses is faster.

However, there are circumstances when optimizing for space can actually be faster than optimizing for time.

For example, consider you are reading records whose length exceeds the block size and the data is being read into a contiguous array. Reading this huge array directly into a user's space is going to be faster than reading it first into the runtime system's internal buffer, then moving the data to the user's space. In this case, it is better to optimize for space; that is, you should *not* buffer the input record.

On the other hand, if the READ is being done into non-contiguous elements of an array, the traditional method of optimizing for time becomes a huge win. Data being read into non-contiguous array elements must be moved, or read, into the user's space one element at a time. In this case, you always want to optimize for time; that is, you should buffer the input data.

If you are reading large, variable length, unformatted records, you should try both buffered and unbuffered I/O to determine which delivers the better performance.

Optimizing for space:

Traditionally, optimizing for space comes at the expense of time.

When the runtime system optimizes for space, it wants to avoid creating a huge internal buffer in order to hold a "very large" record. The size of a "very large" record is clearly subjective, but the rule of thumb here is whether or not a record's size is greater than the specified block size.

If this is the case, the runtime system will read one field at a time from the record, directly into the I/O list items. The optimal record for this optimization is one whose record length exceeds the default block size of 128 KB (or a user-specified block size) and contains "very large" fields.

Note that because fields are read one at a time from the file to the user space, very large records that contain very small fields may see a serious performance issue. In these cases, it may be better to buffer the input. If you are reading large, variable length, unformatted records, you should try both buffered and unbuffered I/O to determine which delivers the better performance.

Optimizing unformatted sequential input based on field size - dynamic buffering:

Dynamic buffering is a hybrid solution that chooses the time/space trade-off on a per field basis. The decision is based on a "field size threshold" supplied by the user, deciding, for every field in every record in the file, regardless of the record length, whether or not to buffer a field from the file to the I/O list item. The runtime system can do this because it knows the size of a field that is being requested before actually attempting to read the field.

When a READ statement is first executed, a read from the file is issued to fill the buffer, regardless of its size. This is necessary so that the runtime system can extract the record size from the first length control field. (Each unformatted sequential record has 4-byte leading and trailing record lengths to facilitate reading both forwards and backwards in the file.) From that point on, dynamic buffering decides whether or not to buffer a field or to read it directly from the file to the I/O list item. If the buffer holds the beginning portion of a large field, it will be moved to the start of the I/O list item and the remainder will be read directly from the file.

The following table shows the various buffering options for unformatted, sequential input:

Buffering Option	How do I get this?	What happens?

Non-buffered	You get this kind of buffering by default, or by specifying:
	• OPEN (BUFFERED=NO)
	- or -
	• FORT_BUFFERED=NO
	- or -
	• `assume nobuffered_io`
		When non-buffered is in effect:
		• The runtime system does not re-allocate its buffer to accommodate large records.
		• Records with lengths > the block size:
		  • Are not buffered.
		  • All fields are moved one at a time from the file to the I/O list item.
		• Records with lengths <= the block size:
		  • Are buffered.
		  • All fields are moved one at a time from the buffer to the I/O list item.

Buffered	You get this kind of buffering by specifying: • OPEN (BUFFERED=YES) - or - • FORT_BUFFERED=YES - or -	When buffered is in effect: • The runtime system re-allocates its buffer to accommodate the largest record read. • All input records are buffered.

Buffering Option	How do I get this?	What happens?
	• `assume buffered_io`	• All fields are moved one at a time from the buffer to the I/O list item
Dynamic buffering	You get this kind of buffering by specifying: • FORT_BUFFERING_THRESHOLD=*n* - and one of - • OPEN (BUFFERED=YES) - or - • FORT_BUFFERED=YES - or - • `assume buffered_io`	When dynamic buffering is in effect: • The runtime system does not re-allocate its buffer to accommodate large records. • All fields with a size <= *n* are buffered and are moved one at a time from the buffer to the I/O list item. • Fields with a size > *n* are not buffered and are moved one at a time from the file to the I/O list item.

Examples

The following example shows an unformatted, sequential READ statement:

```
READ (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

See Also

READ statement
Forms for Sequential READ Statements
Record Types

Forms for Direct-Access READ Statements

Direct-access READ statements transfer input data from external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access READ statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

READ (*eunit*, *format*, *rec* [, *asynchronous*] [, *blank*] [, *decimal*] [, *id*] [, *pad*] [, *pos*] [, *round*] [, *size*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

Unformatted:

READ (*eunit*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

See Also

READ
I/O control-list specifiers
I/O lists

Rules for Formatted Direct-Access READ Statements

Formatted, direct-access READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is read by that input statement.

Examples

The following example shows a formatted, direct-access READ statement:

```
READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

Rules for Unformatted Direct-Access READ Statements

Unformatted, direct-access READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, direct-access READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is less than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is greater than the number of fields in an input record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows unformatted, direct-access READ statements:

```
READ (1, REC=10) LIST(1), LIST(8)
READ (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

Forms for Stream READ Statements

The forms for stream READ statements take the same forms as sequential READ statements. A POS specifier may be present to specify at what file position the READ will start.

You can impose a record structure on a formatted, sequential stream by using a new-line character as a record terminator (see intrinsic function NEW_LINE). There is no record structure in an unformatted, sequential stream.

The INQUIRE statement can be used with the POS specifier to determine the current file position in a stream file.

Examples

The following example shows stream READ statements:

```
READ (12 ) I          !stream reading without POS= specifier
READ (12,POS=10) J    !stream reading with POS= specifier
```

See Also

NEW_LINE

Forms and Rules for Internal READ Statements

Internal READ statements transfer input data from an internal file.

An internal READ statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is also permitted.

An internal READ statement takes one of the following forms:

READ (*iunit*, *format* [, *nml-group*] [, *iostat*] [, *err*] [, *end*] [, *iomsg*]) [*io-list*]

READ (*iunit*, *nml-group* [, *iostat*] [, *err*] [, *end*] [, *iomsg*]) [*io-list*]

For more information on syntax, see READ.

Formatted, internal READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

This form of READ statement behaves as if the format begins with a BN edit descriptor. (You can override this behavior by explicitly specifying the BZ edit descriptor.)

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

Before data transfer occurs, the file is positioned at the beginning of the first record. This record becomes the current record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains.

In list-directed and namelist formatting, character strings have no delimiters.

Examples

The following program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal READ statements to make appropriate conversions from character string representations to binary.

```
INTEGER IVAL
CHARACTER TYPE, RECORD*80
CHARACTER*(*) AFMT, IFMT, OFMT, ZFMT
PARAMETER (AFMT='(Q,A)', IFMT= '(I10)', OFMT= '(O11)',        &  ZFMT= '(Z8)')
ACCEPT AFMT, ILEN, RECORD
TYPE = RECORD(1:1)
```

```
IF (TYPE .EQ. 'D') THEN
    READ (RECORD(2:MIN(ILEN, 11)), IFMT) IVAL
ELSE IF (TYPE .EQ. 'O') THEN
    READ (RECORD(2:MIN(ILEN, 12)), OFMT) IVAL
ELSE IF (TYPE .EQ. 'X') THEN
    READ (RECORD(2:MIN(ILEN, 9)),ZFMT) IVAL
ELSE
    PRINT *, 'ERROR'
END IF
END
```

See Also

I/O control-list specifiers
I/O lists
Rules for List-Directed Sequential READ Statements for details on list-directed input
Rules for Namelist Sequential READ Statement for details on namelist input

Forms for WRITE Statements

This section discusses the various forms you can specify for WRITE statements.

Forms for Sequential WRITE Statements

Sequential WRITE statements transfer output data to external sequential access records. The statements can be formatted by using format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

A sequential WRITE statement takes one of the following forms:

Formatted:

WRITE (*eunit*, *format* [, *advance*] [, *asynchronous*] [, *decimal*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

Formatted - List-Directed:

WRITE (*eunit*, * [, *asynchronous*] [, *decimal*] [, *delim*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

Formatted - Namelist:

WRITE (*eunit*, *nml-group* [, *asynchronous*] [, *decimal*] [, *delim*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*])

Unformatted:

WRITE (*eunit* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

See Also

WRITE
I/O control-list specifiers
I/O lists

Rules for Formatted Sequential WRITE Statements

Formatted, sequential WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for sequential access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

The output list and format specification must not specify more characters for a record than the record size. (Record size is specified by RECL in an OPEN statement.)

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

The following example shows formatted, sequential WRITE statements:

```
WRITE (UNIT=8, FMT='(B)', ADVANCE='NO') C

WRITE (*, "(F6.5)", ERR=25, IOSTAT=IO_STATUS) A, B, C
```

See Also

WRITE statement
Forms for Sequential WRITE Statements

Rules for List-Directed Sequential WRITE Statements

List-directed, sequential WRITE statements transfer data from binary to character form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input. However, there is no guarantee that a REAL internal value transferred as output and then transferred as input as a REAL value will be the same internal value.

The following table shows the default output formats for each intrinsic data type:

Default Formats for List-Directed Output

Data Type	Output Format
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8)	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8)	I22
REAL(4)	1PG15.7E2 [2]
REAL(8)	1PG24.15E3 [2]

Data Type	Output Format
REAL(16)	1PG43.33E4 [2]
COMPLEX(4)	'(',1PG14.7E2,',',1PG14.7E2,')' [2]
COMPLEX(8)	'(',1PG23.15E3,',',1PG23.15E3,')' [2]
COMPLEX(16)	'(',1PG42.33E4,',',1PG42.33E4,')' [2]
CHARACTER	A*w* [1]

[1] Where *w* is the length of the character expression.

[2] If option `assume noold_ldout_format` is in effect, the compiler uses Fortran 2008 standard semantics for output of integer and real values in list-directed and namelist-directed output. This means that for real and complex values, the output is in E or F format depending on the magnitude of the value. For more information, see the description of option assume.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM='QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM='APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

If DECIMAL='POINT', the decimal point in a numeric value is displayed as a period, values are separated by commas, and the separator between the real and imaginary parts of a complex value is a comma. If DECIMAL='COMMA', the decimal point is displayed as a comma, values are separated by semicolons, and the separator between the real and imaginary parts of a complex value is a semicolon.

A literal character constant or complex constant can be longer than an entire record. For complex constants, the end of the record can occur between the comma or semicolon and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record. For literal constants that are longer than an entire record, the constant is continued onto as many records as necessary.

Each output record begins with a blank character for carriage control.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

Examples

Suppose the following statements are specified:

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE (1,*) 'ARRAY VALUES FOLLOW'
WRITE (1,*) A,4
```

The following records are then written to external unit 1:

```
ARRAY VALUES FOLLOW
   3.400000      3.400000      3.400000      3.400000      4
```

The following shows another example:

```
INTEGER       i, j
REAL          a, b
LOGICAL       on, off
```

```
  CHARACTER(20) c
  DATA i /123456/, j /500/, a /28.22/, b /.0015555/
  DATA on /.TRUE./, off/.FALSE./
  DATA c /'Here''s a string'/
  WRITE (*, *)  i, j
  WRITE (*, *)  a, b, on, off
  WRITE (*, *)  c
  END
```

The preceding example produces the following output:

```
    123456         500
  28.22000      1.555500E-03 T F
Here's a string
```

See Also

Rules for Formatted Sequential WRITE Statements
Rules for List-Directed Sequential READ Statements for details on list-directed input

Rules for Namelist Sequential WRITE Statements

Namelist, sequential WRITE statements translate data from internal to external form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input. However, there is no guarantee that a REAL internal value transferred as output and then transferred as input as a REAL value will be the same internal value.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM='QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM='APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

If DECIMAL='POINT', the decimal point in a numeric value is displayed as a period, values are separated by commas, and the separator between the real and imaginary parts of a complex value is a comma. If DECIMAL='COMMA', the decimal point is displayed as a comma, values are separated by semicolons, and the separator between the real and imaginary parts of a complex value is a semicolon.

A literal character constant or complex constant can be longer than an entire record. For complex constants, the end of the record can occur between the comma or semicolon and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record. For literal constants that are longer than an entire record, the constant is continued onto as many records as necessary.

Each output record begins with a blank character for carriage control, except for literal character constants that are continued from the previous record.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Examples

Consider the following statements:

```
CHARACTER*19 NAME(2)/2*' '/
REAL PITCH, ROLL, YAW, POSITION(3)
LOGICAL DIAGNOSTICS
```

```
INTEGER ITERATIONS
TYPE MYTYPE
   INTEGER X
   REAL Y
   CHARACTER(5)Z
END TYPE MYTYPE
TYPE(MYTYPE) :: TYPEVAR = MYTYPE(1,2.0,'ABCDE')
NAMELIST /PARAM/ NAME, PITCH, ROLL, TYPEVAR, YAW, POSITION,        &
         DIAGNOSTICS, ITERATIONS
...
READ (UNIT=1,NML=PARAM)
WRITE (UNIT=2,NML=PARAM)
```

Suppose the following input is read:

```
&PARAM
    NAME(2)(10:)='HEISENBERG',
    PITCH=5.0, YAW=0.0, ROLL=5.0,
    DIAGNOSTICS=.TRUE.
    ITERATIONS=10
/
```

The following is then written to the file connected to unit 2:

```
&PARAM
NAME    = '                         ', '        HEISENBERG',
PITCH   =    5.000000,
ROLL    =    5.000000,
TYPEVAR = 1, 2.0, 'ABCDE'
YAW     =   0.0000000E+00,
POSITION        = 3*0.0000000E+00,
DIAGNOSTICS     = T,
ITERATIONS      =               10
/
```

Note that character values are not enclosed in apostrophes unless the output file is opened with DELIM='APOSTROPHE'. The value of POSITION is not defined in the namelist input, so the current value of POSITION is written.

The following example declares a number of variables, which are placed in a namelist, initialized, and then written to the screen with namelist I/O:

```
      INTEGER(1) int1
      INTEGER    int2, int3, array(3)
      LOGICAL(1) log1
      LOGICAL log2, log3
      REAL    real1
      REAL(8) real2
      COMPLEX z1, z2
      CHARACTER(1)  char1
      CHARACTER(10) char2

      NAMELIST /example/ int1, int2, int3, log1, log2, log3,       &
     &          real1, real2, z1, z2, char1, char2, array

      int1     = 11
      int2     = 12
      int3     = 14
      log1     = .TRUE.
```

```
      log2     = .TRUE.
      log3     = .TRUE.
      real1    = 24.0
      real2    = 28.0d0
      z1       = (38.0,0.0)
      z2       = (316.0d0,0.0d0)
      char1    = 'A'
      char2    = '0123456789'
      array(1) = 41
      array(2) = 42
      array(3) = 43
      WRITE (*, example)
```

The preceding example produces the following output:

```
&EXAMPLE
INT1  =   11,
INT2  =          12,
INT3  =          14,
LOG1  = T,
LOG2  = T,
LOG3  = T,
REAL1 =   24.00000,
REAL2 =   28.0000000000000,
Z1    = (38.00000,0.0000000E+00),
Z2    = (316.0000,0.0000000E+00),
CHAR1 = A,
CHAR2 = 0123456789,
ARRAY =          41,          42,          43
/
```

See Also

NAMELIST
Rules for Formatted Sequential WRITE Statements
Rules for Namelist Sequential READ Statements

Rules for Unformatted Sequential WRITE Statements

Unformatted, sequential WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

This form of WRITE statement writes exactly one record. If there is no I/O item list, the statement writes one null record.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

For a discussion of I/O buffering, see Rules for Unformatted Sequential READ Statements.

Examples

The following example shows an unformatted, sequential WRITE statement:

```
 WRITE (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

Forms for Direct-Access WRITE Statements

Direct-access WRITE statements transfer output data to external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access WRITE statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

WRITE (*eunit*, *format*, *rec* [, *asynchronous*] [, *decimal*] [, *delim*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

Unformatted:

WRITE (*eunit*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

See Also

WRITE
I/O control-list specifiers
I/O lists

Rules for Formatted Direct-Access WRITE Statements

Formatted, direct-access WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for direct access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is written by that output statement.

Examples

The following example shows a formatted, direct-access WRITE statement:

```
WRITE (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

Rules for Unformatted Direct-Access WRITE Statements

Unformatted, direct-access WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows unformatted, direct-access WRITE statements:

```
WRITE (1, REC=10) LIST(1), LIST(8)

WRITE (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

Forms for Stream WRITE Statements

The forms for stream WRITE statements take the same forms as sequential WRITE statements. A POS specifier may be present to specify at what file position the WRITE will start.

After a formatted stream WRITE where no error occurred, the output file is truncated after the byte with the largest POS value. An unformatted stream WRITE does not truncate the output file.

You can impose a record structure on a formatted, sequential stream by using a new-line character as a record terminator (see intrinsic function NEW_LINE). There is no record structure in an unformatted, sequential stream.

The INQUIRE statement can be used with the POS specifier to determine the current file position in a stream file.

See Also

NEW_LINE

Forms and Rules for Internal WRITE Statements

Internal WRITE statements transfer output data to an internal file.

An internal WRITE statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is also permitted.

An internal WRITE statement takes one of the following forms:

WRITE (*iunit*, *format* [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

WRITE (*iunit*, *nml-group* [, *iostat*] [, *err*] [, *iomsg*]) [*io-list*]

For more information on syntax, see WRITE.

Formatted, internal WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an internal file.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the number of characters written in a record is less than the length of the record, the rest of the record is filled with blanks. The number of characters to be written must not exceed the length of the record.

Character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

Examples

The following example shows an internal WRITE statement:

```
INTEGER J, K, STAT_VALUE
CHARACTER*50 CHAR_50
...
WRITE (FMT=*, UNIT=CHAR_50, IOSTAT=STAT_VALUE) J, K
```

See Also

I/O control-list specifiers
I/O lists
Rules for List-Directed Sequential WRITE Statements for details on list-directed output
Rules for Namelist Sequential WRITE Statements for details on namelist output

User-Defined Derived-Type I/O

By default, when a derived-type object is in a formatted I/O statement, it is treated as if all of its components were specified in the component order. The components must be accessible in the scope of the I/O statement and they cannot be pointers or allocatables. In an unformatted I/O statement, a derived-type object is treated as a single value in a processor-dependent form.

User-defined derived-type I/O lets you replace the default I/O processing for a derived-type object. For both unformatted and formatted I/O, a procedures can be invoked that will handle the I/O of the derived type. This is similar to defined operators and defined assignment.

For formatted I/O, the replacement occurs for list-directed formatting, namelist formatting, and for an explicit format with the DT edit descriptor. Other edit descriptors in the explicit format do not have any effect on user-defined I/O.

A procedure can be associated with defined I/O through generic bindings or generic interface blocks. The procedure must conform to the interface that specifies its characteristics. It cannot directly or indirectly use OpenMP* constructs. A defined I/O procedure can also call itself recursively.

An I/O statement that includes a derived-type object and causes a defined I/O procedure to be invoked is called a parent I/O statement. An I/O statement that is executed while a parent statement is being processed, and specifies the unit passed to a defined I/O procedure, is called a child I/O statement.

A defined I/O procedure can be invoked as a parent I/O statement for an external or internal file opened for sequential, direct, or stream access.

This section also has information about how to resolve user-defined I/O procedure references.

Specifying the User-Defined Derived Type

DT Edit Descriptor in User-Defined I/O

The DT edit descriptor passes a character string and integer array to a defined I/O procedure. It takes the following form:

DT [*string*] [(*v-list*)]

The *string* is a default character literal constant delimited by single quotes (' ') or double quotes (" "); no kind parameter can be specified. Its length is the number of characters between the delimiters; two consecutive delimiters are counted as one character. If *string* is not specified, a character string of length zero is passed.

The *v-list* is a list of one or more signed or unsigned integer literal constants; no kind parameter can be specified. If *v-list* is not specified, an integer array of length zero is passed.

A DT edit descriptor must correspond to a list item of a derived type. Also, there should be an accessible interface to a corresponding defined FORMATTED I/O procedure for that derived type.

In a format statement, if the last closing parenthesis of the format string is reached and there are no effective items left, then format processing terminates. But, if there are more items to be processed, then format control reverts to the beginning of the format item which was terminated by the next-to-last right parenthesis.

If there is no such preceding right parenthesis, it reverts to the first left parenthesis of the format specification. During this format reversion, the right parenthesis that is part of a DT edit descriptor is not considered as the next-to-last parenthesis. For example, consider the following:

```
write (10, '(F10.3, I5, 'DT "sample" (1, 2) )' ) 10.1, 3, obj1, 4.7, 1, obj2
```

In the above case, format control reverts to the left parenthesis before F10.3 and not to DT.

Examples

The following are valid ways to specify the DT edit descriptor:

```
DT
DT "z8, i4, e10.2"
DT (1, -1, +1000)
DT 'my type' (0)
```

Associating a Procedure with Defined I/O

Defined I/O Procedures

For a particular derived type and a particular set of kind-type parameter values, there are four possible sets of characteristics for defined I/O procedures: formatted input, formatted output, unformatted input, and unformatted output.

To specify that an I/O procedure should be used for derived-type I/O, specify one of the following with a *defined-io-generic-spec*:

- A generic binding
- An interface block

A *defined-io-generic-spec* is one of the following statements:

- READ (FORMATTED)
- READ (UNFORMATTED)
- WRITE (FORMATTED)
- WRITE (UNFORMATTED)

Generic Bindings

User-defined I/O procedures can be type-bound procedures that use a *defined-io-generic-spec* (see Defined IO Procedures).

Consider the following:

```
TYPE LIST
TYPE(NODE), POINTER :: FIRST
CONTAINS
PROCEDURE :: FMTREAD => LIST_FMTREAD
PROCEDURE :: FMTWRITE => LIST_FMTWRITE
GENERIC,PUBLIC :: READ(FORMATTED) => FMTREAD
GENERIC,PUBLIC :: WRITE(FORMATTED) => FMTWRITE
END TYPE LIST
```

In the above example, LIST_FMTREAD and LIST_FMTWRITE are the type-bound defined I/O procedures. If an object of type LIST is an effective item in a formatted READ statement, LIST_FMTREAD will be called to perform the read operation.

See Also

Type-Bound Procedures
Resolving Defined I/O Procedure References

Generic Interface Block

A generic interface block can be used to associate procedures with defined I/O. The generic identifier should be the *defined-io-generic-spec* (see Defined IO Procedures). This is the only option for sequence or BIND(C) types.

Consider the following:

```
MODULE EXAMPLE

TYPE LIST
INTEGER :: X
END TYPE

INTERFACE READ (FORMATTED)
MODULE PROCEDURE R1
END INTERFACE

CONTAINS
```

```
    SUBROUTINE R1 (DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
    CLASS(LIST), INTENT(INOUT) :: DTV
    INTEGER, INTENT(IN) :: UNIT
    CHARACTER(*), INTENT(IN) :: IOTYPE
    INTEGER, INTENT(IN) :: V_LIST(:)
    INTEGER, INTENT(OUT) :: IOSTAT
    CHARACTER(*), INTENT(INOUT) :: IOMSG
       READ (UNIT, FMT=*, IOSTAT=IOSTAT, IOMSG=IOMSG) DTV%X
    END SUBROUTINE R1

END MODULE EXAMPLE
```

In the above example, R1 is the READ (FORMATTED) defined I/O procedure.

If an object of type LIST is an effective item in a formatted READ statement, R1 will be called.

See Also

TYPE Statement (Derived Types)
Defining Generic Names for Procedures

Characteristics of Defined I/O Procedures

Shown below are the four interfaces that specify the characteristics of the user-defined I/O procedures. The actual specific procedure names and the names of the dummy arguments in these interfaces are arbitrary.

The following names are used in the interfaces:

dtv-type-spec
: Is one of the following:
  - TYPE(*d-name*) for a sequence or BIND(C) type
  - CLASS(*d-name*) for an extensible type

  d-name is the name of the derived type. It cannot be an abstract type. All length type parameters of the derived type must be assumed.

var
: Is a scalar of the derived type. For output, it holds the value to be written. For input, it will be altered in accordance with the values read.

unit
: Is the scalar integer value of the I/O unit on which input or output is taking place. It is a negative number for an internal file or for an external unit that is a NEWUNIT value. It is a processor-dependent number (which may be negative) for the '*' unit.

iotype
: Is the value 'LISTDIRECTED', 'NAMELIST', or 'DT'//string, where string is the character string from the DT edit descriptor.

vlist
: Is a rank-one assumed-shape integer array whose value comes from the parenthetical list of integers from the DT edit descriptor. For list-directed formatting and namelist formatting, *vlist* is a zero-sized integer array.

iostat
: Is a scalar integer variable that must be given a positive value if an error condition occurs. If an end-of-file or end-of-record condition occurs, it must be given the value IOSTAT_END or IOSTAT_EOR (from the intrinsic module ISO_FORTRAN_ENV). In all other cases, it must be given the value zero.

iomsg	Is an assumed-length scalar character variable that must be set to an explanatory message if iostat is given a nonzero value. Otherwise, it must not be altered.

The following interfaces specify the characteristics of the user-defined I/O procedures:

```
SUBROUTINE my_read_formatted (var,unit,iotype,vlist,iostat,iomsg)
dtv-type-spec,INTENT(INOUT) :: var
INTEGER,INTENT(IN) :: unit
CHARACTER(*),INTENT(IN) :: iotype
INTEGER,INTENT(IN) :: vlist(:)
INTEGER,INTENT(OUT) :: iostat
CHARACTER(*),INTENT(INOUT) :: iomsg
END

SUBROUTINE my_read_unformatted (var,unit,iostat,iomsg)
dtv-type-spec,INTENT(INOUT) :: var
INTEGER,INTENT(IN) :: unit
INTEGER,INTENT(OUT) :: iostat
CHARACTER(*),INTENT(INOUT) :: iomsg
END

SUBROUTINE my_write_formatted (var,unit,iotype,vlist,iostat,iomsg)
dtv-type-spec,INTENT(IN) :: var
INTEGER,INTENT(IN) :: unit
CHARACTER(*),INTENT(IN) :: iotype
INTEGER,INTENT(IN) :: vlist(:)
INTEGER,INTENT(OUT) :: iostat
CHARACTER(*),INTENT(INOUT) :: iomsg
END

SUBROUTINE my_write_unformatted (var,unit,iostat,iomsg)
dtv-type-spec,INTENT(IN) :: var
INTEGER,INTENT(IN) :: unit
INTEGER,INTENT(OUT) :: iostat
CHARACTER(*),INTENT(INOUT) :: iomsg
END
```

See Also

ISO_FORTRAN_ENV Module

Defined I/O Data Transfers

The following rules apply to defined I/O data transfers:

- During execution of a defined I/O procedure, there must be no I/O for an external unit except for the unit argument. However, I/O is permitted for internal files.
- You cannot use user-defined I/O in combination with asynchronous I/O.
- No file-positioning commands are permitted in the defined I/O procedure. OPEN, CLOSE, BACKSPACE, ENDFILE, and REWIND statements will not be executed. Any ADVANCE= specifier in a child statement is ignored.
- A record positioning edit descriptor, such as TL and TR, used on the unit by a child data transfer statement will not cause the record position to be positioned before the record position at the time the defined I/O procedure was invoked.
- A child data transfer statement must not specify the ID=, POS=, or REC= specifier in an I/O control list.
- The file position on entry is treated as a left tab limit and there is no record termination on return. However, a child statement with slash (/) edit descriptor, or explicit record termination by a list-directed child I/O, is allowed.

- If the unit is associated with an external file (for example, non-negative, or equal to one of the constants ERROR_UNIT, INPUT_UNIT, or OUTPUT_UNIT from the intrinsic module ISO_FORTRAN_ENV), the current settings for the pad mode, sign mode, etc., can be discovered by using INQUIRE with PAD=, SIGN=, etc. on the unit argument.

  Note that INQUIRE must not be used if the unit is an internal unit passed to a user-defined derived-type I/O procedure from a parent I/O statement. When an internal unit is used with the INQUIRE statement, an error condition will occur, and the variable specified in an IOSTAT= specifier in the INQUIRE statement will be assigned the value IOSTAT_INQUIRE_INTERNAL_UNIT from the intrinsic module ISO_FORTRAN_ENV.

See Also

ISO_FORTRAN_ENV Module

User-Defined Derived-Type I/O

Resolving Defined I/O Procedure References

A generic interface for defined I/O of a derived-type object is one that has both of the following:

- A *defined-io-generic-spec* that is appropriate to the READ or WRITE direction and the form (formatted or unformatted) of the data transfer (see Defined IO Procedures, Generic Bindings, and Generic Interface Block).
- A specific interface whose *var* argument is compatible with the derived-type item.

Within the scope of a *defined-io-generic-spec*, if two procedures have that generic identifier, they must be distinguishable.

For defined I/O procedures, only the *var* argument corresponds to something explicitly written in the program, so it is the *var* that must be distinguishable.

Because *var* arguments are required to be scalar, they cannot differ in rank. So, the *var* must be distinguishable in the type and kind type parameters.

You cannot have two procedures with the same *defined-io-generic-spec*.

See Also

Characteristics of Defined I/O Procedures

Recursive Defined I/O

A defined I/O procedure can invoke itself indirectly.

It can have an I/O statement that includes a derived-type object that results in the invocation of the same procedure. In this case, the defined I/O procedure must be declared RECURSIVE.

Consider the following:

```
! This prints a linked list by calling write on the children
! of the list.

MODULE LIST_MODULE
  IMPLICIT NONE
  TYPE NODE
! This type declaration represents a singly-linked list that also
! contains a user-defined i/o procedure. The name of the procedure
! is arbitrary, but the order of arguments must conform to the
! standard definition.

     INTEGER :: VALUE = -1
     TYPE(NODE), POINTER :: NEXT_NODE => NULL()
   CONTAINS
     PROCEDURE :: PWF
GENERIC :: WRITE(FORMATTED) => PWF ! <=== GENERIC BINDING.
  END TYPE NODE
CONTAINS
```

```
RECURSIVE SUBROUTINE PWF( DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG )
! These arguments are defined in the standard.
    CLASS(NODE), INTENT(IN) :: DTV
    INTEGER, INTENT(IN) :: UNIT
    CHARACTER(LEN=*), INTENT(IN) :: IOTYPE
    INTEGER, DIMENSION(:), INTENT(IN) :: V_LIST
    INTEGER :: IOSTAT
    CHARACTER(LEN=*), INTENT(INOUT) :: IOMSG

! The following is a child i/o statement that is called when user-defined i/o
! statement is invoked.
    WRITE( UNIT=UNIT, FMT='(I9)', IOSTAT=IOSTAT ) DTV%VALUE
    PRINT *, ASSOCIATED(DTV%NEXT_NODE)
    IF(IOSTAT /= 0)RETURN

! It is possible to recursively call the user-defined i/o routine.
    IF(ASSOCIATED(DTV%NEXT_NODE)
      WRITE(UNIT=UNIT, FMT='(/,DT)', IOSTAT=IOSTAT) DTV%NEXT_NODE
    END IF
  END SUBROUTINE PWF
END MODULE LIST_MODULE

PROGRAM LISTE
  USE LIST_MODULE
  IMPLICIT NONE
  INTEGER :: UNIT, IOSTAT, I
  TYPE(NODE), POINTER :: CUR, TO_PRINT

! Create the linked list
  ALLOCATE(CUR)
  CUR % VALUE = 999
  ALLOCATE(TO_PRINT)
  TO_PRINT => CUR

  DO I = 1,10
     ALLOCATE(CUR%NEXT_NODE)
     CUR % VALUE = I
     CUR => CUR%NEXT_NODE
  END DO
  CUR % NEXT_NODE => NULL()
! END CREATION OF LINKED LIST

  DO I = 1,15
     IF(ASSOCIATED(TO_PRINT)) THEN
        PRINT *, I, TO_PRINT%VALUE
        TO_PRINT => TO_PRINT % NEXT_NODE
     END IF
  END DO

! Call the user-defined i/o routine with dt format descriptor.
  WRITE( UNIT=UNIT, FMT='(DT)', IOSTAT=IOSTAT ) CUR
END PROGRAM LISTE
```

See Also

RECURSIVE

Examples of User-Defined Derived-Type I/O

Example 1

The following example shows formatted defined I/O using the DT edit descriptor and both generic type-bound and explicit interface procedures:

```
MODULE TYPES
  TYPE T
     INTEGER :: K(10)
   CONTAINS

! a generic type-bound procedure
     PROCEDURE :: UDIO_READ_ARRAY
     GENERIC :: READ (FORMATTED) => UDIO_READ_ARRAY
  END TYPE T

! an explicit interface
  INTERFACE WRITE(FORMATTED)
     MODULE PROCEDURE UDIO_WRITE_ARRAY
  END INTERFACE
 CONTAINS
  SUBROUTINE UDIO_READ_ARRAY (DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
    CLASS(T), INTENT(INOUT)     :: DTV
    INTEGER, INTENT(IN)         :: UNIT
    CHARACTER(*), INTENT(IN)    :: IOTYPE
    INTEGER, INTENT(IN)         :: V_LIST (:)
    INTEGER, INTENT(OUT)        :: IOSTAT
    CHARACTER(*), INTENT(INOUT) :: IOMSG

! This is the child I/O that gets performed when the procedure
! is called from a parent I/O - it uses list-directed input to read
! the array K

    READ (UNIT, FMT=*, IOSTAT=IOSTAT, IOMSG=IOMSG) DTV%K

    END SUBROUTINE UDIO_READ_ARRAY

  SUBROUTINE UDIO_WRITE_ARRAY (DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
  CLASS(T), INTENT(IN)        :: DTV
  INTEGER, INTENT(IN)         :: UNIT
  CHARACTER(*), INTENT(IN)    :: IOTYPE
  INTEGER, INTENT(IN)         :: V_LIST (:)
  INTEGER, INTENT(OUT)        :: IOSTAT
  CHARACTER(*), INTENT(INOUT) :: IOMSG

! This is the child I/O that gets performed when the procedure
! is called from a parent I/O - it uses list-directed output to write
! the array K

        WRITE (UNIT, FMT=*, IOSTAT=IOSTAT, IOMSG=IOMSG) DTV%K

  END SUBROUTINE UDIO_WRITE_ARRAY

 END MODULE TYPES
```

```
 PROGRAM TEST1
  USE TYPES
  TYPE (T) :: V
  INTEGER  :: COUNTCHAR

   OPEN (1, FILE='TEST.INPUT', FORM='FORMATTED')
    READ (1, FMT='(DT)', ADVANCE='NO', SIZE=COUNTCHAR) V
  CLOSE(UNIT=1)
  WRITE(6, '(DT)') V

 END PROGRAM TEST1
```

Consider that procedure UDIO_READ_ARRAY reads an input file named TEST.INPUT that contains the following:

```
1, 3, 5, 7, 9, 2, 4, 6, 8, 10
```

In this case, the program TEST1 in procedure UDIO_WRITE_ARRAY prints:

```
           1           3           5           7           9           2
           4           6           8          10
```

Example 2

The following example shows list-directed formatted output and user-defined I/O:

```
MODULE M
  TYPE T
     REAL, POINTER :: R (:)
   CONTAINS
     PROCEDURE :: UDIO_WRITE_LD
     GENERIC   :: WRITE(FORMATTED) => UDIO_WRITE_LD
  END TYPE T

CONTAINS
  SUBROUTINE UDIO_WRITE_LD (DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
    CLASS(T), INTENT(IN)            :: DTV
    INTEGER, INTENT(IN)             :: UNIT
    CHARACTER(LEN=*), INTENT(IN)    :: IOTYPE
    INTEGER, INTENT(IN)             :: V_LIST (:)
    INTEGER, INTENT(OUT)            :: IOSTAT
    CHARACTER(LEN=*), INTENT(INOUT) :: IOMSG
    IOSTAT = 0
    PRINT *, SIZE (DTV%R)
    WRITE (UNIT, *) DTV%R
  END SUBROUTINE UDIO_WRITE_LD
END MODULE M


PROGRAM TEST2
  USE M
  TYPE (T)     :: X
  REAL, TARGET :: V (3)

  V = [ SIN (1.0), COS (1.0), TAN (1.0) ]
  X = T (R=V)
  PRINT *, X
END PROGRAM TEST2
```

TEST2 should print "`3 0.8414710 0.5403023 1.557408`".

Example 3

The following example shows user-defined derived-type NAMELIST input/output:

```
! PROGRAM: udio_nml_read_write.f90
!
! This program tests NAMELIST READ and WRITE.  In the WRITE subroutine, there
! are FORMATTED WRITES as well as NAMELIST WRITES.
!
MODULE UDIO
  TYPE MYDT
     INTEGER F1
     INTEGER F2
   CONTAINS
     PROCEDURE :: MYSUBROUTINE
     GENERIC :: READ (FORMATTED) => MYSUBROUTINE
  END TYPE MYDT

  INTERFACE WRITE (FORMATTED)
     MODULE PROCEDURE :: WRITESUBROUTINE
  END INTERFACE
CONTAINS

  SUBROUTINE WRITESUBROUTINE (DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
    CLASS (MYDT),    INTENT(IN)    :: DTV
    INTEGER*4,         INTENT(IN)    :: UNIT
    CHARACTER (LEN=*),     INTENT(IN)    :: IOTYPE
    INTEGER,              INTENT(IN)    :: V_LIST(:)
    INTEGER*4,         INTENT(OUT)    :: IOSTAT
    CHARACTER (LEN=*),    INTENT(INOUT)    :: IOMSG

    INTEGER I, J
    NAMELIST /SUBRT_NML/ I, J

    I=DTV%F1
    J=DTV%F2

    WRITE (UNIT, '(A,2I5.2)', IOSTAT=IOSTAT) IOTYPE, DTV%F1, DTV%F2
    WRITE (UNIT, NML=SUBRT_NML)
  END SUBROUTINE WRITESUBROUTINE

SUBROUTINE MYSUBROUTINE (DTV, UNIT, IOTYPE, V_LIST, IOSTAT, IOMSG)
    CLASS (MYDT),    INTENT(INOUT)    :: DTV
    INTEGER*4,         INTENT(IN)    :: UNIT
    CHARACTER (LEN=*),     INTENT(IN)    :: IOTYPE
    INTEGER,              INTENT(IN)    :: V_LIST(:)
    INTEGER*4,         INTENT(OUT)    :: IOSTAT
    CHARACTER (LEN=*),    INTENT(INOUT)    :: IOMSG

! X and Y are aliases for DTV%F1 and DTV%F2 since field references
! cannot be referenced in a NAMELIST statement

    INTEGER X, Y
    NAMELIST /SUBRT_NML/ X, Y

    READ (UNIT, *) DTV%F1, DTV%F2
```

```
    X = DTV%F1
    Y = DTV%F2

    READ (UNIT, NML=SUBRT_NML, IOSTAT=IOSTAT)

END SUBROUTINE MYSUBROUTINE

END MODULE UDIO

PROGRAM UDIO_PROGRAM
   USE UDIO
   TYPE (MYDT) :: MYDTV
   INTEGER        :: A, B
   NAMELIST /MAIN_NML/ A, MYDTV, B

   OPEN (10, FILE='udio_nml_read_write.in')
   READ (10, NML=MAIN_NML)
   WRITE (6, NML=MAIN_NML)
   CLOSE (10)

END PROGRAM UDIO_PROGRAM
```

The following shows input file 'udio_nml_read_write.in' on unit 10 read by MYSUBROUTINE:

```
&MAIN_NML
A=100
MYDTV=20 30
&SUBRT_NML
X=20
Y=30
/
/B=200
/
```

The following shows output to unit 6 by WRITESUBROUTINE:

```
&MAIN_NML
 A       =         100,
 MYDTV=NAMELIST   20   30
&SUBRT_NML
 I       =          20,
 J       =          30
/
 /B       =         200
/
```

Example 4

The following example shows user-defined derived-type UNFORMATTED input/output:

```
! PROGRAM: udio_unformatted_1.f90
!
! This test first writes unformatted data to a file via user-defined derived type output
! and then reads the data from the file via user-defined derived type input.
!
MODULE UNFORMATTED
    TYPE UNFORMATTED_TYPE
        INTEGER             :: I
```

```
        CHARACTER*25 :: CHAR
    CONTAINS
        PROCEDURE :: MY_UNFMT_WRITE
        GENERIC :: WRITE (UNFORMATTED) => MY_UNFMT_WRITE
    END TYPE UNFORMATTED_TYPE

    INTERFACE READ (UNFORMATTED)
       MODULE PROCEDURE :: MY_UNFMT_READ
    END INTERFACE

CONTAINS
    SUBROUTINE MY_UNFMT_WRITE (DTV, UNIT, IOSTAT, IOMSG)
        CLASS (UNFORMATTED_TYPE), INTENT(IN)    :: DTV
        INTEGER,                  INTENT(IN)    :: UNIT
        INTEGER,                  INTENT(OUT)   :: IOSTAT
        CHARACTER(LEN=*),         INTENT(INOUT) :: IOMSG

        WRITE (UNIT=UNIT, IOSTAT=IOSTAT, IOMSG=IOMSG) DTV%I+1, DTV%CHAR
    END SUBROUTINE MY_UNFMT_WRITE

    SUBROUTINE MY_UNFMT_READ (DTV, UNIT, IOSTAT, IOMSG)
        CLASS (UNFORMATTED_TYPE), INTENT(INOUT)  :: DTV
        INTEGER,                  INTENT(IN)     :: UNIT
        INTEGER,                  INTENT(OUT)    :: IOSTAT
        CHARACTER (LEN=*),        INTENT(INOUT)  :: IOMSG

        READ (UNIT=UNIT, IOSTAT=IOSTAT, IOMSG=IOMSG) DTV%I, DTV%CHAR
        DTV%I = 1-DTV%I
    END SUBROUTINE MY_UNFMT_READ

END MODULE UNFORMATTED

PROGRAM UNFORMATTED_WRITE_PROGRAM
    USE UNFORMATTED

    TYPE (UNFORMATTED_TYPE) :: READ_UNFORMATTED (1,2), UNFORMATTED_OBJECT (3:3,0:1)
    INTEGER :: IOSTAT
    CHARACTER(LEN=100) :: IOMSG

    UNFORMATTED_OBJECT (3, 1) = UNFORMATTED_TYPE (I=71, CHAR='HELLO WORLD.')
    UNFORMATTED_OBJECT (3, 0) = UNFORMATTED_TYPE (I=72, CHAR='WORLD HELLO.')

    OPEN (UNIT=71, FILE='MYUNFORMATTED_DATA.DAT', FORM='UNFORMATTED')
    WRITE (UNIT=71) UNFORMATTED_OBJECT
    CLOSE (UNIT=71)

    OPEN (UNIT=77, FILE='MYUNFORMATTED_DATA.DAT', FORM='UNFORMATTED')
    READ (UNIT=77) READ_UNFORMATTED
    CLOSE (UNIT=77)

    PRINT *, -READ_UNFORMATTED (:,1:2)%I .EQ. UNFORMATTED_OBJECT%I
    PRINT *, READ_UNFORMATTED%CHAR .EQ. UNFORMATTED_OBJECT%CHAR

END PROGRAM UNFORMATTED_WRITE_PROGRAM
```

The following shows output to unit * from program UNFORMATTED_WRITE_PROGRAM:

```
T T
 T T
```

I/O Formatting

A format appearing in an input or output (I/O) statement specifies the form of data being transferred and the data conversion (editing) required to achieve that form. The format specified can be explicit or implicit.

Explicit format is indicated in a format specification that appears in a FORMAT statement or a character expression (the expression must evaluate to a valid format specification).

The format specification contains edit descriptors, which can be data edit descriptors, control edit descriptors, or string edit descriptors.

Implicit format is determined by the processor and is specified using list-directed or namelist formatting.

List-directed formatting is specified with an asterisk (*); namelist formatting is specified with a namelist group name.

List-directed formatting can be specified for advancing sequential files and internal files. Namelist formatting can be specified only for advancing sequential files.

See Also

Rules for List-Directed Sequential READ Statements for details on list-directed input
Rules for List-Directed Sequential WRITE Statements for details on list-directed output
Rules for Namelist Sequential READ Statements for details on namelist input
Rules for Namelist Sequential WRITE Statements for details on namelist output

Format Specifications

A format specification can appear in a FORMAT statement or a character expression. In a FORMAT statement, it is preceded by the keyword FORMAT. A format specification takes one of the following forms:

([*format-items*])

([*format-items,*] *unlimited-format-item*)

format-items

Is *format-items* [[,] *format-item*]...

where *format-item* is one of the following:

[*r*] *data-edit-desc*
control-edit-desc
char-string-edit-desc
[*r*] (*format-items*)

r

(Optional) Is an integer literal constant. This is called a *repeat specification*. The range of *r* is 1 through 2147483647 (2**31-1). If *r* is omitted, it is assumed to be 1.

data-edit-desc

Is one of the data edit descriptors: I, B, O, Z, F, E, EN, ES, EX, D, DT, G, L, or A.

A repeat specification can precede any data edit descriptor.

control-edit-desc

Is one of the control edit descriptors: T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, $, \, and Q.

A repeat specification can precede the slash (/) edit descriptor.

char-string-edit-desc

Is one of the string edit descriptors: H, 'c', and "c", where *c* is a character constant.

unlimited-format-item

Is * (*format-items*). The * indicates an unlimited repeat count.

If more than one edit descriptor is specified, they must be separated by commas or slashes (/).

A comma can be omitted in the following cases:

- Between a P edit descriptor and an immediately following F, E, EN, ES, EX, D, or G edit descriptor
- Before a slash (/) edit descriptor when the optional repeat specification is not present
- After a slash (/) edit descriptor
- Before or after a colon (:) edit descriptor

Description

A FORMAT statement must be labeled.

Named constants are not permitted in format specifications.

If the associated I/O statement contains an I/O list, the format specification must contain at least one data edit descriptor or the control edit descriptor Q.

Blank characters can precede the initial left parenthesis, and additional blanks can appear anywhere within the format specification. These blanks have no meaning unless they are within a character string edit descriptor.

When a formatted input statement is executed, the setting of the BLANK specifier (for the relevant logical unit) determines the interpretation of blanks within the specification. If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks. (For more information on BLANK defaults, see BLANK Specifier in OPEN statements.)

For formatted input, you can use the comma as an external field separator. The comma terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It can also designate null (zero-length) fields.

The following table summarizes the edit descriptors that can be used in format specifications.

Summary of Edit Descriptors

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values.
B	Bw[.m]	Transfers binary values.
BN	BN	Ignores embedded and trailing blanks in a numeric input field.
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.
D	Dw.d	Transfers real values with D exponents.
DT	DT [string] [(v-list)]	Passes a character string and an integer array to a defined I/O procedure.
E	Ew.d[Ee]	Transfers real values with E exponents.
EN	ENw.d[Ee]	Transfers real values with engineering notation.

Code	Form	Effect
ES	ESw.d[Ee]	Transfers real values with scientific notation.
EX	EXw.d[Ee]	Transfers real values with hexadecimal-significands.
F	Fw.d	Transfers real values with no exponent.
G	Gw.d[Ee]	Transfers values of all intrinsic types.
H	nHch[ch...]	Transfers characters following the H edit descriptor to an output record.
I	Iw[.m]	Transfers decimal integer values.
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
P	kP	Interprets certain real numbers with a specified scale factor.
Q	Q	Returns the number of characters remaining in an input record.
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.
SP	SP	Writes optional plus sign (+) into numeric output fields.
SS	SS	Suppresses optional plus sign (+) in numeric output fields.
T	Tn	Tabs to specified position.
TL	TLn	Tabs left the specified number of positions.
TR	TRn	Tabs right the specified number of positions.
X	nX	Skips the specified number of positions.
Z	Zw[.m]	Transfers hexadecimal values.
$	$	Suppresses trailing carriage return during interactive I/O.
:	:	Terminates format control if there are no more items in the I/O list.

Code	Form	Effect
/	[r]/	Terminates the current record and moves to the next record.
\	\	Continues the same record; same as $.
'c'[1]	'c'	Transfers the character literal constant (between the delimiters) to an output record.

[1] These delimiters can also be quotation marks (").

Character Format Specifications

In data transfer I/O statements, a format specifier ([FMT=]format) can be a character expression that is a character array, character array element, or character constant. This type of format is also called a run-time format because it can be constructed or altered during program execution.

The expression must evaluate to a character string whose leading part is a valid format specification (including the enclosing parentheses).

If the expression is a character array element, the format specification must be contained entirely within that element.

If the expression is a character array, the format specification can continue past the first element into subsequent consecutive elements.

If the expression is a character constant delimited by apostrophes, use two consecutive apostrophes ('') to represent an apostrophe character in the format specification; for example:

```
PRINT '("NUM can''t be a real number")'
```

Similarly, if the expression is a character constant delimited by quotation marks, use two consecutive quotation marks ("") to represent a quotation mark character in the format specification.

To avoid using consecutive apostrophes or quotation marks, you can put the character constant in an I/O list instead of a format specification, as follows:

```
PRINT "(A)", "NUM can't be a real number"
```

The following shows another character format specification:

```
WRITE (6, '(I12, I4, I12)') I, J, K
```

In the following example, the format specification changes with each iteration of the DO loop:

```
SUBROUTINE PRINT(TABLE)
REAL TABLE(10,5)
CHARACTER*5 FORCHR(0:5), RPAR*1, FBIG, FMED, FSML
DATA FORCHR(0),RPAR /'(',')'/
DATA FBIG,FMED,FSML /'F8.2,','F9.4,','F9.6,'/
DO I=1,10
  DO J=1,5
    IF (TABLE(I,J) .GE. 100.) THEN
      FORCHR(J) = FBIG
    ELSE IF (TABLE(I,J) .GT. 0.1) THEN
      FORCHR(J) = FMED
    ELSE
      FORCHR(J) = FSML
    END IF
```

```
  END DO
  FORCHR(5)(5:5) = RPAR
  WRITE (6,FORCHR) (TABLE(I,J), J=1,5)
END DO
END
```

The DATA statement assigns a left parenthesis to character array element FORCHR(0), and (for later use) a right parenthesis and three F edit descriptors to character variables.

Next, the proper F edit descriptors are selected for inclusion in the format specification. The selection is based on the magnitude of the individual elements of array TABLE.

A right parenthesis is added to the format specification just before the WRITE statement uses it.

NOTE
Format specifications stored in arrays are recompiled at run time each time they are used. If a Hollerith or character run-time format is used in a READ statement to read data into the format itself, that data is not copied back into the original array, and the array is unavailable for subsequent use as a run-time format specification.

Examples

The following example shows a format specification:

```
       WRITE (*, 9000) int1, real1(3), char1
9000   FORMAT (I5, 3F4.5, A16)
!  I5, 3F5.2, A16 is the format list.
```

The following shows a format example using a character expression:

```
       WRITE (*, '(I5, 3F5.2, A16)')iolist
!  I5, 3F4.5, A16 is the format list.
```

In the following example, the format list is put into an 80-character variable called MYLIST:

```
CHARACTER(80) MYLIST
MYLIST = '(I5, 3F5.2, A16)'
WRITE (*, MYLIST) iolist
```

Consider the following two-dimensional array:

```
   1  2  3
   4  5  6
```

In this case, the elements are stored in memory in the order: 1, 4, 2, 5, 3, 6 as follows:

```
CHARACTER(6) array(3)
DATA array / '(I5', ',3F5.2', ',A16)' /
WRITE (*, array) iolist
```

In the following example, the WRITE statement uses the character array element array(2) as the format specifier for data transfer:

```
CHARACTER(80) array(5)
array(2) = '(I5, 3F5.2, A16)'
WRITE (*, array(2)) iolist
```

See Also

Data edit descriptors
Control edit descriptors
Character string edit descriptors

Data Edit Descriptors

A data edit descriptor causes the transfer or conversion of data to or from its internal representation.

The part of a record that is input or output and formatted with data edit descriptors (or character string edit descriptors) is called a *field*.

Forms for Data Edit Descriptors

A data edit descriptor takes one of the following forms:

[*r*]*c*

[*r*]*cw*

[*r*]*cw.m*

[*r*]*cw.d*

[*r*]*cw.d[Ee]*

r	Is a repeat specification. The range of *r* is 1 through 2147483647 (2**31-1). If *r* is omitted, it is assumed to be 1.
c	Is one of the following format codes: I, B, O, Z, F, E, EN, ES, EX, D, G, L, or A.
w	Is the total number of digits in the field (the field width). If omitted, the system applies default values (see Default Widths for Data Edit Descriptors). The range of *w* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. For I, B, O, Z, D, E, EN, ES, EX, F, and G, the range can start at zero.
m	Is the minimum number of digits that must be in the field (including leading zeros). The range of *m* is 0 through 32767 (2**15-1) on Intel® 64 architecture; 0 through 255 (2**8-1) on IA-32 architecture. *w.m* applies to I, B, O, and Z format edit descriptors.
d	Is the number of digits to the right of the decimal point (the significant digits). The range of *d* is 0 through 255. If *d* exceeds 255 at compile time, a warning is issued and the value 255 is used. If *d* exceeds 255 in a run-time format, no warning or error is issued and the value 255 is used. The number of significant digits is affected if a scale factor is specified for the data edit descriptor. For the G edit descriptor, *d* must be specified if *w* is not zero. *w.d* applies to F, E, EN, ES, EX, G, and D format edit descriptors.
E	Identifies an exponent field.
e	Is the number of digits in the exponent. The range of *e* is 0 through 255. If *e* exceeds 255 at compile time, a warning is issued and the value 255 is used. If *e* exceeds 255 in a run-time format, no warning or error is issued and the value 255 is used. For the G edit descriptor, if *w* is zero, *e* must not be specified.

Description

Standard Fortran allows the field width to be omitted only for the A descriptor. However, Intel® Fortran allows the field width to be omitted for any data edit descriptor.

The *r*, *w*, *m*, *d*, and *e* must all be positive, unsigned, integer literal constants, or the digit 0 where allowed, or variable format expressions -- no kind parameter can be specified. They must not be named constants.

Actual useful ranges for *r*, *w*, *m*, *d*, and *e* may be constrained by record sizes (RECL) and the file system.

The data edit descriptors have the following specific forms:

Integer:	Iw[.m], Bw[.m], Ow[.m], and Zw[.m]
Real and complex:	Fw.d, Ew.d[Ee], ENw.d[Ee], ESw.d[Ee], EXw.d[Ee], Dw.d, and Gw.d[Ee]
Logical:	Lw
Character:	A[w]

The *d* must be specified with F, E, EN, ES, EX, D, and G field descriptors even if *d* is zero. The decimal point is also required. You must specify both *w* and *d*.

A repeat specification can simplify formatting. For example, the following two statements are equivalent:

```
20  FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)

20  FORMAT (3E12.4,4I5)
```

Examples

```
!  This WRITE outputs three integers, each in a five-space field
!  and four reals in pairs of F7.2 and F5.2 values.
      INTEGER(2) int1, int2, int3
      REAL(4) r1, r2, r3, r4
      DATA int1, int2, int3 /143, 62, 999/
      DATA r1, r2, r3, r4 /2458.32, 43.78, 664.55, 73.8/
      WRITE (*,9000) int1, int2, int3, r1, r2, r3, r4
9000  FORMAT (3I5, 2(1X, F7.2, 1X, F5.2))
```

The following output is produced from the above code:

```
  143  62  999 2458.32 43.78  664.55 73.80
```

See Also

General rules for numeric editing
Nested and group repeats

General Rules for Numeric Editing

The following rules apply to input and output data for numeric editing (data edit descriptors I, B, O, Z, F, E, EN, ES, EX, D, and G).

Rules for Input Processing

Leading blanks in the external field are ignored. If the input field is not a hexadecimal-significand number or an IEEE exceptional value, the interpretation of embedded and trailing blanks is determined by the blank interpretation mode. If BLANK='NULL' is in effect (or the BN edit descriptor has been specified) embedded and trailing blanks are ignored; otherwise, they are treated as zeros. An all-blank field is treated as a value of zero.

The following table shows how blanks are interpreted by default:

Type of Unit or File	Default
An explicitly OPENed unit	BLANK='NULL'
An internal file	BLANK='NULL'
A preconnected file[1]	BLANK='NULL'

[1] For interactive input from preconnected files, you should explicitly specify the BN or BZ edit descriptor to ensure desired behavior.

A minus sign must precede a negative value in an external field; a plus sign is optional before a positive value.

In input records, constants can include any valid kind parameter. Named constants are not permitted.

If the data field in a record contains fewer than *w* characters, an input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data. The comma terminates the data field, and can also be used to designate null (zero-length) fields. For more information, see Terminating Short Fields of Input Data.

Rules for Output Processing

The field width *w* must be large enough to include any leading plus or minus sign, and any decimal point or exponent. For example, the field width for an E data edit descriptor must be large enough to contain the following:

- For positive numbers: *d* + 5 or *d* + *e* + 3 characters
- For negative numbers: *d* + 6 or *d* + *e* + 4 characters

For D, E, EN, ES, EX, F, and I edit descriptors, a non-negative value can have a plus sign, depending on which sign edit descriptor is in effect. If a value is negative, the leftmost nonblank character is a minus sign.

If the value is smaller than the field width specified, leading blanks are inserted (the value is right-justified). If the value is too large for the field width specified, the entire output field is filled with asterisks (*).

When the value of the field width is zero, the compiler selects the smallest possible positive actual field width that does not result in the field being filled with asterisks.

See Also

Forms for data edit descriptors
Format Specifications

Integer Editing

Integer editing is controlled by the I (decimal), B (binary), O (octal), and Z (hexadecimal) data edit descriptors.

I Editing

The I edit descriptor transfers decimal integer values. It takes the following form:

I*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width), unless *w* is zero. The *m* has no effect on input, only output.

The specified I/O list item must be of type integer; logical and real items are also allowed if the compiler option `check format` is not specified.

The G edit descriptor can be used to edit integer data; it follows the same rules as I*w*.

Rules for Input Processing

On input, the I data edit descriptor transfers *w* characters from an external field and assigns their integer value to the corresponding I/O list item. The external field data must be an integer constant. *w* must not be zero.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the I edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Input          Value
I4         2788           2788
I3         -26             -26
I9         ^^^^^^312       312
```

Rules for Output Processing

On output, the I data edit descriptor transfers the value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by a sign (a plus sign is optional for positive values, a minus sign is required for negative values), followed by an unsigned integer constant with no leading zeros.

If *m* is specified, the unsigned integer constant will have at least *m* digits, padded with leading zeros if necessary.

If the output list item has the value zero, and *m* is zero, the external field is filled with blanks; if *w* is also zero, the external field is one blank.

If *w* is zero, the external field has the minimum number of characters necessary to represent the value, left justifying the value with no leading blanks. If both *w* and *m* are zero and the internal value is zero, the external field is one blank.

The following shows output using the I edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Value          Output
I3          284           284
I4         -284           -284
I4             0          ^^^0
I5          174           ^^174
I2         3244           **
I3         -473           ***
I7           29.812       An error; the decimal point is invalid
I4.0          0           ^^^^
I4.2          1           ^^01
I4.4          1           0001
I0         -473           -473
I0.4        242           0242
```

See Also

Forms for data edit descriptors
General rules for numeric editing

B Editing

The B data edit descriptor transfers binary (base 2) values. It takes the following form:

B*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width), unless *w* is zero. The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the B data edit descriptor transfers *w* characters from an external field and assigns their binary value to the corresponding I/O list item. The external field must contain only binary digits (0 or 1) or blanks. *w* must not be zero.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the B edit descriptor:

```
Format     Input          Value
B4         1001           9
B1         1              1
B2         ^0             0
B6         ^^^122      An error; the 2 is invalid in binary notation
```

Rules for Output Processing

On output, the B data edit descriptor transfers the binary value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of binary digits) with no leading zeros. A negative value is transferred in internal form.

If *w* is zero, the external field has the minimum number of characters necessary to represent the value. If both *w* and *m* are zero and the internal value is zero, the external field is one blank.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the B edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format       Value         Output
B4            9            1001
B2.2          1            01
B0           42            101010
```

See Also

Forms for data edit descriptors
General rules for numeric editing

O Editing

The O data edit descriptor transfers octal (base 8) values. It takes the following form:

O*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width), unless *w* is zero. The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the O data edit descriptor transfers *w* characters from an external field and assigns their octal value to the corresponding I/O list item. The external field must contain only octal digits (0 through 7) or blanks. *w* must not be zero.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the O edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Input          Value
O5         77777                32767
O4         77777                 4095
O3         97^           An error; the 9 is invalid in octal notation
```

Rules for Output Processing

On output, the O data edit descriptor transfers the octal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of octal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *w* is zero, the external field has the minimum number of characters necessary to represent the value. If both *w* and *m* are zero and the internal value is zero, the external field is one blank.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the O edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Value          Output
O6           32767         ^77777
O12         -32767         ^37777700001
O2           14261         **
O4              27         ^^33
O5            10.5         41050
O4.2               7       ^^07
O4.4               7       0007
O0                83       123
```

See Also

Forms for data edit descriptors
General rules for numeric editing

Z Editing

The Z data edit descriptor transfers hexadecimal (base 16) values. It takes the following form:

Z*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width), unless *w* is zero. The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the Z data edit descriptor transfers *w* characters from an external field and assigns their hexadecimal value to the corresponding I/O list item. The external field must contain only hexadecimal digits (0 though 9 and A (a) through F(f)) or blanks. *w* must not be zero.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the Z edit descriptor:

```
Format    Input             Value
Z3        A94                2708
Z5        A23DEF           664542
Z5        95.AF2       An error; the decimal point is invalid
```

Rules for Output Processing

On output, the Z data edit descriptor transfers the hexadecimal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of hexadecimal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *w* is zero, the external field has the minimum number of characters necessary to represent the value. If both *w* and *m* are zero and the internal value is zero, the external field is one blank.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the Z edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Value         Output
Z4           32767        7FFF
Z9          -32767        ^FFFF8001
Z2              16        10
Z4            -10.5       ****
Z3.3           2708       A94
Z6.4           2708       ^^0A94
Z0         14348303        DAF00F
```

See Also

Forms for data edit descriptors
General rules for numeric editing

Real and Complex Editing

Real and complex editing is controlled by the F, E, D, EN, ES, and EX data edit descriptors. The G, B, O, and Z edit descriptors can also be used to edit real and complex data.

If no field width (*w*) is specified for a real data edit descriptor, the system supplies default values.

Real data edit descriptors can be affected by specified scale factors.

NOTE
Do not use the real data edit descriptors when attempting to parse textual input. These descriptors accept some forms that are purely textual as valid numeric input values. For example, input values T and F are treated as values -1.0 and 0.0, respectively, for .TRUE. and .FALSE.

See Also

Forms for data edit descriptors
General rules for numeric editing
Scale Factor Editing (P)

Default Widths for Data Edit Descriptors for details on system default values for data edit descriptors

F Editing

The F data edit descriptor transfers real values. It takes the following form:

Fw.d

The value of *d* (the number of places after the decimal point) must not exceed the value of *w* (the field width) unless *w* is zero. When *w* is zero, the processor selects the field width. On input, *w* must not be zero.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the F data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The external field data must be an integer or real constant.

An input field is one of the following:

- An IEEE exception specification
- An hexadecimal-significand number
- An optional sign, followed by a string of one or more digits optionally containing a decimal symbol; any blanks are interpreted as zeros.

The basic form can be followed by an exponent in one of the following forms:

- A sign followed by one or more digits
- An E or D followed by zero or more blanks, followed by an optional sign and one or more digits

An exponent containing a D is processed in the same way as an exponent containing an E.

If the input field contains only an exponent letter or decimal point, it is treated as a zero value.

If the input field does not contain a decimal point or an exponent, it is treated as a real number of *w* digits, with *d* digits to the right of the decimal point. (Leading zeros are added, if necessary.)

If the input field contains a decimal point, the location of that decimal point overrides the location specified by the F descriptor.

If the field contains an exponent, that exponent is used to establish the magnitude of the value before it is assigned to the list element.

An input field that is an IEEE exception specification consists of optional blanks, followed by either of the following:

- An optional sign, followed by the string 'INF' or the string 'INFINITY'; this is an IEEE infinity

  This form can not be used if the processor does not support IEEE infinities for the input variable.
- An optional sign, followed by the string 'NAN', optionally followed by zero or more alphanumeric characters enclosed in parentheses, optionally followed by blanks; this is an IEEE Nan

  This form can not be used if the processor does not support IEEE Nans for the input variable.

The NaN value is a quiet NaN if the only nonblank characters in the field are 'NAN' or 'NAN()'.

An input field that is a hexadecimal-significand number contains an optional sign, followed by the digit 0, followed immediately by the letter X, followed by the hexadecimal significand followed by a hexadecimal exponent. A *hexadecimal significand* is a string of one or more hexadecimal characters, optionally containing a decimal symbol. The position of the hexadecimal point is indicated by the decimal symbol. The hexadecimal point implicitly follows the last hexadecimal character if decimal symbol appears in the string. A hexadecimal exponent is the letter P followed by a signed decimal digit string. Embedded blanks are not allowed; trailing blanks are ignored. The value is equal to the significand multiplied by two raised to the power of the exponent. If the optional sign is a minus, the value is negated.

The following shows input using the F edit descriptor:

```
Format     Input          Value
F8.5       123456789        123.45678
F8.5       -1234.567      -1234.56
F8.5       24.77E+2        2477.0
F5.2       1234567.89       123.45
```

Rules for Output Processing

On output, the F data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

For an internal value that is an IEEE infinity, the output field consists of blanks, if needed, followed by a sign (optional if the value is positive and descriptor SP is not in effect), followed by the letters 'Inf' or 'Infinity', right justified within the field.

If *w* is less than 3, the field is filled with asterisks; otherwise, if *w* is less than 8, 'Inf' is produced.

For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by the letters 'NaN' and optionally followed by one to *w* − 5 alphanumeric characters enclosed in parentheses, right justified within the field.

If *w* is less than 3, the field is filled with asterisks.

For an internal value that is neither an IEEE infinity nor an IEEE NaN, the *w* must be greater than or equal to *d*+3 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- At least one digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point

A negative value that is not zero but rounds to zero on output is displayed with a leading minus sign. For example, the value -0.00000001 in F5.1 format will be displayed as -0.0 rather than as 0.0. The setting of compiler option assume [no]std_minus0_rounding can affect this behavior.

The following shows output using the F edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Value              Output
F8.5           2.3547188       ^2.35472
F9.3        8789.7361          ^8789.736
F2.1          51.44            **
F10.4        -23.24352         ^^-23.2435
F5.2         325.013           ******
F5.2           -.2             -0.20
```

See Also

Forms for data edit descriptors
General rules for numeric editing
assume compiler option

E and D Editing

The E and D data edit descriptors transfer real values in exponential form. They take the following form:

Ew.d[Ee]

Dw.d

where *w* is the total field width, *d* is the number of places after the decimal point, and *e* is the number of digits in the exponent.

For the E edit descriptor, if *w* is zero, the processor selects the field width. If *e* is zero, the exponent part contains the minimum number of digits needed to represent the value of the exponent.

For the D edit descriptor, if the value of *w* is zero, the processor selects the field width.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the E and D data edit descriptors transfer *w* characters from an external field and assigns their real value to the corresponding I/O list item. The E and D descriptors interpret and assign input data in the same way as the F data edit descriptor. On input, *w* cannot be zero. The *e*, if present, has no effect on input.

The following shows input using the E and D edit descriptors (the symbol ^ represents a nonprinting blank character):

```
Format        Input              Value
E9.3          734.432E3            734432.0
E12.4         ^^1022.43E             1022.43E-6
E15.3         52.3759663^^^^^          52.3759663
E12.5         210.5271D+10         210.5271E10
BZ,D10.2      12345^^^^^         12345000.0D0
D10.2         ^^123.45^^              123.45D0
D15.3         367.4981763D+04           3.674981763D+06
```

If the I/O list item is single-precision real, the E edit descriptor treats the D exponent indicator as an E indicator.

Rules for Output Processing

On output, the E and D data edit descriptors transfer the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

If *w* is greater than zero, it should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- An optional zero to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
Ew.d	|exp| 99	E+nn	E-nn
	99 < |exp| 999	+nnn	-nnn
Ew.dEe	|exp| 10^e - 1	E+$n_1n_2...n_e$	E-$n_1n_2...n_e$
Dw.d	|exp| 99	D+nn or E+nn	D-nn or E-nn
	99 < |exp| 999	+nnn	-nnn

If an exponent exceeds its specified or implied width, or the number of characters produced exceeds the field width, the entire field of width *w* is filled with asterisks.

The exponent field width (*e*) is optional for the E edit descriptor; if omitted, the default value is 2. If *e* is specified, *w* should be greater than or equal to *d*+*e*+5, or zero.

NOTE
If *w* is greater than zero, it can be as small as *d* + 5 or *d* + *e* + 3, if the optional fields for the sign and the zero are omitted.

For an internal value that is an IEEE infinity or an IEEE NaN, the form of the output field is the same as for Fw.d.

A negative value that is not zero but rounds to zero on output is displayed with a leading minus sign. For example, the value -0.01 in "-5P,E20.5" format will be displayed as -0.00 rather than as 0.00. The setting of compiler option assume [no]std_minus0_rounding can affect this behavior.

The following shows output using the E and D edit descriptors (the symbol ^ represents a nonprinting blank character):

```
Format       Value               Output
E11.2        475867.222            ^^^0.48E+06
E11.5        475867.222               0.47587E+06
E12.3             0.00069           ^^^0.690E
E10.3            -0.5555              -0.556E+00
E5.3             56.12               *****
E14.5E4          -1.001              -0.10010E+0001
E13.3E6           0.000123            0.123E-000003
D14.3             0.0363          ^^^^^0.363D-01
D23.12         5413.87625793      ^^^^^0.541387625793D+04
D9.6              1.2             *********
```

See Also

Forms for data edit descriptors
General rules for numeric editing
Scale Factor Editing (P)
assume compiler option

EN Editing

The EN data edit descriptor transfers values by using engineering notation. It takes the following form:

ENw.d[Ee]

where *w* is the total field width, *d* is the number of places after the decimal point, and *e* is the number of digits in the exponent.

If *w* is zero, the processor chooses the field width. If *e* is present and zero, the exponent part contains the minimal number of digits needed to represent the exponent .

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the EN data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The EN descriptor interprets and assigns input data in the same way as the F data edit descriptor. *w* cannot be zero on input. *e*, if present, has no effect on input.

The following shows input using the EN edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Input           Value
EN11.3       ^^5.321E+00       5.32100
EN11.3       -600.00E-03       -.60000
EN12.3      ^^^3.150E-03        .00315
EN12.3      ^^^3.829E+03    3829.0
```

Rules for Output Processing

On output, the EN data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long if *w* is positive. The real value is output in engineering notation, where the decimal exponent is divisible by 3 and the absolute value of the significand is greater than or equal to 1 and less than 1000 (unless the output value is zero).

If *w* is greater than zero, it should be greater than or equal to *d*+9 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One to three digits to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
ENw.d	|exp| 99	E+nn	E-nn
	99 < |exp| 999	+nnn	-nnn
ENw.dEe	|exp| 10^e - 1	E+$n_1n_2...n_e$	E-$n_1n_2...n_e$

If an exponent exceeds its specified or implied width, or the number of characters produced exceeds the field width, the entire field of width *w* is filled with asterisks.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, *w* should be greater than or equal to *d* + *e* + 5, or zero.

For an internal value that is an IEEE infinity or an IEEE NaN, the form of the output field is the same as for Fw.d.

The following shows output using the EN edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Value          Output
EN11.2     475867.222     ^475.87E+03
EN11.5     475867.222     ***********
EN12.3          0.00069   ^690.000E-06
EN10.3         -0.5555    **********
EN11.2          0.0       ^000.00E-03
```

See Also

Forms for data edit descriptors
General rules for numeric editing

ES Editing

The ES data edit descriptor transfers values by using scientific notation. It takes the following form:

ESw.d[Ee]

where *w* is the total field width, *d* is the number of places after the decimal point, and *e* is the number of digits in the exponent.

If *w* is zero, the processor selects the field width. If *e* is present and zero, the exponent part contains the minimal number of digits needed to represent the exponent.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the ES data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The ES descriptor interprets and assigns input data in the same way as the F data edit descriptor. *w* cannot be zero for input. If *e* is present, it has no effect on input.

The following shows input using the ES edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Input           Value
ES11.3      ^^5.321E+00       5.32100
ES11.3       -6.000E-03       -.60000
ES12.3     ^^^3.150E-03        .00315
ES12.3     ^^^3.829E+03    3829.0
```

Rules for Output Processing

On output, the ES data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long if *w* is positive. The real value is output in scientific notation, where the absolute value of the significand is greater than or equal to 1 and less than 10 (unless the output value is zero).

If w is greater than zero, it should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
ESw.d	|exp| 99	E+nn	E-nn
	99 < |exp| 999	+nnn	-nnn
ESw.dEe	|exp| 10^e - 1	E+$n_1n_2...n_e$	E-$n_1n_2...n_e$

If an exponent exceeds its specified or implied width, or the number of characters produced exceeds the field width, the entire field of width *w* is filled with asterisks.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d* + *e* + 5.

For an internal value that is an IEEE infinity or an IEEE NaN, the form of the output field is the same as for Fw.d.

The following shows output using the ES edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Value          Output
ES11.2     473214.356     ^^^4.73E+05
ES11.5     473214.356     4.73214E+05
ES12.3          0.00069   ^^^6.900E-04
ES10.3         -0.5555    -5.555E-01
ES11.2          0.0       ^0.000E+00
```

See Also

Forms for data edit descriptors

General rules for numeric editing

EX Editing

The EX data edit descriptor transfers real values represented as hexadecimal-significand numbers. It takes the following form:

EXw.d[Ee]

w is the external field width, unless it is zero. *d* is the width of the fractional part of the number, unless it is 0.

If *w* or *d* are zero, the processor picks the external field width or the width of the fractional part, respectively. *d* cannot be zero if the radix of the internal value is not a power of two. The hexadecimal point appears after the first hexadecimal digit and it represented by the decimal symbol.

e, if present and nonzero, is the number of digits in the exponent. If E*e* is not present, or *e* is zero, the exponent contains the minimum number of digits needed to represent the exponent. *e* is ignored on input.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

The form and interpretation is the same as that for Fw.d editing.

Rules for Output Processing

The form of the external field for an internal value that is an IEEE infinity or NaN is the same as for Fw.d. Otherwise, it takes the form:

$[+|-]0X_{x0.x1x2...exp}$

where the plus or minus sign is optional, the period signifies the decimal signal, x0x1x2 are the most significant hexadecimal digits after rounding if d is nonzero, and *exp* is the exponent.

For EXw.dEe with *e* greater than zero, the form is $P[+|-]_{z1z2...ze}$. For EXw.d and EXw.dE0, the form of the exponent is $P[+|-]_{z1z2...zn}$ where *n* is the minimum number of digits required to represent the exponent. The exponent sign is always produced and is plus if the exponent is zero. The choice of the binary exponent is processor dependent.

The following shows possible output using the EX edit descriptor if SS is in effect:

```
Format     Value          Output
EX10.2       42.5          0XA.A0P+2
EX0.0E1    6502.0         0XC.B3P+9
EX9.0E2      -0.0         -0X0.P+00
```

See Also

F Editing
General rules for numeric editing

G Editing

The G data edit descriptor for generalized editing can be used for input or output with any intrinsic type. It takes the following forms:

G*w*

G*w*.*d*

G*w*.*d*E*e*

where *w* is the total field width, *d* is the number of places after the decimal point, and *e* is the number of digits in the exponent.

If *w* is 0, the field width is selected by the processor. If *w* is zero, you can only specify forms G0 or G0.*d*.

If *w* is nonzero, *d* must be specified.

If *e* is present and zero, the exponent part contains the minimal number of digits needed to represent the exponent. For integer, character, and logical data types *d* and *e* are ignored.

When used to specify I/O for integer data, the G*w*, G*w*.*d* and G*w*.*d*E*e* edit descriptors follow the rules for I*w* editing.

When used to specifiy I/O for logical data, the G*w*.*d* and G*w*.*d*E*e* edit descriptors with nonzero *w* follow the rules for L*w* editing. On output, if *w* is 0, the G*w* and G*w*.*d* edit descriptors follow the rules for L1 editing.

When used to specify I/O for character data, the G*w*.*d* and G*w*.*d*.E*e* edit descriptors with nonzero *w* follows the same rules as A*w* editing. For output, when *w* is zero, the G*w* and G*w*.*d* edit descriptors follow the rules for A editing when no *w* is specified.

Rules for Real Input Processing

On input, the G data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The G descriptor interprets and assigns input data in the same way as the F data edit descriptor. *w* cannot be zero on input. If *e* is present, it has no effect on input.

Rules for Real Output Processing

The form in which the value is written depends on the magnitude of the internal value being edited. N is the magnitude of the internal value and *r* is the rounding mode value defined in the table below. If 0 < N < 0.1 - r x 10^{-d-1} or N >= 10^{d} - r, or N is identically 0, *w* is nonzero, and *d* is 0, G*w*.*d* output editing is the same as k PE*w*.*d* output editing and G*w*.*d* E*e* output editing is the same as k PE*w*.*d* E*e* output editing, where k is the scale factor. If 0.1 - r x 10^{-d-1} <= N < 10^{d} - r or N is identically 0 and *d* is not zero, the scale factor has no effect, and the value of N determines the editing as follows:

Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
N = 0	F(w - n).(d -1), n('b')
0.1 - r x 10^{-d-1} <= N < 1 - r x 10^{-d}	F(w - n).d, n('b')
1 - r x 10^{-d} <= N < 10 - r x 10^{-d+1}	F(w - n).(d -1), n('b')
10 - r x 10^{-d+1} <= N < 100 - r x 10^{-d+2}	F(w - n).(d -2), n('b')
.	.
.	.
.	.
10^{d-2} - r x 10^{-2} <= N < 10^{d-1} - r x 10^{-1}	F(w - n).1, n('b')
10^{d-1} - r x 10^{-1} <= N < 10^{d} - r	(w - n).0, n('b')

The 'b' is a blank following the numeric data representation. For Gw.d, n('b') is 4 blanks. For Gw.dEe, n('b') is *e*+2 blanks.

The *r* is defined for each I/O rounding mode as follows:

Rounding Mode	*r*
COMPATIBLE	0.5
NEAREST	0.5 if the higher value is even -0.5 if the lower value is even
UP	1

Rounding Mode	*r*
DOWN	0
ZERO	1 if the internal value is negative 0 if the internal value is positive

Note that the scale factor has no effect on output unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

If *w* is greater than zero, it should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The 4-digit or *e*+2-digit exponent

If *e* is specified and positive, *w* should be greater than or equal to *d* + *e* + 5 if *w* is positive.

If an exponent exceeds its specified or implied width, or the number of characters produced exceeds the field width, the entire field of width *w* is filled with asterisks. However, the field width is not filled with asterisks if the field width is exceeded when optional characters are omitted.

For an internal value that is an IEEE infinity or an IEEE NaN, the form of the output field is the same as for Fw.d.

The following shows output using the G edit descriptor and compares it to output using equivalent F editing (the symbol ^ represents a nonprinting blank character):

```
Value              Format    Output with G     Format    Output with F
       0.01234567  G13.6     ^0.123457E-01     F13.6     ^^^^^0.012346
      -0.12345678  G13.6     -0.123457^^^^     F13.6     ^^^^-0.123457
       1.23456789  G13.6     ^^1.23457^^^^     F13.6     ^^^^^1.234568
      12.34567890  G13.6     ^^12.3457^^^^     F13.6     ^^^^12.345679
     123.45678901  G13.6     ^^123.457^^^^     F13.6     ^^^123.456789
   -1234.56789012  G13.6     ^-1234.57^^^^     F13.6     ^-1234.567890
   12345.67890123  G13.6     ^^12345.7^^^^     F13.6     ^12345.678901
  123456.78901234  G13.6     ^^123457.^^^^     F13.6     123456.789012
-1234567.89012345  G13.6     -0.123457E+07     F13.6     ************
```

If *w* is zero, the G*w* and G*w*.*d* edit descriptors follow the rules for the G.*w*.*d*E*e* edit descriptors on output, but with leading and trailing blanks removed.

See Also

Forms for data edit descriptors
General rules for numeric editing
I data edit descriptor
L data edit descriptor
A data edit descriptor
Scale Factor Editing (P)

Complex Editing

A complex value is an ordered pair of real values. Complex editing is specified by a pair of real edit descriptors, using any combination of the forms: Fw.d, Ew.d[Ee], Dw.d, ENw.d[Ee], ESw.d[Ee], or Gw.d[Ee].

Rules for Input Processing

On input, the two successive fields are read and assigned to the corresponding complex I/O list item as its real and imaginary part, respectively.

The following shows input using complex editing:

```
Format          Input                Value
F8.5,F8.5       1234567812345.67     123.45678, 12345.67
E9.1,F9.3       734.432E8123456789   734.432E8, 123456.789
```

Rules for Output Processing

On output, the two parts of the complex value are transferred under the control of repeated or successive real edit descriptors. The two parts are transferred consecutively without punctuation or blanks, unless control or character string edit descriptors are specified between the pair of real edit descriptors.

The following shows output using complex editing (the symbol ^ represents a nonprinting blank character):

```
Format              Value                  Output
2F8.5               2.3547188, 3.456732    ^2.35472 ^3.45673
E9.2,'^,^',E5.3     47587.222, 56.123      ^0.48E+06^,^*****
```

See Also

Forms for data edit descriptors
General rules for numeric editing
General Rules for Complex Constants for details on complex constants

Logical Editing (L)

The L data edit descriptor transfers logical values. It takes the following form:

Lw

The specified I/O list item must be of type logical or integer.

The G edit descriptor can be used to edit logical data; it follows the same rules as L*w*.

Rules for Input Processing

On input, the L data edit descriptor transfers *w* characters from an external field and assigns their logical value to the corresponding I/O list item. The value assigned depends on the external field data, as follows:

- .TRUE. is assigned if the first nonblank character is .T, T, .t, or t. The logical constant .TRUE. is an acceptable input form.
- .FALSE. is assigned if the first nonblank character is .F, F. .f, or f, or the entire field is filled with blanks. The logical constant .FALSE. is an acceptable input form.

If an other value appears in the external field, an error occurs.

Rules for Output Processing

On output, the L data edit descriptor transfers the following to an external field that is *w* characters long: w - 1 blanks, followed by a T or F (if the value is .TRUE. or .FALSE., respectively).

The following shows output using the L edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Value          Output
L5          .TRUE.         ^^^^T
L1          .FALSE.        F
```

See Also

Forms for data edit descriptors

Character Editing (A)

The A data edit descriptor transfers character or Hollerith values. It takes the following form:

A[w]

If the corresponding I/O list item is of type character, character data is transferred. If the list item is of any other type, Hollerith data is transferred.

The G edit descriptor can be used to edit character data; it follows the same rules as A*w*.

Rules for Input Processing

On input, the A data edit descriptor transfers *w* characters from an external field and assigns them to the corresponding I/O list item.

The maximum number of characters that can be stored depends on the size of the I/O list item, as follows:

- For character data, the maximum size is the length of the corresponding I/O list item.
- For noncharacter data, the maximum size depends on the data type, as shown in the following table:

Size Limits for Noncharacter Data Using A Editing

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL(1) or LOGICAL*1	1
LOGICAL(2) or LOGICAL*2	2
LOGICAL(4) or LOGICAL*4	4
LOGICAL(8) or LOGICAL*8	8
INTEGER(1) or INTEGER*1	1
INTEGER(2) or INTEGER*2	2
INTEGER(4) or INTEGER*4	4
INTEGER(8) or INTEGER*8	8
REAL(4) or REAL*4	4
DOUBLE PRECISION	8
REAL(8) or REAL*8	8
REAL(16) or REAL*16	16
COMPLEX(4) or COMPLEX*8[1]	8
DOUBLE COMPLEX[1]	16
COMPLEX(8) or COMPLEX*16[1]	16
COMPLEX(16) or COMPLEX*32[1]	32

[1] Complex values are treated as pairs of real numbers, so complex editing requires a pair of edit descriptors. (See Complex Editing.)

If *w* is equal to or greater than the length (*len*) of the input item, the rightmost characters are assigned to that item. The leftmost excess characters are ignored.

If *w* is less than *len*, or less than the number of characters that can be stored, *w* characters are assigned to the list item, left-justified, and followed by trailing blanks.

The following shows input using the A edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Input       Value          Data Type
A6          PAGE^#      #              CHARACTER(LEN=1)
A6          PAGE^#      E^#            CHARACTER(LEN=3)
A6          PAGE^#      PAGE^#         CHARACTER(LEN=6)
A6          PAGE^#      PAGE^#^^       CHARACTER(LEN=8)
A6          PAGE^#      #              LOGICAL(1)
A6          PAGE^#      ^#             INTEGER(2)
A6          PAGE^#      GE^#           REAL(4)
A6          PAGE^#      PAGE^#^^       REAL(8)
```

Rules for Output Processing

On output, the A data edit descriptor transfers the contents of the corresponding I/O list item to an external field that is *w* characters long.

If *w* is greater than the size of the list item, the data is transferred to the output field, right-justified, with leading blanks. If *w* is less than or equal to the size of the list item, the leftmost *w* characters are transferred.

The following shows output using the A edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Value          Output
A5          OHMS           ^OHMS
A5          VOLTS          VOLTS
A5          AMPERES        AMPER
```

See Also

Forms for data edit descriptors

Defined I/O Editing (DT)

The DT edit descriptor passes a character string and integer array to a defined I/O procedure. It takes the following form:

DT [*string*] [(*v-list*)]

For more information on this edit descriptor, see DT Edit Descriptor in User-Defined I/O.

Default Widths for Data Edit Descriptors

If *w* (the field width) is omitted for the data edit descriptors, the system applies default values. For the real data edit descriptors, the system also applies default values for *d* (the number of characters to the right of the decimal point), and *e* (the number of characters in the exponent).

These defaults are based on the data type of the I/O list item, and are listed in the following table:

Default Widths for Data Edit Descriptors

Edit Descriptor	Data Type of I/O List Item	w
I, B, O, Z, G	BYTE	7

Edit Descriptor	Data Type of I/O List Item	w
	INTEGER(1), LOGICAL(1)	7
	INTEGER(2), LOGICAL(2)	7
	INTEGER(4), LOGICAL(4)	12
	INTEGER(8), LOGICAL(8)	23
O, Z	REAL(4)	12
	REAL(8)	23
	REAL(16)	44
	CHARACTER*len	MAX(7, 3*len)
L, G	LOGICAL(1), LOGICAL(2), LOGICAL(4), LOGICAL(8)	2
F, E, ES, G, D	REAL(4), COMPLEX(4)	15 **d**: 7 **e**: 2
	REAL(8), COMPLEX(8)	25 **d**: 16 **e**: 2
	REAL(16), COMPLEX(16)	42 **d**: 33 **e**: 3
EN	REAL(4), COMPLEX(4)	15 **d**: 6 **e**: 2
	REAL(8), COMPLEX(8)	25 **d**: 16 **e**: 2
	REAL(16), COMPLEX(16)	42 **d**: 32 **e**: 3
A[1], G	LOGICAL(1)	1
	LOGICAL(2), INTEGER(2)	2
	LOGICAL(4), INTEGER(4)	4
	LOGICAL(8), INTEGER(8)	8
	REAL(4), COMPLEX(4)	4
	REAL(8), COMPLEX(8)	8
	REAL(16), COMPLEX(16)	16
	CHARACTER*len	len

[1] The default is the actual length of the corresponding I/O list item.

Terminating Short Fields of Input Data

On input, an edit descriptor such as Fw.d specifies that *w* characters (the field width) are to be read from the external field.

If the field contains fewer than *w* characters, the input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data.

Padding Short Fields

You can use the OPEN statement specifier PAD='YES' to indicate blank padding for short fields of input data. However, blanks can be interpreted as blanks *or* zeros, depending on which default behavior is in effect at the time. Consider the following:

```
READ (2, '(I5)') J
```

If 3 is input for J, the value of J will be 30000 or 3 depending on which default behavior is in effect (BLANK='NULL' or BLANK='ZERO'). This can give unexpected results.

To ensure that the desired behavior is in effect, explicitly specify the BN or BZ edit descriptor. For example, the following ensures that blanks are interpreted as blanks (and not as zeros):

```
READ (2, '(BN, I5)') J
```

Using Commas to Separate Input Data

You can use a comma to terminate a short data field. The comma has no effect on the *d* part (the number of characters to the right of the decimal point) of the specification.

The comma overrides the *w* specified for the I, B, O, Z, F, E, D, EN, ES, G, and L edit descriptors. For example, suppose the following statements are executed:

```
      READ (5,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

Suppose a record containing the following values is read:

```
1, -2, 1.0, 35
```

The following assignments occur:

```
I = 1
J = -2
A = 1.0
B = 0.35
```

A comma can only terminate fields less than *w* characters long. If a comma follows a field of *w* or more characters, the comma is considered part of the next field.

A null (zero-length) field is designated by two successive commas, or by a comma after a field of *w* characters. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.0D0, 0.0Q0, or .FALSE..

See Also

General Rules for Numeric Editing

Control Edit Descriptors

A control edit descriptor either directly determines how text is displayed or affects the conversions performed by subsequent data edit descriptors.

Forms for Control Edit Descriptors

A control edit descriptor takes one of the following forms:

c

cn

nc

c	Is one of the following format codes: T, TL, TR, X, S, SP, SS, BN, BZ, P, RU, RD, RZ, RN, RC, RP, DC, DP, :, /, \, $, and Q.
n	Is a number of character positions. It must be a positive integer literal constant or a variable format expression. No kind parameter can be specified. It cannot be a named constant. The range of *n* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. Actual useful ranges may be constrained by record sizes (RECL) and the file system.

Description

In general, control edit descriptors are nonrepeatable. The only exception is the slash (/) edit descriptor, which can be preceded by a repeat specification or a * indicating an unlimited repeat count.

The control edit descriptors have the following specific forms:

Positional:	Tn, TLn, TRn, and nX
Sign:	S, SP, and SS
Blank interpretation:	BN and BZ
Rounding mode:	RU, RD, RZ, RN, RC, and RP
Decimal mode:	DC and DP
Scale factor:	kP
Miscellaneous:	:, /, \, $, and Q

The P edit descriptor is an exception to the general control edit descriptor syntax. It is preceded by a scale factor, rather than a character position specifier.

Control edit descriptors can be grouped in parentheses and preceded by a group repeat specification.

See Also

Group repeat specifications
Format Specifications

Positional Editing

The T, TL, TR, and X edit descriptors specify the position where the next character is transferred to or from a record.

On output, these descriptors do not themselves cause characters to be transferred and do not affect the length of the record. If characters are transferred to positions at or after the position specified by one of these descriptors, positions skipped and not previously filled are filled with blanks. The result is as if the entire record was initially filled with blanks.

The TR and X edit descriptors produce the same results.

See Also

Forms for Control Edit Descriptors

T Editing

The T edit descriptor specifies a character position in an I/O record. It takes the following form:

Tn

The *n* is a positive integer literal constant (with no kind parameter) indicating the character position of the record, relative to the left tab limit.

On input, the T descriptor positions the external record at the character position specified by *n*. On output, the T descriptor indicates that data transfer begins at the *n*th character position of the external record.

Examples

In the following examples, the symbol ^ represents a nonprinting blank character.

Suppose a file has a record containing the value `ABC^^^XYZ`, and the following statements are executed:

```
      READ (11,10) VALUE1, VALUE2
 10   FORMAT (T7,A3,T1,A3)
```

The values read first are XYZ, then ABC.

Suppose the following statements are executed:

```
     PRINT 25
25   FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

The following line is printed at the positions indicated:

```
Position 20                    Position 50
   |                              |
   COLUMN 1                       COLUMN 2
```

Note that the first character of the record printed was reserved as a control character.

See Also

Printing of Formatted Records

TL Editing

The TL edit descriptor specifies a character position to the *left* of the current position in an I/O record. It takes the following form:

TLn

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the left of the current character.

If *n* is greater than or equal to the current position, the next character accessed is the first character of the record.

TR Editing

The TR edit descriptor specifies a character position to the *right* of the current position in an I/O record. It takes the following form:

TRn

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

X Editing

The X edit descriptor specifies a character position to the right of the current position in an I/O record. It takes the following form:

nX

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

On output, the X edit descriptor does not output any characters when it appears at the end of a format specification; for example:

```
     WRITE (6,99) K
99   FORMAT ('^K=',I6,5X)
```

Note that the symbol ^ represents a nonprinting blank character. This example writes a record of only 9 characters. To cause *n* trailing blanks to be output at the end of a record, specify a format of `n('^')`.

Sign Editing

The SP, SS and S edit descriptors control the output of the optional plus (+) sign within numeric output fields. These descriptors have no effect during execution of input statements.

These specifiers correspond to the SIGN= specifier values PLUS, SUPPRESS, and PROCESSOR_DEFINED, respectively.

Within a format specification, a sign editing descriptor affects all subsequent I, F, E, EN, ES, EX, D, and G descriptors until another sign editing descriptor occurs.

Examples

Consider the following:

```
      INTEGER i
      REAL r

!     The following statements write:
!     251 +251 251 +251 251
      i = 251
      WRITE (*, 100) i, i, i, i, i
100   FORMAT (I5, SP, I5, SS, I5, SP, I5, S, I5)

!     The following statements write:
!     0.673E+4 +.673E+40.673E+4 +.673E+40.673E+4
      r = 67.3E2
      WRITE (*, 200) r, r, r, r, r
200   FORMAT (E8.3E1, 1X, SP, E8.3E1, SS, E8.3E1, 1X, SP, &
     &            E8.3E1, S, E8.3E1)
```

See Also

Forms for Control Edit Descriptors

SP Editing

The SP edit descriptor causes the processor to produce a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

SP

SS Editing

The SS edit descriptor causes the processor to *suppress* a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

SS

S Editing

The S edit descriptor restores the plus sign as optional for all subsequent positive numeric fields. It takes the following form:

S

The S edit descriptor restores to the processor the discretion of producing plus characters on an optional basis.

Blank Editing

The BN and BZ descriptors control the interpretation of embedded and trailing blanks within numeric input fields. These descriptors have no effect during execution of output statements.

Within a format specification, a blank editing descriptor affects all subsequent I, B, O, Z, F, E, EN, ES, D, and G descriptors until another blank editing descriptor occurs.

The blank editing descriptors override the effect of the BLANK specifier during execution of a particular input data transfer statement. (For more information, see the BLANK specifier in OPEN statements.)

See Also

Forms for Control Edit Descriptors

BN Editing

The BN edit descriptor causes the processor to *ignore* all embedded and trailing blanks in numeric input fields. It takes the following form:

BN

The input field is treated as if all blanks have been removed and the remainder of the field is right-justified. An all-blank field is treated as zero.

Examples

If an input field formatted as a six-digit integer (I6) contains '2 3 4', it is interpreted as ' 234'.

Consider the following code:

```
      READ (*, 100) n
100   FORMAT (BN, I6)
```

If you enter any one of the following three records and terminate by pressing Enter, the READ statement interprets that record as the value 123:

```
   123
123
123   456
```

Because the repeatable edit descriptor associated with the I/O list item n is I6, only the first six characters of each record are read (three blanks followed by 123 for the first record, and 123 followed by three blanks for the last two records). Because blanks are ignored, all three records are interpreted as 123.

The following example shows the effect of BN editing with an input record that has fewer characters than the number of characters specified by the edit descriptors and *iolist*. Suppose you enter 123 and press Enter in response to the following READ statement:

```
READ (*, '(I6)') n
```

The I/O system is looking for six characters to interpret as an integer number. You have entered only three, so the first thing the I/O system does is to pad the record 123 on the right with three blanks. With BN editing in effect, the nonblank characters (123) are right-aligned, so the record is equal to 123.

BZ Editing

The BZ edit descriptor causes the processor to *interpret* all embedded and trailing blanks in numeric input fields as zeros. It takes the following form:

BZ

Examples

The input field ' 23 4 ' is interpreted as ' 23040'. If ' 23 4' is entered, the formatter adds one blank to pad the input to the six-digit integer format (I6), but this extra space is ignored, and the input is interpreted as ' 2304 '. The blanks following the E or D in real-number input are ignored, regardless of the form of blank interpretation in effect.

Suppose you enter 123 and press Enter in response to the following READ statement:

```
READ (*, '(I6)') n
```

The I/O system is looking for six characters to interpret as an integer number. You have entered only three, so the first thing the I/O system does is to pad the record 123 on the right with three blanks. If BZ editing is in effect, those three blanks are interpreted as zeros, and the record is equal to 123000.

Round Editing

The RU, RD, RZ, RN, RC, and RP edit descriptors temporarily change the I/O rounding mode for a connection.

These forms of rounding correspond to the ROUND= specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE, and PROCESSOR DEFINED, respectively. Rounding conforms to the ISO/IEC/IEEE 60559:2011 standard.

The I/O rounding mode affects the conversion of real and complex values in formatted I/O. It affects only D, E, EN, ES, EX, F, and G editing.

Each descriptor continues to be in effect until a different round editing descriptor is encountered or until the end of the current I/O statement.

See Also

Forms for Control Edit Descriptors

RU Editing

The RU edit descriptor causes rounding to the smallest value that is greater than or equal to the original value. It takes the following form:

RU

RD Editing

The RD edit descriptor causes rounding to the largest representable value that is less than or equal to the original value. It takes the following form:

RD

RZ Editing

The RZ edit descriptor causes rounding to the value closest to the original value, but not greater in magnitude. It takes the following form:

RZ

RN Editing

The RN edit descriptor selects NEAREST rounding as specified by the ISO/IEC/IEEE 60559:2011 standard. It takes the following form:

RN

RC Editing

The RC edit descriptor causes rounding to the closer of the two nearest representable values. If the value is halfway between the two values, the one chosen is the one farther from zero. It takes the following form:

RC

RP Editing

The RP edit descriptor causes rounding to be determined by the default settings in the processor, which may correspond to one of the other modes. It takes the following form:

RP

Decimal Editing

The DC and DP edit descriptors temporarily change the decimal edit mode for a connection.

These specifiers correspond to the DECIMAL= specifier values COMMA and POINT, respectively.

The decimal editing mode controls the representation of the decimal symbol during conversion of real and complex values in formatted I/O. It affects only D, E, EN, ES, F, and G editing.

Each descriptor continues to be in effect until a different decimal editing descriptor is encountered or until the end of the current I/O statement.

See Also

Forms for Control Edit Descriptors

DC Editing

The DC edit descriptor changes the decimal editing mode for a connection to a decimal comma. It takes the following form:

DC

Note that during list-directed I/O, a semicolon is used as a value separator in place of a comma.

DP Editing

The DP edit descriptor causes rounding to be determined by the default settings in the processor, which may correspond to one of the other modes. It takes the following form:

DP

Scale-Factor Editing (P)

The P edit descriptor specifies a scale factor, which moves the location of the decimal point in real values and the two real parts of complex values. It takes the following form:

*k*P

The *k* is a signed (sign is optional if positive), integer literal constant specifying the number of positions, to the left or right, that the decimal point is to move (the scale factor). The range of *k* is -128 to 127.

At the beginning of a formatted I/O statement, the value of the scale factor is zero. If a scale editing descriptor is specified, the scale factor is set to the new value, which affects all subsequent real edit descriptors until another scale editing descriptor occurs.

To reinstate a scale factor of zero, you must explicitly specify 0P.

Format reversion does not affect the scale factor. (For more information on format reversion, see Interaction Between Format Specifications and I/O Lists.)

Rules for Input Processing

On input, a positive scale factor moves the decimal point to the left, and a negative scale factor moves the decimal point to the right. (On output, the effect is the reverse.)

On input, when an input field using an F, E, D, EN, ES, EX, or G real edit descriptor contains an explicit exponent, the scale factor has no effect. Otherwise, the internal value of the corresponding I/O list item is equal to the external field data multiplied by 10^{-k}. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right.

The scale factor applies to decimal numbers without an exponent. For hexadecimal-significand numbers, the exponent is mandatory so the scale factor has no effect.

The following shows input using the P edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format    Input         Value
3PE10.5   ^^^37.614^         .037614
3PE10.5   ^^37.614E2     3761.4
-3PE10.5  ^^^^37.614    37614.0
```

The scale factor must precede the first real edit descriptor associated with it, but it need not immediately precede the descriptor. For example, the following all have the same effect:

```
  (3P, I6, F6.3, E8.1)
  (I6, 3P, F6.3, E8.1)
  (I6, 3PF6.3, E8.1)
```

Note that if the scale factor immediately precedes the associated real edit descriptor, the comma separator is optional.

Rules for Output Processing

On output, a positive scale factor moves the decimal point to the right, and a negative scale factor moves the decimal point to the left. (On input, the effect is the reverse.)

On output, the effect of the scale factor depends on which kind of real editing is associated with it, as follows:

- For F editing, the external value equals the internal value of the I/O list item multiplied by 10^{k}. This changes the magnitude of the data.
- For E and D editing, the external decimal field of the I/O list item is multiplied by 10^{k}, and *k* is subtracted from the exponent. This changes the form of the data.

  A positive scale factor decreases the exponent; a negative scale factor increases the exponent.

  For a positive scale factor, *k* must be less than *d* + 2 or an output conversion error occurs.
- For G editing, the scale factor has no effect if the magnitude of the data to be output is within the effective range of the descriptor (the G descriptor supplies its own scaling).

  If the magnitude of the data field is outside G descriptor range, E editing is used, and the scale factor has the same effect as E output editing.
- For EN, ES, and EX editing, the scale factor has no effect.

The following shows output using the P edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Value         Output
1PE12.3    -270.139      ^^-2.701E+02
1P,E12.2   -270.139      ^^^-2.70E+02
-1PE12.2   -270.139      ^^^-0.03E+04
```

Examples

The following shows a FORMAT statement containing a scale factor:

```
     DIMENSION A(6)
     DO 10 I=1,6
10   A(I) = 25.
     WRITE (6, 100) A
100  FORMAT(' ', F8.2, 2PF8.2, F8.2)
```

The preceding statements produce the following results:

```
   25.00  2500.00  2500.00
 2500.00  2500.00  2500.00
```

The following code uses scale-factor editing when reading:

```
      READ (*, 100) a, b, c, d
100   FORMAT (F10.6, 1P, F10.6, F10.6, -2P, F10.6)

      WRITE (*, 200) a, b, c, d
200   FORMAT (4F11.3)
```

If the following data is entered:

```
12340000  12340000  12340000  12340000
   12.34     12.34     12.34     12.34
 12.34e0   12.34e0   12.34e0   12.34e0
 12.34e3   12.34e3   12.34e3   12.34e3
```

The program's output is:

```
   12.340      1.234      1.234   1234.000
   12.340      1.234      1.234   1234.000
   12.340     12.340     12.340     12.340
12340.000  12340.000  12340.000  12340.000
```

The next code shows scale-factor editing when writing:

```
      a = 12.34

      WRITE (*, 100) a, a, a, a, a, a
100   FORMAT (1X, F9.4, E11.4E2, 1P, F9.4, E11.4E2, &
     &        -2P, F9.4, E11.4E2)
```

This program's output is:

```
12.3400 0.1234E+02 123.4000 1.2340E+01   0.1234 0.0012E+04
```

See Also

Forms for Control Edit Descriptors

Slash Editing (/)

The slash edit descriptor terminates data transfer for the current record and starts data transfer for a new record. It takes the following form:

[r]/

The *r* is a repeat specification. It must be a positive default integer literal constant; no kind parameter can be specified.

The range of *r* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. If *r* is omitted, it is assumed to be 1.

Multiple slashes cause the system to skip input records or to output blank records, as follows:

- When *n* consecutive slashes appear between two edit descriptors, *n* - 1 records are skipped on input, or *n* - 1 blank records are output. The first slash terminates the current record. The second slash terminates the first skipped or blank record, and so on.
- When *n* consecutive slashes appear at the beginning or end of a format specification, *n* records are skipped or *n* blank records are output, because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example, suppose the following statements are specified:

```
    WRITE (6,99)
99  FORMAT ('1',T51,'HEADING LINE'//T51,'SUBHEADING LINE'//)
```

The following lines are written:

```
                  Column 50, top of page         |         HEADING LINE
(blank line)         SUBHEADING LINE
(blank line)
(blank line)
```

Note that the first character of the record printed was reserved as a control character (see Printing of Formatted Records).

Examples

```
!     The following statements write spreadsheet column and row labels:
      WRITE (*, 100)
100   FORMAT ('   A     B     C     D     E'                              &
     &        /,' 1',/,' 2',/,' 3',/,' 4',/,' 5')
```

The above example generates the following output:

```
  A     B     C     D     E
1
2
3
4
5
```

See Also

Forms for Control Edit Descriptors

Colon Editing (:)

The colon edit descriptor terminates format control if there are no more items in the I/O list.

Examples

Suppose the following statements are specified:

```
    PRINT 1,3
    PRINT 2,13
1   FORMAT (' I=',I2,' J=',I2)
2   FORMAT (' K=',I2,:,' L=',I2)
```

The above code causes the following lines to be written (the symbol ^ represents a nonprinting blank character):

```
I=^3^J=
K=13
```

The following shows another example:

```
!     The following example writes a= 3.20 b= .99
      REAL a, b, c, d
      DATA a /3.2/, b /.9871515/
```

```
      WRITE (*, 100) a, b
100   FORMAT (' a=', F5.2, :, ' b=', F5.2, :, &
     &        ' c=', F5.2, :, ' d=', F5.2)
      END
```

See Also

Forms for Control Edit Descriptors

Dollar-Sign ($) and Backslash (\ ) Editing

The dollar sign and backslash edit descriptors modify the output of carriage control specified by the first character of the record. They only affect carriage control for formatted files, and have no effect on input.

If the first character of the record is a blank or a plus sign (+), the dollar sign and backslash descriptors suppress carriage return (after printing the record).

For terminal device I/O, when this trailing carriage return is suppressed, a response follows output on the same line. For example, suppose the following statements are specified:

```
      TYPE 100
100   FORMAT (' ENTER RADIUS VALUE ',$)
      ACCEPT 200, RADIUS
200   FORMAT (F6.2)
```

The following prompt is displayed:

```
 ENTER RADIUS VALUE
```

Any response (for example, "12.") is then displayed on the same line:

```
 ENTER RADIUS VALUE    12.
```

If the first character of the record is 0, 1, or ASCII NUL, the dollar sign and backslash descriptors have no effect.

Consider the following:

```
     CHARACTER(20) MYNAME
     WRITE (*,9000)
9000 FORMAT ('Please type your name:',\)
     READ  (*,9001) MYNAME
9001 FORMAT (A20)
     WRITE (*,9002) ' ',MYNAME
9002 FORMAT (1X,A20)
```

This example advances two lines, prompts for input, awaits input on the same line as the prompt, and prints the input.

The following shows the same example using Fortran standard constructs:

```
     CHARACTER(20) MYNAME
     WRITE (*,9000, ADVANCE='NO')
9000 FORMAT ('Please type your name:')
     READ  (*,9001) MYNAME
9001 FORMAT (A20)
     WRITE (*,9002) ' ',MYNAME
9002 FORMAT (1X,A20)
```

See Also

Forms for Control Edit Descriptors

Character Count Editing (Q)

The character count edit descriptor returns the remaining number of characters in the current input record.

The corresponding I/O list item must be of type integer or logical. For example, suppose the following statements are specified:

```
      READ (4,1000) XRAY, KK, NCHRS, (ICHR(I), I=1,NCHRS)
1000  FORMAT (E15.7,I4,Q,(80A1))
```

Two fields are read into variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS, and exactly that many characters are read into the array ICHR. (This instruction can fail if the record is longer than 80 characters.)

If you place the character count descriptor first in a format specification, you can determine the length of an input record.

On output, the character count edit descriptor causes the corresponding I/O list item to be skipped.

Examples

Consider the following:

```
      CHARACTER ICHAR(80)
      READ (4, 1000) XRAY, K, NCHAR, (ICHAR(I), I= 1, NCHAR)
1000  FORMAT (E15.7, I4, Q, 80A1)
```

The preceding input statement reads the variables XRAY and K. The number of characters remaining in the record is NCHAR, specified by the Q edit descriptor. The array ICHAR is then filled by reading exactly the number of characters left in the record. (Note that this instruction will fail if NCHAR is greater than 80, the length of the array ICHAR.) By placing Q in the format specification, you can determine the actual length of an input record.

Note that the length returned by Q is the number of characters left in the record, not the number of reals or integers or other data types. The length returned by Q can be used immediately after it is read and can be used later in the same format statement or in a variable format expression. (See Variable Format Expressions.)

Assume the file Q.DAT contains:

```
1234.567Hello, Q Edit
```

The following program reads in the number REAL1, determines the characters left in the record, and reads those into STR:

```
      CHARACTER STR(80)
      INTEGER LENGTH
      REAL REAL1
      OPEN (UNIT = 10, FILE = 'Q.DAT')
100   FORMAT (F8.3, Q, 80A1)
      READ (10, 100) REAL1, LENGTH, (STR(I), I=1, LENGTH)
      WRITE(*,'(F8.3,2X,I2,2X,<LENGTH>A1)') REAL1, LENGTH, (STR(I), &
     & I= 1, LENGTH)
      END
```

The output on the screen is:

```
1234.567  13  Hello, Q Edit
```

A READ statement that contains only a Q edit descriptor advances the file to the next record. For example, consider that Q.DAT contains the following data:

```
abcdefg
abcd
```

Consider it is then READ with the following statements:

```
      OPEN (10, FILE = "Q.DAT")
      READ(10, 100) LENGTH
100   FORMAT(Q)
      WRITE(*,'(I2)') LENGTH
      READ(10, 100) LENGTH
      WRITE(*,'(I2)') LENGTH
      END
```

The output to the screen would be:

```
7
4
```

See Also

Forms for Control Edit Descriptors

Character String Edit Descriptors

Character string edit descriptors control the output of character strings. The character string edit descriptors are the character constant and H edit descriptor.

Although no string edit descriptor can be preceded by a repeat specification, a parenthesized group of string edit descriptors can be preceded by a repeat specification or a * indicating an unlimited repeat count.

See Also

Nested and Group Repeat Specifications

Character Constant Editing

The character constant edit descriptor causes a character string to be output to an external record. It takes one of the following forms:

'*string*'

"*string*"

The *string* is a character literal constant; no kind parameter can be specified. Its length is the number of characters between the delimiters; two consecutive delimiters are counted as one character.

To include an apostrophe in a character constant that is enclosed by apostrophes, place two consecutive apostrophes ('') in the format specification; for example:

```
50   FORMAT ('TODAY''S^DATE^IS:^',I2,'/',I2,'/',I2)
```

Note that the symbol ^ represents a nonprinting blank character.

Similarly, to include a quotation mark in a character constant that is enclosed by quotation marks, place two consecutive quotation marks ("") in the format specification.

On input, the character constant edit descriptor transfers length of string characters to the edit descriptor.

Examples

Consider the following '(3I5)' format in the WRITE statement:

```
WRITE (10, '(3I5)') I1, I2, I3
```

This is equivalent to:

```
      WRITE (10, 100) I1, I2, I3
100   FORMAT( 3I5)
```

The following shows another example:

```
!     These WRITE statements both output ABC'DEF
!     (The leading blank is a carriage-control character).
      WRITE (*, 970)
970   FORMAT (' ABC''DEF')
      WRITE (*, '('' ABC''''DEF'')')
!     The following WRITE also outputs ABC'DEF. No carriage-
!     control character is necessary for list-directed I/O.
      WRITE (*,*) 'ABC''DEF'
```

Alternatively, if the delimiter is quotation marks, the apostrophe in the character constant `ABC'DEF` requires no special treatment:

```
WRITE (*,*) "ABC'DEF"
```

See Also

Character constants
Format Specifications

H Editing

The H edit descriptor transfers data between the external record and the H edit descriptor itself. The H edit descriptor is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.

An H edit descriptor has the form of a Hollerith constant, as follows:

*n*H*string*

n	Is an unsigned, positive default integer literal constant (with no kind parameter) indicating the number of characters in *string* (including blanks and tabs). The range of *n* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. Actual useful ranges may be constrained by record sizes (RECL) and the file system.
string	Is a string of printable ASCII characters.

On input, the H edit descriptor transfers *n* characters from the external field to the edit descriptor. The first character appears immediately after the letter H. Any characters in the edit descriptor before input are replaced by the input characters. If the edit descriptor appears in a FORMAT statement, the replaced characters are preserved for future uses of that FORMAT statement; otherwise, the replacement is discarded.

On output, the H edit descriptor causes *n* characters following the letter H to be output to an external record.

Examples

```
!     These WRITE statements both print "Don't misspell 'Hollerith'"
!     (The leading blanks are carriage-control characters).
!     Hollerith formatting does not require you to embed additional
!     single quotation marks as shown in the second example.
!
      WRITE (*, 960)
960   FORMAT (27H Don't misspell 'Hollerith')
      WRITE (*, 961)
961   FORMAT (' Don''t misspell ''Hollerith''')
```

See Also

Deleted and Obsolescent Language Features

Format Specifications

Nested and Group Repeat Specifications

Format specifications can include nested format specifications enclosed in parentheses; for example:

```
15   FORMAT (E7.2,I8,I2,(A5,I6))
35   FORMAT (A6,(L8(3I2)),A)
```

A group repeat specification can precede a nested group of edit descriptors; it can be an unsigned integer literal constant, a variable format expression, or * to indicate an unlimited repeat count. For example, the following statements are equivalent, and the second statement shows a group repeat specification:

```
50   FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7,I5,I5)
50   FORMAT (2I8,3(F8.3,E15.7),2I5)
```

If a nested group does not show a repeat count, a default count of 1 is assumed.

Normally, the string edit descriptors and control edit descriptors cannot be repeated (except for slash), but any of these descriptors can be enclosed in parentheses and preceded by a group repeat specification. For example, the following statements are valid:

```
76   FORMAT ('MONTHLY',3('TOTAL'))
100  FORMAT (I8,4(T7),A4)
```

Each of the following can be used to read data at the end of an input record into N elements of array A:

```
    read (10,'(<N>F8.2)') A(1:N)  ! Variable format expression
    read(10,'(*(F8.2))') A(1:N)   ! Unlimited repeat count
```

See Also

String edit descriptors

Control edit descriptors

Forms for Data Edit Descriptors for details on repeat specifications for data edit descriptors

Interaction Between Format Specifications and I/O Lists for details on group repeat specifications and format reversion

Variable Format Expressions

A variable format expression is a numeric expression enclosed in angle brackets (< >) that can be used in a FORMAT statement or in a character format specification.

The numeric expression can be any valid Fortran expression, including function calls and references to dummy arguments.

If the expression is not of type integer, it is converted to integer type before being used.

If the value of a variable format expression does not obey the restrictions on magnitude applying to its use in the format, an error occurs.

Variable format expressions cannot be used with the H edit descriptor, and they are not allowed in character format specifications that are not character constant expressions.

Variable format expressions are evaluated each time they are encountered in the scan of the format. If the value of the variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

Examples

Consider the following statement:

```
  FORMAT (I<J+1>)
```

When the format is scanned, the preceding statement performs an I (integer) data transfer with a field width of J+1. The expression is reevaluated each time it is encountered in the normal format scan.

Consider the following statements:

```
      DIMENSION A(5)
      DATA A/1.,2.,3.,4.,5./

      DO 10 I=1,10
      WRITE (6,100) I
100   FORMAT (I<MAX(I,5)>)
10    CONTINUE

      DO 20 I=1,5
      WRITE (6,101) (A(I), J=1,I)
101   FORMAT (<I>F10.<I-1>)
20    CONTINUE
      END
```

On execution, these statements produce the following output:

```
    1
    2
    3
    4
    5
     6
      7
       8
        9
        10
        1.
       2.0        2.0
      3.00       3.00     3.00
     4.000      4.000    4.000     4.000
    5.0000    5.0000    5.0000    5.0000    5.0000
```

The following shows another example:

```
       WRITE(6,20) INT1
20     FORMAT(I<MAX(20,5)>)

       WRITE(6,FMT=30) REAL2(10), REAL3
30     FORMAT(<J+K>X, <2*M>F8.3)
```

The value of the expression is reevaluated each time an input/output item is processed during the execution of the READ, WRITE, or PRINT statement. For example:

```
       INTEGER width, value
       width=2
       READ (*,10) width, value
10     FORMAT(I1, I <width>)
       PRINT *, value
       END
```

When given input 3123, the program will print 123 and not 12.

See Also

Interaction Between Format Specifications and I/O Lists for details on the synchronization of I/O lists with formats

Printing of Formatted Records

On output, if a file was opened with CARRIAGECONTROL='FORTRAN' in effect, the first character of a record transmitted to a line printer or terminal is typically a character that is not printed, but used to control vertical spacing.

Printing control characters are a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.

The following table lists the valid control characters for printing:

Control Characters for Printing

Character	Meaning	Effect
+	Overprinting	Outputs the record (at the current position in the current line) and a carriage return.
-	One line feed	Outputs the record (at the beginning of the following line) and a carriage return.
0	Two line feeds	Outputs the record (after skipping a line) and a carriage return.
1	Next page	Outputs the record (at the beginning of a new page) and a carriage return.
$	Prompting	Outputs the record (at the beginning of the following line), but no carriage return.
ASCII NUL[1]	Overprinting with no advance	Outputs the record (at the current position in the current line), but no carriage return.

[1] Specify as CHAR(0).

Any other character is interpreted as a blank and is deleted from the print line. If you do not specify a control character for printing, the first character of the record is not printed.

Interaction Between Format Specifications and I/O Lists

Format control begins with the execution of a formatted I/O statement. Each action of format control depends on information provided jointly by the next item in the I/O list (if one exists) and the next edit descriptor in the format specification.

Both the I/O list and the format specification are interpreted from left to right, unless repeat specifications or implied-DO lists appear.

If an I/O list specifies at least one list item, at least one data edit descriptor (I, B, O, Z, F, E, EN, ES, EX, D, G, L, or A) or the Q edit descriptor must appear in the format specification; otherwise, an error occurs.

Each data edit descriptor (or Q edit descriptor) corresponds to one item in the I/O list, except that an I/O list item of type complex requires the interpretation of two F, E, EN, ES, EX, D, or G edit descriptors. No I/O list item corresponds to a control edit descriptor (X, P, T, TL, TR, SP, SS, S, BN, BZ, $, or :), or a character string edit descriptor (H and character constants). For character string edit descriptors, data transfer occurs directly between the external record and the format specification.

When format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding I/O list item specified. If there is such an item, it is transferred under control of the edit descriptor, and then format control proceeds. If there is no corresponding I/O list item, format control terminates.

If there are no other I/O list items to be processed, format control also terminates when the following occurs:

- A colon edit descriptor is encountered.
- The end of the format specification is reached.

If additional I/O list items remain, part or all of the format specification is reused in format reversion.

In format reversion, the current record is terminated and a new one is initiated. Format control then reverts to one of the following (in order) and continues from that point:

1. The group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format specification
2. The initial opening parenthesis of the format specification

Format reversion has no effect on the scale factor (P), the sign control edit descriptors (S, SP, or SS), or the blank interpretation edit descriptors (BN or BZ).

Examples

The data in file FOR002.DAT is to be processed 2 records at a time. Each record starts with a number to be put into an element of a vector B, followed by 5 numbers to be put in a row in matrix A.

FOR002.DAT contains the following data:

```
001 0101 0102 0103 0104 0105
002 0201 0202 0203 0204 0205
003 0301 0302 0303 0304 0305
004 0401 0402 0403 0404 0405
005 0501 0502 0503 0504 0505
006 0601 0602 0603 0604 0605
007 0701 0702 0703 0704 0705
008 0801 0802 0803 0804 0805
009 0901 0902 0903 0904 0905
010 1001 1002 1003 1004 1005
```

The following example shows how several different format specifications interact with I/O lists to process data in file FOR002.DAT:

```
      INTEGER I, J, A(2,5), B(2)
      OPEN (unit=2, access='sequential', file='FOR002.DAT')

      READ (2,100) (B(I), (A(I,J), J=1,5),I=1,2) 1

100   FORMAT (2 (I3, X, 5(I4,X), /) ) 2

      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) 3

999   FORMAT (' B is ', 2(I3, X), ';  A is', /
     1       (' ', 5 (I4, X)) )

      READ (2,200) (B(I), (A(I,J), J=1,5),I=1,2) 4

200   FORMAT (2 (I3, X, 5(I4,X), :/) )
      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) 5
```

```
      READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2) 6
300   FORMAT ( (I3, X, 5(I4,X)) )
      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) 7
      READ (2,400) (B(I), (A(I,J), J=1,5),I=1,2) 8
400   FORMAT ( I3, X, 5(I4,X) )
      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2) 9
      END
```

1 This statement reads B(1); then A(1,1) through A(1,5); then B(2) and A(2,1) through A(2,5).

The first record read (starting with 001) starts the processing of the I/O list.

2 There are two records, each in the format I3, X, 5(I4, X). The slash (/) forces the reading of the second record after A(1,5) is processed. It also forces the reading of the third record after A(2,5) is processed; no data is taken from that record.

3 This statement produces the following output:

```
B is   1   2 ;  A is 101  102  103  104  105 201  202  203  204  205
```

4 This statement reads the record starting with 004. The slash (/) forces the reading of the next record after A(1,5) is processed. The colon (:) stops the reading after A(2,5) is processed, but before the slash (/) forces another read.

5 This statement produces the following output:

```
B is   4   5 ;  A is 401  402  403  404  405 501  502  503  504  505
```

6 This statement reads the record starting with 006. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I3.

7 This statement produces the following output:

```
B is   6   7 ;  A is 601  602  603  604  605 701  702  703  704  705
```

8 This statement reads the record starting with 008. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I4.

9 This statement produces the following output:

```
B is   8  90 ;  A is 801  802  803  804  805 9010 9020 9030 9040 100
```

The record 009 0901 0902 0903 0904 0905 is processed with I4 as "009 " for B(2), which is 90. X skips the next "0". Then "901 " is processed for A(2,1), which is 9010, "902 " for A(2,2), "903 " for A(2,3), and "904 " for A(2,4). The repeat specification of 5 is now exhausted and the format ends. Format reversion causes another record to be read and starts format processing at the left parenthesis before the I4, so "010 " is read for A(2,5), which is 100.

See Also

Data edit descriptors
Control edit descriptors

Q edit descriptor
Character string edit descriptors
Scale Factor Editing (P)

File Operation I/O Statements

The following are file connection, inquiry, and positioning I/O statements:

- BACKSPACE

  Positions a sequential file at the beginning of the preceding record, making it available for subsequent I/O processing.
- CLOSE

  Terminates the connection between a logical unit and a file or device.
- DELETE

  Deletes a record from a relative file.
- ENDFILE

  Writes an end-of-file record to a sequential file and positions the file after this record (the terminal point). For direct access files, truncates the file after the current record.
- FLUSH

  Causes data written to a file to become available to other processes or causes data written to a file outside of Fortran to be accessible to a READ statement.
- INQUIRE

  Requests information on the status of specified properties of a file or logical unit. For more information on specifiers you can use in INQUIRE statements, see INQUIRE Statement Specifiers.
- OPEN

  Connects a Fortran logical unit to a file or device; declares attributes for read and write operations. For more information on specifiers you can use in OPEN statements, see OPEN Statement Specifiers.
- REWIND

  Positions a sequential or direct access file at the beginning of the file (the initial point).
- WAIT

  Performs a wait operation for a specified pending asynchronous data transfer operation.

The following table summarizes I/O statement specifiers:

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
ACCESS=*access*	'SEQUENTIAL', 'DIRECT', 'STREAM', or 'APPEND'	Specifies the method of file access.	INQUIRE, OPEN
ACTION=*permission*	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Specifies file I/O mode.	INQUIRE, OPEN
ADVANCE=*c-expr*	'NO' or 'YES' (default is 'YES')	Specifies formatted sequential data input as advancing, or non-advancing.	READ

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
ASSOCIATEVARIABLE=*var*	Integer variable	Specifies a variable to be updated to reflect the record number of the next sequential record in the file.	OPEN
ASYNCHRONOUS=*asynch*	'YES' or 'NO' (default is 'NO')	Specifies whether or not the I/O is done asynchronously	INQUIRE, OPEN
BINARY=*bin*	'NO' or 'YES'	Returns whether file format is binary.	INQUIRE
BLANK=*blank_control*	'NULL' or 'ZERO' (default is 'NULL')	Specifies whether blanks are ignored in numeric fields or interpreted as zeros.	INQUIRE, OPEN
BLOCKSIZE=*blocksize*	Positive integer variable or expression	Specifies or returns the internal buffer size used in I/O.	INQUIRE, OPEN
BUFFERCOUNT=*bc*	Numeric expression	Specifies the number of buffers to be associated with the unit for multibuffered I/O.	OPEN
BUFFERED=*bf*	'YES' or 'NO' (default is 'NO')	Specifies run-time library behavior following WRITE operations.	INQUIRE, OPEN
CARRIAGECONTROL=*control*	'FORTRAN', 'LIST', or 'NONE'	Specifies carriage control processing.	INQUIRE, OPEN
CONVERT=*form*	'LITTLE_ENDIAN', 'BIG_ENDIAN', 'CRAY', 'FDX', 'FGX', 'IBM', 'VAXD', 'VAXG', or 'NATIVE' (default is 'NATIVE')	Specifies a numeric format for unformatted data.	INQUIRE, OPEN
DEFAULTFILE=*var*	Character expression	Specifies a default file pathname string.	INQUIRE, OPEN
DELIM=*delimiter*	'APOSTROPHE', 'QUOTE' or 'NONE' (default is 'NONE')	Specifies the delimiting character for list-directed or namelist data.	INQUIRE, OPEN
DIRECT=*dir*	'NO' or 'YES'	Returns whether file is connected for direct access.	INQUIRE
DISPOSE=*dis* (or DISP=*dis*)	'KEEP', 'SAVE', 'DELETE', 'PRINT', 'PRINT/DELETE', 'SUBMIT', or 'SUBMIT/DELETE'	Specifies the status of a file after the unit is closed.	OPEN, CLOSE

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
	(default is 'DELETE' for scratch files; 'KEEP' for all other files)		
formatlist	Character variable or expression	Lists edit descriptors. Used in FORMAT statements and format specifiers (the FMT=*formatspec* option) to describe the format of data.	FORMAT, PRINT, READ, WRITE
END=*endlabel*	Integer between 1 and 99999	When an end of file is encountered, transfers control to the statement whose label is specified.	READ
EOR=*eorlabel*	Integer between 1 and 99999	When an end of record is encountered, transfers to the statement whose label is specified.	READ
ERR=*errlabel*	Integer between 1 and 99999	Specifies the label of an executable statement where execution is transferred after an I/O error.	All except PRINT
EXIST=*ex*	.TRUE. or .FALSE.	Returns whether a file exists and can be opened.	INQUIRE
FILE=*file*(or NAME=*name*)	Character variable or expression. Length and format of the name are determined by the operating system	Specifies the name of a file	INQUIRE, OPEN
[FMT=]*formatspec*	Character variable or expression	Specifies an *editlist* to use to format data.	PRINT, READ, WRITE
FORM=*form*	'FORMATTED', 'UNFORMATTED', or 'BINARY'	Specifies a file's format.	INQUIRE, OPEN
FORMATTED=*fmt*	'NO' or 'YES'	Returns whether a file is connected for formatted data transfer.	INQUIRE
IOFOCUS=*iof*	.TRUE. or .FALSE. (default is .TRUE. unless unit '*' is specified)	Specifies whether a unit is the active window in a QuickWin application.	INQUIRE, OPEN
iolist	List of variables of any type, character expression, or NAMELIST	Specifies items to be input or output.	PRINT, READ, WRITE

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
IOSTAT=*iostat*	Integer variable	Specifies a variable whose value indicates whether an I/O error has occurred.	All except PRINT
MAXREC=*var*	Numeric expression	Specifies the maximum number of records that can be transferred to or from a direct access file.	OPEN
MODE=*permission*	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Same as ACTION.	INQUIRE, OPEN
NAMED=*var*	.TRUE. or .FALSE.	Returns whether a file is named.	INQUIRE
NEWUNIT=*u-var*	Scalar integer variable	Is assigned an unused unit number that is automatically chosen. It is always a negative integer.	OPEN
NEXTREC=*nr*	Integer variable	Returns where the next record can be read or written in a file.	INQUIRE
[NML=]*nmlspec*	Namelist name	Specifies a *namelist* group to be input or output.	PRINT, READ, WRITE
NUMBER=*num*	Integer variable	Returns the number of the unit connected to a file.	INQUIRE
OPENED=*od*	.TRUE. or .FALSE.	Returns whether a file is connected.	INQUIRE
ORGANIZATION=*org*	'SEQUENTIAL' or 'RELATIVE' (default is 'SEQUENTIAL')	Specifies the internal organization of a file.	INQUIRE, OPEN
PAD=*pad_switch*	'YES' or 'NO' (default is 'YES')	Specifies whether an input record is padded with blanks when the input list or format requires more data than the record holds, or whether the input record is required to contain the data indicated.	INQUIRE, OPEN
POS=*pos*	Positive integer	Specifies the file storage unit position in a stream file.	INQUIRE, READ, WRITE

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
POSITION=*file_pos*	'ASIS', 'REWIND' or 'APPEND' (default is 'ASIS')	Specifies position in a file.	INQUIRE, OPEN
READ=*rd*	'NO' or 'YES'	Returns whether a file can be read.	INQUIRE
READONLY		Specifies that only READ statements can refer to this connection.	OPEN
READWRITE=*rdwr*	'NO' or 'YES'	Returns whether a file can be both read and written to.	INQUIRE
REC=*rec*	Positive integer variable or expression	Specifies the first (or only) record of a file to be read from, or written to.	READ, WRITE
RECL=*length*(or RECORDSIZE=*length*)	Positive integer variable or expression	Specifies the record length in direct access files, or the maximum record length in sequential files.	INQUIRE, OPEN
RECORDTYPE=*typ*	'FIXED', 'VARIABLE', 'SEGMENTED', 'STREAM', 'STREAM_LF', or 'STREAM_CR'	Specifies the type of records in a file.	INQUIRE, OPEN
SEQUENTIAL=*seq*	'NO' or 'YES'	Returns whether file is connected for sequential access.	INQUIRE
SHARE=*share*	'COMPAT', 'DENYNONE', 'DENYWR', 'DENYRD', or 'DENYRW' (default is 'DENYNONE')	Controls how other processes can simultaneously access a file on networked systems.	INQUIRE, OPEN
SHARED		Specifies that a file is connected for shared access by more than one program executing simultaneously.	OPEN
SIZE=*size*	Integer variable	Returns the number of characters read in a nonadvancing READ before an end-of-record condition occurred.	READ
STATUS=*status*(or TYPE=*status*)	'OLD', 'NEW', 'UNKNOWN' or 'SCRATCH' (default is 'UNKNOWN')	Specifies the status of a file on opening and/or closing.	CLOSE, OPEN

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
TITLE=*name*	Character expression	Specifies the name of a child window in a QuickWin application.	OPEN
UNFORMATTED=*unf*	'NO' or 'YES'	Returns whether a file is connected for unformatted data transfer.	INQUIRE
[UNIT=]*unitspec*	Integer variable or expression	Specifies the unit to which a file is connected.	All except PRINT
USEROPEN=*fname*	Name of a user-written function	Specifies an external function that controls the opening of a file.	OPEN
WRITE=*rd*	'NO' or 'YES'	Returns whether a file can be written to.	INQUIRE

See Also
Data transfer I/O statements
I/O Control List for details on control specifiers

INQUIRE Statement Specifiers

The INQUIRE statement returns information on the status of specified properties of a file or logical unit. For more information, see INQUIRE.

The following table lists the INQUIRE statement specifiers and contains links to their descriptions:

ACCESS	DELIM	NEXTREC	RECL
ACTION	DIRECT	NUMBER	RECORDTYPE
ASYNCHRONOUS	ENCODING	OPENED	ROUND
BINARY	EXIST	ORGANIZATION	SEQUENTIAL
BLANK	FORM	PAD	SHARE
BLOCKSIZE	FORMATTED	PENDING	SIGN
BUFFERED	IOFOCUS	POS	SIZE
CARRIAGECONTROL	MODE	POSITION	UNFORMATTED
CONVERT	NAME	READ	WRITE
DECIMAL	NAMED	READWRITE	

See Also
UNIT control specifier
ERR control specifier
ID control specifier
IOMSG control specifier
IOSTAT control specifier

RECL specifier in OPEN statements
FILE specifier in OPEN statements
DEFAULTFILE specifier in OPEN statements

INQUIRE: ACCESS Specifier

The ACCESS specifier asks how a file is connected. It takes the following form:

ACCESS = *acc*

acc Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is connected for sequential access
'STREAM'	If the file is connected for stream access
'DIRECT'	If the file is connected for direct access
'UNDEFINED'	If the file is not connected

INQUIRE: ACTION Specifier

The ACTION specifier asks which I/O operations are allowed for a file. It takes the following form:

ACTION = *act*

act Is a scalar default character variable that is assigned one of the following values:

'READ'	If the file is connected for input only
'WRITE'	If the file is connected for output only
'READWRITE'	If the file is connected for both input and output
'UNDEFINED'	If the file is not connected

INQUIRE: ASYNCHRONOUS Specifier

The ASYNCHRONOUS specifier asks whether asynchronous I/O is in effect. It takes the following form:

ASYNCHRONOUS = *asyn*

asyn Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit is connected and asynchronous input/output is not in effect.
'YES'	If the file or unit is connected and asynchronous input/output is in effect.

'UNKNOWN'	If the file or unit is not connected.

INQUIRE: BINARY Specifier (W*S)

The BINARY specifier asks whether a file is connected to a binary file. It takes the following form:

BINARY = *bin*

bin Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected to a binary file
'NO'	If the file is connected to a nonbinary file
'UNKNOWN'	If the file is not connected

INQUIRE: BLANK Specifier

The BLANK specifier asks what type of blank control is in effect for a file. It takes the following form:

BLANK = *blnk*

blnk Is a scalar default character variable that is assigned one of the following values:

'NULL'	If null blank control is in effect for the file
'ZERO'	If zero blank control is in effect for the file
'UNDEFINED'	If the file is not connected, or it is not connected for formatted data transfer

INQUIRE: BLOCKSIZE Specifier

The BLOCKSIZE specifier asks about the physical I/O transfer size. It takes the following form:

BLOCKSIZE = *bks*

bks Is a scalar integer variable.

The *bks* is assigned the current size of the physical I/O transfer. If the unit or file is not connected, the value assigned is zero.

INQUIRE: BUFFERED Specifier

The BUFFERED specifier asks whether run-time buffering is in effect. It takes the following form:

BUFFERED = *bf*

bf Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit is connected and buffering is not in effect.
'YES'	If the file or unit is connected and buffering is in effect.
'UNKNOWN'	If the file or unit is not connected.

INQUIRE: CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier asks what type of carriage control is in effect for a file. It takes the following form:

CARRIAGECONTROL = *cc*

cc Is a scalar default character variable that is assigned one of the following values:

'FORTRAN'	If the file is connected with Fortran carriage control in effect
'LIST'	If the file is connected with implied carriage control in effect
'NONE'	If the file is connected with no carriage control in effect
'UNKNOWN'	If the file is not connected

INQUIRE: CONVERT Specifier

The CONVERT specifier asks what type of data conversion is in effect for a file. It takes the following form:

CONVERT = *fm*

fm Is a scalar default character variable that is assigned one of the following values:

'LITTLE_ENDIAN'	If the file is connected with little endian integer and IEEE* floating-point data conversion in effect
'BIG_ENDIAN'	If the file is connected with big endian integer and IEEE floating-point data conversion in effect
'CRAY'	If the file is connected with big endian integer and CRAY* floating-point data conversion in effect

'FDX'	If the file is connected with little endian integer and VAX* processor F_floating, D_floating, and IEEE binary128 data conversion in effect
'FGX'	If the file is connected with little endian integer and VAX processor F_floating, G_floating, and IEEE binary128 data conversion in effect
'IBM'	If the file is connected with big endian integer and IBM* System \370 floating-point data conversion in effect
'VAXD'	If the file is connected with little endian integer and VAX processor F_floating, D_floating, and H_floating in effect
'VAXG'	If the file is connected with little endian integer and VAX processor F_floating, G_floating, and H_floating in effect
'NATIVE'	If the file is connected with no data conversion in effect
'UNKNOWN'	If the file or unit is not connected for unformatted data transfer

INQUIRE: DECIMAL Specifier

The DECIMAL specifier asks which decimal editing mode is in effect for a file connection. It takes the following form:

DECIMAL = *dmode*

dmode Is a scalar default character variable that is assigned one of the following values:

'COMMA'	If a decimal comma is used during decimal editing mode.
'POINT'	If a decimal point is used during decimal editing mode.
'UNDEFINED'	If there is no connection or if the connection is not for formatted I/O.

INQUIRE: DELIM Specifier

The DELIM specifier asks how character constants are delimited in list-directed and namelist output. It takes the following form:

DELIM = *del*

del

Is a scalar default character variable that is assigned one of the following values:

'APOSTROPHE'	If apostrophes are used to delimit character constants in list-directed and namelist output
'QUOTE'	If quotation marks are used to delimit character constants in list-directed and namelist output
'NONE'	If no delimiters are used
'UNDEFINED'	If the file is not connected, or is not connected for formatted data transfer

INQUIRE: DIRECT Specifier

The DIRECT specifier asks whether a file is connected for direct access. It takes the following form:

DIRECT = *dir*

dir

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for direct access
'NO'	If the file is not connected for direct access
'UNKNOWN'	If the file is not connected

INQUIRE: ENCODING Specifier

The ENCODING specifier asks what type of encoding is in effect for a file. It takes the following form:

ENCODING = *enc*

enc

Is a scalar default character expression that is assigned one of the following values:

'UTF-8'	If the file is connected with UTF-8 encoding in effect.
'UNDEFINED'	If the file is connected for unformatted I/O.
'UNKNOWN'	If the processor is unable to determine the encoding form of the file.

INQUIRE: EXIST Specifier

The EXIST specifier asks whether a file exists and can be opened. It takes the following form:

EXIST = *ex*

ex Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file exists and can be opened, or if the specified unit exists
.FALSE.	If the specified file or unit does not exist or if the file exists but cannot be opened

The unit exists if it is a number in the range allowed by the processor.

INQUIRE: FORM Specifier

The FORM specifier asks whether a file is connected for formatted, unformatted, or binary (W*S) data transfer. It takes the following form:

FORM = *fm*

fm Is a scalar default character variable that is assigned one of the following values:

'FORMATTED'	If the file is connected for formatted data transfer
'UNFORMATTED'	If the file is connected for unformatted data transfer
'BINARY'	If the file is connected for binary data transfer
'UNDEFINED'	If the file is not connected

INQUIRE: FORMATTED Specifier

The FORMATTED specifier asks whether a file is connected for formatted data transfer. It takes the following form:

FORMATTED = *fmt*

fmt Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for formatted data transfer
'NO'	If the file is not connected for formatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for formatted data transfer

INQUIRE: IOFOCUS Specifier (W*S)

The IOFOCUS specifier asks if the indicated unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified unit is the active window in a QuickWin application
.FALSE.	If the specified unit is not the active window in a QuickWin application

If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

A value of .TRUE. causes a call to FOCUSQQ immediately before any READ, WRITE, or PRINT statement to that window.

If you use this specifier with a non-Windows application, an error occurs.

INQUIRE: MODE Specifier

MODE is a nonstandard synonym for ACTION.

INQUIRE: NAME Specifier

The NAME specifier returns the name of a file. It takes the following form:

NAME = *nme*

nme Is a scalar default character variable that is assigned the name of the file to which the unit is connected. If the file does not have a name, *nme* is undefined.

The value assigned to *nme* is not necessarily the same as the value given in the FILE specifier. However, the value that is assigned is always valid for use with the FILE specifier in an OPEN statement, unless the value has been truncated in a way that makes it unacceptable. (Values are truncated if the declaration of *nme* is too small to contain the entire value.)

NOTE
The FILE and NAME specifiers are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

See Also

The appropriate manual in your operating system documentation set for details on the maximum size of file pathnames

INQUIRE: NAMED Specifier

The NAMED specifier asks whether a file is named. It takes the following form:

NAMED = *nmd*

nmd

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the file has a name
.FALSE.	If the file does not have a name

INQUIRE: NEXTREC Specifier

The NEXTREC specifier asks where the next record can be read or written in a file connected for direct access. It takes the following form:

NEXTREC = *nr*

nr

Is a scalar integer variable that is assigned a value as follows:

- If the file is connected for direct access and a record (r) was previously read or written, the value assigned is r + 1.
- If no record has been read or written, the value assigned is 1.
- If there are pending asynchronous data transfer operations for the specified file, the value assigned is computed as if all pending data transfers have already completed.
- If the file is not connected for direct access, or if the file position cannot be determined because of an error condition, the value assigned is zero.
- If the file is connected for direct access and a REWIND has been performed on the file, the value assigned is 1.

INQUIRE: NUMBER Specifier

The NUMBER specifier asks the number of the unit connected to a file. It takes the following form:

NUMBER = *num*

num

Is a scalar integer variable.

The *num* is assigned the number of the unit currently connected to the specified file. If there is no unit connected to the file, the value assigned is -1.

INQUIRE: OPENED Specifier

The OPENED specifier asks whether a file is connected. It takes the following form:

OPENED = *od*

od

Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file or unit is connected
.FALSE.	If the specified file or unit is not connected

INQUIRE: ORGANIZATION Specifier

The ORGANIZATION specifier asks how the file is organized. It takes the following form:

ORGANIZATION = *org*

org Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is a sequential file
'RELATIVE'	If the file is a relative file
'UNKNOWN'	If the processor cannot determine the file's organization

INQUIRE: PAD Specifier

The PAD specifier asks whether blank padding was specified for the file. It takes the following form:

PAD = *pd*

pd Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit was connected with PAD='NO'
'YES'	If the file or unit is not connected, or it was connected with PAD='YES'

INQUIRE: PENDING Specifier

The PENDING specifier asks whether previously pending asynchronous data transfers are complete. A data transfer is previously pending if it is not complete at the beginning of execution of the INQUIRE statement. It takes the following form:

PENDING = *pnd*

pnd Is a scalar default logical variable that is assigned the value .TRUE. or .FALSE..

The value is assigned as follows:

- If an ID specifier appears in the INQUIRE statement, the following occurs:
  - If the data transfer specified by ID is complete, then variable *pnd* is set to .FALSE. and INQUIRE performs the WAIT operation for the specified data transfer.
  - If the data transfer specified by ID is not complete, then variable *pnd* is set to .TRUE. and no WAIT operation is performed. The previously pending data transfer remains pending after the execution of the INQUIRE statement.
- If an ID specifier does not appear in the INQUIRE statement, the following occurs:
  - If all previously pending data transfers for the specified unit are complete, then variable *pnd* is set to .FALSE. and the INQUIRE statement performs WAIT operations for all previously pending data transfers for the specified unit.
  - If there are data transfers for the specified unit that are not complete, then variable *pnd* is set to .TRUE. and no WAIT operations are performed. The previously pending data transfers remain pending after the execution of the INQUIRE statement.

See Also

ID Specifier

Example in INQUIRE Statement

INQUIRE: POS Specifier

The POS specifier identifies the file position in file storage units in a stream file. It takes the following form:

POS = *p*

p	Is a scalar integer variable that is assigned the number of the file storage unit immediately following the current position of a file connected for stream access (ACCESS='STREAM').

If the file is positioned at its terminal position, *p* is assigned a value one greater than the number of the highest-numbered file storage unit in the file.

If the file is not connected for stream access or if the position of the file is indeterminate because of previous error conditions, *p* is assigned the value one.

If there are pending asynchronous data operations for the specified file, the value assigned to the POS= specifier is computed as if all pending data transfers have already completed.

INQUIRE: POSITION Specifier

The POSITION specifier asks the position of the file. It takes the following form:

POSITION = *pos*

pos — Is a scalar default character variable that is assigned one of the following values:

'REWIND'	If the file is connected with its position at its initial point
'APPEND'	If the file is connected with its position at its terminal point (or before its end-of-file record, if any)
'ASIS'	If the file is connected without changing its position
'UNDEFINED'	If the file is not connected, or is connected for direct access data transfer and a REWIND statement has not been performed on the unit

INQUIRE: READ Specifier

The READ specifier asks whether a file can be read. It takes the following form:

READ = *rd*

rd — Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be read
'NO'	If the file cannot be read
'UNKNOWN'	If the processor cannot determine whether the file can be read

INQUIRE: READWRITE Specifier

The READWRITE specifier asks whether a file can be both read and written to. It takes the following form:

READWRITE = *rdwr*

rdwr

Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be both read and written to
'NO'	If the file cannot be both read and written to
'UNKNOWN'	If the processor cannot determine whether the file can be both read and written to

INQUIRE: RECL Specifier

The RECL specifier asks the maximum record length for a file. It takes the following form:

RECL = *rcl*

rcl

Is a scalar integer variable that is assigned a value as follows:

- If the file or unit is connected, the value assigned is the maximum record length allowed.
- If the file does not exist, or it is not connected, the value assigned is zero.

  Fortran 2018 standardizes the value assigned to be -1 when the file does not exist, or it is not connected. To get the Fortran 2018 behavior, specify compiler option `assume -noold_inquire_recl`.
- If the file is connected for stream access, the value is undefined. Fortran 2018 standardizes this value to be -2. To get the Fortran 2018 behavior, specify compiler option `assume -noold_inquire_recl`.

The assigned value is expressed in 4-byte units if the file is currently (or was previously) connected for unformatted data transfer and the `assume byterecl` compiler option is not in effect; otherwise, the value is expressed in bytes.

INQUIRE: RECORDTYPE Specifier

The RECORDTYPE specifier asks which type of records are in a file. It takes the following form:

RECORDTYPE = *rtype*

rtype Is a scalar default character variable that is assigned one of the following values:

'FIXED'	If the file is connected for fixed-length records
'VARIABLE'	If the file is connected for variable-length records
'SEGMENTED'	If the file is connected for unformatted sequential data transfer using segmented records
'STREAM'	If the file's records are not terminated
'STREAM_CR'	If the file's records are terminated with a carriage return
'STREAM_LF'	If the file's records are terminated with a line feed
'STREAM_CRLF'	If the file's records are terminated with a carriage return/line feed pair
'UNKNOWN'	If the file is not connected

INQUIRE: ROUND Specifier

The ROUND specifier asks which rounding mode is in effect for a file connection. It takes the following form:

ROUND = *rmode*

rmode Is a scalar default character expression that is assigned one of the following values:

'UP'	If the I/O rounding is set to the smallest representable value that is greater than or equal to the original value.
'DOWN'	If the I/O rounding is set to the largest representable value that is less than or equal to the original value.
'ZERO'	If the I/O rounding is set to the value closest to the original value, but no greater in magnitude than the original value.
'NEAREST'	Conforms to the ISO/IEC/IEEE 60559:2011 standard specification for roundTiesToEven.

'COMPATIBLE'	If the I/O rounding is set to the closer of the two nearest representable values, or the value farther from zero if halfway between them.
'PROCESSOR_DEFINED'	If the I/O rounding mode behaves differently than the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.
'UNDEFINED'	If there is no connection or if the connection is not for formatted I/O.

The rounding modes conform to the corresponding rounding modes specified in the ISO/IEC/IEEE 60559:2011 standard.

INQUIRE: SEQUENTIAL Specifier

The SEQUENTIAL specifier asks whether a file is connected for sequential access. It takes the following form:

SEQUENTIAL = *seq*

seq Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for sequential access
'NO'	If the file is not connected for sequential access
'UNKNOWN'	If the processor cannot determine whether the file is connected for sequential access

INQUIRE: SHARE Specifier

The SHARE specifier asks the current share status of a file or unit. It takes the following form:

SHARE = *shr*

shr Is a scalar default character variable.

On Windows* systems, this variable is assigned one of the following values:

'DENYRW'	If the file is connected for deny-read/write mode
'DENYWR'	If the file is connected for deny-write mode
'DENYRD'	If the file is connected for deny-read mode
'DENYNONE'	If the file is connected for deny-none mode

'UNKNOWN'	If the file or unit is not connected
On Linux* and macOS* systems, this variable is assigned one of the following values:	
'DENYRW'	If the file is connected for exclusive access
'DENYNONE'	If the file is connected for shared access
'NODENY'	If the file is connected with default locking
'UNKNOWN'	If the file or unit is not connected

INQUIRE: SIGN Specifier

The SIGN specifier asks what treatment for plus signs is in effect during a connection. It takes the following form:

SIGN = *sn*

sn Is a scalar default character expression that is assigned one of the following values:

'PLUS'	If the file is connected with addition of plus signs in effect.
'SUPPRESS'	If the file is connected with suppression of plus signs in effect.
'PROCESSOR_DEFINED'	If the file is connected with plus signs added at the discretion of the processor.
'UNDEFINED'	If there is no connection, or if the connection is not for formatted input/output.

INQUIRE: SIZE Specifier

The SIZE specifier asks the size of a file in file storage units. It takes the following form:

SIZE = *sz*

sz Is a scalar integer variable.

The *sz* variable is assigned the size of the file in file storage units. If the file size cannot be determined, the variable is assigned the value -1.

For a file that is connected for stream access, the file size is the number of the highest-numbered file storage unit in the file.

For a file that is connected for sequential or direct access, the file size may be different from the number of storage units implied by the data in the records; the exact relationship is processor-dependent.

If there are pending asynchronous data transfer operations for the specified file, the value assigned to the SIZE= specifier is computed as if the pending data transfers have already completed.

INQUIRE: UNFORMATTED Specifier

The UNFORMATTED specifier asks whether a file is connected for unformatted data transfer. It takes the following form:

UNFORMATTED = *unf*

unf Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for unformatted data transfer
'NO'	If the file is not connected for unformatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for unformatted data transfer

INQUIRE: WRITE Specifier

The WRITE specifier asks whether a file can be written to. It takes the following form:

WRITE = *wr*

wr Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be written to
'NO'	If the file cannot be written to
'UNKNOWN'	If the processor cannot determine whether the file can be written to

OPEN Statement Specifiers

The OPEN statement connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection. For more information, see OPEN.

The following table summarizes the OPEN statement specifiers and contains links to their descriptions:

OPEN Statement Specifiers and Values

Specifier	Values	Function	Default
ACCESS	'DIRECT' 'SEQUENTIAL' 'STREAM' 'APPEND'	Access mode	'SEQUENTIAL'
ACTION	'READ'	File access	'READWRITE'

Specifier	Values	Function	Default
(or MODE)	'WRITE' 'READWRITE'		
ASSOCIATEVARIABLE	var	Next direct access record	No default
ASYNCHRONOUS	'YES' 'NO'	Asynchronous I/O	'NO'
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	n_expr	Physical I/O transfer size	131,072 bytes
BUFFERCOUNT	n_expr	Number of I/O buffers	One
BUFFERED	'YES' 'NO'	Buffering for WRITE operations; buffering for READ operations on variable length, unformatted, input records	'NO' [1]
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	Formatted: 'LIST'[2] Unformatted: 'NONE'
CONVERT	'LITTLE_ENDIAN' 'BIG_ENDIAN' 'CRAY' 'FDX' 'FGX' 'IBM' 'VAXD' 'VAXG' 'NATIVE'	Numeric format specification	'NATIVE'
DECIMAL	'COMMA' 'POINT'	Decimal edit mode	'POINT'
DEFAULTFILE	c_expr	Default file pathname	Current working directory
DELIM	'APOSTROPHE' 'QUOTE' 'NONE'	Delimiter for character constants	'NONE'
DISPOSE(or DISP)	'KEEP' or 'SAVE' 'DELETE' 'PRINT'	File disposition at close	'KEEP'

Specifier	Values	Function	Default
	'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'		
ENCODING	'UTF-8' 'DEFAULT'	Encoding form for a file	'DEFAULT'
ERR	label	Error transfer control	No default
FILE(or NAME)	c_expr	File pathname (file name)	fort.n [3]
FORM	'FORMATTED' 'UNFORMATTED' 'BINARY'	Format type	Depends on ACCESS setting
IOFOCUS	.TRUE. .FALSE.	Active window in QuickWin application	.TRUE.[4]
IOSTAT	var	I/O status	No default
MAXREC	n_expr	Direct access record limit	No limit
NEWUNIT	var	Returns automatically chosen, unused, unit number	No default
NOSHARED	No value	File sharing disallowed	L*X, M*X: SHARED W*32: Not shared
ORGANIZATION	'SEQUENTIAL' 'RELATIVE'	File organization	'SEQUENTIAL'
PAD	'YES' 'NO'	Record padding	'YES'
POSITION	'ASIS' 'REWIND' 'APPEND'	File positioning	'ASIS'
READONLY	No value	Write protection	No default
RECL (or RECORDSIZE)	n_expr	Record length	Depends on RECORDTYPE,ORGANIZATION, and FORM settings [5]
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR'	Record type	Depends on ORGANIZATION, CARRIAGECONTROL, ACCESS, and FORM settings

Specifier	Values	Function	Default
	'STREAM_LF'		
ROUND	'UP' 'DOWN' 'ZERO' 'NEAREST' 'COMPATIBLE' 'PROCESSOR_DEFINED'	Rounding mode	'PROCESSOR_DEFINED' - for `ifort`, this corresponds to 'NEAREST' as in the ISO/IEC/IEEE 60559:2011 standard specification for roundTiesToEven
SHARE	'DENYRW' 'DENYWR'[5] 'DENYRD'[6] 'DENYNONE'	File locking	'DENYWR' [7]
SHARED	No value	File sharing allowed	L*X, M*X: SHARED W*S: Not shared
SIGN	'PLUS' 'SUPPRESS' 'PROCESSOR_DEFINED'	Plus sign mode	'PROCESSOR_DEFINED'
STATUS(or TYPE)	'OLD' 'NEW' 'SCRATCH' 'REPLACE' 'UNKNOWN'	File status at open	'UNKNOWN' [8]
TITLE	c_expr	Title for child window in QuickWin application	No default
UNIT	n_expr	Logical unit number	No default; an io-unit must be specified
USEROPEN	func	User program option	No default

[1] The default is also 'NO' when reading variable length, unformatted records whose length exceeds that of the block size specified for the file.

[2] If you use the compiler option specifying OpenVMS defaults, and the unit is connected to a terminal, the default is 'FORTRAN'.

[3] n is the unit number.

[4] If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

[5] On Linux* and macOS* systems, the default depends only on the FORM setting.

[6] W*S

[7] The default differs under certain conditions (see SHARE Specifier).

[8] The default differs under certain conditions (see STATUS Specifier).

Specifier	Values	Function	Default

Key to Values

```
c_expr: A scalar default character expression
func:   An external function
label:  A statement label
n_expr: A scalar numeric expression
var:    A scalar integer variable
```

See Also

INQUIRE Statement for details on using the INQUIRE statement to get file attributes of existing files

OPEN: ACCESS Specifier

The ACCESS specifier indicates the access method for the connection of the file. It takes the following form:

ACCESS = *acc*

acc Is a scalar default character expression that evaluates to one of the following values:

'DIRECT'	Indicates direct access.
'SEQUENTIAL'	Indicates sequential access.
'STREAM'	Indicates stream access, where the file storage units of the file are accessible sequentially or by position.
'APPEND'	Indicates sequential access, but the file is positioned at the end-of-file record.

The default is 'SEQUENTIAL'.

There are limitations on record access by file organization and record type.

OPEN: ACTION Specifier

The ACTION specifier indicates the allowed I/O operations for the file connection. It takes the following form:

ACTION = *act*

act Is a scalar default character expression that evaluates to one of the following values:

'READ'	Indicates that only READ statements can refer to this connection.
'WRITE'	Indicates that only WRITE, DELETE, and ENDFILE statements can refer to this connection.

'READWRITE'	Indicates that READ, WRITE, DELETE, and ENDFILE statements can refer to this connection.

The default is 'READWRITE'.

However, if compiler option fpscomp general is specified on the command line and *action* is omitted, the system first attempts to open the file with 'READWRITE'. If this fails, the system tries to open the file again, first using 'READ', then using 'WRITE'.

Note that in this case, omitting *action* is not the same as specifying ACTION='READWRITE'. If you specify ACTION='READWRITE' and the file cannot be opened for both read and write access, the attempt to open the file fails. You can use the INQUIRE statement to determine the actual access mode selected.

See Also

fpscomp compiler option

OPEN: ASSOCIATEVARIABLE Specifier

The ASSOCIATEVARIABLE specifier indicates a variable that is updated after each direct access I/O operation, to reflect the record number of the next sequential record in the file. It takes the following form:

ASSOCIATEVARIABLE = *asv*

asv	Is a scalar integer variable. It cannot be a dummy argument to the routine in which the OPEN statement appears.

Direct access READs, direct access WRITEs, and the FIND, DELETE, and REWRITE statements can affect the value of *asv*.

This specifier is valid only for direct access; it is ignored for other access modes.

OPEN: ASYNCHRONOUS Specifier

The ASYNCHRONOUS specifier indicates whether asynchronous I/O is allowed for a unit. It takes the following form:

ASYNCHRONOUS = *asyn*

asyn	Is a scalar expression of type default character that evaluates to one of the following values:

'YES'	Indicates that asynchronous I/O is allowed for a unit.
'NO'	Indicates that asynchronous I/O is not allowed for a unit.

The default is 'NO'.

OPEN: BLANK Specifier

The BLANK specifier indicates how blanks are interpreted in a file. It takes the following form:

BLANK = *blnk*

blnk	Is a scalar default character expression that evaluates to one of the following values:

'NULL'	Indicates all blanks are ignored, except for an all-blank field (which has a value of zero).
'ZERO'	Indicates all blanks (other than leading blanks) are treated as zeros.

The default is 'NULL' (for explicitly OPENed files, preconnected files, and internal files). If you specify compiler option f66 (or OPTIONS/NOF77), the default is 'ZERO'.

If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks.

See Also

Blank Editing for details on the BN and BZ edit descriptors
f66

OPEN: BLOCKSIZE Specifier

The BLOCKSIZE specifier indicates the physical I/O transfer size in bytes for the file. It takes the following form:

BLOCKSIZE = *bks*

bks	Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

If you specify a nonzero number for *bks*, it is rounded up to a multiple of 512 byte blocks. The maximum valid value of BLOCKSIZE is 2147467264.

If you do not specify BLOCKSIZE or you specify zero for *bks*, the default value of 128 KB (131,072 bytes) is assumed. However, if you compile with the `assume buffered_stdout` option, the default blocksize for `stdout` is 8 KB.

The default BLOCKSIZE value can be changed by using the FORT_BLOCKSIZE environment variable or by specifying the BLOCKSIZE parameter on the unit's OPEN. The BLOCKSIZE value can be changed for `stdout` by re-opening the unit corresponding to stdout with an explicit BLOCKSIZE parameter; for example:

```
OPEN(6,ACCESS='SEQUENTIAL',FORM='FORMATTED',BUFFERED='YES',BLOCKSIZE=1048576
```

OPEN: BUFFERCOUNT Specifier

The BUFFERCOUNT specifier indicates the number of buffers to be associated with the unit for multibuffered I/O. It takes the following form:

BUFFERCOUNT = *bc*

bc	Is a scalar numeric expression in the range 1 through 127. If necessary, the value is converted to integer data type before use.

The BLOCKSIZE specifier determines the size of each buffer. For example, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144 bytes.

If you do not specify BUFFERCOUNT or you specify zero for *bc*, the default is 1.

See Also

BLOCKSIZE specifier

OPEN: BUFFERED Specifier

The BUFFERED specifier indicates run-time library behavior for READ and WRITE operations. Buffering of input records only applies to variable length, unformatted records. This specifier takes the following form:

BUFFERED = *bf*

bf Is a scalar default character expression that evaluates to one of the following values:

'NO'	For WRITE operations: Requests that the run-time library send output data to the file system after each operation. For READ operations on variable length, unformatted files: Requests that the run-time library transfer input data directly from disk to user variables for each operation.
'YES'	Requests that the run-time library accumulate data in its internal buffer, possibly across several READ or WRITE operations, before the data is transferred to, or from, the file system. Buffering may improve run-time performance for output-intensive applications.

The default is 'NO' for buffering output. The default is also 'NO' when reading variable length, unformatted records whose length exceeds that of the block size specified for the file.

If BUFFERED='YES' is specified, the request may or may not be honored, depending on the device and other file or connection characteristics.

For direct access, you should specify BUFFERED='YES', although using direct-access I/O to a network file system may be much slower.

If both BLOCKSIZE and BUFFERCOUNT for OPEN have been specified with positive values, their product determines the size in bytes of the buffer for that I/O unit. Otherwise, the default size of the internal buffer is 8 KB (8192 bytes).

NOTE
On Windows systems, the default size of the internal buffer is 1024 bytes if compiler option fpscomp general is used.

The internal buffer will grow to hold the largest single record but will never shrink.

See Also

Rules for Unformatted Sequential READ Statements

OPEN: CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier indicates the type of carriage control used when a file is displayed at a terminal. It takes the following form:

CARRIAGECONTROL = *cc*

cc Is a scalar default character expression that evaluates to one of the following values:

'FORTRAN'	Indicates normal Fortran interpretation of the first character.
'LIST'	Indicates one line feed between records.
'NONE'	Indicates no carriage control processing.

The default for binary (W*S) and unformatted files is 'NONE'. The default for formatted files is 'LIST'. However, if you specify compiler option vms or fpscomp general, and the unit is connected to a terminal, the default is 'FORTRAN'.

On output, if a file was opened with CARRIAGECONTROL='FORTRAN' in effect or the file was processed by the `fortpr` format utility, the first character of a record transmitted to a line printer or terminal is typically a character that is not printed, but is used to control vertical spacing.

See Also

Printing of Formatted Records for details on valid control characters for printing
vms compiler option
fpscomp compiler option

OPEN: CONVERT Specifier

The CONVERT specifier indicates a nonnative numeric format for unformatted data. It takes the following form:

CONVERT = *fm*

fm Is a scalar default character expression that evaluates to one of the following values:

'LITTLE_ENDIAN'[1]	Little endian integer data [2] and IEEE* floating-point data. [3]
'BIG_ENDIAN'[1]	Big endian integer data [2] and IEEE floating-point data. [3]
'CRAY'	Big endian integer data [2] and CRAY* floating-point data of size REAL(8) or COMPLEX(8).
'FDX'	Little endian integer data [2] and VAX* processor floating-point data of format F_floating for REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and IEEE binary128 for REAL(16) or COMPLEX(16).
'FGX'	Little endian integer data [2] and VAX processor floating-point data of format F_floating for REAL(4) or COMPLEX(4),

	G_floating for size REAL(8) or COMPLEX(8), and IEEE binary128 for REAL(16) or COMPLEX(16).
'IBM'	Big endian integer data [2] and IBM* System\370 floating-point data of size REAL(4) or COMPLEX(4) (IBM short 4), and size REAL(8) or COMPLEX(8) (IBM long 8).
'VAXD'	Little endian integer data [2] and VAX processor floating-point data of format F_floating for size REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'VAXG'	Little endian integer data [2] and VAX processor floating-point data of format F_floating for size REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'NATIVE'	No data conversion. This is the default.

[1] INTEGER(1) data is the same for little endian and big endian.

[2] Of the appropriate size: INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8)

[3] Of the appropriate size and type: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), or COMPLEX(16)

You can use CONVERT to specify multiple formats in a single program, usually one format for each specified unit number.

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message appears.

There are other ways to specify numeric format for unformatted files: you can specify an environment variable, compiler option convert, or OPTIONS/CONVERT. The following shows the order of precedence:

Method Used	Precedence
An environment variable	Highest
OPEN (CONVERT=)	.
OPTIONS/CONVERT	.
The convert compiler option	Lowest

Compiler option convert and OPTIONS/CONVERT affect all unit numbers used by the program, while environment variables and OPEN (CONVERT=) affect specific unit numbers.

The following example shows how to code the OPEN statement to read unformatted CRAY* numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20:

```
   OPEN (CONVERT='CRAY', FILE='graph3.dat', FORM='UNFORMATTED',
1      UNIT=15)
   ...
   OPEN (FILE='graph3_native.dat', FORM='UNFORMATTED', UNIT=20)
```

See Also

Data Types, Constants, and Variables for details on supported ranges for data types

convert compiler option

OPEN: DECIMAL Specifier

The DECIMAL specifier controls the representation of the decimal symbol for a connection. It takes the following form:

DECIMAL = *dmode*

dmode Is a scalar default character expression that evaluates to one of the following values:

'COMMA'	Indicates that a decimal comma should be used for decimal editing mode.
'POINT'	Indicates that a decimal point should be used for decimal editing mode.

The default decimal editing mode is 'POINT'.

You can only use this specifier for a formatted I/O connection.

When the mode is DECIMAL='POINT', the decimal point in a numeric input or output value is a period, values are separated by commas in list-directed and NAMELIST I/O, and the separator between the real and imaginary parts of a complex value is a comma.

When the mode is DECIMAL='COMMA', the decimal point in a numeric input or output value is a comma, values are separated by semicolons in list-directed and NAMELIST I/O, and the separator between the real and imaginary parts of a complex value is a semicolon.

The decimal editing mode can be temporarily changed within a READ or WRITE statement by the DECIMAL= specifier or by the corresponding DC and DP edit descriptors.

OPEN: DEFAULTFILE Specifier

The DEFAULTFILE specifier indicates a default file pathname string. It takes the following form:

DEFAULTFILE = *def*

def Is a character expression indicating a default file pathname string.

The default file pathname string is used primarily when accepting file pathnames interactively. File pathnames known to a user program normally appear in the FILE specifier.

DEFAULTFILE supplies a value to the Fortran I/O system that is prefixed to the name that appears in FILE.

If *def* does not end in a slash (/), a slash is added.

If DEFAULTFILE is omitted, the Fortran I/O system uses the current working directory.

OPEN: DELIM Specifier

The DELIM specifier indicates what characters (if any) are used to delimit character constants in list-directed and namelist output. It takes the following form:

DELIM = *del*

del Is a scalar default character expression that evaluates to one of the following values:

'APOSTROPHE'	Indicates apostrophes delimit character constants. All internal apostrophes are doubled.
'QUOTE'	Indicates quotation marks delimit character constants. All internal quotation marks are doubled.
'NONE'	Indicates character constants have no delimiters. No internal apostrophes or quotation marks are doubled.

The default is 'NONE'.

The DELIM specifier is only allowed for files connected for formatted data transfer; it is ignored during input.

OPEN: DISPOSE Specifier

The DISPOSE (or DISP) specifier indicates the status of the file after the unit is closed. It takes one of the following forms:

DISPOSE = *dis*

DISP = *dis*

dis Is a scalar default character expression that evaluates to one of the following values:

'KEEP' or 'SAVE'	Retains the file after the unit closes.
'DELETE'	Deletes the file after the unit closes.
'PRINT' [1]	Submits the file to the line printer spooler and retains it.
'PRINT/DELETE' [1]	Submits the file to the line printer spooler and then deletes it.
'SUBMIT'	Forks a process to execute the file.
'SUBMIT/DELETE'	Forks a process to execute the file, and then deletes the file after the fork is completed.

[1] Use only on sequential files.

The default is 'DELETE' for scratch files. For all other files, the default is 'KEEP'.

OPEN: ENCODING Specifier

The ENCODING specifier indicates the encoding form for a file. It takes the following form:

ENCODING = *enc*

enc — Is a scalar default character expression that evaluates to one of the following values:

'UTF-8'	Indicates that the encoding form of the file is UTF-8 (a unicode file). All characters therein are of ISO 10646 character type, as specified by ISO/IEC 10646-1:2000. This value must not be specified if the processor does not support the ISO 10646 character type.
'DEFAULT'	Indicates that the encoding form of the file is determined by the processor.

The default is 'DEFAULT'.

You can only use this specifier for a formatted I/O connection.

OPEN: FILE Specifier

The FILE specifier indicates the name of the file to be connected to the unit. It takes the following form:

FILE = *name*

name — Is a character or numeric expression.

The *name* can be any pathname allowed by the operating system.

Any trailing blanks in the name are ignored.

If the following conditions occur:

- FILE is omitted
- The unit is not connected to a file
- STATUS='SCRATCH' is not specified
- The corresponding FORT*n* environment variable is not set for the unit number

then Intel® Fortran generates a file name in the form fort.*n*, where *n* is the logical unit number. On Windows systems, if compiler option fpscomp general is specified, omitting FILE implies STATUS='SCRATCH'.

If the file name is stored in a numeric scalar or array, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if it is stored in a character scalar or array, it must not contain a zero byte.

On Windows systems, if the filename is 'USER' or 'CON', input and output are directed to the console. .

In a Windows* QuickWin application, you can specify FILE='USER' to open a child window. All subsequent I/O statements directed to that unit appear in the child window.

On Windows systems, the *name* can be blank (FILE=' ') if the compatibility compiler option fpscomp filesfromcmd is specified. If the *name* is blank, the following occurs:

1. The program reads a filename from the list of arguments (if any) in the command line that started the program. If the argument is a null or blank string (" "), you are prompted for the corresponding filename. Each successive OPEN statement that specifies a blank name reads the next following command-line argument.
2. If no command-line arguments are specified or there are no more arguments in the list, you are prompted for additional filenames.

   Assume the following command line started the program MYPROG (note that quotation marks (") are used):

   myprog first.fil " " third.txt

   MYPROG contains four OPEN statements with blank filenames, in the following order:

```
OPEN (2, FILE = ' ')
OPEN (4, FILE = ' ')
OPEN (5, FILE = ' ')
OPEN (10, FILE = ' ')
```

   Unit 2 is associated with the file FIRST.FIL. Because a blank argument was specified on the command line for the second filename, the OPEN statement for unit 4 produces the following prompt:

```
Filename missing or blank - Please enter name UNIT 4?
```

   Unit 5 is associated with the file THIRD.TXT. Because no fourth file was specified on the command line, the OPEN statement for unit 10 produces the following prompt:

```
Filename missing or blank - Please enter name UNIT 10?
```

See Also

fpscomp compiler option

OPEN: FORM Specifier

The FORM specifier indicates whether the file is being connected for formatted, unformatted, or binary (W*S) data transfer. It takes the following form:

FORM = *fm*

fm Is a scalar default character expression that evaluates to one of the following values:

'FORMATTED'	Indicates formatted data transfer
'UNFORMATTED'	Indicates unformatted data transfer
'BINARY'	Indicates binary data transfer

The default is 'FORMATTED' for sequential access files, and 'UNFORMATTED' for direct access files.

The data is stored and retrieved in a file according to the file's access (set by the ACCESS specifier) and the form of the data the file contains.

A *formatted file* is a sequence of formatted records. Formatted records are a series of ASCII characters terminated by an end-of-record mark (a carriage return and line feed sequence). The records in a formatted direct-access file must all be the same length. The records in a formatted sequential file can have varying lengths. All internal files must be formatted.

An *unformatted file* is a sequence of unformatted records. An unformatted record is a sequence of values. Unformatted direct files contain only this data, and each record is padded to a fixed length with undefined bytes. Unformatted sequential files contain the data plus information that indicates the boundaries of each record.

Binary sequential files are sequences of bytes with no internal structure. There are no records. The file contains only the information specified as I/O list items in WRITE statements referring to the file.

Binary direct files have very little structure. A record length is assigned by the RECL specifier in an OPEN statement. This establishes record boundaries, which are used only for repositioning and padding before and after read and write operations and during BACKSPACE operations. Record boundaries do not restrict the number of bytes that can be transferred during a read or write operation. If an I/O operation attempts to read or write more values than are contained in a record, the read or write operation is continued on the next record.

Fortran standard stream access provides similar functionality to FORM='BINARY'. This is specified using ACCESS='STREAM'.

See Also

Record Access

OPEN: IOFOCUS Specifier (W*S)

The IOFOCUS specifier indicates whether a particular unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof — Is a scalar default logical expression that evaluates to one of the following values:

.TRUE.	Indicates the QuickWin child window is the active window
.FALSE.	Indicates the QuickWin child window is not the active window

If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

A value of .TRUE. causes a call to FOCUSQQ immediately before any READ, WRITE, or PRINT statement to that window. OUTTEXT, OUTGTEXT, or any other graphics routine call does not cause the focus to shift.

OPEN: MAXREC Specifier

The MAXREC specifier indicates the maximum number of records that can be transferred from or to a direct access file while the file is connected. It takes the following form:

MAXREC = *mr*

mr — Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

The default is an unlimited number of records.

OPEN: MODE Specifier

MODE is a nonstandard synonym for ACTION.

OPEN: NAME Specifier

NAME is a nonstandard synonym for FILE.

OPEN: NEWUNIT Specifier

The NEWUNIT specifier opens a file on an unused unit number that is automatically chosen. It also returns the unit number that was chosen. It takes the following form:

NEWUNIT = *u-var*

u-var	Is a scalar integer variable that is assigned the automatically chosen unit number. It is always a negative integer.

If the OPEN is successful, *u-var* can be used in subsequent I/O statements to access the connected file.

If an error occurs during execution of the OPEN statement containing the NEWUNIT= speciifier, the processor does not change the value of the variable.

OPEN: NOSHARED Specifier

The NOSHARED specifier indicates that the file is connected for exclusive access by the program. It takes the following form:

NOSHARED

OPEN: ORGANIZATION Specifier

The ORGANIZATION specifier indicates the internal organization of the file. It takes the following form:

ORGANIZATION = *org*

org Is a scalar default character expression that evaluates to one of the following values

'SEQUENTIAL'	Indicates a sequential file.
'RELATIVE'	Indicates a relative file.

The default is 'SEQUENTIAL'.

OPEN: PAD Specifier

The PAD specifier indicates whether a formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.

The PAD specifier takes the following form:

PAD = *pd*

pd Is a scalar default character expression that evaluates to one of the following values:

'YES'	Indicates the record will be padded with blanks when necessary.
'NO'	Indicates the record will not be padded with blanks. The input record must contain the data required by the input list and format specification.

The default is 'YES'.

This behavior is different from FORTRAN 77, which never pads short records with blanks. For example, consider the following:

```
READ (5,'(I5)') J
```

If you enter 123 followed by a carriage return, FORTRAN 77 turns the I5 into an I3 and J is assigned 123.

However, Intel® Fortran pads the 123 with 2 blanks unless you explicitly open the unit with PAD='NO'.

You can override blank padding by explicitly specifying the BN edit descriptor.

The PAD specifier is ignored during output.

OPEN: POSITION Specifier

The POSITION specifier indicates the position of a file connected for sequential access. It takes the following form:

POSITION = *pos*

pos Is a scalar default character expression that evaluates to one of the following values:

'ASIS'	Indicates the file position is unchanged if the file exists and is already connected. The position is unspecified if the file exists but is not connected.
'REWIND'	Indicates the file is positioned at its initial point.
'APPEND'	Indicates the file is positioned at its terminal point (or before its end-of-file record, if any).

The default is 'ASIS'.

A new file (whether specified as new explicitly or by default) is always positioned at its initial point.

In addition to the POSITION specifier, you can use position statements. The BACKSPACE statement positions a file back one record. The REWIND statement positions a file at its initial point. The ENDFILE statement writes an end-of-file record at the current position and positions the file after it. Note that ENDFILE does not go the end of an existing file, but creates an end-of-file where it is.

OPEN: READONLY Specifier

The READONLY specifier indicates only READ statements can refer to this connection. It takes the following form:

READONLY

READONLY is similar to specifying ACTION='READ', but READONLY prevents deletion of the file if it is closed with STATUS='DELETE' in effect.

The Fortran I/O system's default privileges for file access are READWRITE. If access is denied, the I/O system automatically retries accessing the file for READ access.

However, if you use compiler option vms, the I/O system does not retry accessing for READ access. So, run-time I/O errors can occur if the file protection does not permit WRITE access. To prevent such errors, if you wish to read a file for which you do not have write access, specify READONLY.

OPEN: RECL Specifier

The RECL specifier indicates the length of each record in a file connected for direct access, or the maximum length of a record in a file connected for sequential access.

The RECL specifier takes the following form:

RECL = *rl*

rl	Is a positive numeric expression indicating the length of records in the file. If necessary, the value is converted to integer data type before use.

If the file is connected for formatted data transfer, the value must be expressed in bytes (characters). Otherwise, the value is expressed in 4-byte units (longwords). If the file is connected for unformatted data transfer, the value can be expressed in bytes if compiler option assume byterecl is specified.

Except for segmented records, the *rl* is the length for record data only, it does not include space for control information. If *rl* is too large, you can exhaust your program's virtual memory resources trying to create room for the record.

The length specified is interpreted depending on the type of records in the connected file, as follows:

- For segmented records, RECL indicates the maximum length for any segment (including the four bytes of control information).
- For fixed-length records, RECL indicates the size of each record; it *must* be specified. If the records are unformatted, the size must be expressed as an even multiple of four.

  You can use the RECL specifier in an INQUIRE statement to get the record length before opening the file.
- For variable-length records, RECL indicates the maximum length for any record.

If you read a fixed-length file with a record length different from the one used to create the file, indeterminate results can occur.

The maximum length for *rl* depends on the record type and the setting of the CARRIAGECONTROL specifier, as shown in the following table:

Maximum Record Lengths (RECL)

Record Type	CARRIAGECONTROL	Formatted (size in bytes)
Fixed-length	'NONE'	Unlimited
Variable-length	'NONE'	2147483640 (2**31-8)
Segmented	'NONE'	32764 (2**15-4)
Stream	'NONE'	2147483647 (2**31-1)
Stream_CR	'LIST'	2147483647 (2**31-1)
	'FORTRAN'	2147483646 (2**31-2)
Stream_LF	'LIST'	2147483647 (2**31-1) [1]
	'FORTRAN'	2147483646 (2**31-2)

[1] L*X only

The default value depends on the setting of the RECORDTYPE specifier, as shown in the following table:

Default Record Lengths (RECL)

RECORDTYPE	RECL value
'FIXED'	None; value must be explicitly specified.
All other settings	132 bytes for formatted records; 510 longwords for unformatted records.[1]

[1]To change the default record length values, you can use environment variable FORT_FMT_RECL or FORT_UFMT_RECL.

For formatted records with other than RECORDTYPE='FIXED', the default RECL is 132.

There is a property of list-directed sequential WRITE statements called the *right margin*. If you do not specify RECL as an OPEN statement specifier or in environmental variable FORT_FMT_RECL, the right margin value defaults to 80. When RECL is specified, the right margin is set to the value of RECL. If the length of a

list-directed sequential WRITE exceeds the value of the *right margin* value, the remaining characters will wrap to the next line. Therefore, writing 100 characters will produce two lines of output, and writing 180 characters will produce three lines of output. You can turn off the *right margin* using the `wrap-margin` compiler option or the FORT_FMT_NO_WRAP_MARGIN environment variable.

See Also

assume:byterecl compiler option
vms compiler option
Record Transfer

OPEN: RECORDSIZE Specifier

RECORDSIZE is a nonstandard synonym for RECL.

OPEN: RECORDTYPE Specifier

The RECORDTYPE specifier indicates the type of records in a file. It takes the following form:

RECORDTYPE = *typ*

typ Is a scalar default character expression that evaluates to one of the following values:

'FIXED'	Indicates fixed-length records.
'VARIABLE'	Indicates variable-length records.
'SEGMENTED'	Indicates segmented records.
'STREAM'	Indicates stream-type variable length data with no record terminators.
'STREAM_LF'	Indicates stream-type variable length records, terminated with a line feed.
'STREAM_CR'	Indicates stream-type variable length records, terminated with a carriage return.
'STREAM_CRLF'	Indicates stream-type variable length records, terminated with a carriage return/line feed pair.

When you open a file, default record types are as follows:

'FIXED'	For relative files
'FIXED'	For direct access sequential files
'STREAM_LF'	For formatted sequential access files on Linux* and macOS* systems
'STREAM_CRLF'	For formatted sequential access files on Windows systems
'VARIABLE'	For unformatted sequential access files

A *segmented record* is a logical record consisting of segments that are physical records. Since the length of a segmented record can be greater than 65,535 bytes, only use segmented records for unformatted sequential access to disk or raw magnetic tape files.

Files containing segmented records can be accessed only by unformatted sequential data transfer statements.

If an output statement does not specify a full record for a file containing fixed-length records, the following occurs:

- In formatted files, the record is filled with blanks
- In unformatted files, the record is filled with zeros

OPEN: ROUND Specifier

The ROUND specifier indicates the I/O rounding mode for the duration of a connection. It takes the following form:

ROUND = *rmode*

rmode Is a scalar default character expression that evaluates to one of the following values:

'UP'	The smallest representable value that is greater than or equal to the original value.
'DOWN'	The largest representable value that is less than or equal to the original value.
'ZERO'	The value closest to the original value, but no greater in magnitude than the original value.
'NEAREST'	Conforms to the ISO/IEC/IEEE 60559:2011 standard specification for `roundTiesToEven`.
'COMPATIBLE'	The closer of the two nearest representable values. If the value is halfway between the two values, the one chosen is the one farther from zero.
'PROCESSOR_DEFINED'	The value is determined by the default settings in the processor, which may correspond to one of the other modes.

The default I/O rounding mode is 'PROCESSOR_DEFINED'. For `ifort`, this corresponds to 'NEAREST'.

The rounding modes conform to the corresponding rounding modes specified in the ISO/IEC/IEEE 60559:2011 standard.

You can only use this specifier for a formatted I/O connection.

The rounding mode can be temporarily changed within a READ or WRITE statement by the corresponding RU, RD, RZ, RN, RC, and RP edit descriptors.

OPEN: SHARE Specifier

The SHARE specifier indicates whether file locking is implemented while the unit is open. It takes the following form:

SHARE = *shr*

shr

Is a scalar default character expression.

On Windows* systems, this expression evaluates to one of the following values:

'DENYRW'	Indicates deny-read/write mode. No other process can open the file.
'DENYWR'	Indicates deny-write mode. No process can open the file with write access.
'DENYRD'	Indicates deny-read mode. No process can open the file with read access.
'DENYNONE'	Indicates deny-none mode. Any process can open the file in any mode.

On Linux* and macOS* systems, this expression evaluates to one of the following values:

'DENYRW'	Indicates exclusive access for cooperating processes.
'DENYNONE'	Indicates shared access for cooperating processes.

On Windows systems, the default is 'DENYWR'. However, if you specify compiler option fpscomp general or the SHARED specifier, the default is 'DENYNONE'.

On Linux and macOS* systems, no restrictions are applied to file opening if you do not use a locking mechanism.

'COMPAT' is accepted for compatibility with previous versions. It is equivalent to 'DENYNONE'.

Use the ACCESS specifier in an INQUIRE statement to determine the access permission for a file.

Be careful not to permit other users to perform operations that might cause problems. For example, if you open a file intending only to read from it, and want no other user to write to it while you have it open, you could open it with ACTION='READ' and SHARE='DENYRW'. Other users would not be able to open it with ACTION='WRITE' and change the file.

Suppose you want several users to read a file, and you want to make sure no user updates the file while anyone is reading it. First, determine what type of access to the file you want to allow the original user. Because you want the initial user to read the file only, that user should open the file with ACTION='READ'. Next, determine what type of access the initial user should allow other users; in this case, other users should be able only to read the file. The first user should open the file with SHARE='DENYWR'. Other users can also open the same file with ACTION='READ' and SHARE='DENYWR'.

See Also

fpscomp compiler option

OPEN: SHARED Specifier

The SHARED specifier indicates that the file is connected for shared access by more than one program executing simultaneously. It takes the following form:

SHARED

On Linux* and macOS* systems, shared access is the default for the Fortran I/O system. On Windows* systems, it is the default if SHARED or compiler option fpscomp general is specified.

See Also

fpscomp compiler option

OPEN: SIGN Specifier

The SIGN specifier controls the optional plus characters in formatted numeric output created during the connection. It takes the following form:

SIGN = *sn*

sn Is a scalar default character expression that evaluates to one of the following values:

'PLUS'	Indicates that the processor should produce a plus sign in any subsequent position where it would be otherwise optional. This has the same effect as the SP edit descriptor.
'SUPPRESS'	Indicates that the processor should suppress a plus sign in any subsequent position where it would be otherwise optional. This has the same effect as the SS edit descriptor.
'PROCESSOR_DEFINED'	Indicates that the processor determines whether a plus sign is added in any subsequent position where it would be otherwise optional. This has the same effect as the S edit descriptor.

The default is 'PROCESSOR_DEFINED'.

The setting can be overwritten by a SIGN= specifier in a WRITE statement.

OPEN: STATUS Specifier

The STATUS specifier indicates the status of a file when it is opened. It takes the following form:

STATUS = *sta*

sta Is a scalar default character expression that evaluates to one of the following values:

'OLD'	Indicates an existing file.
'NEW'	Indicates a new file; if the file already exists, an error occurs. Once the file is created, its status changes to 'OLD'.
'SCRATCH'	Indicates a new file that is unnamed (called a scratch file). When the file is closed or the program terminates, the scratch file is deleted.
'REPLACE'	Indicates the file replaces another. If the file to be replaced exists, it is deleted and a new file is created with the same name. If the file to be replaced does not exist, a new file is created and its status changes to 'OLD'.
'UNKNOWN'	Indicates the file may or may not exist. If the file does not exist, a new file is created and its status changes to 'OLD'.

Scratch files go into a temporary directory and are visible while they are open. Scratch files are deleted when the unit is closed or when the program terminates normally, whichever occurs first.

To specify the path for scratch files, you can use one of the following environment variables:

- On Windows*: FORT_TMPDIR, TMP, or TEMP, searched in that order
- On Linux* and macOS*: FORT_TMPDIR or TMPDIR, searched in that order

If no environment variable is defined, the default is the current directory.

The default is 'UNKNOWN'. This is also the default if you implicitly open a file by using WRITE. However, if you implicitly open a file using READ, the default is 'OLD'. If you specify compiler option f66 (or OPTIONS/NOF77), the default is 'NEW'.

NOTE
The STATUS specifier can also appear in CLOSE statements to indicate the file's status after it is closed. However, in CLOSE statements the STATUS values are the same as those listed for the DISPOSE specifier.

See Also

f66 compiler option

OPEN: TITLE Specifier (W*S)

The TITLE specifier indicates the name of a child window in a QuickWin application. It takes the following form:

TITLE = *name*

name	Is a character expression.

If TITLE is specified in a non-Quickwin application, a run-time error occurs.

OPEN: TYPE Specifier

TYPE is a nonstandard synonym for STATUS.

OPEN: USEROPEN Specifier

The USEROPEN specifier lets you pass control to a routine that directly opens a file. The file can use system calls or library routines to establish a special context that changes the effect of subsequent Fortran I/O statements.

The USEROPEN specifier takes the following form:

USEROPEN = *function-name*

function-name	Is the name of an external function. The external function can be written in Fortran, C, or other languages. The return value is the *file descriptor*. On Linux and macOS*, the *file descriptor* is a 4-byte integer on both 32-bit and 64-bit systems. On Windows, the *file descriptor* is a 4-byte integer on 32-bit systems and an 8-byte integer on 64-bit systems.

If the function is written in Fortran, do not execute a Fortran OPEN statement to open the file named in USEROPEN.

The Intel® Fortran Run-time Library (RTL) I/O support routines call the function named in USEROPEN in place of the system calls normally used when the file is first opened for I/O.

On Windows* systems, the Fortran RTL normally calls CreateFile( ) to open a file. When USEROPEN is specified, the called function opens the file (or pipe, etc.) by using CreateFile( ) and returns the *handle* of the file (return value from CreateFile( )) when it returns control to the calling Fortran program.

On Linux* and macOS* systems, the Fortran RTL normally calls the open function to open a file. When USEROPEN is specified, the called function opens the file by calling open and returns the file descriptor of the file when it returns control to the calling Fortran program.

When opening the file, the called function usually specifies options different from those provided by a normal Fortran OPEN statement.

NOTE
You may get unexpected results if you specify OPEN with a filename and a USEROPEN specifier that opens a different filename, and then use a CLOSE statement with STATUS=DELETE (or DISPOSE=DELETE). In this case, the run-time library assumes you want to delete the file named in the OPEN statement, not the one you specified in the USEROPEN function.

For more information about how to use the USEROPEN specifier, see User-Supplied OPEN Procedures/ USEROPEN Specifier.

Compilation Control Lines and Statements

In addition to specifying options on the compiler command line, you can specify the following lines and statements in a program unit to influence compilation:

- The INCLUDE Line

  Incorporates external source code into programs.

- The OPTIONS Statement

  Sets options usually specified in the compiler command line. OPTIONS statement settings override command line options.

Directive Enhanced Compilation

Directive enhanced compilation is performed by using compiler directives. Compiler directives are special commands that let you perform various tasks during compilation. They are similar to compiler options, but can provide more control within your program.

Compiler directives are preceded by a special prefix that identifies them to the compiler.

Syntax Rules for Compiler Directives

The following syntax rules apply to all general and OpenMP* Fortran compiler directives. You must follow these rules precisely to compile your program properly and obtain meaningful results.

A directive prefix (tag) takes one of the following forms:

General compiler directives:	!DIR$
OpenMP Fortran compiler directives:	!$OMP

The following prefix forms can be used in place of !DIR$: *c*DIR$, *c*DEC$, or !MS$, where *c* is one of the following: C (or c), !, or *.

The following prefix forms can be used in place of !$OMP: *c*$OMP, where *c* is one of the following: C (or c), !, or *.

The following are source form rules for directive prefixes:

- In fixed and tab source forms, prefixes begin with !, C (or c), or *.

  The prefix must appear in columns 1 through 5; column 6 must be a blank or a tab (except for prefix !MS $). From column 7 on, blanks are insignificant, so the directive can be positioned anywhere on the line after column 6.
- In free source form, prefixes begin with !.

  The prefix can appear in any column, but it cannot be preceded by any nonblank, nontab characters on the same line.
- In all source forms, directives spelled with two keywords can be separated by an optional space; for example, "LOOP COUNT" and "LOOPCOUNT" are both valid spellings for the same directive. However, when a directive name is preceded by the prefix "NO", this is not considered to be two keywords. For example, "NO DECLARE" is not a valid spelling; the only valid spelling for this directive is "NODECLARE".

A compiler directive ends in column 72 (or column 132, if compiler option extend-source is specified).

General compiler directives and OpenMP Fortran directives can be continued in the same way as Fortran statements can be continued:

- In fixed form, the first line of the directive {initial line} has the directive prefix is in columns 1 through 5 and has a blank, a tab, or a zero in column 6; each continued line of the directive has the directive prefix in columns 1 through 5 and has a character other than a blank, a tab, or a zero in column 6.
- In free form, the initial line of the directive ends with an ampersand followed by an optional comment beginning with an exclamation point. Each continued line of the directive has the directive prefix optionally preceded by blanks or tabs, followed by an ampersand optionally preceded by blanks or tabs.

A comment beginning with an ! can follow a compiler directive on the same line.

Additional Fortran statements (or directives) cannot appear on the same line as the compiler directive.

Compiler directives cannot appear within a continued Fortran statement.

Blank common used in a compiler directive is specified by two slashes (/ /).

If the source line starts with a valid directive prefix but the directive is not recognized, the compiler displays an informational message and ignores the line.

General Compiler Directives

Compiler directives are specially formatted comments in the source file which provide information to the compiler. Some directives, such as line length or conditional compilation directives provide the compiler information which is used in interpreting the source file. Other directives, such as optimization directives provide hints or suggestions to the compiler, which, in some cases, may be ignored or overridden by the compiler based on the heuristics of the optimizer and/or code generator. You do not need to specify a compiler option to enable general directives. If the directive is ignored by the compiler, no diagnostic message is issued.

The following general compiler directives are available:

- ALIAS

  Specifies an alternate external name to be used when referring to external subprograms.
- ASSUME

  Provides heuristic information to the compiler optimizer.
- ASSUME_ALIGNED

  Specifies that an entity in memory is aligned.
- ATTRIBUTES

  Specifies properties for data objects and procedures.
- BLOCK_LOOP and NOBLOCK_LOOP

  Enables or disables loop blocking for the immediately following nested DO loops.
- DECLARE and NODECLARE

  Generates or disables warnings for variables that have been used but not declared.
- DEFINE and UNDEFINE

  Defines (or undefines) a symbolic variable whose existence (or value) can be tested during conditional compilation.
- DISTRIBUTE POINT

  Suggests a location at which a DO loop may be split.
- FIXEDFORMLINESIZE

  Sets the line length for fixed-form source code.
- FMA and NOFMA

  Enables (or disables) the compiler to allow generation of fused multiply-add (FMA) instructions, also known as floating-point contractions.
- FREEFORM and NOFREEFORM

  Specifies free-format or fixed-format source code.
- IDENT

  Specifies an identifier for an object module.
- IF and IF DEFINED

  Specifies a conditional compilation construct.
- INLINE, FORCEINLINE, NOINLINE

  Tell the compiler to perform the specified inlining on routines within statements or DO loops.
- INTEGER

  Specifies the default integer kind.
- IVDEP

  Assists the compiler's dependence analysis of iterative DO loops.
- LOOP COUNT

  Specifies the typical trip count for a DO loop; this assists the optimizer.

- MESSAGE

  Specifies a character string to be sent to the standard output device during the first compiler pass.
- NOFUSION

  Prevents a loop from fusing with adjacent loops.
- OBJCOMMENT

  Specifies a library search path in an object file.
- OPTIMIZE and NOOPTIMIZE

  Enables or disables optimizations for the program unit.
- OPTIONS

  Affects data alignment and warnings about data alignment.
- PACK

  Specifies the memory alignment of derived-type items.
- PARALLEL and NOPARALLEL

  Facilitates or prevents auto-parallelization by assisting the compiler's dependence analysis of theimmediately following DO loop.
- PREFETCH and NOPREFETCH

  Enables or disables hint to the compiler to prefetch data from memory.
- PSECT

  Modifies certain characteristics of a common block.
- REAL

  Specifies the default real kind.
- SIMD

  Requires and controls SIMD vectorization of loops.
- STRICT and NOSTRICT

  Disables or enables language features not found in the language standard specified on the command line.
- UNROLL and NOUNROLL

  Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop.
- UNROLL_AND_JAM and NOUNROLL_AND_JAM

  Enables or disables loop unrolling and jamming.
- VECTOR and NOVECTOR

  Overrides default heuristics for vectorization of DO loops.

See Also

Syntax Rules for Compiler Directives

Rules for General Directives that Affect DO Loops

Rules for Placement of Directives

Directives can have different effects depending on placement:

- Some directives affect the next statement only.
- Some directives can appear only at the beginning of a scope and affect the rest of that scope.
- Some directives affect the statements starting at that point in the source and continue until changed by another directive or until the end of the scope containing the directive.

The scope can be a contained procedure or a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. A directive does not affect modules invoked with the USE statement in the program unit that contains it, but it does affect INCLUDE statements that follow the directive.

Certain directives may appear before program units or between program units in a source file. These directives affect only the next program unit that lexically follows the directive. The effect of the directive ceases at the end of the affected program unit. For example:

```
!dir$ integer:2
program m
integer k
print *, kind(k), kind(42)    ! this prints 2, 2 which means the directive took effect
call sub()
end

subroutine sub()
integer kk
print *, kind(kk), kind(-42)  ! this prints 4, 4 because the INTEGER:2 directive has no effect
here
end
```

The following directives have this behavior:

- ALIAS
- ATTRIBUTES
- DECLARE and NODECLARE
- DEFINE and UNDEFINE
- FIXEDFORMLINESIZE
- FREEFORM and NOFREEFORM
- IDENT
- INTEGER
- OPTIMIZE and NOOPTIMIZE
- PACK
- PREFETCH
- PSECT
- REAL
- STRICT and NOSTRICT

Other rules may apply to these directives. For more information, see the description of each directive.

Rules for General Directives that Affect DO Loops

This table lists the general directives that affect DO loops:

BLOCK_LOOP and NOBLOCK_LOOP	PARALLEL and NOPARALLEL
CODE_ALIGN	PREFETCH and NOPREFETCH
DISTRIBUTE POINT	SIMD
FORCEINLINE	UNROLL and NOUNROLL
INLINE and NOINLINE	UNROLL_AND_JAM
IVDEP	VECTOR and NOVECTOR

LOOP COUNT
NOFUSION
NOUNROLL_AND_JAM

The following rules apply to all of the general directives:

- The directive must precede the DO statement for each DO loop it affects. The DO statement can be any of the following:
  - A counted DO loop
  - A DO WHILE loop
  - A DO CONCURRENT loop
- No source code lines, other than the following, can be placed between the directive statement and the DO statement:
  - One of the other general directives that affect DO loops
  - An OpenMP* Fortran PARALLEL DO directive
  - Comment lines
  - Blank lines

Other rules may apply to these directives. For more information, see the description of each directive.

See Also

Rules for Loop Directives that Affect Array Assignment Statements

Rules for Loop Directives that Affect Array Assignment Statements

When certain loop directives precede an array assignment statement, they affect the implicit loops that are generated by the compiler.

The following loop directives can affect array assignment statements:

BLOCK_LOOP and NOBLOCK_LOOP	NOVECTOR
CODE_ALIGN	PARALLEL and NOPARALLEL
DISTRIBUTE POINT	PREFETCH and NOPREFETCH
FORCEINLINE	SIMD
INLINE and NOINLINE	UNROLL and NOUNROLL
IVDEP	UNROLL_AND_JAM
LOOP COUNT	VECTOR
NOFUSION	
NOUNROLL_AND_JAM	

Usually only one of the general directives can precede the array assignment statement (`one-dimensional-array = expression`) to affect it. The BLOCK_LOOP and NOBLOCK_LOOP directives may precede array assignment statements with any rank variable on the left-hand-side of the assignment.

Other rules may apply to the general directives. For more information, see the description of each directive.

Examples

Consider the following:

```
            REAL A(10), B(10)
            ...
       !DIR$ IVDEP
            A = B + 3
```

This has the same effect as writing the following explicit loop:

```
      !DIR$ IVDEP
            DO I = 1, 10
               A (I) = B (I) + 3
            END DO
```

See Also

Rules for General Directives that Affect DO Loops

OpenMP* Fortran Compiler Directives

Intel® Fortran supports OpenMP* Fortran compiler directives that comply with OpenMP Fortran Application Program Interface (API) specification Version TR4: Version 5.0.

To use these directives, you must specify compiler option `-qopenmp` (Linux* and macOS*) or `/Qopenmp` (Windows*).

OpenMP directives are specially formatted Fortran comment lines embedded in the source file which provide the compiler with hints and suggestions for parallelization, optimization, vectorization, and offloading code to accelerator hardware. The compiler uses the information specified in the directives with compiler heuristic algorithms to generate more efficient code. At times, these heuristics may choose to ignore or override the information provided by a directive. If the directive is ignored by the compiler, no diagnostic message is issued.

Options that use OpenMP* are available for both Intel® microprocessors and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP constructs and features that may perform differently on Intel® microprocessors vs. non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, thread affinity, and binding.

This section discusses clauses used in multiple OpenMP* Fortran directives, conditional compilation rules, nesting and binding rules, and the following directives:

- ATOMIC directive

  Specifies that a specific memory location is to be updated atomically.
- BARRIER directive

  Synchronizes all the threads in a team.
- CANCEL directive

  Requests cancellation of the innermost enclosing region of the type specified, and causes the encountering implicit or explicit task to proceed to the end of the canceled construct.
- CANCELLATION POINT directive

  Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified.
- CRITICAL directive

  Restricts access for a block of code to only one thread at a time.
- DECLARE REDUCTION directive

  Declares a user-defined reduction for one or more types.
- DECLARE SIMD directive

Generates a SIMD procedure.

- DECLARE TARGET directive

  Causes the creation of a device-specific version of a named routine that can be called from a target region.
- DISTRIBUTE directive

  Specifies that loop iterations will be executed by thread teams in the context of their implicit tasks.
- DISTRIBUTE PARALLEL DO directive

  Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
- DISTRIBUTE PARALLEL DO SIMD directive

  Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
- DISTRIBUTE SIMD

  Specifies a loop that will be distributed across the master threads of the teams region. It will be executed concurrently using SIMD instructions.
- DO directive

  Specifies that the iterations of the immediately following DO loop must be executed in parallel.
- DO SIMD directive

  Specifies a loop that can be executed concurrently using SIMD instructions.
- FLUSH directive

  Specifies synchronization points where the threads in a team must have a consistent view of memory.
- MASTER directive

  Specifies a block of code to be executed by the master thread of the team.
- ORDERED directive

  Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations.
- PARALLEL directive

  Defines a parallel region.
- PARALLEL DO directive

  Defines a parallel region that contains a single DO directive.
- PARALLEL DO SIMD directive

  Specifies a loop that can be executed concurrently using SIMD instructions. It provides a shortcut for specifying a PARALLEL construct containing one SIMD loop construct and no other statement.
- PARALLEL SECTIONS directive

  Defines a parallel region that contains a single SECTIONS directive.
- PARALLEL WORKSHARE directive

  Defines a parallel region that contains a single WORKSHARE directive.
- SCAN directive

  Specifies a scan computation that updates each list item in each iteration of the loop the directive appears in.
- SECTIONS directive

  Specifies that the enclosed SECTION directives define blocks of code to be divided among threads in a team.
- SIMD directive

  Requires and controls SIMD vectorization of loops.
- SINGLE directive

  Specifies a block of code to be executed by only one thread in a team at a time.
- TARGET directive

  Creates a device data environment and executes the construct on the same device.
- TARGET DATA directive

  Creates a device data environment for the extent of the region.

- TARGET ENTER DATA

  Specifies that variables are mapped to a device data environment.
- TARGET EXIT DATA

  Specifies that variables are unmapped from a device data environment.
- TARGET PARALLEL

  Creates a device data environment in a parallel region and executes the construct on that device.
- TARGET PARALLEL DO

  Provides an abbreviated way to specify a TARGET directive containing a PARALLEL DO directive and no other statements.
- TARGET PARALLEL DO SIMD

  Specifies a TARGET construct that contains a PARALLEL DO SIMD construct and no other statement.
- TARGET SIMD

  Specifies a TARGET construct that contains a SIMD construct and no other statement.
- TARGET TEAMS

  Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the master thread in each team executing the structured block.
- TARGET TEAMS DISTRIBUTE

  Creates a device data environment and executes the construct on the same device. It also specifies that loop iterations will be shared among the master threads of all thread teams in a league created by a TEAMS construct.
- TARGET TEAMS DISTRIBUTE PARALLEL DO

  Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct.
- TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD

  Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.
- TARGET TEAMS DISTRIBUTE SIMD

  Creates a device data environment and executes the construct on the same device. It also specifies that loop iterations will be shared among the master threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
- TARGET UPDATE directive

  Makes the list items in the device data environment consistent with their corresponding original list items.
- TASK directive

  Defines a task region.
- TASKGROUP directive

  Specifies a wait for the completion of all child tasks of the current task and all of their descendant tasks.
- TASKLOOP directive

  Specifies that the iterations of one or more associated DO loops should be executed in parallel using OpenMP* tasks. The iterations are distributed across tasks that are created by the construct and scheduled to be executed.
- TASKLOOP SIMD directive

  Specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using OpenMP* tasks.
- TASKWAIT directive

  Specifies a wait on the completion of child tasks generated since the beginning of the current task.
- TASKYIELD directive

  Specifies that the current task can be suspended at this point in favor of execution of a different task.
- TEAMS

Creates a group of thread teams to be used in a parallel region.

- TEAMS DISTRIBUTE

  Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies that loop iterations will be shared among the master threads of all thread teams in a league created by a TEAMS construct.
- TEAMS DISTRIBUTE PARALLEL DO

  Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
- TEAMS DISTRIBUTE PARALLEL DO SIMD

  Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams. The loop will be distributed across the master threads of the teams region, which will be executed concurrently using SIMD instructions.
- TEAMS DISTRIBUTE SIMD

  Creates a league of thread teams to execute the structured block in the master thread of each team. It also specifies a loop that will be distributed across the master threads of the teams region. The loop will be executed concurrently using SIMD instructions.
- THREADPRIVATE directive

  Makes named common blocks private to each thread, but global within the thread.
- WORKSHARE directive

  Divides the work of executing a block of statements or constructs into separate units.

The OpenMP parallel directives can be grouped into the categories. For more information about the categories for these directives, see OpenMP* Directives Summary.

See Also

Syntax Rules for Compiler Directives

qopenmp, Qopenmp compiler option

Clauses Used in Multiple OpenMP* Fortran Directives

The OpenMP web site http://www.openmp.org/

Clauses Used in Multiple OpenMP* Fortran Directives

This topic summarizes clauses that are used in more than one OpenMP* Fortran directive.

Other clauses (or keywords) are available for some OpenMP* Fortran directives. For more information, see each directive description.

Some of the OpenMP* Fortran directives have clauses (or options) you can specify to control the scope attributes of variables for the duration of the directive.

Data Scope Attribute Clauses

Name	Description
DEFAULT	Lets you specify a scope for all variables in the lexical extent of a parallel region.
FIRSTPRIVATE	Provides a superset of the functionality provided by the PRIVATE clause. It declares one or more variables to be private to each thread in a team, and initializes each of them with the value of the corresponding original variable.
IN_REDUCTION	Specifies that a task participates in a reduction.
LASTPRIVATE	Provides a superset of the functionality provided by the PRIVATE clause. It declares one or more variables to be private to an implicit task, and causes the corresponding original variable to be updated after the end of the region.
LINEAR	Specifies that all variables in a list are private to a SIMD task and that they have a linear relationship within the iteration space of a loop.
PRIVATE	Declares one or more variables to be private to each thread in a team.
REDUCTION	Performs a reduction operation on the specified variables.
SHARED	Specifies variables that will be shared by all the threads in a team.

The data copying clauses let you copy data values from private or threadprivate variables in one implicit task or thread to the corresponding variables in other implicit tasks or threads in the team.

Data Copying Clauses

Name	Description
COPYIN	Specifies that the data in the master thread of the team is to be copied to the thread private copies of the common block at the beginning of the parallel region.
COPYPRIVATE	Uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. The COPYPRIVATE clause can only appear in the END SINGLE directive.

The data motion clause MAP is used in OpenMP* Fortran TARGET directives. This data motion clause does not modify the values of any of the internal control variables (ICVs).

List items that appear in this data motion clause may have corresponding new list items created in the device data environment that is associated with the construct. If a new list item is created, a new list item of the same type, kind, and rank is allocated. The initial value of the new list item is undefined.

The original list items and new list items may share storage. This means that data races can occur. Data races are caused by unintended sharing of data; for example, when WRITEs to either item by one task or device are followed by a READ of the other item by another task or device without intervening synchronization.

Data Motion Clause

Name	Description
MAP	Maps a variable from the data environment of the current task to the data environment of the device associated with the construct.

The following are other clauses that can be used in more than one OpenMP* Fortran directive.

Miscellaneous Clauses

Name	Description
ALIGNED	Specifies that all variables in a list are aligned.
COLLAPSE	Specifies how many loops are associated with the loop construct.
DEPEND	Enforces additional constraints on the scheduling of a task by enabling dependences between sibling tasks in the task region.
DEVICE	Specifies the target device for certain TARGET directives.
FINAL	Specifies that the generated task will be a final task.
IF	Specifies a conditional expression. If the expression evaluates to .FALSE., the construct is not executed.
MERGEABLE	Specifies that the implementation may generate a merged task.
NOWAIT	Specifies that threads may resume execution before the execution of the region completes.
PRIORITY	Specifies that the generated tasks have the indicated priority for execution.
UNTIED	Specifies that the task is never tied to the thread that started its execution.

Conditional Compilation Rules

The OpenMP* Fortran API lets you conditionally compile Intel® Fortran statements if you use the appropriate directive prefix.

The prefix depends on which source form you are using, although !$ is valid in all forms.

The prefix must be followed by a valid Intel Fortran statement on the same line.

Free Source Form

The free source form conditional compilation prefix is !$. This prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Free-form source rules apply to the directive line.

Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix with optional white space before and after the ampersand.

Fixed Source Form

For fixed source form programs, the conditional compilation prefix is one of the following: !$, C$ (or c$), or * $.

The prefix must start in column one and appear as a single string with no intervening white space. Fixed-form source rules apply to the directive line.

After the prefix is replaced with two spaces, initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six. For example, the following forms for specifying conditional compilation are equivalent:

```
c23456789
!$    IAM = OMP_GET_THREAD_NUM( ) +
!$   *      INDEX
#IFDEF _OPENMP
      IAM = OMP_GET_THREAD_NUM( ) +
     *      INDEX
#ENDIF
```

Nesting and Binding Rules

This section describes the dynamic nesting and binding rules for OpenMP* Fortran API directives.

Binding Rules

The following rules apply to dynamic binding:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO directive.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL directive.

Nesting Rules

The following rules apply to dynamic nesting:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other.
- DO, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Examples

The following example shows nested PARALLEL regions:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
        DO I =1, N
!$OMP PARALLEL SHARED(I,N)
!$OMP DO
        DO J =1, N
          CALL WORK(I,J)
        END DO
!$OMP END PARALLEL
      END DO
!$OMP END PARALLEL
```

Note that the inner and outer DO directives bind to different PARALLEL regions.

The following shows a variation of the preceding example:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
        DO I =1, N
           CALL SOME_WORK(I,N)
        END DO
!$OMP END PARALLEL
      ...
      SUBROUTINE SOME_WORK(I,N)
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
        DO J =1, N
           CALL WORK(I,J)
        END DO
!$OMP END PARALLEL
        RETURN
        END
```

Equivalent Compiler Options

Some compiler directives and compiler options have the same effect, as shown in the table below. However, compiler directives can be turned on and off throughout a program, while compiler options remain in effect for the whole compilation unless overridden by a compiler directive.

The following table shows compiler directives and equivalent command-line compiler options:

Compiler Directive	Equivalent Command-Line Compiler Option
DECLARE	`-warn declarations` (Linux* and macOS*) `/warn:declarations` (Windows*)
NODECLARE	`-warn nodeclarations` (Linux and macOS*) `/warn:nodeclarations` (Windows)
DEFINE *name*	`-D`*name* (Linux and macOS*) `/D`*name* (Windows)
FIXEDFORMLINESIZE:*length*	`-extend-source` [*size*] (Linux and macOS*) `/extend-source`[:*size*] (Windows)

Compiler Directive	Equivalent Command-Line Compiler Option
FREEFORM	`-free` or `-nofixed` (Linux and macOS*) `/free` or `/nofixed` (Windows)
NOFREEFORM	`-nofree` or `-fixed` (Linux and macOS*) `/nofree` or `/fixed` (Windows)
INTEGER:*size*	`-integer_size`*size* (Linux and macOS*) `/integer_size:`*size* (Windows)
OBJCOMMENT	`/libdir:user` (Windows)
OPTIMIZE[: *n*]	`-O`*n* (Linux and macOS*) `/O`*n* (Windows)
NOOPTIMIZE	`-O0` (Linux and macOS*) `/Od` (Windows)
PACK:*alignment*	`-align` *recnbytes* (Linux and macOS*) `/align:`*recnbytes* (Windows)
REAL:*size*	`-real-size`*size* (Linux and macOS*) `/real-size:`*size* (Windows)
STRICT	`-warn stderrors` with `-stand f08` (Linux and macOS*) `/warn:stderrors` with `/stand:f08` (Windows)
NOSTRICT	`-warn nostderrors` (Linux and macOS*) `/warn:nostderrors` (Windows)

Scope and Association

Program entities are identified by names, labels, input/output unit numbers, operator symbols, or assignment symbols. For example, a variable, a derived type, or a subroutine is identified by its name.

Scope refers to the area in which a name is recognized. A scoping unit is the program or part of a program in which a name is defined or known. It can be any of the following:

- An entire executable program
- A single scoping unit
- A single statement (or part of a statement)

The region of the program in which a name is known and accessible is referred to as the scope of that name. These different scopes allow the same name to be used for different things in different regions of the program.

Association is the language concept that allows different names to refer to the same entity in a particular region of a program.

Scope

Program entities have the following kinds of scope (as shown in the table below):

- Global

  Entities that are accessible throughout an executable program. The name of a global entity must be unique. It cannot be used to identify any other global entity in the same executable program.
- Scoping unit (Local scope)

  Entities that are declared within a scoping unit. These entities are local to that scoping unit. The names of local entities are divided into classes (see the table below).

  A *scoping unit* is one of the following:

  - A derived-type definition
  - A procedure interface body (excluding any derived-type definitions and interface bodies contained within it)
  - A program unit or subprogram (excluding any derived-type definitions, interface bodies, and subprograms contained within it)

  A scoping unit that immediately surrounds another scoping unit is called the host scoping unit. Named entities within the host scoping unit are accessible to the nested scoping unit by host association, unless access is blocked by an IMPORT ONLY or IMPORT NONE statement. For information about host association, see Use and Host Association.

  Once an entity is declared in a scoping unit, its name can be used throughout that scoping unit. An entity declared in another scoping unit is a different entity even if it has the same name and properties.

  Within a scoping unit, a local entity name that is not generic or an OpenMP* reduction identifier must be unique within its class. However, the name of a local entity in one class can be used to identify a local entity of another class.

  Within a scoping unit, a generic name can be the same as any one of the procedure names in the interface block. A generic name can be the same as the name of a derived type.

  An OpenMP* reduction identifier can be the same as any local entity name.

  A component name has the same scope as the derived type of which it is a component. It can appear only within a component designator of a structure of that type.

  For information on interactions between local and global names, see the table below.
- Statement

  Entities that are accessible only within a statement or part of a statement; such entities cannot be referenced in subsequent statements.

  The name of a statement entity can also be the name of a common block, or a local or construct variable accessible in the same scoping unit or construct; in this case, the name is interpreted within the statement as that of the statement entity.
- Construct

  Construct entities are accessible during the execution of the construct in which they are declared. Entities declared in the specification part of a BLOCK construct, associate names in ASSOCIATE, SELECT RANK, and SELECT TYPE constructs, and index variables in FORALL statements or constructs, or DO CONCURRENT constructs are construct entities. An index name of an inner FORALL statement or construct, or of a DO CONCURRENT construct cannot be the same as that of an index variable of an outer (containing) FORALL or DO CONCURRENT construct.

  An associate name of a SELECT TYPE construct has a different scope for each block of the construct; the declared type, dynamic type, type parameters, rank or bounds may differ from block to block. An associate name for a SELECT RANK construct has a different scope for each block of the construct since the rank of the associate name is different in each block. An associate name for an ASSOCIATE construct has the scope of the ASSOCIATE construct.

Scope of Program Entities

Entity	Scope
Program units	Global
Common blocks[1]	Global

Entity	Scope	
External procedures	Global	
Intrinsic procedures	Global[2]	
Module procedures	Local	Class I
Internal procedures	Local	Class I
Dummy procedures	Local	Class I
Statement functions	Local	Class I
Derived types	Local or construct	Class I
Components of derived types	Local	Class II
Named constants	Local or construct	Class I
Named constructs	Local or construct	Class I
Namelist group names	Local	Class I
OpenMP* reduction identifier	Local	Class I
Generic identifiers	Local or construct	Class I
Argument keywords in procedures	Local	Class III
Variables	Local or construct	Class I
Variables that are dummy arguments in statement functions	Statement	
DO variables in an implied-DO list[3] of a DATA statement, or an array constructor	Statement	
DO index variables of a FORALL statement or construct, or a DO CONCURRENT construct	Construct	
Associate names	Construct	
Intrinsic operators	Global	
Defined operators	Local or construct	
Statement labels	Local	
External I/O unit numbers	Global	
Intrinsic assignment	Global[4]	
Defined assignment	Local or construct	

[1] Names of common blocks can also be used to identify local entities.

[2] If an intrinsic procedure is not used in a scoping unit, its name can be used as a local entity within that scoping unit. For example, if intrinsic function COS is not used in a program unit, COS can be used as a local variable there.

[3] The DO variable in an implied-DO list of an I/O list has local scope.

Entity	Scope
[4] The scope of the assignment symbol (=) is global, but it can identify additional operations (see Defining Generic Assignment).	

Scoping units can contain other scoping units. For example, the following shows six scoping units:

```
MODULE MOD_1                    ! Scoping unit 1
   ...                          ! Scoping unit 1
CONTAINS                        ! Scoping unit 1
  FUNCTION FIRST                !   Scoping unit 2
    TYPE NAME                   !     Scoping unit 3
    ...                         !     Scoping unit 3
    END TYPE NAME               !     Scoping unit 3
    ...                         !   Scoping unit 2
  CONTAINS                      !   Scoping unit 2
    SUBROUTINE SUB_B            !       Scoping unit 4
      TYPE PROCESS              !         Scoping unit 5
      ...                       !         Scoping unit 5
      END TYPE PROCESS          !         Scoping unit 5
      INTERFACE                 !         Scoping unit 5
        SUBROUTINE SUB_A        !           Scoping unit 6
        ...                     !           Scoping unit 6
          BLOCK                 !             Scoping unit 7
          ...                   !             Scoping unit 7
          END BLOCK             !             Scoping unit 7
        END SUBROUTINE SUB_A    !           Scoping unit 6
      END INTERFACE             !         Scoping unit 5
    END SUBROUTINE SUB_B        !       Scoping unit 4
  END FUNCTION   FIRST          !   Scoping unit 2
END MODULE                      ! Scoping unit 1
```

See Also

Derived data types
Defining Generic Names for Procedures
Intrinsic procedures
Program Units and Procedures
Use and host association
Construct association
BLOCK construct
Defining Generic Operators
Defining Generic Assignment
PRIVATE Attributes and Statements
PUBLIC Attributes and Statements
IMPORT statement

Unambiguous Generic Procedure References

When a generic procedure reference is made, a specific procedure is invoked. If the following rules are used, the generic reference will be unambiguous:

- Two dummy arguments are said to be distinguishable if any of the following are true:
  - One is a procedure and the other is a data object
  - One has the ALLOCATABLE attribute and the other has the POINTER attribute without INTENT (IN)
  - They are both data objects or both known to be functions, and they have different type and kind parameters, or different rank

  - One is a function with nonzero rank and the other is not known to be a function
- Within a scoping unit, two procedures that have the same generic name must both be subroutines or both be functions. One of the procedures must have a nonoptional dummy argument that is one of the following:
  - Not present by position or argument keyword in the other argument list
  - Is present, but is distinguishable from the dummy argument in the other argument list
- Within a scoping unit, two procedures that have the same generic operator must both have the same number of arguments or both define assignment. One of the procedures must have a dummy argument that corresponds, by position in the argument list, to a dummy argument of the other procedure that is distinguishable from the dummy argument in the other argument list.

When an interface block extends an intrinsic procedure, operator, or assignment, the rules apply as if the intrinsic consists of a collection of specific procedures, one for each allowed set of arguments.

When a generic procedure is accessed from a module, the rules apply to all the specific versions, even if some of them are inaccessible by their specific names.

See Also

Defining Generic Names for Procedures for details on generic procedure names

Resolving Procedure References

The procedure name in a procedure reference is either established to be generic or specific, or is not established. The rules for resolving a procedure reference differ depending on whether the procedure is established and how it is established.

This section discusses the following topics:

- References to Generic Names
- References to Specific Names
- References to Nonestablished Names

References to Generic Names

Within a scoping unit, a procedure name is established to be generic if any of the following is true:

- The scoping unit contains an interface block with that procedure name.
- The procedure name matches the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is established to be generic in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be generic in a host scoping unit.

To resolve a reference to a procedure name established to be generic, the following rules are used in the order shown:

1. If an interface block with that procedure name appears in one of the following, the reference is to the specific procedure providing that interface:

   a. The scoping unit that contains the reference
   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with one of the specific interfaces of the interface block.
2. If the procedure name is specified with the INTRINSIC attribute in one of the following, the reference is to that intrinsic procedure:

   a. The same scoping unit
   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with the interface of that intrinsic procedure.
3. If the following is true, the reference is resolved by applying rules 1 and 2 to the host scoping unit:

   a. The procedure name is established to be generic in the host scoping unit
   b. There is agreement between the scoping unit and the host scoping unit as to whether the procedure is a function or subroutine name.

4. If none of the preceding rules apply, the reference must be to the generic intrinsic procedure with that name. The reference must be consistent with the interface of that intrinsic procedure.

Examples

The following example shows how a module can define three separate procedures and give them a generic name DUP through an interface block. Although the main program calls all three by the generic name, there is no ambiguity since the arguments are of different data types, and DUP is a function rather than a subroutine. The module UN_MOD must give each procedure a different name.

```
MODULE UN_MOD
!

CONTAINS
   subroutine dup1(x,y)
   real x,y
   print *, ' Real arguments', x, y
   end subroutine dup1

   subroutine dup2(m,n)
   integer m,n
   print *, ' Integer arguments', m, n
   end subroutine dup2

   character function dup3 (z)
   character(len=2) z
   dup3 = 'String argument '// z
   end function dup3

END MODULE

program unclear
!
! shows how to use generic procedure references

USE UN_MOD
INTERFACE DUP
   MODULE PROCEDURE dup1, dup2, dup3
END INTERFACE

real a,b
integer c,d
character (len=2) state

a = 1.5
b = 2.32
c = 5
d = 47
state = 'WA'

call dup(a,b)
call dup(c,d)
print *, dup(state)       !actual output is 'S' only
END
```

Note that the function DUP3 only prints one character, since module UN_MOD specifies no length parameter for the function result.

If the dummy arguments x and y for DUP were declared as integers instead of reals, then any calls to DUP would be ambiguous. If this is the case, a compile-time error results.

The subroutine definitions, DUP1, DUP2, and DUP3, must have different names. The generic name is specified in the first line of the interface block, and in the example is DUP.

References to Specific Names

In a scoping unit, a procedure name is established to be specific if it is not established to be generic and any of the following is true:

- The scoping unit contains an interface body with that procedure name.
- The scoping unit contains an internal procedure, module procedure, or statement function with that procedure name.
- The procedure name is the same as the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is specified with the EXTERNAL attribute in that scoping unit.
- The procedure name is established to be specific in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be specific in a host scoping unit.

To resolve a reference to a procedure name established to be specific, the following rules are used in the order shown:

1. If either of the following is true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:

   a. The scoping unit is a subprogram, and it contains an interface body with that procedure name.
   b. The procedure name has been declared EXTERNAL, and the procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.
2. If the scoping unit contains an interface body or the procedure name has been declared EXTERNAL, and Rule 1 does not apply, the reference is to an external procedure with that name.
3. If the scoping unit contains an internal procedure or statement function with that procedure name, the reference is to that entity.
4. If the procedure name has been declared INTRINSIC in the scoping unit, the reference is to the intrinsic procedure with that name.
5. If the scoping unit contains a USE statement that makes the name of a module procedure accessible, the reference is to that procedure. Sinec the USE statement allows renaming, the name referenced may differ from the name of the module procedure.)
6. If none of the preceding rules apply, the reference is resolved by applying these rules to the host scoping unit.

References to Nonestablished Names

In a scoping unit, a procedure name is not established if it is not determined to be generic or specific.

To resolve a reference to a procedure name that is not established, the following rules are used in the order shown:

1. If both of the following are true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:

   a. The scoping unit is a subprogram.
   b. The procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.
2. If both of the following are true, the procedure is an intrinsic procedure and the reference is to that intrinsic procedure:

   a. The procedure name matches the name of an intrinsic procedure.
   b. There is agreement between the intrinsic procedure definition and the reference of the name as a function or subroutine.
3. If neither of the preceding rules apply, the reference is to an external procedure with that name.

See Also

Function references
USE statement
CALL Statement for details on subroutine references
Defining Generic Names for Procedures for details on generic procedure names

Association

Association allows different program units to access the same value through different names. Entities are associated when each is associated with the same storage location.

There are four kinds of association:

- Name association
- Pointer association
- Storage association
- Inheritance association

The following example shows name, pointer, and storage association between an external program unit and an external procedure.

Example of Name, Pointer, and Storage Association

```
! Scoping Unit 1: An external program unit

REAL A, B(4)
REAL, POINTER :: M(:)
REAL, TARGET :: N(12)
COMMON /COM/...
EQUIVALENCE (A, B(1))        ! Storage association between A and B(1)
M => N                       ! Pointer association
CALL P (actual-arg,...)
...

! Scoping Unit 2: An external procedure
SUBROUTINE P (dummy-arg,...)  ! Name and storage association between
                              !    these arguments and the calling
                              !    routine's arguments in scoping unit 1
  COMMON /COM/...             ! Storage association with common block COM
                              !    in scoping unit 1

  REAL Y
  CALL Q (actual-arg,...)
  CONTAINS
    SUBROUTINE Q (dummy-arg,...) ! Name and storage association
between
                                 !   these arguments and the calling
                                 !   routine's arguments in host procedure
                                 !   P (subprogram Q has host association
                                 !   with procedure P)
      Y = 2.0*(Y-1.0)       ! Name association with Y in host procedure P
    ...
```

The following example shows inheritance association:

```
TYPE POINT                  ! A base type
REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT   ! An extension of TYPE(POINT)
                                      ! Components X and Y, and component name POINT,
                                      !   are inherited from the parent type POINT
INTEGER :: COLOR
END TYPE COLOR_POINT
```

Name Association

Name association allows an entity to be accessed from different scoping units by the same name or by different names. There are five types of name association: argument, use, host, linkage, and construct.

Argument Association

Arguments are the values passed to and from functions and subroutines through calling program argument lists.

Execution of a procedure reference establishes argument association between an actual argument and its corresponding dummy argument. The name of a dummy argument can be different from the name of its associated actual argument (if any).

When the procedure completes execution, the argument association is terminated. A dummy argument of that procedure can be associated with an entirely different actual argument in a subsequent invocation of the procedure.

Association Between Actual Arguments and Dummy Data Objects

A scalar dummy argument of a nonelemental procedure can be associated only with a scalar actual argument.

If the actual argument is scalar, the corresponding dummy argument must be scalar unless the actual argument is of type default character, of type character with the C character kind, or is an element or substring of an element of an array that is not an assumed-shape or pointer array. If the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments must agree.

If a scalar dummy argument is of type default character, the length of the dummy argument must be less than or equal to the length of the actual argument. The dummy argument becomes associated with the leftmost *len* characters of the actual argument. If an array dummy argument is of type default character and is not assumed shape, it becomes associated with the leftmost characters of the actual argument element sequence and it must not extend beyond the end of that sequence.

If a dummy argument is not allocatable and is not a pointer, it must be type compatible with the associated actual argument. If a dummy argument is allocatable or a pointer, the associated actual argument must be polymorphic only if the dummy argument is polymorphic, and the declared type of the actual argument must be the same as the declared type of the dummy argument.

If the dummy argument is a pointer, the actual argument must be a pointer and the nondeferred type parameters and ranks must agree. If a dummy argument is allocatable, the actual argument must be allocatable and the nondeferred type parameters and ranks must agree. The actual argument can have an allocation status of unallocated.

At the invocation of the procedure, the pointer association status of an actual argument associated with a pointer dummy argument becomes undefined if the dummy argument has INTENT(OUT).

The dynamic type of a polymorphic, allocatable, or pointer dummy argument can change as a result of execution of an allocate statement or pointer assignment in the subprogram. Because of this, the corresponding actual argument needs to be polymorphic and have a declared type that is the same as the declared type of the dummy argument, or an extension of that type.

The values of assumed type parameters of a dummy argument are assumed from the corresponding type parameters of the associated actual argument.

Except in references to intrinsic inquiry functions, if the dummy argument is not a pointer and the corresponding actual argument is a pointer, the actual argument must be associated with a target and the dummy argument becomes the argument associated with that target.

Except in references to intrinsic inquiry functions, if the dummy argument is not allocatable and the actual argument is allocatable, the actual argument must be allocated.

If the dummy argument has the VALUE attribute, it becomes associated with a definable anonymous data object whose initial value is that of the actual argument. Subsequent changes to the value or definition status of the dummy argument do not affect the actual argument.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure.

If the dummy argument has the TARGET attribute, does not have the VALUE attribute, is either a scalar or an assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, then the following is true:

- Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure.
- When execution of the procedure completes, any pointers that do not become undefined and are associated with the dummy argument remain associated with the actual argument.

Any pointers associated with the dummy argument become undefined when execution of the procedure completes in these cases:

- If the dummy argument has the TARGET attribute and the corresponding actual argument does not have the TARGET attribute, or is an array section with a vector subscript.
- If the dummy argument has the TARGET attribute and the VALUE attribute.

If a nonpointer dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument must be definable. If a dummy argument has INTENT (OUT), the corresponding actual argument becomes undefined at the time the association is established, except for components of an object of derived type for which default initialization has been specified. If the dummy argument is not polymorphic and the type of the actual argument is an extension type of the type of the dummy argument, only the part of the actual argument that is of the same type as the dummy argument becomes undefined.

Association Between Actual Arguments and Dummy Procedure Entities

If a dummy argument is a procedure pointer, the associated actual argument must be a procedure pointer, a reference to a function that returns a procedure pointer, or a reference to the NULL intrinsic function.

If a dummy argument is a dummy procedure without the POINTER attribute, the associated actual argument must be the specific name of an external, module, dummy, or intrinsic procedure, an associated procedure pointer, or a reference to a function that returns an associated procedure pointer. If the specific name is also a generic name, only the specific procedure is associated with the dummy argument.

For generic declarations, a dummy argument is type, kind, and rank compatible with another dummy argument if following is true:

- The first is type compatible with the second
- The kind type parameters of the first have the same values as the corresponding kind type parameters of the second

- Both have the same rank or either is assumed-rank

If an external procedure name or a dummy procedure name is used as an actual argument, its interface must be explicit or it must be explicitly declared to have the EXTERNAL attribute.

If the interface of the dummy argument is explicit, the procedure characteristics must be the same for the associated actual argument and the corresponding dummy argument, except that a pure actual argument can be associated with a dummy argument that is not pure, and an elemental intrinsic actual procedure can be associated with a dummy procedure that is prohibited from being elemental.

If the interface of the dummy argument is implicit and either the name of the dummy argument is explicitly typed or it is referenced as a function, the dummy argument must not be referenced as a subroutine and the actual argument must be a function, function procedure pointer, or dummy procedure.

If the interface of the dummy argument is implicit and a reference to it appears as a subroutine reference, the actual argument must be a subroutine, subroutine procedure pointer, or dummy procedure.

An assumed-rank dummy argument can correspond to an actual argument of any rank. If the actual argument has rank zero (is a scalar), the dummy argument has rank zero; the shape is a zero-sized array and the LBOUND and UBOUND intrinsic functions, with no DIM argument, return zero-sized arrays. If the actual argument has rank greater than zero, the rank and extents of the dummy argument are assumed from the actual argument, including the lack of a final extent in the case of an assumed-size array. If the actual argument is an array and the dummy argument is allocatable or a pointer, the bounds of the dummy argument are assumed from the actual argument.

An assumed-type dummy argument must not correspond to an actual argument that is of a derived type that has type parameters, type-bound procedures, or final subroutines.

Association For C Interoperability

When a Fortran procedure that has an INTENT(OUT) allocatable dummy argument is invoked by a C function, and the actual argument in the C function is the address of a C descriptor that describes an allocated allocatable variable, the variable is deallocated upon entry to the Fortran procedure.

When a C function is invoked from a Fortran procedure by means of an interface with an INTENT(OUT) allocatable dummy argument, and the actual argument in the reference to the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

See Also

Argument Association in Procedures
Procedure Characteristics

Use and Host Association Overview

Use association allows the entities in a module to be accessible to other scoping units. *Host association* allows the entities in a host scoping unit to be accessible to an internal subprogram, a module subprogram, or submodule program.

Use association and host association remain in effect throughout the execution of the executable program.

An interface body does not access named entities by host association, but it can access entities by use association.

Examples

The following example shows host and use association:

```
MODULE SHARE_DATA
  REAL Y, Z
END MODULE

PROGRAM DEMO
  USE SHARE_DATA         ! All entities in SHARE_DATA are available
```

```
  REAL B, Q              !   through use association.
  ...
  CALL CONS (Y)
CONTAINS
  SUBROUTINE CONS (Y)    ! Y is a local entity (dummy argument).
    REAL C, Y
    ...
    Y = B + C + Q + Z    ! B and Q are available through host association.
    ...                  !   C is a local entity, explicitly declared.
  END SUBROUTINE CONS    !   is available through use association.
END PROGRAM DEMO
```

See Also

Use Association
Host Association
Scope for details on entities with local scope

Use Association

Use association allows the entities in a module to be accessible to other scoping units. This association remains in effect throughout the execution of the program. The mechanism for use association is the USE statement. The USE statement provides access to all public entities in the module, unless ONLY is specified. In this case, only the entities named in the ONLY list can be accessed.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible. A name that appears in the scoping unit as an external name in an EXTERNAL statement is a global name, and any entity of the host that has this as its nongeneric name is inaccessible.

For more information on rules that apply to use association, see the USE statement.

See Also

USE statement
Host Association
Scope for details on entities with local scope

Host Association

Host association allows the entities in a host scoping unit to be accessible to an internal subprogram, a module subprogram, or submodule program. This association remains in effect throughout the execution of the program.

The following also have access to entities from its host:

- A module procedure interface body
- A derived-type definition
- An interface body that is not a separate interface body has access to the named entities from its host that are made accessible by IMPORT statements in the interface body.

The accessed entities are known by the same name and have the same attributes as in the host, except that a local entity can have the ASYNCHRONOUS attribute even if the host entity does not, and a noncoarray local entity can have the VOLATILE attribute even if the host entity does not. The accessed entities can be named data objects, derived types, abstract interfaces, procedures, generic identifiers, and namelist groups.

Entities that are local to a procedure are not accessible to their hosts.

The following are considered local identifiers within a scoping unit so they are not accessible to their hosts:

- A function name in a statement function, an entity declaration in a type declaration statement, or the name of an entity declared by an interface body, unless any of these are global identifiers.
- An object name in an entity declaration in a type declaration statement or a POINTER, SAVE, ALLOCATABLE, or TARGET statement.
- A type name in a derived type statement.

- A procedure pointer that has the EXTERNAL attribute.
- The name of a variable that is wholly or partially initialized in a DATA statement.
- The name of an object that is wholly or partially initialized in an EQUIVALENCE statement.
- A namelist group name in a NAMELIST statement.
- A generic name in a generic specification in an INTERFACE statement.
- The name of a named construct.
- The name of a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement, or a statement function.
- A named constant defined in a PARAMETER statement.
- An array name in a DIMENSION statement.
- A variable name in a common block object in a COMMON statement.
- A result name in a FUNCTION or ENTRY statement.
- An intrinsic procedure name in an INTRINSIC statement.

A name that appears in an ASYNCHRONOUS or VOLATILE statement is not necessarily the name of a local variable. In an internal or module procedure, if a variable that is accessible via host association is specified in an ASYNCHRONOUS or VOLATILE statement, that host variable is given the ASYNCHRONOUS or VOLATILE attribute in the local scope.

If an intrinsic procedure is accessed by means of host association, it must be established to be intrinsic in the host scoping unit by one of the following methods:

- It must be explicitly given the INTRINSIC attribute.
- It must be invoked as an intrinsic procedure.
- It must be accessed from a module or from its host where it is established to be intrinsic.

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association can be changed within the procedure. After execution of the procedure, the pointer association remains current, unless the execution caused the target to become undefined. If this occurs, the host associated pointer becomes undefined.

NOTE
Implicit declarations can cause problems for host association. It is recommended that you use IMPLICIT NONE in both the host and the contained procedure, and that you explicitly declare all entities.

When all entities are explicitly declared, local declarations override host declarations, and host declarations that are not overridden are available in the contained procedure.

Examples

The following example shows how a host and an internal procedure can use host-associated entities:

```
program INTERNAL
! shows use of internal subroutine and CONTAINS statement
   real a,b,c
   call find
   print *, c
contains
   subroutine find
     read *, a,b
     c = sqrt(a**2 + b**2)
   end subroutine find
end
```

In this case, the variables `a`, `b`, and `c` are available to the internal subroutine `find` through host association. They do not have to be passed as arguments to the internal procedure. In fact, if they are, they become local variables to the subroutine and hide the variables declared in the host program.

Conversely, the host program knows the value of `C`, when it returns from the internal subroutine that has defined `C`.

See Also

Use Association
Scope for details on entities with local scope

Linkage Association

Linkage association occurs between the following:

- A module variable that has the BIND attribute and the C variable with which it interoperates
- A Fortran common block and the C variable with which it interoperates

This association remains in effect throughout the execution of the program.

See Also

USE statement
Host Association
Scope for details on entities with local scope

Construct Association

Construct association establishes an association between each selector and the corresponding associate name of an ASSOCIATE, SELECT RANK, or SELECT TYPE construct.

If the selector is allocatable, it must be allocated. The associate name is associated with the data object and does not have the ALLOCATABLE attribute.

If the selector has the POINTER attribute, it must be associated. The associate name is associated with the target of the pointer and does not have the POINTER attribute; it has the TARGET attribute if the selector is a variable that has the POINTER or TARGET attribute.

If the selector is a variable other than an array section having a vector subscript, the association is with the data object specified by the selector; otherwise, the association is with the value of the selector expression, which is evaluated before the execution of the block.

Each associate name remains associated with the corresponding selector throughout the execution of the executed block. Within the block, each selector is known by, and can be accessed by, the corresponding associate name. When the construct is terminated, the association is terminated.

See Also

USE statement
Host Association
Scope for details on entities with local scope

Additional Attributes Of Associate Names

In an ASSOCIATE or SELECT TYPE construct, the rank of each entity identified by an associate name has the same rank as its corresponding selector. The lower bound of each dimension is the result of the intrinsic function LBOUND applied to the corresponding dimension of selector. The upper bound of each dimension is one less than the sum of the lower bound and the extent. The entity identified by the associate name does not have the ALLOCATABLE or POINTER attributes. If the corresponding selector has the POINTER or TARGET attribute, the entity identified by the associate name has the TARGET attribute.

Each entity identified by an associate name in an ASSOCIATE, SELECT RANK, or SELECT TYPE construct has the same corank as the corresponding selector. The cobounds of each codimension of the entity are the same as those of the selector.

In an ASSOCIATE, SELECT RANK, or SELECT TYPE construct, if a selector has the ASYNCHRONOUS or VOLATILE attribute, the entity associated with the corresponding associate name also has that attribute. If the selector is polymorphic, the associated entity has the same dynamic type and type parameters as the selector. If the selector has the OPTIONAL attribute, it must be present. If the selector is contiguous, the associated entity is also contiguous.

The associated entity is a variable. In an ASSOCIATE or SELECT TYPE construct, if the selector is not definable, the associated entity is not definable and cannot be defined or undefined. An associate name or a subobject of the associate name is not allowed in a variable definition context or a pointer association context if the selector is not allowed in a variable definition context.

In a SELECT RANK construct, the selector has assumed rank, and assumed rank entities are not otherwise definable. However, in the block following a RANK(*) statement the associate name is the name of a one dimensional assumed-size array, and in the block following a RANK (*scalar-int-const-expr*), the variable has the specified rank. In these cases, if the selector is otherwise definable ignoring that it is assumed rank, the associated entity may be defined or undefined.

See Also

ASSOCIATE
SELECT RANK
SELECT TYPE

Pointer Association

A pointer can be associated with a target. At different times during the execution of a program, a pointer can be undefined, associated with different targets, or be disassociated. The initial association status of a pointer is undefined. A pointer can become associated by the following:

- Pointer assignment (pointer => target)

  The target must be associated, or specified with the TARGET attribute. If the target is allocatable, it must be currently allocated.
- Allocation (successful execution of an ALLOCATE statement)

  The ALLOCATE statement must reference the pointer.

A pointer becomes disassociated if any of the following occur:

- The pointer is nullified by a NULLIFY statement.
- The pointer is deallocated by a DEALLOCATE statement.
- The pointer is assigned a disassociated pointer or the NULL intrinsic function.
- The pointer is an ultimate component of an object of a type for which default initialization is specified for the component and one of the following is true:
  - A procedure is invoked with this object as an actual argument corresponding to a nonpointer, nonallocatable, dummy argument with INTENT (OUT).
  - A procedure is invoked with this object as an unsaved, nonpointer, nonallocatable local object that is not accessed by use or host association.
  - This object is allocated.

When a pointer is associated with a target, the definition status of the pointer is defined or undefined, depending on the definition status of the target. A target is undefined in the following cases:

- If it was never allocated
- If it is not deallocated through the pointer
- If a RETURN or END statement causes it to become undefined

If a pointer is associated with a definable target, the definition status of the pointer can be defined or undefined, according to the rules for a variable.

The association status of a pointer becomes undefined when a DO CONCURRENT construct is terminated and the pointer's association status was changed in more than one iteration of the construct.

If the association status of a pointer is disassociated or undefined, the pointer must not be referenced or deallocated.

Whatever its association status, a pointer can always be nullified, allocated, or associated with a target. When a pointer is nullified, it is disassociated. When a pointer is allocated, it becomes associated, but is undefined. When a pointer is associated with a target, its association and definition status are determined by its target.

See Also

Pointer assignments
NULL intrinsic function
Dynamic Allocation for details on the ALLOCATE, DEALLOCATE, and NULLIFY statements

Storage Association

Storage association describes the relationships that exist among data objects. It is the association of two or more data objects that occurs when two or more storage sequences share, or are aligned with, one or more *storage units*. Storage sequences are used to describe relationships among variables, common blocks, and result variables.

See Also

Storage Units and Storage Sequence
Array Association

Storage Units and Storage Sequence

A *storage unit* is a fixed unit of physical memory allocated to certain data. A *storage sequence* is a sequence of storage units. The size of a storage sequence is the number of storage units in the storage sequence. A storage unit can be numeric, character, or unspecified.

A nonpointer scalar of type default real, integer, or logical occupies one numeric storage unit. A nonpointer scalar of type double precision real or default complex occupies two contiguous numeric storage units. In Intel® Fortran, one numeric storage unit corresponds to 4 bytes of memory.

A nonpointer scalar of type default character with character length 1 occupies one character storage unit. A nonpointer scalar of type default character with character length *len* occupies *len* contiguous character storage units. In Intel® Fortran, one character storage unit corresponds to 1 byte of memory.

A nonpointer scalar of nondefault data type occupies a single unspecified storage unit. The number of bytes corresponding to the unspecified storage unit differs depending on the data type.

The following table lists the storage requirements (in bytes) for the intrinsic data types:

Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
BYTE	1
LOGICAL	2, 4, or 8 [1]
LOGICAL(1)	1
LOGICAL(2)	2
LOGICAL(4)	4
LOGICAL(8)	8
INTEGER	2, 4, or 8 [1]
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4

Data Type	Storage Requirements (in bytes)
INTEGER(8)	8
REAL	4, 8, or 16[2]
REAL(4)	4
DOUBLE PRECISION	8
REAL(8)	8
REAL(16)	16
COMPLEX	8, 16, or 32[2]
COMPLEX(4)	8
DOUBLE COMPLEX	16
COMPLEX(8)	16
COMPLEX(16)	32
CHARACTER	1
CHARACTER*len	len [3]
CHARACTER*(*)	assumed-length [4]

[1] Depending on default integer, LOGICAL and INTEGER can have 2, 4, or 8 bytes. The default allocation is four bytes.

[2] Depending on default real, REAL can have 4, 8, or 16 bytes and COMPLEX can have 8, 16, or 32 bytes. The default allocations are four bytes for REAL and eight bytes for COMPLEX.

[3] The value of len is the number of characters specified. The largest valid value is 2**31-1 on IA-32 architecture; 2**63-1 on Intel® 64 architecture. Negative values are treated as zero.

[4] The assumed-length format *(*) applies to dummy arguments, PARAMETER statements, or character functions, and indicates that the length of the actual argument or function is used. (See Assumed-Length Character Arguments.)

A nonpointer scalar of sequence derived type occupies a sequence of storage sequences corresponding to the components of the structure, in the order they occur in the derived-type definition. Note that a sequence derived type has a SEQUENCE statement.

A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and is different for each combination of type, type parameters, and rank.

The definition status and value of a data object affects the definition status and value of any storage-associated entity.

When two objects occupy the same storage sequence, they are totally storage-associated. When two objects occupy parts of the same storage sequence, they are partially associated. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement can cause total or partial storage association of storage sequences.

See Also

Assumed-Length Character Arguments
COMMON
ENTRY
EQUIVALENCE

Array Association

A nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order.

Two or more arrays are associated when each one is associated with the same storage location. They are partially associated when part of the storage associated with one array is the same as part or all of the storage associated with another array.

If arrays with different data types are associated, or partially associated, with the same storage location, and the value of one array is defined (for example, by assignment), the value of the other array becomes undefined. This happens because an element of an array is considered defined only if the storage associated with it contains data of the same type as the array name.

An array element, array section, or whole array is defined by a DATA statement before program execution. Note that the array properties must be declared in a previous specification statement. During program execution, array elements and sections are defined by an assignment or input statement, and entire arrays are defined by input statements.

See Also

Arrays
DATA statement
Array Elements for details on array element order

Inheritance Association

Inheritance association occurs between components of a parent component and components inherited by type extension into an extended type.

This association is not affected by the accessibility of the inherited components.

See Also

Storage Association
Pointer Association

Deleted and Obsolescent Language Features

Fortran 90 identified some FORTRAN 77 features to be obsolescent. Fortran 95 deleted some Fortran 90 features, and identified some Fortran 90 language features to be obsolescent.

Fortran 90 identified some FORTRAN 77 features to be obsolescent. Fortran 95 deleted some Fortran 90 obsolescent features, and identified some Fortran 90 language features to become obsolescent. Subsequent standard revisions have continued to identify some features of previous standards as obsolescent and to delete selected features that were previously deemed obsolescent.

Fortran 2008 has also identified some earlier standard language features to be obsolescent. Features considered obsolescent may be removed from future revisions of the Fortran Standard.

Fortran 2018 identified the following language features as obsolescent: the COMMON and EQUIVALENCE statement, the FORALL statement and construct, BLOCK DATA subprograms, labeled DO loops, and specific names for intrinsic procedures. The arithmetic IF statement and the non-block form of DO constructs were deleted in Fortran 2018.

To have these features flagged, you can specify compiler option `stand`.

NOTE
Intel® Fortran fully supports features identified as deleted from or obsolescent in the Fortran Standard.

See Also
stand compiler option

Deleted Language Features in the Fortran Standard

Some language features, considered redundant in older versions of the Fortran Standard, are not included in the current Fortran Standard. However, they are still fully supported by Intel® Fortran.

In the examples below, both forms are supported by Intel® Fortran, but the Fortran 2008 Standard only supports the second form:

- ASSIGN and assigned GO TO statements

  The ASSIGN statement, when assigning a label for use with the assigned GO TO statement, can be replaced by assigning the integer value of the label to an integer variable; the assigned GO TO statement can then be replaced by an IF statement that tests the integer variable for various values and then goes to the label that represents that value. For example, replace:

```
ASSIGN 10 TO J
...
ASSIGN 20 TO J
...
GO TO J
```

  with:

```
J = 10
...
J = 20
...
IF (J .EQ. 10) THEN
    GO TO 10
ELSE IF (J .EQ. 20) THEN
    GO TO 20
END IF
```

- Arithmetic IF statement

  The arithmetic IF statement can be replace by an IF-THEN-ELSE construct or a CASE SELECT construct.

  For example, replace:

```
IF (expr) 10, 20, 30
```

  with:

```
IF (expr .LT. 0) THEN
  GO TO 10
ELSE IF (expr .EQ. 0) THEN
  GO TO 20
ELSE
  GO TO 30
ENDIF
```

- Assigned FORMAT specifier

The assigned FORMAT specifier sets an integer variable to the label of a FORMAT statement and then the integer variable is used in an I/O statement instead of the FORMAT label. You can replace the integer variable with a character variable whose value is the contents of the FORMAT statement, and then use the character variable in the I/O statement as the format. For example, replace:

```
ASSIGN 1000 TO IFMT
...
WRITE (6, IFMT) X, Y
...
1000     FORMAT (2E10.2)
```

with:

```
CHARACTER(20) :: IFMT = "(2E10.2)"
...
WRITE (6, IFMT) X, Y
```

- The non-block form of a DO statement

  The non-block form of a DO loop contains a statement label in the DO statement identifying the terminal statement of the DO. The terminal statement may be an executable statement and may be shared with another non-block DO statement. Use an END DO statement as the terminal statement for each DO loop.

  You should not use statement labels on the terminating statement and in the DO statement, because labeled DO loops are now obsolescent.

  For example, replace the following:

```
   DO 10 I = 1, N
     DO 10 J = 1, M
10 A(I, J) = F_OF (I, J)
```

  with this:

```
DO I = 1, N
  DO J = 1, M
    A(I, J) = F_OF (I, J)
  END DO
END DO
```

- Branching to an END IF statement from outside its IF block

  The END IF statement can no longer be a target of a GO TO statement that is outside the IF block that ends with that END IF statement. Use a CONTINUE statement after the END IF as the target of the GO TO statement. For example, replace:

```
IF ...
    GO TO 100
ELSE IF ...
100  END IF
```

  with:

```
IF ...
     GO TO 100
ELSE IF ...
END IF
100  CONTINUE
```

- H edit descriptor

  Replace the H edit descriptor of the form *n*H*characters* with "*characters*". Remember to double any quotes or apostrophes in the string *characters*.
- PAUSE statement

The PAUSE statement displays a character string on the standard output device and then suspends program execution until any character is typed on the standard input device. You can replace the PAUSE statement with a WRITE statement followed by a READ statement. For example, replace:

```
PAUSE " don't forget to buy milk"
```

with:

```
WRITE (6, *) " don't forget to buy milk"
READ (5, *)    ! no io-list is necessary, the input is ignored
```

- Real and double precision DO control variables and DO loop control expressions

  REAL variables of any KIND can no longer be used as DO loop control variables and expressions. You can replace such DO loops with the appropriate DO WHILE loop that explicitly initializes, increments, and tests the REAL variable. For example, replace:

```
DO X = 0.1, 0.5, 0.01
   ...
END DO
```

  with:

```
X = 0.1
DO WHILE (X .LE. 0.5)
   ...
   X = X + 0.01
END DO
```

- Vertical format control

  Formatted output to certain printing output units used to result in the first character of each record being interpreted as controlling vertical spacing on the unit. There is no standard way to detect whether output to such a unit should result in such vertical format control and no way to specify that it should be applied. The effect can be achieved by post-processing a formatted file after it is created to interpret the first character as some form of control character. This is left to the user.

Intel Fortran flags these features if you specify compiler option stand.

See Also

Obsolescent Language Features in the Fortran Standard

`stand` compiler option

Obsolescent Language Features in the Fortran Standard

Some language features, considered redundant in older versions of the Fortran Standard, are identified as obsolescent in the current Fortran Standard.

Intel® Fortran flags these features if you specify compiler option stand.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate returns (labels in an argument list)

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program use a CASE construct to test the value and perform operations.
- Assumed-length character functions

  To replace this functionality, it is recommended that you use one of the following:

  - An automatic character-length function, where the length of the function result is declared in a specification expression
  - A subroutine whose arguments correspond to the function result and the function arguments

  Dummy arguments of a function can still have assumed character length; this feature is not obsolescent.
- BLOCK DATA subprograms

BLOCK DATA was used to initialize COMMON block variables. This can be achieved with initialization of MODULE data when it is declared. MODULE data is the preferred method of sharing data between compilation units, and for replacing COMMON blocks.

- CHARACTER*(*) form of CHARACTER declaration

  To replace this functionality, it is recommended that you use the Fortran 90 forms of specifying a length selector in CHARACTER declarations (see Declaration Statements for Character Types).
- COMMON blocks

  To replace this functionality, it is recommended data specified in COMMON be declared in a MODULE which can be made available through USE association where the data is needed.
- Computed GO TO statement

  To replace this functionality, it is recommended that you use a CASE construct.
- DATA statements among executable statements

  This functionality has been included since FORTRAN 66, but is considered to be a potential source of errors.
- ENTRY statement

  To replace this functionality, it is recommended that you use multiple module procedures that can access shared data in the module.
- EQUIVALENCE

  The use of storage association thru EQUIVALENCE statements is not recommended.
- Fixed source form

  Newer methods of entering data have made this source form obsolescent and error-prone.

  The recommended method for coding is to use free source form.
- FORALL statement and construct

  These were added to the language with the expectation they would result in very efficient and possibly parallel code. The complexity and many restrictions prevented compilers from taking advantage of them.

  The DO CONCURRENT construct makes FORALL redundant. Data manipulations which can be done with FORALL can be done more effectively with pointers using rank remapping. Both the FORALL statement form and the construct became obsolescent in Fortran 2018.
- Labeled DO loops

  To replace this functionality, it is recommended that you use an END DO statements to terminate loops, and CYCLE statements to branch to the end of the loop to start the next iteration of the loop (or exit the loop, if the last iteration of the loop takes the branch).
- Statement functions

  To replace this functionality, it is recommended that you use an internal function.
- Specific names of intrinsic functions that also have generic names

  To replace this functionality, it is recommended that you use the generic names.

Additional Language Features

To facilitate compatibility with older versions of Fortran, Intel® Fortran provides the following additional language features:

- The DEFINE FILE statement
- The ENCODE and DECODE statements
- The FIND statement
- The INTERFACE TO statement
- FORTRAN 66 Interpretation of the EXTERNAL Statement
- An alternative syntax for the PARAMETER statement
- The VIRTUAL statement
- An alternative syntax for binary, octal, and hexadecimal constants
- An alternative syntax for a record specifier

- An alternate syntax for the DELETE statement
- An alternative form for namelist external records
- The integer POINTER statement
- Record structures

These language features are particularly useful in porting older Fortran programs to Standard Fortran. However, you should avoid using them in new programs on these systems, and in new programs for which portability to other Standard Fortran implementations is important.

FORTRAN 66 Interpretation of the EXTERNAL Statement

If you specify compiler option f66, the EXTERNAL statement is interpreted in the way that was specified by the FORTRAN 66 (FORTRAN IV) standard. This interpretation became incompatible with FORTRAN 77 and later revisions of the Fortran standard.

The FORTRAN 66 interpretation of the EXTERNAL statement combines the functionality of the INTRINSIC statement with that of the EXTERNAL statement.

This lets you use subprograms as arguments to other subprograms. The subprograms to be used as arguments can be either user-supplied procedures or Fortran intrinsic procedures.

The FORTRAN 66 EXTERNAL statement takes the following form:

EXTERNAL [*]*v* [, [*]*v*] ...

*	Specifies that a user-supplied external procedure is to be used instead of a Fortran intrinsic procedure having the same name. This modifier is not standard FORTRAN 66, but was an extension in some FORTRAN 66 compilers, and provides the FORTRAN 77 meaning of EXTERNAL where required.
v	Is the name of a subprogram or the name of a dummy argument associated with the name of a subprogram.

Description

The FORTRAN 66 EXTERNAL statement declares that each name in its list is an external procedure name. Such a name can then be used as an actual argument to a subprogram, which then can use the corresponding dummy argument in a function reference or CALL statement.

However, when used as an argument, a complete function reference represents a value, not a subprogram name; for example, SQRT(B) in CALL SUBR(A, SQRT(B), C). Therefore, it does not need to be defined in an EXTERNAL statement. Note that the incomplete reference SQRT *would* need to be defined in an EXTERNAL statement.

Examples

The following example, when compiled with compiler option f66, shows the FORTRAN 66 EXTERNAL statement:

```
Main Program                          Subprograms
EXTERNAL SIN, COS, *TAN, SINDEG       SUBROUTINE TRIG(X,F,Y)
   .                                  Y = F(X)
   .                                  RETURN
   .                                  END
CALL TRIG(ANGLE, SIN, SINE)
   .
   .                                  FUNCTION TAN(X)
   .                                  TAN = SIN(X)/COS(X)
CALL TRIG(ANGLE, COS, COSINE)         RETURN
   .                                  END
   .
   .
```

```
CALL TRIG(ANGLE, TAN, TANGNT)        FUNCTION SINDEG(X)/
   .                                 SINDEG = SIN(X*3.1459/180)
   .                                 RETURN
   .                                 END
CALL TRIG(ANGLED, SINDEG, SINE)
```

The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
Y = SINDEG(X)
```

The functions SIN and COS are examples of trigonometric functions supplied in the Fortran intrinsic procedure library. The function TAN is also supplied in the library, but the asterisk (*) in the EXTERNAL statement specifies that the user-supplied function be used, instead of the intrinsic function. The function SINDEG is also a user-supplied function. Because no library function has the same name, no asterisk is required.

See Also

f66 compiler option

Alternative Syntax for the PARAMETER Statement

The PARAMETER statement discussed here is similar to the one discussed in PARAMETER; they both assign a name to a constant. However, this PARAMETER statement differs from the other one in the following ways:

- Its list is not bounded with parentheses.
- The form of the constant, rather than implicit or explicit typing of the name, determines the data type of the variable.

This PARAMETER statement takes the following form:

PARAMETER *c* = *expr* [, *c* = *expr*] ...

c	Is the name of the constant.
expr	Is a constant expression. It can be of any data type.

Description

Each name *c* becomes a constant and is defined as the value of expression *expr*. Once a name is defined as a constant, it can appear in any position in which a constant is allowed. The effect is the same as if the constant were written there instead of the name.

The name of a constant cannot appear as part of another constant, except as the real or imaginary part of a complex constant. For example:

```
  PARAMETER I=3
  PARAMETER M=I.25            ! Not allowed
  PARAMETER N=(1.703, I)      ! Allowed
```

The name used in the PARAMETER statement identifies only the name's corresponding constant in that program unit. Such a name can be defined only once in PARAMETER statements within the same program unit.

The name of a constant assumes the data type of its corresponding constant expression. The data type of a parameter constant cannot be specified in a type declaration statement. Nor does the initial letter of the constant's name implicitly affect its data type.

Examples

The following are valid examples of this form of the PARAMETER statement:

```
PARAMETER PI=3.1415927, DPI=3.141592653589793238D0
PARAMETER PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

See Also

PARAMETER for details on compile-time constant expressions

Alternative Syntax for Binary, Octal, and Hexadecimal Constants

In Intel® Fortran, you can use an alternative syntax for binary, octal, and hexadecimal constants. The following table shows the alternative syntax and equivalents:

Constant	Alternative Syntax	Equivalent
Binary	'0..1'B	B'0..1'
Octal	'0..7'O	O'0..7'
Hexadecimal	'0..F'X	Z'0..F'
	X'0..F'	

You can use a quotation mark (") in place of an apostrophe in all the above syntax forms.

For information on the # syntax for integers not in base 10, see Integer Constants.

See Also

Binary constants
Octal constants
Hexadecimal constants

Alternative Syntax for a Record Specifier

In Intel® Fortran, you can specify the following form for a record specifier in an I/O control list:

'*r*

r	Is a numeric expression with a value that represents the position of the record to be accessed using direct access I/O.

The value must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file. If necessary, a record number is converted to integer data type before being used.

If this nonkeyword form is used in an I/O control list, it must immediately follow the nonkeyword form of the io-unit specifier.

Alternative Syntax for the DELETE Statement

In Intel® Fortran, you can specify the following form of the DELETE statement when deleting records from a relative file:

DELETE (*io-unit* '*r* [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit	Is the number of the logical unit containing the record to be deleted.
r	Is the positional number of the record to be deleted.
label	Is the label of an executable statement that receives control if an error condition occurs.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

This form deletes the direct access record specified by *r*.

See Also
DELETE statement

Alternative Form for Namelist External Records

In Intel® Fortran, you can use the following form for an external record:

$*group-name*[*object* = *value* [*object* = *value*] ...$[END]

group-name	Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit.
object	Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks, but it can be preceded or followed by blanks.
value	Is a null value, a constant (or list of constants), a repetition of constants in the form r*c, or a repetition of null values in the form r*.

If more than one *object*=*value* or more than one value is specified, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks.

See Also
NAMELIST statement
Rules for Namelist Sequential READ Statements for details on namelist input
Rules for Namelist Sequential WRITE Statements for details on namelist output

Record Structures

The record structure was defined in earlier versions of Intel® Fortran as a language extension. It is still supported, although its functionality has been replaced by Standard Fortran derived data types. Record structures in existing code can be easily converted to Standard Fortran derived type structures for portability, but can also be left in their old form. In most cases, an Intel Fortran record and a Standard Fortran derived type can be used interchangeably.

Intel Fortran record structures are similar to Standard Fortran derived types.

A *record structure* is an aggregate entity containing one or more elements. (Record elements are also called fields or components.) You can use records when you need to declare and operate on multi-field data structures in your programs.

Creating a record is a two-step process:

1. You must define the form of the record with a multistatement *structure declaration*.

2. You must use a RECORD statement to declare the record as an entity with a name. (More than one RECORD statement can refer to a given structure.)

Examples

Intel Fortran record structures, using only intrinsic types, easily convert to Standard Fortran derived types. The conversion can be as simple as replacing the keyword STRUCTURE with TYPE and removing slash (/) marks. The following shows an example conversion:

Record Structure	Standard Fortran Derived-Type

`STRUCTURE /employee_name/`
`  CHARACTER*25  last_name`
`  CHARACTER*15  first_name`
`END STRUCTURE`
`STRUCTURE /employee_addr/`
`  CHARACTER*20  street_name`
`  INTEGER(2)  street_number`
`  INTEGER(2)  apt_number`
`  CHARACTER*20  city`
`  CHARACTER*2  state`
`  INTEGER(4)  zip`
`END STRUCTURE`
	`TYPE employee_name`
	`  CHARACTER*25  last_name`
	`  CHARACTER*15  first_name`
	`END TYPE`
	`TYPE employee_addr`
	`  CHARACTER*20  street_name`
	`  INTEGER(2)  street_number`
	`  INTEGER(2)  apt_number`
	`  CHARACTER*20  city`
	`  CHARACTER*2  state`
	`  INTEGER(4)  zip`
	`END TYPE`

The record structures can be used as subordinate record variables within another record, such as the `employee_data` record. The equivalent Standard Fortran derived type would use the derived-type objects as components in a similar manner, as shown below:

Record Structure	Standard Fortran Derived-Type

`STRUCTURE /employee_data/`
`  RECORD  /employee_name/ name`
`  RECORD  /employee_addr/ addr`
`  INTEGER(4) telephone`
`  INTEGER(2) date_of_birth`
`  INTEGER(2) date_of_hire`
`  INTEGER(2) social_security(3)`
`  LOGICAL(2) married`
`  INTEGER(2) dependents`
`END STRUCTURE`
	`TYPE employee_data`
	`  TYPE (employee_name) name`
	`  TYPE (employee_addr) addr`
	`  INTEGER(4) telephone`
	`  INTEGER(2) date_of_birth`
	`  INTEGER(2) date_of_hire`
	`  INTEGER(2) social_security(3)`
	`  LOGICAL(2) married`
	`  INTEGER(2) dependents`
	`END TYPE`

See Also

RECORD Statement

Structure Declarations

A *structure declaration* defines the field names, types of data within fields, and order and alignment of fields within a record. Fields and structures can be initialized, but records cannot be initialized. For more information, see STRUCTURE.

See Also

Type Declarations
Substructure Declarations
Union Declarations

Type Declarations within Record Structures

The syntax of a type declaration within a record structure is identical to that of a normal Fortran type statement.

The following rules and behavior apply to type declarations in record structures:

- %FILL can be specified in place of a field name to leave space in a record for purposes such as alignment. This creates an unnamed field.

  %FILL can have an array specification; for example:

  ```
  INTEGER %FILL (2,2)
  ```

  Unnamed fields cannot be initialized. For example, the following statement is invalid and generates an error message:

  ```
  INTEGER %FILL /1980/
  ```
- Initial values can be supplied in field declaration statements. Unnamed fields cannot be initialized; they are always undefined.
- Field names must always be given explicit data types. The IMPLICIT statement does not affect field declarations.
- Any required array dimensions must be specified in the field declaration statements. DIMENSION statements cannot be used to define field names.
- Adjustable or assumed sized arrays and assumed-length CHARACTER declarations are not allowed in field declarations.

Substructure Declarations

A field within a structure can itself be a structured item composed of other fields, other structures, or both. You can declare a substructure in two ways:

- By nesting structure declarations within other structure or union declarations (with the limitation that you cannot refer to a structure inside itself at any level of nesting).

  One or more field names must be defined in the STRUCTURE statement for the substructure, because all fields in a structure must be named. In this case, the substructure is being used as a field within a structure or union.

  Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict.
- By using a RECORD statement that specifies another previously defined record structure, thereby including it in the structure being declared.

See the example in STRUCTURE for a sample structure declaration containing both a nested structure declaration (TIME) and an included structure (DATE).

References to Record Fields

References to record fields must correspond to the kind of field being referenced. Aggregate field references refer to composite structures (and substructures). Scalar field references refer to singular data items, such as variables.

An operation on a record can involve one or more fields.

Record field references take one of the following forms:

Aggregate Field Reference:

record-name [.*aggregate-field-name*] ...

Scalar Field Reference:

record-name [.*aggregate-field-name*] ... .*scalar-field-name*

record-name	Is the name used in a RECORD statement to identify a record.
aggregate-field-name	Is the name of a field that is a substructure (a record or a nested structure declaration) within the record structure identified by the record name.
scalar-field-name	Is the name of a data item (having a data type) defined within a structure declaration.

Description

Records and record fields cannot be used in DATA statements, but individual fields can be initialized in the STRUCTURE definition.

An automatic array cannot be a record field.

A scalar field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names followed by the name of a scalar field. A scalar field reference refers to a single data item (having a data type) and can be treated like a normal reference to a Fortran variable or array element.

You can use scalar field references in statement functions and in executable statements. However, they cannot be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements, or as the control variable in an indexed DO-loop.

Type conversion rules for scalar field references are the same as those for variables and array elements.

An aggregate field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names.

You can only assign an aggregate field to another aggregate field (record = record) if the records have the same structure. Intel® Fortran supports no other operations (such as arithmetic or comparison) on aggregate fields.

Intel Fortran requires qualification on all levels. While some languages allow omission of aggregate field names when there is no ambiguity as to which field is intended, Intel Fortran requires all aggregate field names to be included in references.

You can use aggregate field references in unformatted I/O statements; one I/O record is written no matter how many aggregate and array name references appear in the I/O list. You cannot use aggregate field references in formatted, namelist, and list-directed I/O statements.

You can use aggregate field references as actual arguments and record dummy arguments. The declaration of the dummy record in the subprogram must match the form of the aggregate field reference passed by the calling program unit; each structure must have the same number and types of fields in the same order. The order of map fields within a union declaration is irrelevant.

Records are passed by reference. Aggregate field references are treated like normal variables. You can use adjustable arrays in RECORD statements that are used as dummy arguments.

Examples

The following examples show record and field references. Consider the following structure declarations:

Structure DATE:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
STRUCTURE
```

Structure APPOINTMENT:

```
STRUCTURE /APPOINTMENT/
  RECORD /DATE/      APP_DATE
  STRUCTURE /TIME/   APP_TIME(2)
    INTEGER*1        HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20       APP_MEMO(4)
  LOGICAL*1          APP_FLAG
END STRUCTURE
```

The following RECORD statement creates a variable named NEXT_APP and a 10-element array named APP_LIST. Both the variable and each element of the array take the form of the structure APPOINTMENT.

```
RECORD /APPOINTMENT/    NEXT_APP,APP_LIST(10)
```

Each of the following examples of record and field references are derived from the previous structure declarations and RECORD statement:

Aggregate Field References

- The record NEXT_APP:

```
NEXT_APP
```

- The field APP_DATE, a 4-byte array field in the record array APP_LIST(3):

```
APP_LIST(3).APP_DATE
```

Scalar Field References

- The field APP_FLAG, a LOGICAL field of the record NEXT_APP:

```
NEXT_APP.APP_FLAG
```

- The first character of APP_MEMO(1), a CHARACTER*20 field of the record NEXT_APP:

```
NEXT_APP.APP_MEMO(1)(1:1)
```

> **NOTE**
> Because periods are used in record references to separate fields, you should avoid using relational operators (.EQ., .XOR.), logical constants (.TRUE., .FALSE.), and logical expressions (.AND., .NOT., .OR.) as field names in structure declarations. Dots can also be used instead of % to separate fields of a derived type.

Consider the following example:

```
module mod
   type T1_t
     integer :: i
   end type T1_t
  type T2_t
     type (T1_t) :: eq
     integer     :: i
  end type T2_t

  interface operator (.eq.)
     module procedure eq_func
  end interface operator (.eq.)
contains
  function eq_func(t2, i) result (rslt)
  type(T2_t), intent (in) :: t2
  integer,    intent (in) :: i
```

```
      rslt = t2%eq%i + i
  end function eq_func
end module mod

use mod
type(T2_t) :: t2
integer    :: i
  t2%eq%i = 0
  t2%i    = -10
  i       = -10
  print *, t2.eq.i, (t2).eq.i
end
```

In this case, the reference "t2.eq.i" prints 0. The reference "(t2).eq.i" will invoke eq_func and will print -10.

See Also

RECORD statement

STRUCTURE
for details on structure declarations

UNION
for details on UNION and MAP statements

Aggregate Assignment

For aggregate assignment statements, the variable and expression must have the same structure as the aggregate they reference.

The aggregate assignment statement assigns the value of each field of the aggregate on the right of an equal sign to the corresponding field of the aggregate on the left. Both aggregates must be declared with the same structure.

Examples

The following example shows valid aggregate assignments:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE

RECORD /DATE/ TODAY, THIS_WEEK(7)
STRUCTURE /APPOINTMENT/
  ...
  RECORD /DATE/ APP_DATE
END STRUCTURE

RECORD /APPOINTMENT/ MEETING

DO I =  1,7
  CALL GET_DATE (TODAY)
  THIS_WEEK(I) =     TODAY
  THIS_WEEK(I).DAY = TODAY.DAY + 1
END DO
MEETING.APP_DATE =   TODAY
```

Additional Character Sets

This topic contains information about the additional character sets you can use in your Fortran programs.

See Also

Character Sets for details on the Standard Fortran character set

Character and Key Code Charts for Windows*

This topic contains the ASCII and ANSI character code charts, and the Key code charts that are available on Windows* OS.

ASCII Character Codes for Windows*

The ASCII character code charts contain the decimal and hexadecimal values of the extended ASCII (American Standards Committee for Information Interchange) character set. The extended character set includes the ASCII character set (Chart 1) and 128 other characters for graphics and line drawing (Chart 2), often called the "IBM* character set".

ASCII Character Codes Chart 1 (W*S)

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	sp	64	40	@	96	60	`
^A	1	01	☺	SOH	33	21	!	65	41	A	97	61	a
^B	2	02	☻	STX	34	22	"	66	42	B	98	62	b
^C	3	03	♥	ETX	35	23	#	67	43	C	99	63	c
^D	4	04	♦	EOT	36	24	$	68	44	D	100	64	d
^E	5	05	♣	ENQ	37	25	%	69	45	E	101	65	e
^F	6	06	♠	ACK	38	26	&	70	46	F	102	66	f
^G	7	07	•	BEL	39	27	'	71	47	G	103	67	g
^H	8	08	◘	BS	40	28	(	72	48	H	104	68	h
^I	9	09	○	HT	41	29	)	73	49	I	105	69	i
^J	10	0A	◙	LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B	♂	VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C	♀	FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D	♪	CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E	♫	SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F	☼	SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10	►	SLE	48	30	0	80	50	P	112	70	p
^Q	17	11	◄	CS1	49	31	1	81	51	Q	113	71	q
^R	18	12	↕	DC2	50	32	2	82	52	R	114	72	r
^S	19	13	‼	DC3	51	33	3	83	53	S	115	73	s
^T	20	14	¶	DC4	52	34	4	84	54	T	116	74	t
^U	21	15	§	NAK	53	35	5	85	55	U	117	75	u
^V	22	16	▬	SYN	54	36	6	86	56	V	118	76	v
^W	23	17	↨	ETB	55	37	7	87	57	W	119	77	w
^X	24	18	↑	CAN	56	38	8	88	58	X	120	78	x
^Y	25	19	↓	EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A	→	SUB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B	←	ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C	∟	FS	60	3C	<	92	5C	\	124	7C	|
^]	29	1D	↔	GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^_	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂†

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

ASCII Character Codes Chart 2: IBM* Character Set (W*S)

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▒
178	B2	▒
179	B3	▒
180	B4	▒
181	B5	▒
182	B6	▒
183	B7	▒
184	B8	▒
185	B9	▒
186	BA	▒
187	BB	▒
188	BC	▒
189	BD	▒
190	BE	▒
191	BF	▒

Dec	Hex	Char
192	C0	▒
193	C1	▒
194	C2	▒
195	C3	▒
196	C4	▒
197	C5	▒
198	C6	▒
199	C7	▒
200	C8	▒
201	C9	▒
202	CA	▒
203	CB	▒
204	CC	▒
205	CD	▒
206	CE	▒
207	CF	▒
208	D0	▒
209	D1	▒
210	D2	▒
211	D3	▒
212	D4	▒
213	D5	▒
214	D6	▒
215	D7	▒
216	D8	▒
217	D9	▒
218	DA	▒
219	DB	▒
220	DC	▒
221	DD	▒
222	DE	▒
223	DF	▒

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

ANSI Character Codes for Windows*

The ANSI character code chart lists the extended character set of most of the programs used by Windows* operating systems. The codes of the ANSI (American National Standards Institute) character set from 32 through 126 are displayable characters from the ASCII character set. The ANSI characters displayed as solid blocks are undefined characters and may appear differently on output devices.

ANSI Character Codes Chart (W*S)

0	■	32		64	@	96	`	128	■	160		192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	130	‚	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	131	ƒ	163	£	195	Ã	227	ã
4	■	36	$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	133	…	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
7	■	39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
8	■	40	(	72	H	104	h	136	ˆ	168	¨	200	È	232	è
9	■	41	)	73	I	105	i	137	‰	169	©	201	É	233	é
10	■	42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
11	■	43	+	75	K	107	k	139	‹	171	«	203	Ë	235	ë
12	■	44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
13	■	45	-	77	M	109	m	141	■	173	-	205	Í	237	í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
17	■	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	147	“	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	148	”	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	149	•	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	151	—	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	152	˜	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
27	■	59	;	91	[	123	{	155	›	187	»	219	Û	251	û
28	■	60	<	92	\	124	|	156	œ	188	¼	220	Ü	252	ü
29	■	61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
30	■	62	>	94	^	126	~	158	■	190	¾	222	Þ	254	þ
31	■	63	?	95	_	127	■	159	Ÿ	191	¿	223	ß	255	ÿ

■ Indicates that this character isn't supported by Windows.

TT Indicates that this character is available only in TrueType fonts.

Key Codes for Windows*

Some keys, such as function keys, cursor keys, and ALT+KEY combinations, have no ASCII code. When a key is pressed, a microprocessor within the keyboard generates an "extended scan code" of two bytes.

The first (low-order) byte contains the ASCII code, if any. The second (high-order) byte has the scan code--a unique code generated by the keyboard when a key is either pressed or released. Because the extended scan code is more extensive than the standard ASCII code, programs can use it to identify keys which do not have an ASCII code.

For more details on key codes, see:

- Key Codes Chart 1
- Key Codes Chart 2

Key Codes Chart 1 (W*S)

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1	01	NUL§
1!	2	02	49	31	1	33	21	!				120	78	NUL
2@	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3#	4	04	51	33	3	35	23	#				122	7A	NUL
4$	5	05	52	34	4	36	24	$				123	7B	NUL
5%	6	06	53	35	5	37	25	%				124	7C	NUL
6^	7	07	54	36	6	94	5E	^	30	1E	RS	125	7D	NUL
7&	8	08	55	37	7	38	26	&				126	7E	NUL
8*	9	09	56	38	8	42	2A	*				127	7F	NUL
9(	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
-_	12	0C	45	2D	-	95	5F	_	31	1F	US	130	82	NUL
=+	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F		14	0E	NUL§
TAB	15	0F	9	09		15	0F	NUL	148	94	NUL§	15	A5	NUL§
Q	16	10	113	71	q	81	51	Q	17	11	DC1	16	10	NUL
W	17	11	119	77	w	87	57	W	23	17	ETB	17	11	NUL
E	18	12	101	65	e	69	45	E	5	05	ENQ	18	12	NUL
R	19	13	114	72	r	82	52	R	18	12	DC2	19	13	NUL
T	20	14	116	74	t	84	54	T	20	14	SO	20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19	EM	21	15	NUL
U	22	16	117	75	u	85	55	U	21	15	NAK	22	16	NUL
I	23	17	105	69	i	73	49	I	9	09	TAB	23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F	SI	24	18	NUL
P	25	19	112	70	p	80	50	P	16	10	DLE	25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B	ESC	26	1A	NUL§
]}	27	1B	93	5D	]	125	7D	}	29	1D	GS	27	1B	NUL§
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	28	1C	NUL§
ENTER♪	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	166	A6	NUL§
LCTRL	29	1D												
RCTRL♪	29	1D												
A	30	1E	97	61	a	65	41	A	1	01	SOH	30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13	DC3	31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04	EOT	32	20	NUL
F	33	21	102	66	f	70	46	F	6	06	ACK	33	21	NUL
G	34	22	103	67	g	71	47	G	7	07	BEL	34	22	NUL
H	35	23	104	68	h	72	48	H	8	08	BS	35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A	LF	36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B	VT	37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C	FF	38	26	NUL
;:	39	27	59	3B	;	58	3A	:				39	27	NUL§
'"	40	28	39	27	'	34	22	"				40	28	NUL§
`~	41	29	96	60	`	126	7E	~				41	29	NUL§
LSHIFT	42	2A												
|	43	2B	92	5C	\	124	7C	|	28	1C	FS			
Z	44	2C	122	7A	z	90	5A	Z	26	1A	SUB	44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18	CAN	45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03	ETX	46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16	SYN	47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02	STX	48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E	SO	49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D	CR	50	32	NUL
,<	51	33	44	2C	,	60	3C	<				51	33	NUL§
.>	52	34	46	2E	.	62	3E	>				52	34	NUL§

Key Codes Chart 2 (W*S)

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
/?	53	35	47	2F	/	63	3F	?				53	34	NUL§
GRAY /£	53	35	47	2F	/	63	3F	?	149	95	NUL	164	A5	NUL
R SHIFT	54	36												
*PRTSC	55	37	42	2A	*	PRTSC		††	16	10				
L ALT	56	38												
R ALT£	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	42	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
F11£	87	57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12£	88	58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
HOME£	71	47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72	48	72	48	NUL	56	38	8	141	8D	NUL§			
UP£	72	48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
PGUP£	73	49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY-	74	4A				45	2D	–						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
LEFT£	75	4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGHT£	77	4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY+	78	4E				43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
END£	79	4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80	50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN£	80	50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
PGDN£	81	51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82	52	82	52	NUL	48	30	0	146	92	NUL§			
INS£	82	52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL
DEL	83	53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL£	83	53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

† Extended codes return 0 (NUL) or E0 (decimal 224) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.
§ These key combinations are only recognized on extended keyboards.
£ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60h), it appears as two bytes (E0h) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16h, only E0h is seen since the interrupt only gives one-byte scan codes.
†† Under MS-DOS, SHIFT + PRTSC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

ASCII Character Set for Linux* and macOS*

This topic describes the ASCII character set that is available on Linux* and macOS* operating systems.

The ASCII character set contains characters with decimal values 0 through 127. The first half of each of the numbered columns identifies the character as you would enter it on a terminal or as you would see it on a printer. Except for SP and HT, the characters with names are nonprintable. In the figure, the characters with names are defined as follows:

NUL	Null	DC1	Device Control 1 (XON)
SOH	Start of Heading	DC2	Device Control 2
STX	Start of Text	DC3	Device Control 1 (XOFF)
ETX	End of Text	DC4	Device Control 4

EOT	End of Transmission	NAK	Negative Acknowledge
ENQ	Enquiry	SYN	Synchronous Idle
ACK	Acknowledge	ETB	End of Transmission Block
BEL	Bell	CAN	Cancel
BS	Backspace	EM	End of Medium
HT	Horizontal Tab	SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tab	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	SP	Space
DLE	Data Link Escape	DEL	Delete

The remaining half of each column identifies the character by the binary value of the byte; the value is stated in three radixes—octal, decimal, and hexadecimal. For example, the uppercase letter A has, under ASCII conventions, a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.

ASCII Character Set (L*X, M*X)

Data Representation Models

Several of the numeric intrinsic functions are defined by a model set for integers (for each intrinsic kind used) and reals (for each real kind used). The bit functions are defined by a model set for bits (binary digits).

The following intrinsic functions provide information on the data representation models:

Intrinsic function	Model	Value returned
BIT_SIZE	Bit	The number of bits (s) in the bit model
DIGITS	Integer or Real	The number of significant digits in the model for the argument
EPSILON	Real	The number that is almost negligible when compared to one
EXPONENT	Real	The value of the exponent part of a real argument
FRACTION	Real	The fractional part of a real argument

Intrinsic function	Model	Value returned
HUGE	Integer or Real	The largest number in the model for the argument
MAXEXPONENT	Real	The maximum exponent in the model for the argument
MINEXPONENT	Real	The minimum exponent in the model for the argument
NEAREST	Real	The nearest different machine-representable number in a given direction
PRECISION	Real	The decimal precision (real or complex) of the argument
RADIX	Integer or Real	The base of the model for the argument
RANGE	Integer or Real	The decimal exponent range of the model for the argument
RRSPACING	Real	The reciprocal of the relative spacing near the argument
SCALE	Real	The value of the exponent part (of the model for the argument) changed by a specified value
SET_EXPONENT	Real	The value of the exponent part (of the model for the argument) set to a specified value
SPACING	Real	The value of the absolute spacing of model numbers near the argument
TINY	Real	The smallest positive number in the model for the argument

For more information on the range of values for each data type (and kind), see Data and I/O in the *Compiler Reference*.

Model for Integer Data

In general, the model set for integers is defined as follows:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

The following values apply to this model set:

- *i* is the integer value.
- *s* is the sign (either +1 or -1).
- *q* is the number of digits (a positive integer).
- *r* is the radix (an integer greater than 1).
- w_k is a nonnegative number less than *r*.

The model for INTEGER(4) follows:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

The following example shows the general integer model for *i* = -20 using a base (*r*) of 2:

$$i = (-1) \times (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4)$$

$$i = (-1) \times (4 + 16)$$

$$i = -1 \times 20$$

$$i = -20$$

Model for Real Data

The model set for reals, in general, is defined as one of the following:

$$x = 0$$

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

The following values apply to this model set:

- *x* is the real value.
- *s* is the sign (either +1 or -1).
- *b* is the base (real radix; an integer greater than 1; *b* = 2 in Intel® Fortran).
- *p* is the number of mantissa digits (an integer greater than 1). The number of digits differs depending on the real format, as follows:

REAL(4)	IEEE binary32	24
REAL(8)	IEEE binary64	53
REAL(16)	IEEE binary128	113

- *e* is an integer in the range e_{min} to e_{max} inclusive. This range differs depending on the real format, as follows:

		e_{min}	e_{max}
REAL(4)	IEEE binary32	-125	128
REAL(8)	IEEE binary64	-1021	1024
REAL(16)	IEEE binary128	-16381	16384

- f_k is a nonnegative number less than *b* (f_1 is also nonzero).

For x = 0, its exponent *e* and digits f_k are defined to be zero.

The model set for single-precision real (REAL(4)) is defined as one of the following:

$$x = 0$$

$$x = s \times 2^{e} \times \left[1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right], -125 \le e \le 128$$

The following example shows the general real model for x = 20.0 using a base (b) of 2:

$$x = 1 \times 2^{5} \times (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$$

$$x = 1 \times 32 \times (.5 + .125)$$

$$x = 32 \times (.625)$$

$$x = 20.0$$

Model for Bit Data

The model set for bits (binary digits) interprets a binary digit w located at position k of a nonnegative integer scalar object based on a model nonnegative integer defined by the following:

$$j = \sum_{k=0}^{s-1} w_k \times 2^{k}$$

The following values apply to this model set:

- j is the integer value.
- s is the number of bits (the length of a sequence of bits).
- w_k is a bit value of 0 or 1. This defines a sequence of bits $w_{s-1}...w_0$, with w_{s-1} the leftmost bit and w_0 the rightmost bit. The positions of bits in the sequence are numbered from right to left, with the position of the rightmost bit being zero.

The interpretation of a negative integer as a sequence of bits is processor dependent.

The inquiry function BIT_SIZE provides the value of the parameter s of the model.

The following example shows the bit model for j = 1001 (integer 9) using a bit number (s) of 4:

$$1 \quad 0 \quad 0 \quad 1$$

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$

$$w_3 \quad w_2 \quad w_1 \quad w_0$$

$$j = (w_0 \times 2^{0}) + (w_1 \times 2^{1}) + (w_2 \times 2^{2}) + (w_3 \times 2^{3})$$

$$j = 1 + 0 + 0 + 8$$

$$j = 9$$

See Also

BIT_SIZE

Bit Sequence Comparisons

When bit sequences of unequal length are compared, the shorter sequence is padded with zero bits on the left, so that it is the same length as the longer sequence.

Bit sequences are compared from left to right, one bit at a time, until unequal bits are found, or until all bits have been compared and found to be equal.

If unequal bits are found, the sequence with zero in the unequal position is considered to be less than the sequence with one in the unequal position. Otherwise the sequences are considered to be equal.

Library Modules and Run-Time Library Routines

Intel® Fortran provides a library of modules that contain routines you can use in your programs:

Module Name	Description
IFAUTO	Interfaces to Automation library routines
IFCOM	Interfaces to COM library routines
IFCOMMONALLOC	Interface to a user-defined routine to dynamically allocate commons
IFCORE	Interfaces to miscellaneous run-time library routines
IFESTABLISH	Interface to a routine to handle Run-Time Library (RTL) errors
IFLOGM	Interfaces to routines from the dialog library
IFMT	Defines derived types for Thread/Process/Synchronization Win32 APIs
IFNLS	Interfaces to National Language Support routines
IFPORT	Interfaces to portability routines
IFPOSIX	Interfaces to Posix-compliant routines
IFQWIN	Interfaces to QuickWin and Graphics library routines
IFWINTY	Defines Fortran parameter constants and derived data types for use with Windows APIs
KERNEL32	Interfaces to the Windows APIs provided by kernel32.dll

NOTE
The same routine name may appear in different modules. These routines may have different semantics, so be careful you are using the module that contains the routine that will produce the results you want.

Run-Time Library Routines

Intel® Fortran provides library modules containing the following routines:

- Routines that help you write programs for graphics, QuickWin, and other applications (in modules IFQWIN, IFLOGM, and IFCORE):
  - QuickWin routines (W*S)
  - Graphics routines (W*S)
  - Dialog routines (W*S)
  - Miscellaneous run-time routines
- Routines systems that help you write programs using Component Object Model (COM) and Automation servers (in modules IFCOM and IFAUTO):
  - COM routines (W*S)
  - AUTO routines (W*S)
- Portability routines that help you port your programs to or from other systems, or help you perform basic I/O to serial ports on Windows* systems (in module IFPORT).
- National Language Support routines that help you write foreign language programs for international markets (in module IFNLS). These routines are only available on Windows* systems.
- POSIX routines that help you write Fortran programs that comply with the POSIX* Standard (in module IFPOSIX).
- ESTABLISHQQ lets you specify a routine to handle Run-Time Library (RTL) errors (in module IFESTABLISH).
- FOR__SET_FTN_ALLOC lets you specify a user-defined routine to dynamically allocate commons. The caller of FOR__SET_FTN_ALLOC must include module IFCOMMONALLOC.

When you include the statement USE *module-name* in your program, these library routines are automatically linked to your program if called.

You can restrict what is accessed from a USE module by adding ONLY clauses to the USE statement.

NOTE
The same routine name may appear in different modules. These routines may have different semantics, so be careful you are using the module that contains the routine that will produce the results you want.

See Also
USE

Overview of NLS and MCBS Routines (Windows*)

The NLS and MCBS routines are only available on Windows* systems. These library routines for handling extended and multibyte character sets are divided into three categories:

- Locale Setting and Inquiry routines to set locales (local code sets) and inquire about their current settings

  At program startup, the current language and country setting is retrieved from the operating system. The user can change this setting through the Control Panel Regional Settings icon. The current codepage is also retrieved from the system.

  There is a system default console codepage and a system default Windows codepage. Console programs retrieve the system console codepage, while Windows programs (including QuickWin applications) retrieve the system Windows codepage.

  The NLS Library provides routines to determine the current locale (local code set), to return parameters of the current locale, to provide a list of all the system supported locales, and to set the locale to another language, country and/or codepage. These routines are summarized in the following table. Note that the locales and codepages set with these routines affect only the program or console that calls the routine. They do not change the system defaults or affect other programs or consoles.
- NLS Formatting routines to format dates, currency, and numbers

  You can set time, date, currency and number formats from the Control Panel, by clicking on the Regional Settings icon. The NLS Library also provides formatting routines for the current locale. These routines are summarized in the following table. These routines return strings in the current codepage, set by default at program start or by NLSSetLocale.

All the formatting routines return the number of bytes in the formatted string (not the number of characters, which can vary if multibyte characters are included). If the output string is longer than the formatted string, the output string is blank padded. If the output string is shorter than the formatted string, an error occurs, NLS$ErrorInsufficientBuffer is returned, and nothing is written to the output string.

- Multibyte Character Set (MBCS) routines for using multi-byte characters

  Examples of multibyte character sets are Japanese, Korean, and Chinese.

  - The MBCS inquiry routines provide information on the maximum length of multibyte characters, the length, number and position of multibyte characters in strings, and whether a multibyte character is a leading or trailing byte. These routines are summarized in the following table. The NLS library provides a parameter, MBLenMax, defined in the NLS module to be the longest length (in bytes) of any character, in any codepage. This parameter can be useful in comparisons and tests. To determine the maximum character length of the current codepage, use the MBCurMax function.
  - There are four MBCS conversion routines. Two convert Japan Industry Standard (JIS) characters to Microsoft Kanji characters or vice versa. Two convert between a codepage multibyte character string and a Unicode string.
  - There are several MBCS Fortran equivalent routines. They are the exact equivalents of Fortran intrinsic routines except that the MBCS equivalents allow character strings to contain multibyte characters.

Examples

The following example uses Locale Setting and Inquiry routines:

```
    USE IFNLS
    INTEGER(4) strlen, status
    CHARACTER(40) str
    strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
    print *, str    ! prints Monday
    strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
    print *, str    ! prints Tuesday
    strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME3, str)
    print *, str    ! prints Wednesday
! Change locale to Spanish, Mexico
    status = NLSSetLocale("Spanish", "Mexico")
    strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
    print *, str    ! prints lunes
    strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
    print *, str    ! prints martes
    strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME3, str)
    print *, str    ! prints miércoles
    END
```

The following example uses NLS Formatting routines:

```
USE IFNLS
INTEGER(4) strlen, status
CHARACTER(40) str
strlen = NLSFormatTime(str)
print *, str             ! prints               11:42:24 AM
strlen = NLSFormatDate(str, flags= NLS$LongDate)
print *, str             ! prints        Friday, July 14, 2000
status = NLSSetLocale ("Spanish", "Mexico")
strlen = NLSFormatTime(str)
print *, str             ! prints               11:42:24
print *, str             ! prints viernes 14 de julio de 2000
```

The following example uses Multibyte Character Set (MBCS) inquiry routines:

```
  USE IFNLS
  CHARACTER(4) str
  INTEGER status
  status = NLSSetLocale ("Japan")
  str = " ･, " ¿"
  PRINT '(1X,''String by char = '',\)'
  DO i = 1, len(str)
     PRINT '(A2,\)',str(i:i)
  END DO
  PRINT '(/,1X,''MBLead = '',\)'
  DO i = 1, len(str)
     PRINT '(L2,\)',mblead(str(i:i))
  END DO
  PRINT '(/,1X,''String as whole = '',A,\)',str
  PRINT '(/,1X,''MBStrLead = '',\)'
  DO i = 1, len(str)
     PRINT '(L1,\)',MBStrLead(str,i)
  END DO
  END
```

This code produces the following output for *str* = ･ , " ¿

```
String by char  =  ･ ･ ･ﾝ
MBLead          =  T T T F
String as whole = 高徳
MBStrLead       = TFTF
```

The following example uses Multibyte Character Set (MBCS) Fortran equivalent routines:

```
  USE IFNLS
  INTEGER(4) i, len(7), infotype(7)
  CHARACTER(10) str(7)
  LOGICAL(4) log4
  data infotype / NLS$LI_SDAYNAME1, NLS$LI_SDAYNAME2, &
 & NLS$LI_SDAYNAME3, NLS$LI_SDAYNAME4, &
 & NLS$LI_SDAYNAME5, NLS$LI_SDAYNAME6, &
 & NLS$LI_SDAYNAME7 /
  WRITE(*,*) 'NLSGetLocaleInfo'
  WRITE(*,*) '----------------'
  WRITE(*,*) ' '
  WRITE(*,*) 'Getting the names of the days of the week...'
  DO i = 1, 7
     len(i) = NLSGetLocaleInfo(infotype(i), str(i))
     WRITE(*, 11) 'len/str/hex = ', len(i), str(i), str(i)
  END DO
11 FORMAT (1X, A, I2, 2X, A10, 2X, '[', Z20, ']')
  WRITE(*,*) ' '
  WRITE(*,*) 'Lexically comparing the names of the days...'
  DO i = 1, 6
     log4 = MBLGE(str(i), str(i+1), NLS$IgnoreCase)
     WRITE(*, 12) 'Is day ', i, ' GT day ', i+1, '? Answer = ', log4
  END DO
12 FORMAT (1X, A, I1, A, I1, A, L1)
```

```
  WRITE(*,*) ' '
  WRITE(*,*) 'Done.'
  END
```

This code produces the following output when the locale is Japan:

```
NLSGetLocaleInfo
----------------

Getting the names of the days of the week...
len/str/hex =  6  月曜日      [8C8E976A93FA20202020]
len/str/hex =  6  火曜日      [89CE976A93FA20202020]
len/str/hex =  6  水曜日      [9085976A93FA20202020]
len/str/hex =  6  木曜日      [96D8976A93FA20202020]
len/str/hex =  6  金曜日      [8BE0976A93FA20202020]
len/str/hex =  6  土曜日      [9379976A93FA20202020]
len/str/hex =  6  日曜日      [93FA976A93FA20202020]

Lexically comparing the names of the days...
Is day 1 GT day 2?  Answer = T
Is day 2 GT day 3?  Answer = F
Is day 3 GT day 4?  Answer = F
Is day 4 GT day 5?  Answer = T
Is day 5 GT day 6?  Answer = F
Is day 6 GT day 7?  Answer = F

Done.
```

See Also

National Language Support Library Routines
Standard Fortran Routines That Handle MBCS Characters
USE

Standard Fortran Routines That Handle MBCS Characters (Windows*)

This section describes Fortran routines that work as usual even if MBCS characters are included in strings.

Because a space can never be a lead or tail byte, many routines that deal with spaces work as expected on strings containing MBCS characters. Such functions include:

- ADJUSTL (*string*)
- ADJUSTR (*string*)
- TRIM (*string*)

Some routines work with the computer collating sequence to return a character in a certain position in the sequence or the position in the sequence of a certain character. These functions are not dependent on a particular collating sequence. (You should note, however, that elsewhere in this manual the ASCII collating sequence is mentioned in reference to these functions.) Such functions use *position* and *c* values between 0 and 255 (inclusive) and include:

- ACHAR (*position*)
- CHAR (*position*[,*kind*])
- IACHAR (*c*)
- ICHAR (*c*)

Because Fortran uses character lengths instead of NULLs to indicate the length of a string, some functions work solely from the length of the string, and not with the contents of the string. This function works as usual on strings containing MBCS characters, and include:

REPEAT (*string*, *ncopies*)

Overview of Portability Routines

This section summarizes portability routines.

Information Retrieval Routines

Information retrieval routines return information about system commands, command-line arguments, environment variables, and process or user information.

Group, user, and process ID are INTEGER(4) variables. Login name and host name are character variables. The functions GETGID and GETUID are provided for portability, but always return 1.

Process Control Routines

Process control routines control the operation of a process or subprocess. You can wait for a subprocess to complete with either SLEEP or ALARM, monitor its progress and send signals via KILL, and stop its execution with ABORT.

In spite of its name, KILL does not necessarily stop execution of a program. Rather, the routine signaled could include a handler routine that examines the signal and takes appropriate action depending on the code passed.

Note that when you use SYSTEM, commands are run in a separate shell. Defaults set with the SYSTEM function, such as current working directory or environment variables, do not affect the environment the calling program runs in.

The portability library does not include the FORK routine. On Linux* and macOS* systems, FORK creates a duplicate image of the parent process. Child and parent processes each have their own copies of resources, and become independent from one another.

On Windows* systems, you can create a child process (called a thread), but both parent and child processes share the same address space and share system resources. If you need to create another process, use the CreateProcessWindows API routine.

Numeric Values and Conversion Routines

Numeric values and conversion routines are available for calculating Bessel functions, data type conversion, and generating random numbers. Some of these functions have equivalents in Standard Fortran, in which case the standard Fortran routines should be used.

Data object conversion can be accomplished by using the INT intrinsic function instead of LONG or SHORT. The intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED perform the same functions as the random number functions listed in the table showing numeric values and conversion routines.

Other bit manipulation functions such as AND, XOR, OR, LSHIFT, and RSHIFT are intrinsic functions. You do not need the IFPORT module to access them. Standard Fortran includes many bit operation routines, which are listed in the Bit Operation and Representation Routines table.

Input and Output Routines

The portability library contains routines that change file properties, read and write characters and buffers, and change the offset position in a file. These input and output routines can be used with standard Fortran input or output statements such as READ or WRITE on the same files, provided that you take into account the following:

- When used with direct files, after an FSEEK, GETC, or PUTC operation, the record number is the number of the next whole record. Any subsequent normal Fortran I/O to that unit occurs at the next whole record. For example, if you seek to absolute location 1 of a file whose record length is 10, the NEXTREC returned by an INQUIRE would be 2. If you seek to absolute location 10, NEXTREC would still return 2.
- On units with CARRIAGECONTROL='FORTRAN' (the default), PUTC and FPUTC characters are treated as carriage control characters if they appear in column 1.
- On sequentially formatted units, the C string "\n"c, which represents the carriage return/line feed escape sequence, is written as CHAR(13) (carriage return) and CHAR(10) (line feed), instead of just line feed, or CHAR(10). On input, the sequence 13 followed by 10 is returned as just 10. (The length of character string "\n"c is 1 character, whose ASCII value, indicated by ICHAR('\n'c), is 10.)
- Reading and writing is in a raw form for direct files. Separators between records can be read and overwritten. Therefore, be careful if you continue using the file as a direct file.

I/O errors arising from the use of these routines result in an Intel® Fortran run-time error.

Some portability file I/O routines have equivalents in Standard Fortran. For example, you could use the ACCESS function to check a file specified by name for accessibility according to mode. It tests a file for read, write, or execute permission, as well as checking to see if the file exists. It works on the file attributes as they exist on disk, not as a program's OPEN statement specifies them.

Instead of ACCESS, you can use the INQUIRE statement with the ACTION specifier to check for similar information. (The ACCESS function always returns 0 for read permission on FAT files, meaning that all files have read permission.)

Date and Time Routines

Various date and time routines are available to determine system time, or convert it to local time, Greenwich Mean Time, arrays of date and time elements, or an ASCII character string.

DATE and TIME are available as either a function or subroutine. Because of the name duplication, if your programs do not include the USE IFPORT statement, each separately compiled program unit can use only one of these versions. For example, if a program calls the subroutine TIME once, it cannot also use TIME as a function.

Standard Fortran includes date and time intrinsic subroutines. For more information, see DATE_AND_TIME.

Error Handling Routines

Error handling routines detect and report errors.

IERRNO error codes are analogous to *errno* on Linux* and macOS* systems. The IFPORT module provides parameter definitions for many of UNIX's *errno* names, found typically in `errno.h` on UNIX systems.

IERRNO is updated only when an error occurs. For example, if a call to the GETC function results in an error, but two subsequent calls to PUTC succeed, a call to IERRNO returns the error for the GETC call. Examine IERRNO immediately after returning from one of the portability library routines. Other Standard Fortran routines might also change the value to an undefined value.

If your application uses multithreading, remember that IERRNO is set on a per-thread basis.

System, Drive, or Directory Control and Inquiry Routines

You can retrieve information about devices, directories, and files with the functions listed below. File names can be long file names or UNC file names. A forward slash in a path name is treated as a backslash. All path names can contain drive specifications.

Standard Fortran provides the INQUIRE statement, which returns detailed file information either by file name or unit number. Use INQUIRE as an equivalent to FSTAT, LSTAT, or STAT. LSTAT and STAT return the same information; STAT is the preferred function.

Serial Port Routines (Windows* only)

The serial port I/O (SPORT_*xxx*) routines help you perform basic input and output to serial ports. These routines are available only on systems using IA-32 architecture.

Additional Routines

You can also use portability routines for program call and control, keyboards and speakers, file management, arrays, floating-point inquiry and control, IEEE* functionality, and other miscellaneous uses.

NOTE
On Windows* systems, all portability routines that take path names also accept long file names or UNC (Universal Naming Convention) file names. A forward slash in a path name is treated as a backslash. All path names can contain drive specifications as well as MBCS (multiple-byte character set) characters.

See Also

Portability Routines
Overview of Serial Port I/O Routines

Overview of Serial Port I/O Routines (Windows*)

The serial port I/O (SPORT_*xxx*) routines help you perform basic input and output to serial ports. These routines are available only on Windows* systems.

The programming model is much the same as a normal file except the user does a connect (SPORT_CONNECT, SPORT_CONNECT_EX) and release (SPORT_RELEASE) to the port instead of an open and close of a file.

Two types of read and write operations (as determined in a mode on the connect call) are provided:

- Read and write arbitrary data from/to the port using SPORT_READ_DATA and SPORT_WRITE_DATA.
- Read and writes line-terminated data using SPORT_READ_LINE and SPORT_WRITE_LINE.

Once any I/O operation has been requested on the port, an additional thread is started that keeps a read outstanding on the port so that data will not be missed.

The SPORT_SET_STATE, SPORT_SET_STATE_EX, and SPORT_SET_TIMEOUTS (W*32 W*64) routines allow you to set basic port parameters such as baud rate, stop bits, timeouts, and so on. Additionally, you can call SPORT_GET_HANDLE to return the Windows* handle to the port so that you can call Windows* Communication Functions to implement additional needs.

Calling the Serial Port I/O Routines

The SPORT_*xxx* routines are functions that return an error status:

- An error status of 0 (zero) indicates success
- Other values are Windows* error values that indicate an error

As described in the calling syntax, these routines require the following `USE` statement:

```
USE IFPORT
```

The `USE IFPORT`statement includes the routine definitions in the compilation. You may also need to add a `USE IFWINTY` statement to your program because some Windows* constants may be required that are typically defined in the IFWINTY module.

Many arguments are optional. If a constant is used where an argument is both input and output, a probe for writeability is done before the output. Thus, in many cases, a constant may be used without creating a temporary variable. It should be noted, however, that doing so may not be compatible with all Fortran implementations.

Run-Time Behavior of the Serial Port I/O Routines

To help ensure that data overruns do not occur, the SPORT_*xxx* run-time support creates a separate thread that maintains an outstanding read to the connected port. This thread is started when any read or write operation is performed to the port using the affiliated read/write routine. As such, port parameters must not be changed after you have started reading or writing to the port. Instead, you should set up the port parameters after connecting to the port and then leave them unchanged until after the port has been released.

If the parameters of the port must be changed more dynamically, use the SPORT_CANCEL_IO routine to ensure that no I/O is in progress. Additionally, that call will kill the helper thread so that it will automatically pick up the new, correct, parameters when it restarts during the next I/O operation.

Serial Port Usage

Depending upon the application, serial port programming can be very simple or very complex . The SPORT_*xxx* routines are intended to provide a level of support that will help the user implement simple applications as well as providing a foundation that can be used for more complex applications. Users needing to implement full serial port protocols (such as a PPP/SLIP implementation or some other complex protocol) should use the Windows* Communication Functions directly to achieve the detailed level of control needed in those cases. Simple tasks, such as communicating with a terminal or some other data collection device are well suited for implementations using the SPORT_*xxx* routines.

You should first familiarize yourself with the hardware connection to the serial device. Typical connections used today involve either a 9 pin/wire connector or a 25 pin/wire connector. Many cables do not implement all 9 or 25 connections in order to save on costs. For certain applications these subset cables may work just fine but others may require the full 9 or 25 connections. All cables will implement the Receive/SendData signals as well as the SignalGround. Without these signals, there can be no data transfer. There are two other categories of important signals:

- Signals used for flow control

  Flow control signals tell the device/computer on the other end of the cable that data may be sent or that they should wait. Typically, the RequestToSend/ClearToSend signals are used for this purpose. Other signals such as DataSetReady or DataTerminalReady may also be used. Make sure that the cable used implements all the signals required by your hardware/software solution. Special characters (normally as XON/XOFF) may also be used to control the flow of data instead of or in addition to the hardware signals. Check your specific application to see what cabling is needed.
- Signals that indicate status or state of a modem or phone connection.

  These signals may not be required if the connection between the computer and the device is direct and not through a modem. This signals typically convey information such as the state of the carrier (CarrierDetect) or if the phone line is ringing (Ring). Again, make sure the cable used implements all the signals required for your application.

After the correct physical connection has been set up the programmer must become familiar with the data protocol used to communicate with the remote device/system.

Many simple devices terminate parcels of data with a "record terminator" (often a carriage return or line feed character). Other devices may simply send data in fixed length packets or packets containing some sort of count information. The two types of I/O routines provided by the SPORT_*xxx* support (line oriented using SPORT_READ_LINE and SPORT_WRITE_LINE or transfer raw data using SPORT_READ_DATA and SPORT_WRITE_DATA) can handle these two types of data transfer. The programmer must become familiar with the particular application to determine which type of I/O is more appropriate to use.

The SPORT_*xxx* routines call Windows* routines. For example, the SPORT_SET_STATE routine calls the routine SetCommState, which uses the DCB Communications Structure.

See Also

Portability Routines for a list of the SPORT routines

Summary of Language Extensions

This appendix summarizes the Intel® Fortran language extensions to the ANSI/ISO Fortran 2003 Standard.

Most extensions are available on all supported operating systems. However, some extensions are limited to one or more platforms. If an extension is limited, it is labeled.

Language Extensions: Source Forms

The following are extensions to the methods and rules for source forms:

- Tab-formatting as a method to code lines
- The letter D as a debugging statement indicator in column 1 of fixed or tab source form
- An optional statement field width of 132 columns for fixed or tab source form
- An optional sequence number field for fixed source form
- Up to 511 continuation lines in a source program

Language Extensions: Names

As an extension, the dollar sign ($) is a valid character in names, and can be the first character.

Language Extensions: Character Sets

The following are extensions to the standard character set:

- The Tab (<Tab>) character (see Character Sets)
- ASCII Character Code Chart 2 -- IBM* Character Set
- ANSI Character Code Chart
- Key Code Charts

Language Extensions: Intrinsic Data Types

The following are data-type extensions:

BYTE	INTEGER*1	REAL*16
DOUBLE COMPLEX	INTEGER*2	COMPLEX*8
LOGICAL*1	INTEGER*4	COMPLEX*16
LOGICAL*2	INTEGER*8	COMPLEX*32
LOGICAL*4	REAL*4	
LOGICAL*8	REAL*8	

See Also

Intrinsic Data Types

Language Extensions: Constants

Hollerith constants are allowed as an extension.

C Strings are allowed as extensions in character constants.

Language Extensions: Expressions and Assignment

When operands of different intrinsic data types are combined in expressions, conversions are performed as necessary (see Data Type of Numeric Expressions).

Binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

The following are extensions allowed in logical expressions:

- .XOR. as a synonym for .NEQV.
- Integers as valid logical items
- Logical operators applied to integers bit-by-bit

Language Extensions: Specification Statements

The following specification attributes and statements are extensions:

- AUTOMATIC attribute and statement
- STATIC attribute and statement

Language Extensions: Execution Control

The following control statements are extensions:

- ASSIGN
- Assigned GO TO
- IF - Arithmetic
- Non-block form of a DO statement
- PAUSE

These are older Fortran features that have been deleted from the Fortran Standard. Intel® Fortran fully supports these features.

Language Extensions: Compilation Control Lines and Statements

The following line option and statement are extensions that can influence compilation:

- [/[NO]LIST], which can be specified in an INCLUDE line
- The OPTIONS statement

Language Extensions: Built-In Functions

The following built-in functions are extensions:

- %VAL, %REF, and %LOC, which facilitate references to non-Fortran procedures
- %FILL, which can be used in record structure type definitions

Language Extensions: I/O Statements

The following I/O statements are extensions:

- The ACCEPT statement
- The REWRITE statement
- The TYPE statement, which is a synonym for the PRINT statement

Language Extensions: I/O Formatting

The following are extensions allowed in I/O Formatting:

- The Q edit descriptor
- The dollar sign ($) edit descriptor and carriage-control character
- The backslash (\) edit descriptor
- The ASCII NUL carriage-control character
- Variable format expressions
- The H edit descriptor

  This is an older Fortran feature that has been deleted in Fortran 95. Intel® Fortran fully supports this feature.

Language Extensions: File Operation Statements

The following statement specifiers and statements are extensions:

- CLOSE statement specifiers:
  - STATUS values: 'SAVE' (as a synonym for 'KEEP'), 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT/DELETE'
  - DISPOSE (or DISP)
- DELETE statement
- INQUIRE statement specifiers:
  - BINARY (W*S)
  - BLOCKSIZE
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - FORM values: 'UNKNOWN', 'BINARY' (W*S)
  - IOFOCUS (W*S)
  - MODE as a synonym for ACTION
  - ORGANIZATION
  - RECORDTYPE
  - SHARE (W*S)

  See also INQUIRE Statement.
- OPEN statement specifiers:
  - ACCESS values: 'APPEND'
  - ASSOCIATEVARIABLE
  - BLOCKSIZE
  - BUFFERCOUNT
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - DISPOSE (or DISP)
  - FORM value: 'BINARY' (W*S)
  - IOFOCUS (W*S)
  - MAXREC
  - MODE as a synonym for ACTION
  - NAME as a synonym for FILE
  - NOSHARED

- ORGANIZATION
- READONLY
- RECORDSIZE as a synonym for RECL
- RECORDTYPE
- SHARE (W*S)
- SHARED
- TITLE (W*S)
- TYPE as a synonym for STATUS
- USEROPEN

See also OPEN Statement.

Language Extensions: Compiler Directives

The following General Directives are extensions:

- ALIAS
- ASSUME
- ASSUME_ALIGNED
- ATTRIBUTES
- BLOCK_LOOP and NOBLOCK_LOOP
- DECLARE and NODECLARE
- DEFINE and UNDEFINE
- DISTRIBUTE POINT
- FIXEDFORMLINESIZE
- FMA and NOFMA
- FORCEINLINE
- FREEFORM and NOFREEFORM
- IDENT
- IF and IF DEFINED
- INLINE and NOINLINE
- INTEGER
- IVDEP
- LOOP COUNT
- MESSAGE
- NOFUSION
- OBJCOMMENT
- OPTIMIZE and NOOPTIMIZE
- OPTIONS
- PACK
- PARALLEL and NOPARALLEL (loop)
- PREFETCH and NOPREFETCH
- PSECT
- REAL
- SIMD
- STRICT and NOSTRICT
- UNROLL and NOUNROLL
- UNROLL_AND_JAM and NOUNROLL_AND_JAM
- VECTOR and NOVECTOR

Language Extensions: Intrinsic Procedures

The following intrinsic procedures are extensions available on all platforms:

A to D

ACOSD	BIEOR	COSD	DBLEQ

AIMIN0	BIOR	COTAND	DCMPLX
AJMAX0	BITEST	CQABS	DCONJG
AJMIN0	BJTEST	CQCOS	DCOSD
AKMAX0	BKTEST	CQEXP	DCOTAN
AKMIN0	BMOD	CQLOG	DCOTAND
AND	BMVBITS	CQSIN	DERF
ASINH	BNOT	CQSQRT	DERFC
ATAN2D	BSHFT	CQTAN	DFLOAT
ATAND	BSHFTC	CTAN	DFLOTI
BABS	BSIGN	DACOSD	DFLOTJ
BADDRESS	CACHESIZE	DACOSH	DFLOTK
BBCLR	CDABS	DASIND	DIMAG
BBITS	CDCOS	DASINH	DNUM
BBSET	CDEXP	DATAN2D	DREAL
AIMAX0	CDLOG	DATAND	DSHIFTL
BBTEST	CDSIN	DATAN	DSHIFTR
BDIM	CDSQRT	COTAN	DSIND
BIAND	CDTAN	DATE	DTAND

E to I

EOF	HMOD	IIDNNT	IMVBITS
ERRSNS	HMVBITS	IIEOR	ININT
EXIT	HNOT	IIFIX	INOT
FLOATI	HSHFT	IINT	INT1
FLOATJ	HSHFTC	IIOR	INT2
FLOATK	HSIGN	IIQINT	INT4
FP_CLASS	HTEST	IIQNNT	INT8
FREE	IADDR	IISHFT	INT_PTR_KIND
GETARG	IARG	IISHFTC	INUM
HABS	IARGC	IISIGN	IQINT
HBCLR	IBCHNG	IIXOR	IQNINT
HBITS	IDATE	IJINT	ISHA
HBSET	IIABS	ILEN	ISHC
HDIM	IIAND	IMAG	ISHL
HFIX	IIBCLR	IMAX0	ISNAN

HIAND	IIBITS	IMAX1	IXOR
HIEOR	IIBSET	IMIN0	IZEXT
HIOR	IIDIM	IMIN1	
HIXOR	IIDINT	IMOD	

J to P

JFIX	JISIGN	KIBSET	KMOD
JIABS	JIXOR	KIDIM	KMVBITS
JIAND	JMAX0	KIDINT	KNINT
JIBCLR	JMAX1	KIDNNT	KNOT
JIBITS	JMIN0	KIEOR	KNUM
JIBSET	JMIN1	KIFIX	KZEXT
JIDIM	JMOD	KINT	LOC
JIDINT	JMVBITS	KIOR	LSHIFT
JIDNNT	JNINT	KIQINT	LSHFT
JIEOR	JNOT	KIQNNT	MALLOC
JIFIX	JNUM	KISHFT	MCLOCK
JINT	JZEXT	KISHFTC	MM_PREFETCH
JIOR	KDIM	KISIGN	NARGS
JIQINT	KIABS	KMAX0	NUMARG
JIQNNT	KIAND	KMAX1	OR
JISHFT	KIBCLR	KMIN0	
JISHFTC	KIBITS	KMIN1	

Q to Z

QABS	QCOSH	QNINT	SIZEOF
QACOS	QCOTAN	QNUM	SNGLQ
QACOSD	QCOTAND	QREAL	TAND
QACOSH	QDIM	QSIGN	TIME
QARCOS	QERF	QSIN	TRAILZ
QASIN	QERFC	QSIND	XOR
QASIND	QEXP	QSINH	ZABS
QASINH	QEXT	QSQRT	ZCOS
QATAN	QEXTD	QTAN	ZEXP
QATAN2	QFLOAT	QTAND	ZEXT
QATAN2D	QIMAG	QTANH	ZLOG

QATAND	QINT	RAN	ZSIN
QATANH	QLOG	RANF	ZSQRT
QCMPLX	QLOG10	RANDU	ZTAN
QCONJG	QMAX1	RNUM	
QCOS	QMIN1	RSHIFT	
QCOSD	QMOD	SIND	

Language Extensions: Additional Language Features

The following are language extensions that facilitate compatibility with other versions of Fortran:

- DEFINE FILE statement
- ENCODE and DECODE statements
- FIND statement
- The INTERFACE TO statement
- FORTRAN 66 Interpretation of the EXTERNAL statement
- An alternative syntax for the PARAMETER statement
- VIRTUAL statement
- AND, OR, XOR, IMAG, LSHIFT, RSHIFT intrinsics (see the *A to Z Reference*)
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for an I/O record specifier
- An alternate syntax for the DELETE statement
- An alternative form for namelist external records
- The integer POINTER statement
- Record structures

Language Extensions: Run-Time Library Routines

The following run-time library routines are available as extensions:

- Run-Time Library Routines
- OpenMP* Run-time Library Routines for Fortran

A to Z Reference

This section contains the following:

- Language Summary Tables

  This section organizes the Fortran functions, subroutines, and statements by the operations they perform. You can use the tables to locate a particular routine for a particular task.
- The descriptions of all Fortran statements, intrinsics, directives, and module library routines, which are listed in alphabetical order.

  In the description of routines, pointers and handles are INTEGER(4) on IA-32 architecture and INTEGER(8) on Intel® 64 architecture.

The Fortran compiler understands statements and intrinsic functions in your program without any additional information, such as that provided in modules.

However, modules must be included in programs that contain the following routines:

- Quickwin routines and graphics routines (W*S)

These routines require a USE IFQWIN statement to include the library and graphics modules.

- Portability routines

  It is recommended that you specify USE IFPORT when accessing routines in the portability library.
- Serial port I/O routines

  These routines require a USE IFPORT statement to access the portability library. These routines are only available on Windows* systems.
- NLS routines (W*S)

  These routines require a USE IFNLS statement to access the NLS library.
- POSIX* routines

  These routines require a USE IFPOSIX statement to access the POSIX library.
- Dialog routines (W*S)

  These routines require a USE IFLOGM statement to access the dialog library.
- Component Object Module (COM) routines (W*S)

  These routines require a USE IFCOM statement to access the COM library.
- Automation server routines (W*S)

  These routines require a USE IFAUTO statement to access the AUTO library.
- Miscellaneous Run-Time Routines

  Most of these routines require a USE IFCORE statement to obtain the proper interfaces.

Whenever required, these USE module statements are prominent in the *A to Z Reference*.

In addition to the appropriate USE statement, for some routines you must specify the types of libraries to be used when linking.

Language Summary Tables

Statements for Program Unit Calls and Definitions

The following table lists statements used for program unit definition and procedure call and return.

Name	Description
BLOCK DATA	Identifies a block-data subprogram.
CALL	Executes a subroutine.
COMMON	Delineates variables shared between program units.
CONTAINS	Identifies the start of module procedures within a host module, contained procedures within a procedure, or bound procedures within a type.
ENTRY	Specifies a secondary entry point to a subroutine or external function.
EXTERNAL	Declares a name to be that of a user-defined subroutine or function, making it passable as an argument.
FUNCTION	Identifies a program unit as a function.
INCLUDE	Inserts the contents of a specified file into the source file.
INTERFACE	Specifies an explicit interface for external functions and subroutines.
INTRINSIC	Declares a predefined function.

Name	Description
MODULE	Identifies a module program unit.
PROGRAM	Identifies a program unit as a main program.
RETURN	Returns control to the program unit that called a subroutine or function.
SUBROUTINE	Identifies a program unit as a subroutine.
USE	Gives a program unit access to a module.

Statements Affecting Variables

The following table lists statements that affect variables.

Name	Description
AUTOMATIC	Declares a variable on the stack, rather than at a static memory location.
BYTE	Specifies variables as the BYTE data type; BYTE is equivalent to INTEGER(1).
CHARACTER	Specifies variables as the CHARACTER data type.
CODIMENSION	Specifies that an entity is a coarray and specifies its corank and cobounds, if any.
COMPLEX	Specifies variables as the COMPLEX data type.
DATA	Assigns initial values to variables.
DIMENSION	Specifies that an entity is an array and specifies its rank and bounds.
DOUBLE COMPLEX	Specifies variables as the DOUBLE COMPLEX data type, equivalent to COMPLEX(8).
DOUBLE PRECISION	Specifies variables as the DOUBLE-PRECISION real data type, equivalent to REAL(8).
EQUIVALENCE	Specifies that two or more variables or arrays share the same memory location.
IMPLICIT	Specifies the default types for variables and functions.
INTEGER	Specifies variables as the INTEGER data type.
LOGICAL	Specifies variables as the LOGICAL data type.
MAP	Within a UNION statement, delimits a group of variable type declarations that are to be ordered contiguously within memory.
NAMELIST	Declares a group name for a set of variables to be read or written in a single statement.
PARAMETER	Equates a constant expression with a name.
PROTECTED	Specifies limitations on the use of module entities.

Name	Description
REAL	Specifies variables as the REAL data type.
RECORD	Declares one or more variables of a user-defined structure type.
SAVE	Causes variables to retain their values between invocations of the procedure in which they are defined.
STATIC	Declares a variable is in a static memory location, rather than on the stack.
STRUCTURE	Defines a new variable type, composed of a collection of other variable types.
TYPE	Defines a new variable type, composed of a collection of other variable types.
UNION	Within a structure, causes two or more maps to occupy the same memory locations.
VOLATILE	Specifies that the value of an object is totally unpredictable based on information available to the current program unit.

Statements for Input and Output

The following table lists statements used for input and output.

Name	Procedure Type	Description
ACCEPT	Statement	Similar to a formatted, sequential READ statement.
BACKSPACE	Statement	Positions a file to the beginning of the previous record.
CLOSE	Statement	Disconnects the specified unit.
DELETE	Statement	Deletes a record from a relative file.
ENDFILE	Statement	Writes an end-of-file record or truncates a file.
FLUSH	Statement	Causes data written to a file to become available to other processes or causes data written to a file outside of Fortran to be accessible to a READ statement.
INQUIRE	Statement	Returns the properties of a file or unit.
OPEN	Statement	Associates a unit number with an external device or file.
PRINT(or TYPE)	Statement	Displays data on the screen.

Name	Procedure Type	Description
READ	Statement	Transfers data from a file to the items in an I/O list.
REWIND	Statement	Repositions a file to its first record.
REWRITE	Statement	Rewrites the current record.
WRITE	Statement	Transfers data from the items in an I/O list to a file

Compiler Directives

The following tables list available compiler directives.

Each OpenMP* Fortran directive name is preceded by the prefix *c*$OMP, where *c* is one of the following: !, C (or c), or *; for example, !$OMP ATOMIC.

Compiler directives are specially formatted comments in the source file which provide information to the compiler. Some directives, such as line length or conditional compilation directives provide the compiler information which is used in interpreting the source file. Other directives, such as optimization directives provide hints or suggestions to the compiler, which, in some cases, may be ignored or overridden by the compiler based on the heuristics of the optimizer and/or code generator. If the directive is ignored by the compiler, no diagnostic message is issued.

You do not need to specify a compiler option to enable general directives.

Some directives may perform differently on Intel® microprocessors than on non-Intel microprocessors.

General Directives

Name	Description
ALIAS	Specifies an alternate external name to be used when referring to an external subprogram.
ASSUME	Provides heuristic information to the compiler optimizer.
ASSUME_ALIGNED	Specifies that an entity in memory is aligned.
ATTRIBUTES	Applies attributes to variables and procedures.
BLOCK_LOOP	Enables loop blocking for the immediately following nested DO loops.
DECLARE	Generates warning messages for undeclared variables.
DEFINE	Creates a variable whose existence can be tested during conditional compilation.
DISTRIBUTE POINT	Suggests a location at which a DO loop may be split.
ELSE	Marks the beginning of an alternative conditional-compilation block to an IF directive construct.

Name	Description
ELSEIF	Marks the beginning of an alternative conditional-compilation block to an IF directive construct.
ENDIF	Marks the end of a conditional-compilation block.
FIXEDFORMLINESIZE	Sets fixed-form line length. This directive has no effect on freeform code.
FMA	Tells the compiler to allow generation of fused multiply-add (FMA) instructions, also known as floating-point contractions.
FORCEINLINE	Specifies that a routine should be inlined whenever the compiler can do so.
FREEFORM	Uses freeform format for source code.
IDENT	Specifies an identifier for an object module.
IF	Marks the beginning of a conditional-compilation block.
IF DEFINED	Marks the beginning of a conditional-compilation block.
INLINE	Specifies that the routines can be inlined.
INTEGER	Selects default integer size.
IVDEP	Assists the compiler's dependence analysis of iterative DO loops.
LOOP COUNT	Specifies the typical trip loop count for a DO loop; this assists the optimizer.
MESSAGE	Sends a character string to the standard output device.
NOBLOCK_LOOP	Disables loop blocking for the immediately following nested DO loops.
NODECLARE	(Default) Turns off warning messages for undeclared variables.
NOFMA	Disables the generation of FMA instructions.
NOFREEFORM	(Default) Uses standard FORTRAN 77 code formatting column rules.
NOFUSION	Prevents a loop from fusing with adjacent loops.
NOINLINE	Specifies that a routine should not be inlined.
NOPARALLEL	Disables auto-parallelization for an immediately following DO loop.
NOOPTIMIZE	Disables optimizations for the program unit.
NOPREFETCH	Disables a data prefetch from memory.

Name	Description
NOSTRICT	(Default) Disables a previous STRICT directive.
NOUNROLL	Disables the unrolling of a DO loop.
NOUNROLL_AND_JAM	Disables loop unrolling and jamming.
NOVECTOR	Disables vectorization of a DO loop.
OBJCOMMENT	Specifies a library search path in an object file.
OPTIMIZE	Enables optimizations for the program unit.
OPTIONS	Controls whether fields in records and data items in common blocks are naturally aligned or packed on arbitrary byte boundaries.
PACK	Specifies the memory alignment of derived-type items.
PARALLEL	Helps auto-parallelization by assisting the compiler's dependence analysis of an immediately following DO loop.
PREFETCH	Hints to the compiler to prefetch data from memory.
PSECT	Modifies certain characteristics of a common block.
REAL	Selects default real size.
SIMD	Requires and controls SIMD vectorization of loops.
STRICT	Disables Intel® Fortran features not in the language standard specified on the command line.
UNDEFINE	Removes a symbolic variable name created with the DEFINE directive.
UNROLL	Tells the compiler's optimizer how many times to unroll a DO loop.
UNROLL_AND_JAM	Enables loop unrolling and jamming.
VECTOR	Overrides default heuristics for vectorization of DO loops.

OpenMP* Fortran Directives

OpenMP directives are specially formatted Fortran comment lines embedded in the source file that provide the compiler with hints and suggestions for parallelization, optimization, vectorization and offloading code to accelerator hardware. The compiler uses the information specified in the directives with compiler heuristic algorithms to generate more efficient code. At times, these heuristics may choose to ignore or override the information provided by a directive. If the directive is ignored by the compiler, no diagnostic message is issued.

To use the following directives, you must specify compiler option `[q or Q]openmp`. For more information, refer to the option description in the Compiler Options reference.

Name	Description
ATOMIC	Specifies that a specific memory location is to be updated atomically.
BARRIER	Synchronizes all the threads in a team.
CANCEL	Requests cancellation of the innermost enclosing region of the type specified, and causes the encountering implicit or explicit task to proceed to the end of the canceled construct.
CANCELLATION POINT	Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified.
CRITICAL	Restricts access for a block of code to only one thread at a time.
DECLARE REDUCTION	Declares a user defined reduction for one or more types.
DECLARE SIMD	Generates a SIMD procedure.
DECLARE TARGET	Specifies that named variables, common blocks, functions, and subroutines are mapped to a device.
DISTRIBUTE	Specifies that loop iterations will be executed by thread teams in the context of their implicit tasks.
DISTRIBUTE PARALLEL DO	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
DISTRIBUTE PARALLEL DO SIMD	Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
DISTRIBUTE SIMD	Specifies a loop that will be distributed across the master threads of the teams region. It will be executed concurrently using SIMD instructions.
DO	Specifies that the iterations of the immediately following DO loop must be executed in parallel.
DO SIMD	Specifies a loop that can be executed concurrently using SIMD instructions.
FLUSH	Specifies synchronization points where the implementation must have a consistent view of memory.
MASTER	Specifies a block of code to be executed by the master thread of the team.
ORDERED	Specifies a block of code to be executed sequentially.

Name	Description
PARALLEL	Defines a parallel region.
PARALLEL DO	Defines a parallel region that contains a single DO directive.
PARALLEL DO SIMD	Specifies a loop that can be executed concurrently using SIMD instructions. It provides a shortcut for specifying a PARALLEL construct containing one SIMD loop construct and no other statement.
PARALLEL SECTIONS	Defines a parallel region that contains SECTIONS directives.
PARALLEL WORKSHARE	Defines a parallel region that contains a single WORKSHARE directive.
SECTION	Appears within a SECTIONS directive construct to indicate a block (section) of code. It is optional for the first block of code within the SECTIONS directive construct.
SECTIONS	Specifies a block of code to be divided among threads in a team (a worksharing area).
SIMD	Requires and controls SIMD vectorization of loops.
SINGLE	Specifies a block of code to be executed by only one thread in a team.
TARGET	Creates a device data environment and executes the construct on the same device.
TARGET DATA	Creates a device data environment for the extent of the region.
TARGET ENTER DATA	Specifies that variables are mapped to a device data environment.
TARGET EXIT DATA	Specifies that variables are unmapped from a device data environment.
TARGET PARALLEL	Creates a device data environment in a parallel region and executes the construct on that device.
TARGET PARALLEL DO	Provides an abbreviated way to specify a TARGET directive containing a PARALLEL DO directive and no other statements.
TARGET PARALLEL DO SIMD	Specifies a TARGET construct that contains a PARALLEL DO SIMD construct and no other statement.
TARGET SIMD	Specifies a TARGET construct that contains a SIMD construct and no other statement.

Name	Description
TARGET TEAMS	Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the master thread in each team executing the structured block.
TARGET TEAMS DISTRIBUTE	Creates a device data environment and executes the construct on the same device. It also specifies that loop iterations will be shared among the master threads of all thread teams in a league created by a TEAMS construct.
TARGET TEAMS DISTRIBUTE PARALLEL DO	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct.
TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.
TARGET TEAMS DISTRIBUTE SIMD	Creates a device data environment and executes the construct on the same device. It also specifies that loop iterations will be shared among the master threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
TARGET UPDATE	Makes the list items in the device data environment consistent with their corresponding original list items.
TASK	Defines a task region.
TASKGROUP	Specifies a wait for the completion of all child tasks of the current task and all of their descendant tasks.
TASKLOOP	Specifies that the iterations of one or more associated DO loops should be executed in parallel using OpenMP* tasks. The iterations are distributed across tasks that are created by the construct and scheduled to be executed.
TASKLOOP SIMD	Specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using OpenMP* tasks.
TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task.

Name	Description
TASKYIELD	Specifies that the current task can be suspended in favor of execution of a different task.
TEAMS DISTRIBUTE	Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies that loop iterations will be shared among the master threads of all thread teams in a league created by a TEAMS construct.
TEAMS DISTRIBUTE PARALLEL DO	Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
TEAMS DISTRIBUTE PARALLEL DO SIMD	Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams. The loop will be distributed across the master threads of the teams region, which will be executed concurrently using SIMD instructions.
TEAMS DISTRIBUTE SIMD	Creates a league of thread teams to execute the structured block in the master thread of each team. It also specifies a loop that will be distributed across the master threads of the teams region. The loop will be executed concurrently using SIMD instructions.
THREADPRIVATE	Makes named common blocks private to a thread but global within the thread.
WORKSHARE	Divides the work of executing a block of statements or constructs into separate units.

See Also

qopenmp, Qopenmp compiler option

Program Control Statements

The following table lists statements that affect program control.

Statements

Name	Description
CALL	Transfers control to a subroutine.
CASE	Within a SELECT CASE construct, marks a block of statements that are executed if an associated value matches the SELECT CASE expression.
CASE DEFAULT or CLASS DEFAULT	Identifies the block of statements to be executed in a SELECT CASE construct if the value of the expression does not match any of the CASE selectors, or in a SELECT TYPE construct if the dynamic type of the selector does not match the type of any of the TYPE IS or CLASS IS statements.
CLASS IS	Within a SELECT TYPE construct, marks a block of statements that are executed if the type or dynamic type of an expression matches the type of the CLASS IS expression.
CONTINUE	Often used as the target of GOTO or as the terminal statement in a DO loop; performs no operation.
CYCLE	Advances control to the end statement of a DO loop; the intervening loop statements are not executed.
DO	Marks the beginning of a loop construct. Statements through and including the ending statement may be executed repeatedly.
DO CONCURRENT	Marks the beginning of a DO CONCURRENT construct. The order of executions of iterations of a DO CONCURRENT construct are indeterminate.
DO WHILE	Evaluates statements in the DO WHILE loop, through and including the ending statement, until a logical condition becomes .FALSE..
ELSE	Marks an optional branch in an IF construct.
ELSE IF	Marks an optional branch in an IF construct.
ELSEWHERE	Marks an optional branch in a WHERE construct.
END	Marks the end of a program unit. Execution of the END [PROGRAM] statement of the main program initiates normal termination for the image that executes it.
END DO	Marks the end of a series of statements in a DO, DO CONCURRENT, or DO WHILE construct.
END FORALL	Marks the end of a series of statements following a block FORALL statement.
END IF	Marks the end of a series of statements following a block IF statement.
END SELECT	Marks the end of a SELECT CASE or SELECT TYPE statement.

Name	Description
END WHERE	Marks the end of a series of statements following a block WHERE statement.
ERROR STOP	Initiates error termination for all images.
EXIT	Terminates execution of a DO loop or other construct. Execution continues with the first statement that follows the construct.
FAIL_IMAGE	Causes the image that executes it to execute no more statements and stop participating in program execution.
FORALL	Indicates a loop construct where the output from one iteration cannot change the input to another.
GOTO	Transfers control to a specified part of the program.
IF	Controls conditional execution of other statements.
PAUSE	Suspends program execution and, optionally, executes operating-system commands.
SELECT CASE	Transfers program control to a block of statements, determined by the value of an expression.
SELECT TYPE	Transfers the program to a block of statements, determined by the dynamic type of an expression.
STOP	Initiates normal termination for the image that executes the statement.
TYPE IS	Within a SELECT TYPE construct, marks a block of statements that are executed if the type or dynamic type of an expression matches the type of the TYPE IS expression.
WHERE	Controls conditional execution of array assignments and elemental function calls.

Inquiry Intrinsic Functions

The following table lists inquiry intrinsic functions.

Name	Description
ALLOCATED	Determines whether an allocatable variable is allocated.
ASSOCIATED	Determines if a pointer is associated or if two pointers are associated with the same target.
BIT_SIZE	Returns the number of bits in an integer type.
CACHESIZE	Returns the size of a level of the memory cache.
COMMAND_ARGUMENT_COUNT	Returns the number of command-line arguments.
DIGITS	Returns number of significant digits for data of the same type as the argument.

Name	Description
EOF	Determines whether a file is at or beyond the end-of-file record.
EPSILON	Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as the argument.
HUGE	Returns the largest number that can be represented by numbers of the type of the argument.
IARGC	Returns the index of the last command-line argument.
INT_PTR_KIND	Returns the INTEGER KIND that will hold an address.
KIND	Returns the value of the kind parameter of the argument.
LBOUND	Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.
LEN	Returns the length of a character expression.
LOC	Returns the address of the argument.
MAXEXPONENT	Returns the largest positive decimal exponent for data of the same type as the argument.
MINEXPONENT	Returns the largest negative decimal exponent for data of the same type as the argument.
NARGS	Returns the total number of command-line arguments, including the command.
PRECISION	Returns the number of significant digits for data of the same type as the argument.
PRESENT	Determines whether an optional argument is present.
RADIX	Returns the base for data of the same type as the argument.
RANGE	Returns the decimal exponent range for data of the same type as the argument.
SELECTED_INT_KIND	Returns the value of the kind parameter of integers in range *r*.
SELECTED_REAL_KIND	Returns the value of the kind parameter of reals with (optional) first argument digits and (optional) second argument exponent range. At least one optional argument is required.
SHAPE	Returns the shape of an array or scalar argument.
SIZEOF	Returns the number of bytes of storage used by the argument.
TINY	Returns the smallest positive number that can be represented by numbers of type the argument.

Name	Description
UBOUND	Returns the upper bounds for all dimensions of an array, or the upper bound for a specified dimension.

Random Number Intrinsic Procedures

The following table lists random number intrinsic procedures.

Name	Procedure Type	Description
RAN	Intrinsic function	Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1.
RANF	Intrinsic function	Generates a random number between 0.0 and RAND_MAX.
RANDOM_NUMBER	Intrinsic subroutine	Returns a pseudorandom real value greater than or equal to zero and less than one.
RANDOM_SEED	Intrinsic subroutine	Changes the starting point of RANDOM_NUMBER; takes one or no arguments.
RANDU	Intrinsic subroutine	Computes a pseudorandom number as a single-precision value.

The portability routines RANF, RANDOM, and SEED also supply this functionality.

Atomic Intrinsic Subroutines

The following table lists atomic intrinsic subroutines.

Atomic Operation

Name	Procedure Type	Description
ATOMIC_ADD	Intrinsic Subroutine	Performs atomic addition.
ATOMIC_AND	Intrinsic Subroutine	Performs atomic bitwise AND.
ATOMIC_CAS	Intrinsic Subroutine	Performs atomic compare and swap.
ATOMIC_DEFINE	Intrinsic Subroutine	Performs atomically defines a variable.
ATOMIC_FETCH_ADD	Intrinsic Subroutine	Performs atomic fetch and add.
ATOMIC_FETCH_AND	Intrinsic Subroutine	Performs atomic fetch and bitwise AND.
ATOMIC_FETCH_OR	Intrinsic Subroutine	Performs atomic fetch and bitwise OR.

Name	Procedure Type	Description
ATOMIC_FETCH_XOR	Intrinsic Subroutine	Performs atomic fetch and bitwise exclusive OR.
ATOMIC_OR	Intrinsic Subroutine	Performs atomic bitwise OR.
ATOMIC_REF	Intrinsic Subroutine	Performs atomically references a variable.
ATOMIC_XOR	Intrinsic Subroutine	Performs atomic bitwise exclusive OR.

Collective Intrinsic Subroutines

The following table lists collective intrinsic subroutines.

Collective Operation

Name	Procedure Type	Description
CO_BROADCAST	Intrinsic Subroutine	Broadcasts a value to other images.
CO_MAX	Intrinsic Subroutine	Finds maximum value across images.
CO_MIN	Intrinsic Subroutine	Finds minimum value across images.
CO_REDUCE	Intrinsic Subroutine	Performs user-defined reduction across images.
CO_SUM	Intrinsic Subroutine	Performs sum reduction across images.

Date and Time Intrinsic Subroutines

The following table lists date and time intrinsic subroutines.

Name	Procedure Type	Description
CPU_TIME	Intrinsic subroutine	Returns the processor time in seconds.
DATE	Intrinsic subroutine	Returns the ASCII representation of the current date (in dd-mmm-yy form).
DATE_AND_TIME	Intrinsic subroutine	Returns the date and time. This is the preferred procedure for date and time.
IDATE	Intrinsic subroutine	Returns three integer values representing the current month, day, and year.
SYSTEM_CLOCK	Intrinsic subroutine	Returns data from the system clock.

Name	Procedure Type	Description
TIME	Intrinsic subroutine	Returns the ASCII representation of the current time (in hh:mm:ss form).

The portability routines GETDAT, GETTIM, SETDAT, and SETTIM also supply this functionality.

Keyboard and Speaker Library Routines

The following table lists keyboard and speaker library routines.

Name	Routine Type	Description
GETCHARQQ	Run-time Function	Returns the next keyboard keystroke.
BEEPQQ	Portability subroutine	Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.
GETSTRQQ	Run-time function	Reads a character string from the keyboard using buffered input.
PEEKCHARQQ	Run-time function	Checks the buffer to see if a keystroke is waiting.

Statements and Intrinsic Procedures for Memory Allocation and Deallocation

The following table lists statements and intrinsic procedures that are used for memory allocation and deallocation.

Name	Procedure Type	Description
ALLOCATE	Statement	Dynamically establishes allocatable array dimensions.
ALLOCATED	Intrinsic Function	Determines whether an allocatable array is allocated.
DEALLOCATE	Statement	Frees the storage space previously reserved in an ALLOCATE statement.
FREE	Intrinsic Subroutine	Frees the memory block specified by the integer pointer argument.
MALLOC	Intrinsic Function	Allocates a memory block of size bytes and returns an integer pointer to the block.
MOVE_ALLOC	Intrinsic Subroutine	Moves an allocation from one allocatable object to another.

Intrinsic Functions for Arrays

The following table lists intrinsic functions for arrays.

Name	Description
ALL	Determines whether all array values meet the conditions in a mask along a (optional) dimension.
ANY	Determines whether any array values meet the conditions in a mask along a (optional) dimension.
COSHAPE	Returns the sizes of codimensions of a coarray.
COUNT	Counts the number of array elements that meet the conditions in a mask along a (optional) dimension.
CSHIFT	Performs a circular shift along a (optional) dimension.
DIMENSION	Specifies that an entity is an array and specifies its rank and bounds.
DOT_PRODUCT	Performs dot-product multiplication on vectors (one-dimensional arrays).
EOSHIFT	Shifts elements off one end of *array* along a (optional) dimension and copies (optional) boundary values in other end.
LBOUND	Returns lower dimensional bounds of an array along a (optional) dimension.
LCOBOUND	Returns the lower cobounds of a coarray.
MATMUL	Performs matrix multiplication on matrices (two-dimensional arrays).
MAXLOC	Returns the location of the maximum value in an array meeting conditions in a (optional) mask along a (optional) dimension.
MAXVAL	Returns the maximum value in an array along a (optional) dimension that meets conditions in a (optional) mask.
MERGE	Merges two arrays according to conditions in a mask.
MINLOC	Returns the location of the minimum value in an array meeting conditions in a (optional) mask along a (optional) dimension.
MINVAL	Returns the minimum value in an array along a (optional) dimension that meets conditions in a (optional) mask.
PACK	Packs an array into a vector (one-dimensional array) of a (optional) size using a mask.
PRODUCT	Returns product of elements of an array along a (optional) dimension that meet conditions in a (optional) mask.
RESHAPE	Reshapes an array with (optional) subscript order, padded with (optional) array elements.
SHAPE	Returns the shape of an array.

Name	Description
SIZE	Returns the extent of an array along a (optional) dimension.
SPREAD	Replicates an array by adding a dimension.
SUM	Sums array elements along a (optional) dimension that meet conditions of an (optional) mask.
TRANSPOSE	Transposes a two-dimensional array.
UBOUND	Returns upper dimensional bounds of an array along a (optional) dimension.
UCOBOUND	Returns the upper cobounds of a coarray.
UNPACK	Unpacks a vector (one-dimensional array) into an array under a mask padding with values from a field.

Intrinsic Functions for Numeric and Type Conversion

The following table lists intrinsic functions for numeric and type conversion.

Name	Description
ABS	Returns the absolute value of the argument.
AIMAG	Returns imaginary part of complex number *z*.
AINT	Truncates the argument to a whole number of a specified (optional) kind.
AMAX0	Returns largest value among integer arguments as real.
AMIN0	Returns smallest value among integer arguments as real.
ANINT	Rounds to the nearest whole number of a specified (optional) kind.
CEILING	Returns smallest integer greater than the argument.
CMPLX	Converts the first argument and (optional) second argument to complex of a (optional) kind.
CONJG	Returns the conjugate of a complex number.
DBLE	Converts the argument to double precision type.
DCMPLX	Converts the argument to double complex type.
DFLOAT	Converts an integer to double precision type.
DIM	Returns the first argument minus the second argument if positive; else 0.
DPROD	Returns double-precision product of two single precision arguments.

Name	Description
FLOAT	Converts the argument to REAL(4).
FLOOR	Returns the greatest integer less than or equal to the argument.
IFIX	Converts a single-precision real argument to an integer argument by truncating.
IMAG	Same as AIMAG.
INT	Converts a value to integer type.
LOGICAL	Converts between logical arguments of (optional) kind.
MAX	Returns largest value among arguments.
MAX1	Returns largest value among real arguments as integer.
MIN	Returns smallest value among arguments.
MIN1	Returns smallest value among real arguments as integer
MOD	Returns the remainder of the first argument divided by the second argument.
MODULO	Returns the first argument modulo the second argument.
NINT	Returns the nearest integer to the argument.
REAL	Converts a value to real type.
SIGN	Returns absolute value of the first argument times the sign of the second argument.
SNGL	Converts a double-precision argument to single-precision real type.
TRANSFER	Transforms first argument into type of second argument with (optional) size if an array.
ZEXT	Extends the argument with zeros.

Trigonometric, Exponential, Root, and Logarithmic Intrinsic Procedures

The following table lists intrinsic procedures for trigonometric, exponential, root, and logarithmic operations.

NOTE
Many routines in the LIBM library (Math Library) are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Name	Description
ACOS	Returns the arccosine of the argument, expressed in radians between 0 and pi.

Name	Description
ACOSD	Returns the arccosine of the argument, expressed in degrees between 0 and 180.
ALOG	Returns natural log of the argument.
ALOG10	Returns common log (base 10) of the argument.
ASIN	Returns the arcsine of the argument, expressed in radians between ±pi/2.
ASIND	Returns the arcsine of the argument, expressed in degrees between ±90°.
ATAN	Returns the arctangent of the argument, expressed in radians between ±pi/2.
ATAND	Returns the arctangent of the argument, expressed in degrees between ±90°.
ATAN2	Returns the arctangent of the first argument divided by the second argument, expressed in radians between ±pi.
ATAN2D	Returns the arctangent of the first argument divided by the second argument, expressed in degrees between ±180°.
CCOS	Returns complex cosine of the argument.
CDCOS	Returns the double-precision complex cosine of the argument.
CDEXP	Returns double-precision complex exponential value of the argument.
CDLOG	Returns the double-precision complex natural log of the argument.
CDSIN	Returns the double-precision complex sine of the argument.
CDSQRT	Returns the double-precision complex square root of the argument.
CEXP	Returns the complex exponential value of the argument.
CLOG	Returns the complex natural log of the argument.
COS	Returns the cosine of the argument, which is in radians.
COSD	Returns the cosine of the argument, which is in degrees.
COSH	Returns the hyperbolic cosine of the argument.
COTAN	Returns the cotangent of the argument, which is in radians.
COTAND	Returns the cotangent of the argument, which is in degrees.

Name	Description
CSIN	Returns the complex sine of the argument.
CSQRT	Returns the complex square root of the argument.
DACOS	Returns the double-precision arccosine of the argument radians between 0 and pi.
DACOSD	Returns the arccosine of the argument in degrees between 0 and 180.
DASIN	Returns the double-precision arcsine of the argument in radians between ±pi/2.
DASIND	Returns the double-precision arcsine of the argument degrees between ±90°.
DATAN	Returns the double-precision arctangent of the argument radians between ±pi/2.
DATAND	Returns the double-precision arctangent of the argument degrees between ±90°.
DATAN2	Returns the double-precision arctangent of the first argument divided by the second argument, expressed in radians between ±pi.
DATAN2D	Returns the double-precision arctangent of the first argument divided by the second argument, expressed in degrees between ±180°.
DCOS	Returns the double-precision cosine of the argument, which is in radians.
DCOSD	Returns the double-precision cosine of the argument, which is in degrees.
DCOSH	Returns the double-precision hyperbolic cosine of the argument.
DCOTAN	Returns the double-precision cotangent of the argument.
DEXP	Returns the double-precision exponential value of the argument.
DLOG	Returns the double-precision natural log of the argument.
DLOG10	Returns the double-precision common log (base 10) of the argument.
DSIN	Returns the double-precision sin of the argument, whch is in radians.
DSIND	Returns the double-precision sin of the argument, which is in degrees.
DSINH	Returns the double-precision hyperbolic sine of the argument.
DSQRT	Returns the double-precision square root of the argument.

Name	Description
DTAN	Returns the double-precision tangent of the argument, which is in radians.
DTAND	Returns the double-precision tangent of the argument, which is in degrees.
DTANH	Returns the double-precision hyperbolic tangent of the argument.
EXP	Returns the exponential value of the argument.
EXP10	Returns the base 10 exponential value of the argument.
LOG	Returns the natural log of the argument.
LOG10	Returns the common log (base 10) of the argument.
SIN	Returns the sine of the argument, which is in radians.
SIND	Returns the sine of the argument, which is in degrees.
SINH	Returns the hyperbolic sine of the argument.
SQRT	Returns the square root of the argument.
TAN	Returns the tangent of the argument, which is in radians.
TAND	Returns the tangent of the argument, which is in degrees.
TANH	Returns the hyperbolic tangent of the argument.

Intrinsic Functions for Floating-Point Inquiry and Control

The following table lists intrinsic functions for floating-point inquiry and control.

Certain functions (EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, and SPACING) return values related to components of the model set of real numbers. For a description of this model, see the Model for Real Data.

Name	Description
DIGITS	Returns number of significant digits for data of the same type as the argument.
EPSILON	Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as the argument.
EXPONENT	Returns the exponent part of the representation of *x*.
FRACTION	Returns the fractional part of the representation of the argument.
HUGE	Returns largest number that can be represented by data of type the argument.

Name	Description
MAXEXPONENT	Returns the largest positive decimal exponent for data of the same type as the argument.
MINEXPONENT	Returns the largest negative decimal exponent for data of the same type as the argument.
NEAREST	Returns the nearest different machine representable number to the first argument in the direction of the sign of the second argument.
PRECISION	Returns the number of significant digits for data of the same type as the argument.
RADIX	Returns the base for data of the same type as the argument.
RANGE	Returns the decimal exponent range for data of the same type as the argument.
RRSPACING	Returns the reciprocal of the relative spacing of numbers near the argument.
SCALE	Multiplies the first argument by 2 raised to the power of the second argument.
SET_EXPONENT	Returns a number whose fractional part is the first argument and whose exponential part is the second argument.
SPACING	Returns the absolute spacing of numbers near the argument.
TINY	Returns smallest positive number that can be represented by data of type of the argument.

The portability routines GETCONTROLFPQQ, GETSTATUSFPQQ, LCWRQQ, SCWRQQ, SETCONTROLFPQQ, and SSWRQQ also supply this functionality.

Character Intrinsic Functions

The following table lists character intrinsic functions.

Name	Description
ACHAR	Returns character in a specified position in the ASCII character set.
ADJUSTL	Adjusts left, removing leading blanks and inserting trailing blanks.
ADJUSTR	Adjusts right, removing trailing blanks and inserting leading blanks.
CHAR	Returns character in a specified position in the processor's character set of (optional) kind.
IACHAR	Returns the position of the argument in the ASCII character set.

Name	Description
ICHAR	Returns the position of the argument in the processor's character set.
INDEX	Returns the starting position of a substring in a string, leftmost or (optional) rightmost occurance.
LEN	Returns the size of the argument.
LEN_TRIM	Returns the number of characters in the argument, not counting trailing blanks.
LGE	Tests whether the the first argument is greater than or equal to the second argument, based on the ASCII collating sequence.
LGT	Tests whether the first argument is greater than the second argument, based on the ASCII collating sequence.
LLE	Tests whether the first argument is less than or equal to the second argument, based on the ASCII collating sequence.
LLT	Tests whether the first argument is less than the second argument, based on the ASCII collating sequence.
REPEAT	Concatenates multiple copies of a string.
SCAN	Scans a string for any characters in a set and returns leftmost or (optional) rightmost position where a match is found.
TRIM	Removes trailing blanks from a string.
VERIFY	Returns the position of the leftmost or (optional) rightmost character in the argument string not in a set, or zero if all characters in the set are present.

Intrinsic Procedures for Bit Operation and Representation

The following tables list intrinsic procedures for bit operation and representation.

Bit Operation

Name	Procedure Type	Description
BIT_SIZE	Intrinsic Function	Returns the number of bits in integers of the type the argument.
BTEST	Intrinsic Function	Tests a bit in a position of the argument; true if bit is 1.
IAND	Intrinsic Function	Performs a logical AND.
IBCHNG	Intrinsic Function	Reverses value of bit in a position of the argument.

Name	Procedure Type	Description
IBCLR	Intrinsic Function	Clears the bit in a position of the argument to zero.
IBITS	Intrinsic Function	Extracts a sequence of bits of length from the argument starting in a position.
IBSET	Intrinsic Function	Sets the bit in a position of the argument to one.
IEOR	Intrinsic Function	Performs an exclusive OR.
IOR	Intrinsic Function	Performs an inclusive OR.
ISHA	Intrinsic Function	Shifts the argument arithmetically left or right by shift bits; left if shift positive, right if shift negative. Zeros shifted in from the right, ones shifted in from the left.
ISHC	Intrinsic Function	Performs a circular shift of the argument left or right by shift bits; left if shift positive, right if shift negative. No bits lost.
ISHFT	Intrinsic Function	Shifts the argument logically left or right by shift bits; left if shift positive, right if shift negative. Zeros shifted in from opposite end.
ISHFTC	Intrinsic Function	Performs a circular shift of the rightmost bits of (optional) size by shift bits. No bits lost.
ISHL	Intrinsic Function	Shifts the argument logically left or right by shift bits. Zeros shifted in from opposite end.
MVBITS	Intrinsic Subroutine	Copies a sequence of bits from one integer to another.
NOT	Intrinsic Function	Performs a logical complement.

Bit Representation

Name	Procedure Type	Description
LEADZ	Intrinsic Function	Returns leading zero bits in an integer.
POPCNT	Intrinsic Function	Returns number of 1 bits in an integer.
POPPAR	Intrinsic Function	Returns the parity of an integer.
TRAILZ	Intrinsic Function	Returns trailing zero bits in an integer.

QuickWin Library Routines (W*S)

The following table lists Quickwin library routines.

Programs that use these routines must access the appropriate library with USE IFQWIN. These routines are restricted to Windows* systems.

Name	Routine Type	Description
ABOUTBOXQQ	Function	Adds an About Box with customized text.
APPENDMENUQQ	Function	Appends a menu item.
CLICKMENUQQ	Function	Sends menu click messages to the application window.
DELETEMENUQQ	Function	Deletes a menu item.
FOCUSQQ	Function	Makes a child window active, and gives focus to the child window.
GETACTIVEQQ	Function	Gets the unit number of the active child window.
GETEXITQQ	Function	Gets the setting for a QuickWin application's exit behavior.
GETHWNDQQ	Function	Gets the true windows handle from window with the specified unit number.
GETWINDOWCONFIG	Function	Returns the current window's properties.
GETWSIZEQQ	Function	Gets the size of the child or frame window.
GETUNITQQ	Function	Gets the unit number corresponding to the specified windows handle. Inverse of GETHWNDQQ.
INCHARQQ	Function	Reads a keyboard input and return its ASCII value.

Name	Routine Type	Description
INITIALSETTINGS	Function	Controls initial menu settings and initial frame window settings.
INQFOCUSQQ	Function	Determines which window is active and has the focus.
INSERTMENUQQ	Function	Inserts a menu item.
INTEGERTORGB	Subroutine	Converts a true color value into its red, green and blue components.
MESSAGEBOXQQ	Function	Displays a message box.
MODIFYMENUFLAGSQQ	Function	Modifies a menu item state.
MODIFYMENUROUTINEQQ	Function	Modifies a menu item's callback routine.
MODIFYMENUSTRINGQQ	Function	Changes a menu item's text string.
PASSDIRKEYSQQ	Function	Determines the behavior of direction and page keys.
REGISTERMOUSEEVENT	Function	Registers the application-defined routines to be called on mouse events.
RGBTOINTEGER	Function	Converts a trio of red, green and blue values to a true color value for use with RGB functions and subroutines.
SETACTIVEQQ	Function	Makes the specified window the current active window without giving it the focus.
SETEXITQQ	Function	Sets a QuickWin application's exit behavior.
SETMESSAGEQQ	Subroutine	Changes any QuickWin message, including status bar messages, state messages and dialog box messages.
SETMOUSECURSOR	Function	Sets the mouse cursor for the window in focus.
SETWINDOWCONFIG	Function	Configures the current window's properties.
SETWINDOWMENUQQ	Function	Sets the Window menu to which current child window names will be appended.
SETWSIZEQQ	Function	Sets the size of the child or frame window.
UNREGISTERMOUSEEVENT	Function	Removes the callback routine registered by REGISTERMOUSEEVENT.

Name	Routine Type	Description
WAITONMOUSEEVENT	Function	Blocks return until a mouse event occurs.

Graphics Library Routines (W*S)

The following table lists library routines for graphics.

Programs that use these routines must access the appropriate library with USE IFQWIN. These routines are restricted to Windows* systems.

Name	Routine Type	Description
ARC, ARC_W	Functions	Draws an arc.
CLEARSCREEN	Subroutine	Clears the screen, viewport, or text window.
DISPLAYCURSOR	Function	Turns the cursor off and on.
ELLIPSE, ELLIPSE_W	Functions	Draws an ellipse or circle.
FLOODFILL, FLOODFILL_W	Functions	Fills an enclosed area of the screen with the current color index, using the current fill mask.
FLOODFILLRGB, FLOODFILLRGB_W	Functions	Fills an enclosed area of the screen with the current RGB color, using the current fill mask.
GETARCINFO	Function	Determines the end points of the most recently drawn arc or pie.
GETBKCOLOR	Function	Returns the current background color index.
GETBKCOLORRGB	Function	Returns the current background RGB color.
GETCOLOR	Function	Returns the current color index.
GETCOLORRGB	Function	Returns the current RGB color.
GETCURRENTPOSITION, GETCURRENTPOSITION_W	Subroutines	Returns the coordinates of the current graphics-output position.
GETFILLMASK	Subroutine	Returns the current fill mask.
GETFONTINFO	Function	Returns the current font characteristics.
GETGTEXTEXTENT	Function	Determines the width of the specified text in the current font.
GETGTEXTROTATION	Function	Get the current text rotation angle.
GETIMAGE, GETIMAGE_W	Subroutines	Stores a screen image in memory.
GETLINESTYLE	Function	Returns the current line style.

Name	Routine Type	Description
GETLINEWIDTHQQ	Function	Returns the current line width or the line width set by the last call to SETLINEWIDTHQQ.
GETPHYSCOORD	Subroutine	Converts viewport coordinates to physical coordinates.
GETPIXEL, GETPIXEL_W	Functions	Returns a pixel's color index.
GETPIXELRGB, GETPIXELRGB_W	Functions	Returns a pixel's RGB color.
GETPIXELS	Function	Returns the color indices of multiple pixels.
GETPIXELSRGB	Function	Returns the RGB colors of multiple pixels.
GETTEXTCOLOR	Function	Returns the current text color index.
GETTEXTCOLORRGB	Function	Returns the current text RGB color.
GETTEXTPOSITION	Subroutine	Returns the current text-output position.
GETTEXTWINDOW	Subroutine	Returns the boundaries of the current text window.
GETVIEWCOORD, GETVIEWCOORD_W	Subroutines	Converts physical or window coordinates to viewport coordinates.
GETWINDOWCOORD	Subroutine	Converts viewport coordinates to window coordinates.
GETWRITEMODE	Function	Returns the logical write mode for lines.
GRSTATUS	Function	Returns the status (success or failure) of the most recently called graphics routine.
IMAGESIZE, IMAGESIZE_W	Functions	Returns image size in bytes.
INITIALIZEFONTS	Function	Initializes the font library.
LINETO, LINETO_W	Functions	Draws a line from the current position to a specified point.
LINETOAR	Function	Draws a line between points in one array and corresponding points in another array.
LINETOAREX	Function	Similar to LINETOAR, but also lets you specify color and line style.
LOADIMAGE, LOADIMAGE_W	Functions	Reads a Windows bitmap file (.BMP) and displays it at the specified location.

Name	Routine Type	Description
MOVETO, MOVETO_W	Subroutines	Moves the current position to the specified point.
OUTGTEXT	Subroutine	Sends text in the current font to the screen at the current position.
OUTTEXT	Subroutine	Sends text to the screen at the current position.
PIE, PIE_W	Functions	Draws a pie slice.
POLYBEZIER, POLYBEZIER_W	Functions	Draws one or more Bezier curves.
POLYBEZIERTO, POLYBEZIERTO_W	Functions	Draws one or more Bezier curves.
POLYGON, POLYGON_W	Functions	Draws a polygon.
POLYLINEQQ	Function	Draws a line between successive points in an array.
PUTIMAGE, PUTIMAGE_W	Subroutines	Retrieves an image from memory and displays it.
RECTANGLE, RECTANGLE_W	Functions	Draws a rectangle.
REMAPALLPALETTERGB	Function	Remaps a set of RGB color values to indices recognized by the current video configuration.
REMAPPALETTERGB	Function	Remaps a single RGB color value to a color index.
SAVEIMAGE, SAVEIMAGE_W	Functions	Captures a screen image and saves it as a Windows bitmap file.
SCROLLTEXTWINDOW	Subroutine	Scrolls the contents of a text window.
SETBKCOLOR	Function	Sets the current background color.
SETBKCOLORRGB	Function	Sets the current background color to a direct color value rather than an index to a defined palette.
SETCLIPRGN	Subroutine	Limits graphics output to a part of the screen.
SETCOLOR	Function	Sets the current color to a new color index.
SETCOLORRGB	Function	Sets the current color to a direct color value rather than an index to a defined palette.
SETFILLMASK	Subroutine	Changes the current fill mask to a new pattern.

Name	Routine Type	Description
SETFONT	Function	Finds a single font matching the specified characteristics and assigns it to OUTGTEXT.
SETGTEXTROTATION	Subroutine	Sets the direction in which text is written to the specified angle.
SETLINESTYLE	Subroutine	Changes the current line style.
SETLINEWIDTHQQ	Subroutine	Sets the width of a solid line drawn using any of the supported graphics functions.
SETPIXEL, SETPIXEL_W	Functions	Sets color of a pixel at a specified location.
SETPIXELRGB, SETPIXELRGB_W	Functions	Sets RGB color of a pixel at a specified location.
SETPIXELS	Subroutine	Sets the color indices of multiple pixels.
SETPIXELSRGB	Subroutine	Sets the RGB color of multiple pixels.
SETTEXTCOLOR	Function	Sets the current text color to a new color index.
SETTEXTCOLORRGB	Function	Sets the current text color to a direct color value rather than an index to a defined palette.
SETTEXTCURSOR	Function	Sets the height and width of the text cursor for the window in focus.
SETTEXTPOSITION	Subroutine	Changes the current text position.
SETTEXTWINDOW	Subroutine	Sets the current text display window.
SETVIEWORG	Subroutine	Positions the viewport coordinate origin.
SETVIEWPORT	Subroutine	Defines the size and screen position of the viewport.
SETWINDOW	Function	Defines the window coordinate system.
SETWRITEMODE	Function	Changes the current logical write mode for lines.
WRAPON	Function	Turns line wrapping on or off.

Portability Library Routines

The following tables list library routines for portability.

If you have programs that use these routines, it is recommended that they access the portability library with USE IFPORT.

Some routines in this library can be called with different sets of arguments, and some can be used as a function or a subroutine. In these cases, the arguments and calling mechanism determine the meaning of the routine. The IFPORT module contains generic interface blocks that give procedure definitions for these routines.

Fortran contains intrinsic procedures for many of the portability functions. The portability routines are extensions to the Fortran 2008 standard. For portability and performance, you should write code using Fortran standard intrinsic procedures whenever possible.

Information Retrieval

Name	Procedure Type	Description
FOR_IFCORE_VERSION	Function	Returns the version of the Fortran run-time library (ifcore).
FOR_IFPORT_VERSION	Function	Returns the version of the Fortran portability library (ifport).
FSTAT	Function	Returns information about a logical file unit.
GETCWD	Function	Returns the pathname of the current working directory.
GETENV	Subroutine	Searches the environment for a given string and returns its value if found.
GETGID	Function	Returns the group ID of the user.
GETLOG	Subroutine	Returns the user's login name.
GETPID	Function	Returns the process ID of the process.
GETUID	Function	Returns the user ID of the user of the process.
HOSTNAM[1]	Function	Returns the name of the user's host.
ISATTY	Function	Checks whether a logical unit number is a terminal.
LSTAT	Function	Returns information about a named file. STAT is the preferred form of this function.
RENAME	Function	Renames a file.
STAT	Function	Returns information about a named file.
UNLINK	Function	Deletes the file given by path.

Process Control

Name	Procedure Type	Description
ABORT	Subroutine	Stops execution of the current process, clears I/O buffers, and writes a string to external unit 0.

Name	Procedure Type	Description
ALARM	Function	Executes an external subroutine after waiting a specified number of seconds.
KILL	Function	Sends a signal code to the process given by ID.
SIGNAL	Function	Changes the action for signal.
SLEEP	Subroutine	Suspends program execution for a specified number of seconds.
SYSTEM	Function	Executes a command in a separate shell.

Numeric Values and Conversion

Name	Procedure Type	Description
BESJ0, BESJ1,BESJN, BESY0,BESY1, BESYN	Functions	Return single-precision values of Bessel functions of the first and second kind of orders 1, 2, and n, respectively.
BIC, BIS	Subroutines	Perform bit level clear, set, and test for integers.
BIT	Function	Performs bit level clear, set, and test for integers.
CDFLOAT	Function	Converts a COMPLEX(4) argument to DOUBLE PRECISION type.
COMPLINT, COMPLREAL, COMPLLOG	Functions	Return a BIT-WISE complement or logical .NOT. of the argument.
CSMG	Function	Performs an effective BIT-WISE store under mask.
DBESJ0, DBESJ1,DBESJN, DBESY0,DBESY1, DBESYN	Functions	Return double-precision values of Bessel functions of the first and second kind of orders 1, 2, and n, respectively.
DFLOATI, DFLOATJ, DFLOATK	Functions	Convert an integer to double-precision real type.
DRAND, DRANDM	Functions	Return random numbers between 0.0 and 1.0.
DRANSET	Subroutine	Sets the seed for the random number generator.
IDFLOAT	Function	Converts an INTEGER(4) argument to double-precision real type.
IFLOATI, IFLOATJ	Functions	Convert an integer to single-precision real type.

Name	Procedure Type	Description
INMAX	Function	Returns the maximum positive value for an integer.
INTC	Function	Converts an INTEGER(4) argument to INTEGER(2) type.
IRAND, IRANDM	Functions	Return a positive integer in the range 0 through 2**31-1 or 2**15-1 if called without an argument.
IRANGET	Subroutine	Returns the current seed.
IRANSET	Subroutine	Sets the seed for the random number generator.
JABS	Function	Computes an absolute value.
LONG	Function	Converts an INTEGER(2) argument to INTEGER(4) type.
QRANSET	Subroutine	Sets the seed for a sequence of pseudo-random numbers.
RAND, RANDOM[2]	Functions	Return random values in the range 0 through 1.0.
RANF	Function	Generates a random number between 0.0 and RAND_MAX.
RANGET	Subroutine	Returns the current seed.
RANSET	Subroutine	Sets the seed for the random number generator.
SEED	Subroutine	Changes the starting point of RANDOM.
SHORT	Function	Converts an INTEGER(4) argument to INTEGER(2) type.
SRAND	Subroutine	Seeds the random number generator used with IRAND and RAND.

Input and Output

Name	Procedure Type	Description
ACCESS	Function	Checks a file for accessibility according to mode.
CHMOD	Function	Changes file attributes.
FGETC	Function	Reads a character from an external unit.
FLUSH	Function	Flushes the buffer for an external unit to its associated file.
FPUTC	Function	Writes a character to an external unit.

Name	Procedure Type	Description
FSEEK	Subroutine	Repositions a file on an external unit.
FTELL, FTELLI8	Function	Return the offset, in bytes, from the beginning of the file.
GETC	Function	Reads a character from unit 5.
GETPOS, GETPOSI8	Functions	Return the offset, in bytes, from the beginning of the file.
PUTC	Function	Writes a character to unit 6.

Date and Time

Name	Procedure Type	Description
CLOCK	Function	Returns current time in HH:MM:SS format using a 24-hour clock.
CLOCKX	Subroutine	Returns the processor clock to the nearest microsecond.
CTIME	Function	Converts system time to a 24-character ASCII string.
DATE[3]	Subroutine or Function	Returns the current system date.
DATE4	Subroutine	Returns the current system date.
DCLOCK	Function	Returns the elapsed time in seconds since the start of the current process.
DTIME	Function	Returns CPU time since later of (1) start of program, or (2) most recent call to DTIME.
ETIME	Function	Returns elapsed CPU time since the start of program execution.
FDATE	Subroutine or Function	Returns the current date and time as an ASCII string.
GETDAT	Subroutine	Returns the date.
GETTIM	Subroutine	Returns the time.
GMTIME	Subroutine	Returns Greenwich Mean Time as a 9-element integer array.
IDATE[3]	Subroutine	Returns the date either as one 3-element array or three scalar parameters (month, day, year).
IDATE4	Subroutine	Returns the date either as one 3-element array or three scalar parameters (month, day, year).
ITIME	Subroutine	Returns current time as a 3-element array (hour, minute, second).

Name	Procedure Type	Description
JDATE[3]	Function	Returns current date as an 8-character string with the Julian date.
JDATE4	Function	Returns current date as a 10-character string with the Julian date.
LTIME	Subroutine	Returns local time as a 9-element integer array.
RTC	Function	Returns number of seconds since 00:00:00 GMT, Jan 1, 1970.
SECNDS	Function	Returns number of seconds since midnight, less the value of its argument.
SETDAT	Function	Sets the date.
SETTIM	Function	Sets the time.
TIME	Subroutine or Function	As a subroutine, returns time formatted as HH:MM:SS; as a function, returns time in seconds since 00:00:00 GMT, Jan 1, 1970.
TIMEF	Function	Returns the number of seconds since the first time this function was called (or zero).

Error Handling

Name	Procedure Type	Description
GETLASTERROR	Function	Returns the last error set.
GETLASTERRORQQ	Function	Returns the last error set by a run-time function or subroutine.
IERRNO	Function	Returns the last code error.
SETERRORMODEQQ	Subroutine	Sets the mode for handling critical errors.

Program Control

Name	Procedure Type	Description
RAISEQQ	Function	Sends an interrupt to the executing program, simulating an interrupt from the operating system.
RUNQQ	Function	Calls another program and waits for it to execute.
SIGNALQQ	Function	Controls signal handling.
SLEEPQQ	Subroutine	Delays execution of the program for the specified time.

System, Drive, and Directory

Name	Procedure Type	Description
CHDIR	Function	Changes the current working directory.
CHANGEDIRQQ	Function	Makes the specified directory the current (default) directory.
CHANGEDRIVEQQ	Function	Makes the specified drive the current drive.
DELDIRQQ	Function	Deletes a specified directory.
GETDRIVEDIRQQ	Function	Returns the current drive and directory path.
GETDRIVESIZEQQ	Function	Gets the size of the specified drive.
GETDRIVESQQ	Function	Reports the drives available to the system.
GETENVQQ	Function	Gets a value from the current environment.
MAKEDIRQQ	Function	Makes a directory with the specified directory name.
SETENVQQ	Function	Adds a new environment variable, or sets the value of an existing one.
SYSTEMQQ	Function	Executes a command by passing a command string to the operating system's command interpretor.

Speaker

Name	Procedure Type	Description
BEEPQQ	Subroutine	Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.

File Management

Name	Procedure Type	Description
DELFILESQQ	Function	Deletes the specified files in a specified directory.
FINDFILEQQ	Function	Searches for a file in the directories specified in the PATH environment variable.
FULLPATHQQ	Function	Returns the full path for a specified file or directory.
GETFILEINFOQQ	Function	Returns information about files with names that match a request string.

Name	Procedure Type	Description
PACKTIMEQQ	Subroutine	Packs time values for use by SETFILETIMEQQ.
RENAMEFILEQQ	Function	Renames a file.
SETFILEACCESSQQ	Function	Sets file-access mode for the specified file.
SETFILETIMEQQ	Function	Sets modification time for a given file.
SPLITPATHQQ	Function	Breaks a full path into four components.
UNPACKTIMEQQ	Subroutine	Unpacks a file's packed time and date value into its component parts.

Arrays

Name	Procedure Type	Description
BSEARCHQQ	Function	Performs a binary search for a specified element on a sorted one-dimensional array of non-structure data types (derived types are not allowed).
SORTQQ	Subroutine	Sorts a one-dimensional array of non-structure data types (derived types are not allowed).

Floating-Point Inquiry and Control

Name	Procedure Type	Description
CLEARSTATUSFPQQ	Subroutine	Clears the exception flags in the floating-point processor status word.
GETCONTROLFPQQ	Subroutine	Returns the value of the floating-point processor control word.
GETSTATUSFPQQ	Subroutine	Returns the value of the floating-point processor status word.
LCWRQQ	Subroutine	Same as SETCONTROLFPQQ.
SCWRQQ	Subroutine	Same as GETCONTROLFPQQ.
SETCONTROLFPQQ	Subroutine	Sets the value of the floating-point processor control word.
SSWRQQ	Subroutine	Same as GETSTATUSFPQQ.

IEEE Functionality

Name	Procedure Type	Description
IEEE_FLAGS	Function	Sets, gets, or clears IEEE flags.
IEEE_HANDLER	Function	Establishes a handler for IEEE exceptions.

Serial Port I/O[4]

Name	Procedure Type	Description
SPORT_CANCEL_IO	Function	Cancels any I/O in progress to the specified port.
SPORT_CONNECT	Function	Establishes the connection to a serial port and defines certain usage parameters.
SPORT_CONNECT_EX	Function	Establishes the connection to a serial port, defines certain usage parameters, and defines the size of the internal buffer for data reception.
SPORT_GET_HANDLE	Function	Returns the Windows* handle associated with the communications port.
SPORT_GET_STATE	Function	Returns the baud rate, parity, data bits, and stop bit settings of the communications port.
SPORT_GET_STATE_EX	Function	Returns the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.
SPORT_GET_TIMEOUTS	Function	Returns the user selectable timeouts for the serial port.
SPORT_PEEK_DATA	Function	Returns information about the availability of input data.
SPORT_PEEK_LINE	Function	Returns information about the availability of input records.
SPORT_PURGE	Function	Executes a purge function on the specified port.
SPORT_READ_DATA	Function	Reads available data from the port specified.
SPORT_READ_LINE	Function	Reads a record from the port specified.
SPORT_RELEASE	Function	Releases a serial port that has previously been connected.
SPORT_SET_STATE	Function	Sets the baud rate, parity, data bits and stop bit settings of the communications port.
SPORT_SET_STATE_EX	Function	Sets the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.
SPORT_SET_TIMEOUTS	Function	Sets the user selectable timeouts for the serial port.
SPORT_SHOW_STATE	Function	Displays the state of a port.

Name	Procedure Type	Description
SPORT_SPECIAL_FUNC	Function	Executes a communications function on a specified port.
SPORT_WRITE_DATA	Function	Outputs data to a specified port.
SPORT_WRITE_LINE	Function	Outputs data to a specified port and follows it with a record terminator.

Miscellaneous

Name	Procedure Type	Description
LNBLNK	Function	Returns the index of the last non-blank character in a string.
QSORT	Subroutine	Returns a sorted version of a one-dimensional array of a specified number of elements of a named size.
RINDEX	Function	Returns the index of the last occurrence of a substring in a string.
SCANENV	Subroutine	Scans the environment for the value of an environment variable.
TTYNAM	Subroutine	Checks whether a logical unit is a terminal.

[1] This routine can also be specified as HOSTNM.

[2] There is also a RANDOM subroutine in the portability library.

[3] The two-digit year return value of DATE, IDATE, and JDATE may cause problems with the year 2000. Use the intrinsic subroutine DATE_AND_TIME instead.

[4] W*S

National Language Support Library Routines (W*S)

The following table lists library routines for National Language Support (NLS).

Programs that use these routines must access the NLS library with USE IFNLS. These routines are restricted to Windows* systems.

Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Name	Routine Type	Description
MBCharLen	Function	Returns the length of the first multibyte character in a string.
MBConvertMBToUnicode	Function	Converts a character string from a multibyte codepage to a Unicode string.
MBConvertUnicodeToMB	Function	Converts a Unicode string to a multibyte character string of the current codepage.

Name	Routine Type	Description
MBCurMax	Function	Returns the longest possible mutlibyte character for the current codepage.
MBINCHARQQ	Function	Same as INCHARQQ, but can read a single multibyte character at once.
MBINDEX	Function	Same as INDEX, except that multibyte characters can be included in its arguments.
MBJISToJMS	Function	Converts a Japan Industry Standard (JIS) character to a Kanji (Shift JIS or JMS) character.
MBJMSToJIS	Function	Converts a Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character.
MBLead	Function	Determines whether a given character is the first byte of a multibyte character.
MBLen	Function	Returns the number of multibyte characters in a string, including trailing spaces.
MBLen_Trim	Function	Returns the number of multibyte characters in a string, not including trailing spaces.
MBLGE, MBLGT, MBLLE, MBLLT,MBLEQ, MBLNE	Function	Same as LGE, LGT, LLE, and LLT, and the logical operators .EQ. and .NE., except that multibyte characters can be included in their arguments.
MBNext	Function	Returns the string position of the first byte of the multibyte character immediately after the given string position.
MBPrev	Function	Returns the string position of the first byte of the multibyte character immediately before the given string position.
MBSCAN	Function	Same as SCAN, except that multibyte characters can be included. in its arguments
MBStrLead	Function	Performs a context sensitive test to determine whether a given byte in a character string is a lead byte.
MBVERIFY	Function	Same as VERIFY, except that multibyte characters can be included in its arguments.

Name	Routine Type	Description
NLSEnumCodepages	Function	Returns an array of valid codepages for the current console.
NLSEnumLocales	Function	Returns an array of locales (langauge/country combinations) installed on the system.
NLSFormatCurrency	Function	Formats a currency number according to conventions of the current locale (language/country).
NLSFormatDate	Function	Formats a date according to conventions of the current locale (language/country).
NLSFormatNumber	Function	Formats a number according to conventions of the current locale (language/country).
NLSFormatTime	Function	Formats a time according to conventions of the current locale (language/country).
NLSGetEnvironmentCodepage	Function	Returns the current codepage for the system Window or console.
NLSGetLocale	Subroutine	Returns the current language, country, and/or codepage.
NLSGetLocaleInfo	Function	Returns information about the current locale.
NLSSetEnvironmentCodepage	Function	Sets the codepage for the console.
NLSSetLocale	Function	Sets the current language, country, and codepage.

POSIX* Library Procedures

The following table lists library procedures for POSIX*.

Programs that use POSIX procedures must access the appropriate libraries with USE IFPOSIX. The IPX *nnnn* routines are functions; the PXF *nnnn* routines are subroutines, except for the routines named PXFIS *nnnn* and PXFWIF *nnnn*.

Name	Description
IPXFARGC	Returns the index of the last commans-line argument.
IPXFCONST	Returns the value associated with a constant defined in the C POSIX standard.
IPXFLENTRIM	Returns the index of the last non-blank character in an input string.
IPXFWEXITSTATUS[1]	Returns the exit code of a child process.

Name	Description
IPXFWSTOPSIG[1]	Returns the number of the signal that caused a child process to stop.
IPXFWTERMSIG[1]	Returns the number of the signal that caused a child process to terminate.
PXF(type)GET	Gets the value stored in a component (or field) of a structure.
PXF(type)SET	Sets the value of a component (or field) of a structure.
PXFA(type)GET	Gets the array values stored in a component (or field) of a structure.
PXFA(type)SET	Sets the value of an array component (or field) of a structure.
PXFACCESS	Determines the accessibility of a file.
PXFALARM	Schedules an alarm.
PXFCALLSUBHANDLE	Calls the associated subroutine.
PXFCFGETISPEED[1]	Returns the input baud rate from a `termios` structure.
PXFCFGETOSPEED[1]	Returns the output baud rate from a `termios` structure.
PXFCFSETISPEED[1]	Sets the input baud rate in a `termios` structure.
PXFCFSETOSPEED[1]	Sets the output baud rate in a `termios` structure.
PXFCHDIR	Changes the current working directory.
PXFCHMOD	Changes the ownership mode of the file.
PXFCHOWN[1]	Changes the owner and group of a file.
PXFCLEARENV	Clears the process environment.
PXFCLOSE	Closes the file associated with the descriptor.
PXFCLOSEDIR	Closes the directory stream.
PXFCNTL[1]	Manipulates an open file descriptor.
PXFCONST	Returns the value associated with a constant.
PXFCREAT	Creates a new file or rewrites an existing file.
PXFCTERMID[1]	Generates a terminal pathname.
PXFDUP, PXFDUP2	Duplicates an existing file descriptor.
PXFE(type)GET	Gets the value stored in an array element component (or field) of a structure.
PXFE(type)SET	Sets the value of an array element component (or field) of a structure.
PXFEXECV, PXFEXECVE, PXFEXECVP	Executes a new process by passing command-line arguments.

Name	Description
PXFEXIT, PXFFASTEXIT	Exits from a process.
PXFFDOPEN	Opens an external unit.
PXFFFLUSH	Flushes a file directly to disk.
PXFFGETC	Reads a character from a file.
PXFFILENO	Returns the file descriptor associated with a specified unit.
PXFFORK[1]	Creates a child process that differs from the parent process only in its PID.
PXFFPATHCONF	Gets the value for a configuration option of an opened file.
PXFFPUTC	Writes a character to a file.
PXFFSEEK	Modifies a file position.
PXFFSTAT	Gets a file's status information.
PXFFTELL	Returns the relative position in bytes from the beginning of the file.
PXFGETARG	Gets the specified command-line argument.
PXFGETC	Reads a character from standard input unit 5.
PXFGETCWD	Returns the path of the current working directory.
PXFGETEGID[1]	Gets the effective group ID of the current process.
PXFGETENV	Gets the setting of an environment variable.
PXFGETEUID[1]	Gets the effective user ID of the current process.
PXFGETGID[1]	Gets the real group ID of the current process.
PXFGETGRGID[1]	Gets group information for the specified GID.
PXFGETGRNAM[1]	Gets group information for the named group.
PXFGETGROUPS[1]	Gets supplementary group IDs.
PXFGETLOGIN	Gets the name of the user.
PXFGETPGRP[1]	Gets the process group ID of the calling process.
PXFGETPID	Gets the process ID of the calling process.
PXFGETPPID	Gets the process ID of the parent of the calling process.
PXFGETPWNAM[1]	Gets password information for a specified name.
PXFGETPWUID[1]	Gets password information for a specified UID.
PXFGETSUBHANDLE	Returns a subroutine handle for a subroutine.
PXFGETUID[1]	Gets the real user ID of the current process.
PXFISATTY[1]	Tests whether a file descriptor is connected to a terminal.

Name	Description
PXFISBLK	Tests for a block special file.
PXFISCHR	Tests for a character file.
PXFISCONST	Tests whether a string is a valid constant name.
PXFISDIR	Tests whether a file is a directory.
PXFISFIFO	Tests whether a file is a special FIFO file.
PXFISREG	Tests whether a file is a regular file.
PXFKILL	Sends a signal to a specified process.
PXFLINK[1]	Creates a link to a file or directory.
PXFLOCALTIME	Converts a given elapsed time in seconds to local time.
PXFLSEEK	Positions a file a specified distance in bytes.
PXFMKDIR	Creates a new directory.
PXFMKFIFO[1]	Creates a new FIFO.
PXFOPEN	Opens or creates a file.
PXFOPENDIR	Opens a directory and associates a stream with it.
PXFPATHCONF	Gets the value for a configuration option of an opened file.
PXFPAUSE	Suspends process execution.
PXFPIPE[1]	Creates a communications pipe between two processes.
PXFPOSIXIO[1]	Sets the current value of the POSIX I/O flag.
PXFPUTC	Outputs a character to logical unit 6 (stdout).
PXFREAD	Reads from a file.
PXFREADDIR	Reads the current directory entry.
PXFRENAME	Changes the name of a file.
PXFREWINDDIR	Resets the position of the stream to the beginning of the directory.
PXFRMDIR	Removes a directory.
PXFSETENV	Adds a new environment variable or sets the value of an environment variable.
PXFSETGID[1]	Sets the effective group ID of the current process.
PXFSETPGID[1]	Sets the process group ID.
PXFSETSID[1]	Creates a session and sets the process group ID.
PXFSETUID[1]	Sets the effective user ID of the current process.
PXFSIGACTION[1]	Changes the action associated with a specific signal.

Name	Description
PXFSIGADDSET[1]	Adds a signal to a signal set.
PXFSIGDELSET[1]	Deletes a signal from a signal set.
PXFSIGEMPTYSET[1]	Empties a signal set.
PXFSIGFILLSET[1]	Fills a signal set.
PXFSIGISMEMBER	Tests whether a signal is a member of a signal set.
PXFSIGPENDING[1]	Examines pending signals.
PXFSIGPROCMASK[1]	Changes the list of currently blocked signals.
PXFSIGSUSPEND[1]	Suspends the process until a signal is received.
PXFSLEEP	Forces the process to sleep.
PXFSTAT	Gets a file's status information.
PXFSTRUCTCOPY	Copies the contents of one structure to another.
PXFSTRUCTCREATE	Creates an instance of the specified structure.
PXFSTRUCTFREE	Deletes the instance of a structure.
PXFSYSCONF	Gets values for system limits or options.
PXFTCDRAIN[1]	Waits until all output written has been transmitted.
PXFTCFLOW[1]	Suspends the transmission or reception of data.
PXFTCFLUSH[1]	Discards terminal input data, output data, or both.
PXFTCGETATTR[1]	Reads current terminal settings.
PXFTCGETPGRP[1]	Gets the foreground process group ID associated with the terminal.
PXFTCSENDBREAK[1]	Sends a break to the terminal.
PXFTCSETATTR[1]	Writes new terminal settings.
PXFTCSETPGRP[1]	Sets the foreground process group associated with the terminal.
PXFTIME	Gets the system time.
PXFTIMES	Gets process times.
PXFTTYNAM[1]	Gets the terminal pathname.
PXFUCOMPARE	Compares two unsigned integers.
PXFUMASK	Sets a new file creation mask and gets the previous one.
PXFUNAME	Gets the operation system name.
PXFUNLINK	Removes a directory entry.
PXFUTIME	Sets file access and modification times.
PXFWAIT[1]	Waits for a child process.

Name	Description
PXFWAITPID[1]	Waits for a specific PID.
PXFWIFEXITED[1]	Determines if a child process has exited.
PXFWIFSIGNALED[1]	Determines if a child process has exited because of a signal.
PXFWIFSTOPPED[1]	Determines if a child process has stopped.
PXFWRITE	Writes to a file.

[1] L*X, M*X

Dialog Library Routines (W*S)

The following table lists routines from the dialog library.

Programs that use these routines must access the Dialog library with USE IFLOGM. These routines are restricted to Windows* systems.

Name	Routine Type	Description
DLGEXIT	Subroutine	Closes an open dialog.
DLGFLUSH	Subroutine	Updates the display of a dialog box.
DLGGET	Function	Retrieves values of dialog control variables.
DLGGETCHAR	Function	Retrieves values of dialog control variables of type Character.
DLGGETINT	Function	Retrieves values of dialog control variables of type Integer.
DLGGETLOG	Function	Retrieves values of dialog control variables of type Logical.
DLGINIT	Function	Initializes a dialog.
DLGINITWITHRESOURCEHANDLE	Function	Initializes a dialog.
DLGISDLGMESSAGE	Function	Determines whether a message is intended for a modeless dialog box and, if it is, processes it.
DLGISDLGMESSAGEWITHDLG	Function	Determines whether a message is intended for a specific modeless dialog box and, if it is, processes it.
DLGMODAL	Function	Displays a dialog and processes dialog selections from user.
DLGMODALWITHPARENT	Function	Displays a dialog in a specific parent window and processes dialog selections from user.
DLGMODELESS	Function	Displays a modeless dialog box.

Name	Routine Type	Description
DLGSENDCTRLMESSAGE	Function	Sends a message to a dialog box control.
DLGSET	Function	Assigns values to dialog control variables.
DLGSETCHAR	Function	Assigns values to dialog control variables of type Character.
DLGSETCTRLEVENTHANDLER	Function	Assigns your own event handlers to ActiveX controls in a dialog box.
DLGSETINT	Function	Assigns values to dialog control variables of type Integer.
DLGSETLOG	Function	Assigns values to dialog control variables of type Logical.
DLGSETRETURN	Subroutine	Sets the return value for DLGMODAL.
DLGSETSUB	Function	Assigns procedures (callback routines) to dialog controls.
DLGSETTITLE	Subroutine	Sets the title of a dialog box.
DLGUNINIT	Subroutine	Deallocates memory occupied by an initialized dialog.

See Also

Additional Documentation: Creating Fortran Applications that Use Windows* OS Features

COM and Automation Library Routines (W*S)

The following tables list COM and Automation libary routines.

Programs that use COM routines must access the appropriate libraries with USE IFCOM. Programs that use automation routines must access the appropriate libraries with USE IFAUTO. Some procedures also require the USE IFWINTY module.

The COM and Automation routines are restricted to Windows* systems.

Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Component Object Model (COM) Procedures (USE IFCOM)

Name	Routine Type	Description
COMAddObjectReference	Function	Adds a reference to an object's interface.
COMCLSIDFromProgID[1]	Subroutine	Passes a programmatic identifier and returns the corresponding class identifier.
COMCLSIDFromString[1]	Subroutine	Passes a class identifier string and returns the corresponding class identifier.

Name	Routine Type	Description
COMCreateObject[1]	Subroutine	A generic routine that executes either COMCreateObjectByProgID or COMCreateObjectByGUID.
COMCreateObjectByGUID[1]	Subroutine	Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.
COMCreateObjectByProgID	Subroutine	Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.
COMGetActiveObjectByGUID[1]	Subroutine	Passes a class identifier and returns a pointer to the interface of a currently active object.
COMGetActiveObjectByProgID	Subroutine	Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.
COMGetFileObject	Subroutine	Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
COMInitialize	Subroutine	Initializes the COM library.
COMIsEqualGUID[1]	Function	Determines whether two GUIDs are the same.
COMQueryInterface[1]	Subroutine	Passes an interface identifier and returns a pointer to an object's interface.
COMReleaseObject	Function	Indicates that the program is done with a reference to an object's interface.
COMStringFromGUID[1]	Subroutine	Passes a GUID and returns a string of printable characters.
COMUninitialize	Subroutine	Uninitializes the COM library.

Automation Server Procedures (USE IFAUTO)

Name	Routine Type	Description
AUTOAddArg[1]	Subroutine	Passes an argument name and value and adds the argument to the argument list data structure.
AUTOAllocateInvokeArgs	Function	Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.
AUTODeallocateInvokeArgs	Subroutine	Deallocates an argument list data structure.

Name	Routine Type	Description
AUTOGetExceptInfo	Subroutine	Retrieves the exception information when a method has returned an exception status.
AUTOGetProperty[1]	Function	Passes the name or identifier of the property and gets the value of the Automation object's property.
AUTOGetPropertyByID	Function	Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.
AUTOGetPropertyInvokeArgs	Function	Passes an argument list data structure and gets the value of the Automation object's property specified in the argument list's first argument.
AUTOInvoke	Function	Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.
AUTOSetProperty[1]	Function	Passes the name or identifier of the property and a value, and sets the value of the Automation object's property.
AUTOSetPropertyByID	Function	Passes the member ID of the property and sets the value of the Automation object's property, using the argument list's first argument.
AUTOSetPropertyInvokeArgs	Function	Passes an argument list data structure and sets the value of the Automation object's property specified in the argument list's first argument.

[1]These routines also require USE IFWINTY.

Miscellaneous Run-Time Library Routines

The following table lists miscellaneous run-time library routines.

Programs that use most of these routines should contain a USE IFCORE statement to obtain the proper interfaces to these routines.

Name	Procedure Type	Description
COMMITQQ	Function	Forces the operating system to execute any pending write operations for a file.

Name	Procedure Type	Description
FOR_DESCRIPTOR_ASSIGN[1]	Subroutine	Creates an array descriptor in memory.
FOR_GET_FPE	Function	Returns the current settings of floating-point exception flags.
FOR_LFENCE	Subroutine	Inserts a memory load fence instruction that ensures completion of preceding load instructions.
FOR_MFENCE	Subroutine	Inserts a memory access fence instruction that ensures completion of preceding memory access instructions.
for_rtl_finish_[2]	Function	Cleans up the Fortran run-time environment.
for_rtl_init_[2]	Function	Initializes the Fortran run-time environment.
FOR_SET_FPE	Function	Sets the floating-point exception flags.
FOR__SET_FTN_ALLOC	Function	Tells the Fortran Run-Time Library (RTL) to use a user-defined routine to dynamically allocate commons.
FOR_SET_REENTRANCY	Function	Controls the type of reentrancy protection that the Fortran Run-Time Library (RTL) exhibits.
FOR_SFENCE	Subroutine	Inserts a memory store fence instruction that ensures completion of preceding store instructions.
GERROR	Subroutine	Returns a message for the last error detected by a Fortran run-time routine.
GETCHARQQ	Function	Returns the next keystroke.
GETEXCEPTIONPTRSQQ[1]	Function	Returns a pointer to C run-time exception information pointers appropriate for use in signal handlers established with SIGNALQQ or direct calls to the C rtl signal( ) routine.
GETSTRQQ	Function	Reads a character string from the keyboard using buffered input.
PEEKCHARQQ	Function	Checks the buffer to see if a keystroke is waiting.

Name	Procedure Type	Description
PERROR	Subroutine	Sends a message to the standard error stream, preceded by a specified string, for the last detected error.
TRACEBACKQQ	Subroutine	Provides traceback information.

[1] W*S

[2] You do not need a USE IFCORE statement for this routine.

A to B

A to B

ABORT

Portability Subroutine: *Flushes and closes I/O buffers, and terminates program execution.*

Module

USE IFPORT

Syntax

`CALL ABORT[string]`

string	(Input; optional) Character*(*). Allows you to specify an abort message at program termination. When ABORT is called, "abort:" is written to external unit 0, followed by *string*. If omitted, the default message written to external unit 0 is "abort: Fortran Abort Called."

This subroutine causes the program to terminate and an exit code value of 134 is returned to the program that launched it.

Example

```
USE IFPORT
!The following prints "abort: Fortran Abort Called"
CALL ABORT
!The following prints "abort: Out of here!"
Call ABORT ("Out of here!")
```

See Also

EXIT
STOP

ABOUTBOXQQ (W*S)

QuickWin Function: *Specifies the information displayed in the message box that appears when the user selects the About command from a QuickWin application's Help menu.*

Module

USE IFQWIN

Syntax

`result=ABOUTBOXQQ(cstring)`

cstring	(Input; output) Character*(*). Null-terminated C string.

Results

The value of the result is INTEGER(4). It is zero if successful; otherwise, nonzero.

If your program does not call ABOUTBOXQQ, the QuickWin run-time library supplies a default string.

Example

```
        USE IFQWIN
        INTEGER(4) dummy
! Set the About box message
        dummy = ABOUTBOXQQ ('Matrix Multiplier\r      Version 1.0'C)
```

ABS

Elemental Intrinsic Function (Generic): *Computes an absolute value.*

Syntax

`result=ABS(a)`

a	(Input) Must be of type integer, real, or complex.

Results

The result has the same type and kind type parameter as *a* except if *a* is complex value, the result type is real. If *a* is an integer or real value, the value of the result is | *a* |; if *a* is a complex value (X, Y), the result is the real value SQRT (X**2 + Y**2).

Specific Name	Argument Type	Result Type
BABS	INTEGER(1)	INTEGER(1)
IIABS[1]	INTEGER(2)	INTEGER(2)
IABS [2]	INTEGER(4)	INTEGER(4)
KIABS	INTEGER(8)	INTEGER(8)
ABS	REAL(4)	REAL(4)
DABS	REAL(8)	REAL(8)
QABS	REAL(16)	REAL(16)
CABS [3]	COMPLEX(4)	REAL(4)
CDABS[4]	COMPLEX(8)	REAL(8)
CQABS	COMPLEX(16)	REAL(16)

Specific Name	Argument Type	Result Type

[1]Or HABS.

[2]Or JIABS. For compatibility with older versions of Fortran, IABS can also be specified as a generic function, which allows integer arguments of any kind and produces a result of type default INTEGER.

[3]The setting of compiler options specifying real size can affect CABS, making CABS generic, which allows a complex argument of any kind to produce a result of default REAL.

[4]This function can also be specified as ZABS.

Example

ABS (-7.4) has the value 7.4.

ABS ((6.0, 8.0)) has the value 10.0.

The following ABS.F90 program calculates two square roots, retaining the sign:

```
        REAL mag(2), sgn(2), result(2)
        WRITE (*, '(A)') ' Enter two signed magnitudes: '
        READ (*, *) mag
        sgn = SIGN((/1.0, 1.0/), mag) ! transfer the signs to 1.0s
        result = SQRT (ABS (mag))
! Restore the sign by multiplying by -1 or +1:
        result = result * sgn
        WRITE (*, *) result
        END
```

ABSTRACT INTERFACE

Statement: *Defines an abstract interface.*

Syntax

```
ABSTRACT INTERFACE
   [interface-body]...
END INTERFACE
```

interface-body

Is one or more function or subroutine subprograms or a procedure pointer. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

The subprogram must *not* contain a statement function or a DATA, ENTRY, or FORMAT statement; an entry name can be used as a procedure name.

The subprogram can contain a USE statement.

Description

An abstract interface block defines an interface whose name can be used in a PROCEDURE declaration statement to declare subprograms with identical arguments and characteristics.

An abstract interface block cannot contain a PROCEDURE statement or a MODULE PROCEDURE statement.

Interface blocks can appear in the specification part of the program unit that invokes the external or dummy procedure.

An interface block must not appear in a block data program unit.

An interface block comprises its own scoping unit, and does not inherit anything from its host through host association.

The function or subroutine named in the interface-body cannot have the same name as a keyword that specifies an intrinsic type.

To make an interface block available to multiple program units (through a USE statement), place the interface block in a module.

Example

Previously, within an interface block, you needed to individually declare subroutines and functions that had the same argument keywords and characteristics. For example:

```
INTERFACE
SUBROUTINE SUB_ONE (X)
  REAL, INTENT(IN) :: X
END SUBROUTINE SUB_ONE
SUBROUTINE SUB_TWO (X)
  REAL, INTENT(IN) :: X
END SUBROUTINE SUB_TWO
...
END INTERFACE
```

Now you can use an abstract interface to specify a subprogram name for these identical arguments and characteristics. For example:

```
ABSTRACT INTERFACE
SUBROUTINE TEMPLATE (X)
  REAL, INTENT(IN) :: X
END SUBROUTINE TEMPLATE
END INTERFACE
```

You can then use the subprogram in the abstract interface as a template in a PROCEDURE statement to declare procedures. For example:

```
PROCEDURE (TEMPLATE) :: SUB_ONE, SUB_TWO, ...
```

See Also

INTERFACE
PROCEDURE
FUNCTION
SUBROUTINE
Procedure Interfaces
Use and Host Association

ACCEPT

Statement: *Transfers input data.*

Syntax

Formatted:

ACCEPT *form*[,*io-list*]

Formatted - List-Directed:

ACCEPT *[,*io-list*]

Formatted - Namelist:

ACCEPT *nml*

form	Is the nonkeyword form of a format specifier (no FMT=).
io-list	Is an I/O list.
*	Is the format specifier indicating list-directed formatting.
nml	Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

The ACCEPT statement is the same as a formatted, sequential READ statement, except that an ACCEPT statement must never be connected to user-specified I/O units. You can override this restriction by using environment variable FOR_ACCEPT.

Example

In the following example, character data is read from the implicit unit and binary values are assigned to each of the five elements of array CHARAR:

```
     CHARACTER*10 CHARAR(5)
     ACCEPT 200, CHARAR
200  FORMAT (5A10)
```

ACCESS Function

Portability Function: *Determines if a file exists and how it can be accessed.*

Module

USE IFPORT

Syntax

`result = ACCESS(name,mode)`

name	(Input) Character*(*). Name of the file whose accessibility is to be determined.
mode	(Input) Character*(*). Modes of accessibility to check for. Must be a character string of length one or greater containing only the characters "r", "w", "x", or "" (a blank). These characters are interpreted as follows.

Character	Meaning
r	Tests for read permission
w	Tests for write permission
x	Tests for execute permission. On Windows* systems, the extension of *name* must be .COM, .EXE, .BAT, .CMD, .PL, .KSH, or .CSH.
(blank)	Tests for existence

The characters within *mode* can appear in any order or combination. For example, wrx and r are legal forms of *mode* and represent the same set of inquiries.

Results

The value of the result is INTEGER(4). It is zero if all inquiries specified by *mode* are true. If either argument is invalid, or if the file cannot be accessed in all of the modes specified, one of the following error codes is returned:

- EACCES: Access denied; the file's permission setting does not allow the specified access
- EINVAL: The mode argument is invalid
- ENOENT: File or path not found

For a list of error codes, see IERRNO.

The *name* argument can contain either forward or backward slashes for path separators.

On Windows* systems, all files are readable. A test for read permission always returns 0.

Example

```
use ifport
! checks for read and write permission on the file "DATAFILE.TXT"
J = ACCESS ("DATAFILE.TXT", "rw")
PRINT *, J
! checks whether "DATAFILE.TXT" is executable. It is not, since
! it does not end in .COM, .EXE, .BAT, or .CMD
J = ACCESS ("DATAFILE.TXT","x")
PRINT *, J
```

See Also

INQUIRE
GETFILEINFOQQ

ACHAR

Elemental Intrinsic Function (Generic): *Returns the character in a specified position of the ASCII character set, even if the processor's default character set is different. It is the inverse of the IACHAR function. In Intel® Fortran, ACHAR is equivalent to the CHAR function.*

Syntax

`result = ACHAR (i [, kind])`

i	(Input) Is of type integer.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is character with length 1. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default character. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *i* has a value within the range 0 to 127, the result is the character in position *i* of the ASCII character set; otherwise, it is processor defined. ACHAR (IACHAR(C)) has the value C for any character C capable of representation in the default character set. For a complete list of ASCII character codes, see Character and Key Code Charts.

Example

ACHAR (71) has the value 'G'.

ACHAR (63) has the value '?'.

See Also

CHAR
IACHAR
ICHAR

ACOS

Elemental Intrinsic Function (Generic): *Produces the arccosine of x.*

Syntax

```
result = ACOS (x)
```

x	(Input) Must be of type real, where | *x* | must be less than or equal to 1, or of type complex.

Results

The result type and kind are the same as *x* and is expressed in radians.

If the result is real, the value is expressed in radians and lies in the range 0 <= ACOS(X) <= pi.

If the result is complex, the real part is expressed in radians and lies in the range 0 <= REAL(ACOS (x)) <= pi.

Specific Name	Argument Type	Result Type
ACOS [1]	REAL(4)	REAL(4)
DACOS [1]	REAL(8)	REAL(8)
QACOS [2]	REAL(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect ACOS and DACOS.

[2]Or QARCOS.

Example

ACOS (0.68032123) has the value .8225955.

ACOSD

Elemental Intrinsic Function (Generic): *Produces the arccosine of x.*

Syntax

```
result = ACOSD (x)
```

x	(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results

The result type and kind are the same as *x* and are expressed in degrees. The value lies in the range −90 to 90 degrees.

Specific Name	Argument Type	Result Type
ACOSD [1]	REAL(4)	REAL(4)
DACOSD [1]	REAL(8)	REAL(8)
QACOSD	REAL(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect ACOSD and DACOSD.

Example

ACOSD (0.886579) has the value 27.55354.

ACOSH

Elemental Intrinsic Function (Generic): *Produces the hyperbolic arccosine of x.*

Syntax

```
result = ACOSH (x)
```

x	(Input) Must be of type real, where *x* must be greater than or equal to 1, or of type complex.

Results

The result type and kind are the same as *x*.

If the result is complex, the real part is non-negative, and the imaginary part is expressed in radians and lies in the range -pi <= AIMAG (ACOSH (x)) <= pi.

Specific Name	Argument Type	Result Type
ACOSH [1]	REAL(4)	REAL(4)
DACOSH [1]	REAL(8)	REAL(8)
QACOSH	REAL(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect ACOSH and DACOSH.

Example

ACOSH (1.0) has the value 0.0.

ACOSH (180.0) has the value 5.8861.

ADJUSTL

Elemental Intrinsic Function (Generic): *Adjusts a character string to the left, removing leading blanks and inserting trailing blanks.*

Syntax

```
result = ADJUSTL (string)
```

string (Input) Must be of type character.

Results

The result type is character with the same length and kind parameter as *string*. The value of the result is the same as *string*, except that any leading blanks have been removed and inserted as trailing blanks.

Example

```
CHARACTER(16) STRING
STRING= ADJUSTL('   Fortran 90   ') ! returns 'Fortran 90      '

ADJUSTL ('    SUMMERTIME')     ! has the value 'SUMMERTIME    '
```

See Also

ADJUSTR

ADJUSTR

Elemental Intrinsic Function (Generic): *Adjusts a character string to the right, removing trailing blanks and inserting leading blanks.*

Syntax

```
result = ADJUSTR (string)
```

string (Input) Must be of type character.

Results

The result type is character with the same length and kind parameter as *string*.

The value of the result is the same as *string*, except that any trailing blanks have been removed and inserted as leading blanks.

Example

```
CHARACTER(16) STRING
STRING= ADJUSTR('   Fortran 90   ') ! returns '     Fortran 90'

ADJUSTR ('SUMMERTIME    ')     ! has the value '    SUMMERTIME'
```

See Also

ADJUSTL

AIMAG

Elemental Intrinsic Function (Generic): *Returns the imaginary part of a complex number. This function can also be specified as IMAG.*

Syntax

```
result = AIMAG (z)
```

z (Input) Must be of type complex.

Results

The result type is real with the same kind parameter as *z*. If *z* has the value (x, y), the result has the value y.

Specific Name	Argument Type	Result Type
AIMAG [1]	COMPLEX(4)	REAL(4)
DIMAG	COMPLEX(8)	REAL(8)
QIMAG	COMPLEX(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect AIMAG.

To return the real part of complex numbers, use REAL.

Example

AIMAG ((4.0, 5.0)) has the value 5.0.

If C is complex, C%IM is the same as AIMAG (C).

The program AIMAG.F90 applies the quadratic formula to a polynomial and allows for complex results:

```
REAL a, b, c
COMPLEX ans1, ans2, d
WRITE ( *, 100)
100 FORMAT ('  Enter A, b, and c of the  ',   &
                          'polynomial ax**2 + bx + c:  '\)
READ (*, *) a, b, c
d = CSQRT (CMPLX (b**2 - 4.0*a*c))  ! d is either:
                                    !  0.0 + i root, or
                                    !  root + i 0.0
ans1 = (-b + d) / (2.0 * a)
ans2 = (-b + d) / (2.0 * a)
WRITE (*, 200)
200 FORMAT (/ '  The roots are:'  /)
WRITE (*, 300) REAL(ans1), AIMAG(ans1), &
                  REAL(ans2), AIMAG(ans2)
300 FORMAT (' X = ', F10.5, ' + i', F10.5)
END
```

See Also

CONJG
DBLE

AINT

Elemental Intrinsic Function (Generic): *Truncates a value to a whole number.*

Syntax

```
result = AINT (a [,kind])
```

a (Input) Must be of type real.

kind (Input; optional) Must be a scalar integer constant expression.

Results

The result type is real. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter is that of *a*.

The result is defined as the largest integer whose magnitude does not exceed the magnitude of *a* and whose sign is the same as that of *a*. If | *a* | is less than 1, AINT(*a*) has the value zero.

Specific Name	Argument Type	Result Type
AINT [1]	REAL(4)	REAL(4)
DINT [1]	REAL(8)	REAL(8)
QINT	REAL(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect AINT and DINT.

To round rather than truncate, use ANINT.

Example

AINT (3.678) has the value 3.0.

AINT (-1.375) has the value -1.0.

```
REAL r1, r2
REAL(8) r3(2)
r1 = AINT(2.6)   ! returns the value 2.0
r2 = AINT(-2.6)  ! returns the value -2.0
r3 = AINT((/1.3, 1.9/), KIND = 8)       ! returns the values
                                        ! (1.0D0, 1.0D0)
```

See Also

ANINT

ALARM

Portability Function: *Causes a subroutine to begin execution after a specified amount of time has elapsed.*

Module

USE IFPORT

Syntax

`result = ALARM (time,proc)`

time	(Input) Integer. Specifies the time delay, in seconds, between the call to ALARM and the time when *proc* is to begin execution. If *time* is 0, the alarm is turned off and no routine is called.
proc	(Input) Name of the procedure to call. The procedure takes no arguments and must be declared EXTERNAL.

Results

The return value is INTEGER(4). It is zero if no alarm is pending. If an alarm is pending (has already been set by a previous call to ALARM), it returns the number of seconds remaining until the previously set alarm is to go off, rounded up to the nearest second.

After ALARM is called and the timer starts, the calling program continues for *time* seconds. The calling program then suspends and calls *proc*, which runs in another thread. When *proc* finishes, the alarm thread terminates, the original thread resumes, and the calling program resets the alarm. Once the alarm goes off, it is disabled until set again.

If *proc* performs I/O or otherwise uses the Fortran library, you need to compile it with one of the multithread libraries.

The thread that *proc* runs in has a higher priority than any other thread in the process. All other threads are essentially suspended until *proc* terminates, or is blocked on some other event, such as I/O.

No alarms can occur after the main process ends. If the main program finishes or any thread executes an EXIT call, than any pending alarm is deactivated before it has a chance to run.

Example

```
USE IFPORT
INTEGER(4) numsec, istat
EXTERNAL subprog
numsec = 4
write *, "subprog will begin in ", numsec, " seconds"
ISTAT = ALARM (numsec, subprog)
```

See Also

RUNQQ

ALIAS Directive

General Compiler Directive: *Declares alternate external names for external subprograms.*

Syntax

```
!DIR$ ALIAS internal-name,external-name
```

internal-name	The name of the entity as used in the current program unit. It can be a procedure name, a COMMON block name, a module variable that is initialized, or a PARAMETER in a module. It may not be the name of an internal procedure.
external-name	A name or a character constant, delimited by apostrophes or quotation marks.

If a name is specified, the name (in uppercase) is used as the external name for the specified *internal-name.* If a character constant is specified, it is used as is; the string is not changed to uppercase, nor are blanks removed.

The ALIAS directive affects only the external name used for references to the specified *internal-name.*

Names that are not acceptable to the linker will cause link-time errors.

See Also

ATTRIBUTES ALIAS
ATTRIBUTES DECORATE
ATTRIBUTES DEFAULT
General Compiler Directives
Syntax Rules for Compiler Directives

ALIGNED Clause

Parallel Directive Clause: *Specifies that all variables in a list are aligned.*

Syntax

```
ALIGNED (list [:n])
```

list	Is the name of one or more variables. Each name must be separated by a comma. Any variable that appears in *list* cannot appear in more than one ALIGNED clause.
n	Must be a constant positive integer expression; it indicates the number of bytes for the alignment. If *n* is not specified, the compiler uses the default alignment specified for SIMD instructions on the target platform.

The ALIGNED clause declares that the location of each list item is aligned to the number of bytes expressed in the optional alignment parameter *n* of the ALIGNED clause. If a list item has the ALLOCATABLE attribute, its allocation status must be allocated. If it has the POINTER attribute, its association status must be associated. If the type of a list item is type(C_PTR) or a Cray pointer, the item must be defined.

NOTE
Be careful when using the ALIGNED clause. Instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception if some of the access patterns are actually unaligned.

ALL

Transformational Intrinsic Function (Generic): *Determines if all values are true in an entire array or in a specified dimension of an array.*

Syntax

```
result = ALL (mask [,dim])
```

mask	(Input) Must be a logical array.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *mask*.

Results

The result is an array or a scalar of type logical.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result is true only if all elements of *mask* are true, or *mask* has size zero. The result has the value false if any element of *mask* is false.

An array result has the same type and kind parameters as *mask*, and a rank that is one less than *mask*. Its shape is (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of *mask*.

Each element in an array result is true only if all elements in the one dimensional array defined by *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) are true.

Example

```
       LOGICAL mask( 2, 3), AR1(3), AR2(2)
       mask = RESHAPE((/.TRUE., .TRUE., .FALSE., .TRUE., .FALSE., &
                       .FALSE./),(/2,3/))
! mask is true false false
! true true false
      AR1 = ALL(mask,DIM = 1) ! evaluates the elements column by
               ! column yielding [true false false]
      AR2 = ALL(mask,DIM = 2) ! evaluates the elements row by row
               ! yielding [false false].
```

ALL ((/.TRUE., .FALSE., .TRUE./)) has the value false because some elements of MASK are not true.

ALL ((/.TRUE., .TRUE., .TRUE./)) has the value true because *all* elements of MASK are true.

A is the array

```
   [ 1  5  7 ]
   [ 3  6  8 ]
```

and B is the array

```
   [ 0  5  7 ]
   [ 2  6  9 ].
```

ALL (A .EQ. B, DIM=1) tests to see if all elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, false) because only the second column has elements that are all equal.

ALL (A .EQ. B, DIM=2) tests to see if all elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (false, false) because each row has some elements that are not equal.

See Also

ANY
COUNT

ALLOCATABLE

Statement and Attribute: *Specifies that an object is allocatable. The shape of an allocatable array is determined when an ALLOCATE statement is executed, dynamically allocating space for the array. A character object may have a deferred length that is determined when the object is allocated with an ALLOCATE statement. A allocatable scalar object of any type may be allocated with an ALLOCATE statement.*

Syntax

The ALLOCATABLE attribute can be specified in a type declaration statement or an ALLOCATABLE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] ALLOCATABLE [, *att-ls*] :: *a*[(*d-spec*)] [[*coarray-spec*]][, *a*[(*d-spec*)] [[*coarray-spec*]]...

Statement:

ALLOCATABLE [::] *a*[(*d-spec*)] [[*coarray-spec*]][, *a*[(*d-spec*)] [[*coarray-spec*]]...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
a	Is the name of the allocatable object.
d-spec	Is a deferred-shape specification (: [, :] ...), where each colon represents a dimension of the array or a deferred-coshape specification.
coarray-spec	Is a deferred-coshape specification. The left bracket and right bracket are required.

Description

A character object declaration uses LEN=: to indicate it is deferred length.

If the array is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

When the allocatable object is no longer needed, it can be deallocated by execution of a DEALLOCATE statement.

An allocatable object cannot be specified in a COMMON, EQUIVALENCE, DATA, or NAMELIST statement.

Allocatable objects are not saved by default. If you want to retain the values of an allocatable object across procedure calls, you must specify the SAVE attribute for the object.

Example

```
! Method for declaring and allocating objects.

INTEGER, ALLOCATABLE :: matrix(:,:)        ! deferred shape
REAL, ALLOCATABLE    :: vector(:)
CHARACTER, ALLOCATABLE :: c1               ! char scalar, len=1
INTEGER, ALLOCATABLE :: k                  ! numeric scalar
CHARACTER(LEN=:), ALLOCATABLE :: c2, c3(:) ! deferred length, shape

ALLOCATE(matrix(3,5),vector(-2:N+2))       ! specifies shapes
ALLOCATE(c1)                               ! specifies scalar, len=1
ALLOCATE(k)                                ! specifies scalar
ALLOCATE(character(len=5) :: c2)           ! specifies scalar, len=5
ALLOCATE(character(len=3) :: c3(2:4))      ! specifies length, shape
```

The following example shows a type declaration statement specifying the ALLOCATABLE attribute:

```
REAL, ALLOCATABLE :: Z(:, :, :)
```

The following is an example of the ALLOCATABLE statement:

```
REAL A, B(:)
ALLOCATABLE :: A(:,:), B
```

See Also

Type Declarations

Compatible attributes
ALLOCATE
DEALLOCATE
Arrays
Allocation of Allocatable Arrays

SAVE

ALLOCATE

Statement: *Dynamically creates storage for allocatable variables and pointer targets.*

Syntax

```
ALLOCATE ([type::] object[(s-spec[, s-spec]...)] [, object[(s-spec[, s-spec]...)] ]...
[[coarray-spec]]...[, alloc-opt[, alloc-opt]...])
```

type

Is a data type specifier. If specified, the kind type parameters of each *object* must be the same as the corresponding *type* parameter values and each object must be type compatible with the specified type. The type must not have an ultimate subcomponent that is a coarray.

You cannot specify *type* if you specify SOURCE= or MOLD=.

object

Is the object to be allocated. It is a variable name or structure component, and must be a pointer or an allocatable object. The object can be of type character with zero length. If the object is a coarray and type is specified, type may not be C_PTR or C_FUNPTR. The object may not be coindexed.

s-spec

Is an array shape specification in the form [lower-bound:]upper-bound. Each bound must be a scalar integer expression. The number of shape specifications must be the same as the rank of the *object*.

You can specify an *s-spec* list for each array object in an ALLOCATE statement.

coarray-spec

Is a coarray shape specification in the form:

[[lower-bound:] upper-bound,] ... [lower-bound:] *

The brackets around coarray-spec are required. A coarray shape specification can only appear if the object is a coarray. The number of coarray specifications must be one less than the corank of the coarray object.

alloc-opt

Can be any of the following keywords:

STAT=*stat-var*

(Output) The *stat-var* is a scalar integer variable in which the status of the allocation is stored.

ERRMSG=*err-var*

(Output) The *err-var* is a scalar default character variable in which an error condition is stored if such a condition occurs.

SOURCE=*source-expr*
MOLD=*source-expr*

(Input) The *source-expr* is an expression that is scalar or has the same rank as *object*.

If MOLD= or SOURCE= is specified, *type* cannot be specified and each object must have the same rank as *source-expr* unless *source-expr* is scalar.

Only one of MOLD= or SOURCE= can appear in the same ALLOCATE statement.

If *source-expr* appears and its declared type is C_PTR or C_FUNPTR, *object* cannot be a coarray.

The declared type of *source-expr* cannot be EVENT_TYPE or LOCK_TYPE, or have a subcomponent of either type, or have a subcomponent that is a coarray.

You can specify STAT=, ERRMSG=, and one of MOLD= or SOURCE= in the same ALLOCATE statement. The keywords can appear in any order.

Description

The storage space allocated is uninitialized unless SOURCE= is specified or the type of the object is default initialized.

A bound in *s-spec* must not be an expression containing an array inquiry function whose argument is any allocatable object in the same ALLOCATE statement; for example, the following is not permitted:

```
INTEGER ERR
INTEGER, ALLOCATABLE :: A(:), B(:)
...
ALLOCATE(A(10:25), B(SIZE(A)), STAT=ERR)  ! A is invalid as an argument to function SIZE
```

If *object* is an array, *s-spec* must appear and the number of *s-specs* must equal the rank of *object*, or *source-expr* must appear and have the same rank as *object* and the shape of *object* is that of *source-expr*. If both *s-spec* and SOURCE=*source-expr* appear, *source-expr* must be scalar. If *object* is scalar, no *s-spec* should appear. If SOURCE=*source-expr* appears, *object* is initialized with the value of *source-expr*.

A type parameter value in *type* can be an asterisk if and only if each *object* is a dummy argument for which the corresponding type parameter is assumed.

If a STAT= variable, ERRMSG= variable, or *source-expr* is specified, it must not be allocated in the ALLOCATE statement in which it appears; nor can they depend on the value, bounds, length type parameters, allocation status, or association status of any object in the same ALLOCATE statement.

If the allocation is successful, the STAT= variable becomes defined with the value zero, and the definition status of the ERRMSG= variable remains unchanged. If the allocation is not successful, an error condition occurs, and the STAT= variable is set to a positive integer value (representing the run-time error); the ERRMSG= variable contains a descriptive message about the error condition.

If no STAT= variable is specified and an error condition occurs, the program initiates error termination.

If the allocation is successful and *source-expr* is specified, the dynamic type and value of the allocated object becomes that of the source expression. If the value of a non-deferred length type parameter of *object* is different from the value of the corresponding type parameter of *source-expr*, an error condition occurs.

If any object has a deferred type parameter, is unlimited polymorphic, or is of ABSTRACT type, either *type* or *source-expr* must appear.

If *object* is a coarray, *source-expr* must not have a dynamic type of C_PTR, C_FUNPTR, EVENT_TYPE, or LOCK_TYPE, or have a subcomponent whose dynamic type is EVENT_TYPE or LOCK_TYPE. The *object* cannot be a coindexed object.

When an ALLOCATE statement is executed for an *object* that is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment following the statement is delayed until all other images have executed the same statement the same number of times.

If an ALLOCATE or DEALLOCATE statement with a coarray allocatable object is executed when one or more images of the current team has initiated normal termination, the STAT= variable becomes defined with the processor-dependent positive integer value of the constant STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. Otherwise, if an allocatable object is a coarray and one or more images of the current team has failed, the STAT= variable becomes defined with the processor-dependent positive integer value of the constant STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV.

If any other error condition occurs during execution of the ALLOCATE or DEALLOCATE statement, the STAT= variable becomes defined with a processor-dependent positive integer value different from STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

If an ALLOCATE or DEALLOCATE statement with a coarray allocatable object is executed when one or more images of the current team has failed, each allocatable object is successfully allocated or deallocated on the active images of the current team. If any other error occurs, the allocation status of each allocatable object is processor dependent:

- Successfully allocated allocatable objects have the allocation status of allocated, or associated if the allocate object is has the POINTER attribute.
- Successfully deallocated allocatable objects have the allocation status of deallocated, or disassociated if the allocatable object has the POINTER attribute.
- An allocatable object that was not successfully allocated or deallocated has its previous allocation status, or its previous association status if it has the POINTER attribute.

If MOLD= appears and *source-expr* is a variable, its value does not have to be defined.

To release the storage for an object, use DEALLOCATE.

To determine whether an allocatable object is currently allocated, use the ALLOCATED intrinsic function.

To determine whether a pointer is currently associated with a target, use the ASSOCIATED intrinsic function.

Example

```
!Method for creating and allocating deferred shape arrays.
INTEGER,ALLOCATABLE::matrix(:,:)
REAL, ALLOCATABLE:: vector(:)
. . .
ALLOCATE (matrix(3,5),vector(-2:N+2))
. . .
```

The following shows another example of the ALLOCATE statement:

```
INTEGER J, N, ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE(A(0:80), B(-3:J+1, N), STAT = ALLOC_ERR)
```

The above example allocates the scalar objects s and t to be 15 by 25 matrices with the value of r.

Consider the following:

```
REAL(KIND=8) :: r(15, 25)
REAL(KIND=8), ALLOCATABLE :: s(:,:), t(:,:)
...
ALLOCATE(s,SOURCE=r)
ALLOCATE(t,SOURCE=r)
```

The above example allocates `s` and `t` to be 15 by 25 arrays with the values of `r`.

See Also

ALLOCATABLE
ALLOCATED
DEALLOCATE

ASSOCIATED
POINTER
Dynamic Allocation
Pointer Assignments
ISO_FORTRAN_ENV Module

ALLOCATED

Inquiry Intrinsic Function (Generic): *Indicates whether an allocatable array or allocatable scalar is currently allocated.*

Syntax

`result = ALLOCATED ([ARRAY=]array)`

`result = ALLOCATED ([SCALAR=]scalar)`

array	(Input) Must be an allocatable array.
scalar	(Input) Must be an allocatable scalar.

Results

The result is a scalar of type default logical.

The result has the value true if argument *array* or *scalar* is currently allocated; it has the value false if the argument is not currently allocated.

NOTE
When the argument keyword ARRAY is used, *array* must be an allocatable array. When the argument keyword SCALAR is used, *scalar* must be an allocatable scalar.

Example

```
REAL, ALLOCATABLE :: A(:)
...
IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

Consider the following:

```
REAL, ALLOCATABLE, DIMENSION (:,:,:) :: E
PRINT *, ALLOCATED (E)       ! Returns the value false
ALLOCATE (E (12, 15, 20))
PRINT *, ALLOCATED (E)       ! Returns the value true
```

See Also

ALLOCATABLE
ALLOCATE
DEALLOCATE
Arrays
Dynamic Allocation

ANINT

Elemental Intrinsic Function (Generic): *Calculates the nearest whole number.*

Syntax

```
result = ANINT (a[,kind] )
```

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of *a*. If *a* is greater than zero, ANINT (*a*) has the value AINT (*a* + 0.5); if *a* is less than or equal to zero, ANINT (*a*) has the value AINT (*a* - 0.5).

Specific Name	Argument Type	Result Type
ANINT [1]	REAL(4)	REAL(4)
DNINT [1]	REAL(8)	REAL(8)
QNINT	REAL(16)	REAL(16)

[1]The setting of compiler options specifying real size can affect ANINT and DNINT.

To truncate rather than round, use AINT.

Example

ANINT (3.456) has the value 3.0.

ANINT (-2.798) has the value -3.0.

Consider the following:

```
      REAL r1, r2
      r1 = ANINT(2.6)     ! returns the value 3.0
      r2 = ANINT(-2.6)    ! returns the value -3.0

! Calculates and adds tax to a purchase amount.
      REAL amount, taxrate, tax, total
      taxrate = 0.081
      amount = 12.99
      tax = ANINT (amount * taxrate * 100.0) / 100.0
      total = amount + tax
      WRITE (*, 100) amount, tax, total
  100  FORMAT ( 1X, 'AMOUNT', F7.2 /
     + 1X, 'TAX ', F7.2 /
     + 1X, 'TOTAL ', F7.2)
      END
```

See Also

NINT

ANY

Transformational Intrinsic Function (Generic):
Determines if any value is true in an entire array or in a specified dimension of an array.

Syntax

```
result = ANY (mask [, dim])
```

mask	(Input) Must be a logical array.
dim	(Input; optional) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *mask*.

Results

The result is an array or a scalar of type logical.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result is true if any elements of *mask* are true. The result has the value false if no element of *mask* is true, or *mask* has size zero.

An array result has the same type and kind parameters as *mask*, and a rank that is one less than *mask*. Its shape is (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *mask*.

Each element in an array result is true if any elements in the one dimensional array defined by *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) are true.

Example

```
    LOGICAL mask( 2, 3), AR1(3), AR2(2)
    logical, parameter :: T = .true.
    logical, parameter :: F = .false.
    DATA mask /T, T, F, T, F, F/
! mask is     true false false
!             true true false
    AR1 = ANY(mask,DIM = 1)  ! evaluates the elements column by
                             !   column yielding [true true false]
    AR2 = ANY(mask,DIM = 2)  ! evaluates the elements row by row
                             !   yielding [true true]
```

ANY ((/.FALSE., .FALSE., .TRUE./)) has the value true because one element is true.

A is the array

```
  [ 1  5  7 ]
  [ 3  6  8 ]
```

and B is the array

```
  [ 0  5  7 ]
  [ 2  6  9 ].
```

ANY (A .EQ. B, DIM=1) tests to see if any elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, true) because the second and third columns have at least one element that is equal.

ANY (A .EQ. B, DIM=2) tests to see if any elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (true, true) because each row has at least one element that is equal.

See Also

ALL
COUNT

APPENDMENUQQ (W*S)

QuickWin Function: *Appends a menu item to the end of a menu and registers its callback subroutine.*

Module

USE IFQWIN

Syntax

```
result = APPENDMENUQQ (menuID, flags, text,routine)
```

menuID
(Input) INTEGER(4). Identifies the menu to which the item is appended, starting with 1 as the leftmost menu.

flags
(Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see the Results section below). The following constants are available:

- $MENUGRAYED - Disables and grays out the menu item.
- $MENUDISABLED - Disables but does not gray out the menu item.
- $MENUENABLED - Enables the menu item.
- $MENUSEPARATOR - Draws a separator bar.
- $MENUCHECKED - Puts a check by the menu item.
- $MENUUNCHECKED - Removes the check by the menu item.

text
(Input) Character*(*). Menu item name. Must be a null-terminated C string, for example, 'WORDS OF TEXT'C.

routine
(Input) EXTERNAL. Callback subroutine that is called if the menu item is selected. All routines take a single LOGICAL parameter that indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- WINPRINT - Prints the program.
- WINSAVE - Saves the program.
- WINEXIT - Terminates the program.
- WINSELECTTEXT - Selects text from the current window.
- WINSELECTGRAPHICS - Selects graphics from the current window.
- WINSELECTALL - Selects the entire contents of the current window.
- WININPUT - Brings to the top the child window requesting input and makes it the current window.
- WINCOPY - Copies the selected text and/or graphics from the current window to the Clipboard.
- WINPASTE - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.
- WINCLEARPASTE - Clears the paste buffer.
- WINSIZETOFIT - Sizes output to fit window.
- WINFULLSCREEN - Displays output in full screen.
- WINSTATE - Toggles between pause and resume states of text output.
- WINCASCADE - Cascades active windows.
- WINTILE - Tiles active windows.
- WINARRANGE - Arranges icons.
- WINSTATUS - Enables a status bar.
- WININDEX - Displays the index for QuickWin help.
- WINUSING - Displays information on how to use Help.
- WINABOUT - Displays information about the current QuickWin application.
- NUL - No callback routine.

Results

The result type is logical. It is .TRUE. if successful; otherwise, .FALSE..

You do not need to specify a menu item number, because APPENDMENUQQ always adds the new item to the bottom of the menu list. If there is no item yet for a menu, your appended item is treated as the top-level menu item (shown on the menu bar), and *text* becomes the menu title. APPENDMENUQQ ignores the callback routine for a top-level menu item if there are any other menu items in the menu. In this case, you can set *routine* to NUL.

If you want to insert a menu item into a menu rather than append to the bottom of the menu list, use INSERTMENUQQ.

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to APPENDMENUQQ as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY (ALT+R in the example) as an alternative to selecting the item with the mouse.

Example

```
      USE IFQWIN
      LOGICAL(4) result
      CHARACTER(25) str
      ...
! Append two items to the bottom of the first (FILE) menu
      str    = '&Add to File Menu'C ! 'A' is a quick-access key
      result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
      str    = 'Menu Item &2b'C ! '2' is a quick-access key
      result = APPENDMENUQQ(1, $MENUENABLED, str, WINCASCADE)
! Append an item to the bottom of the second (EDIT) menu
      str    = 'Add to Second &Menu'C ! 'M' is a quick-access key
      result = APPENDMENUQQ(2, $MENUENABLED, str, WINTILE)
```

See Also

INSERTMENUQQ
DELETEMENUQQ
MODIFYMENUFLAGSQQ
MODIFYMENUROUTINEQQ
MODIFYMENUSTRINGQQ

ARC, ARC_W (W*S)

Graphics Functions: *Draw elliptical arcs using the current graphics color.*

Module

USE IFQWIN

Syntax

`result = ARC (x1,y1,x2,y2,x3,y3,x4,y4)`

`result = ARC_W (wx1,wy1,wx2,wy2,wx3,wy3,wx4,wy4)`

x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
x3, y3	(Input) INTEGER(2). Viewport coordinates of start vector.
x4, y4	(Input) INTEGER(2). Viewport coordinates of end vector.
wx1,wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.
wx3, wy3	(Input) REAL(8). Window coordinates of start vector.
wx4, wy4	(Input) REAL(8). Window coordinates of end vector.

Results

The result type is INTEGER(2). It is nonzero if successful; otherwise, 0. If the arc is clipped or partially out of bounds, the arc is considered successfully drawn and the return is 1. If the arc is drawn completely out of bounds, the return is 0.

The center of the arc is the center of the bounding rectangle defined by the points (*x1*, *y1*) and (*x2*, *y2*) for ARC and (*wx1*, *wy1*) and (*wx2*, *wy2*) for ARC_W.

The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*) for ARC and (*wx3*, *wy3*) for ARC_W. It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*) for ARC and (*wx4*, *wy4*) for ARC_W.

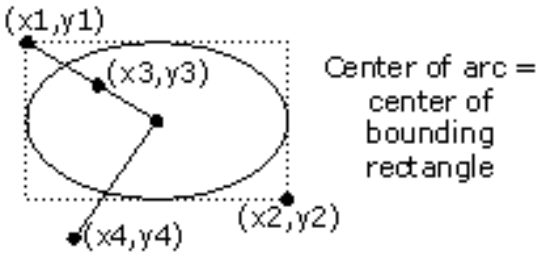
ARC uses the view-coordinate system. ARC_W uses the window-coordinate system. In each case, the arc is drawn using the current color.

NOTE
The ARC routine described here is a QuickWin graphics routine. If you are trying to use the Microsoft* Platform SDK version of the Arc routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Arc.

Example

This program draws the arc shown below.

```
USE IFQWIN
INTEGER(2) status, x1, y1, x2, y2, x3, y3, x4, y4

x1 = 80; y1 = 50
x2 = 240; y2 = 150
x3 = 120; y3 = 75
x4 = 90; y4 = 180
status = ARC( x1, y1, x2, y2, x3, y3, x4, y4 )
END
```



ASIN

Elemental Intrinsic Function (Generic): *Produces the arcsine of an argument.*

Syntax

```
result = ASIN (x)
```

x	(Input) Must be of type real, where | *x* | must be less than or equal to 1, or of type complex.

Results

The result type and kind are the same as *x*.

If the result is real, it is expressed in radians and lies in the range - pi/2 <= ASIN (x) <= pi/2.

If the result is complex, the real part is expressed in radians and lies in the range pi/2 <= REAL (ASIN (x)) <= pi/2.

Specific Name	Argument Type	Result Type
ASIN	REAL(4)	REAL(4)
DASIN	REAL(8)	REAL(8)
QASIN	REAL(16)	REAL(16)

Example

ASIN (0.79345021) has the value 0.9164571.

ASIND

Elemental Intrinsic Function (Generic): *Produces the arcsine of x.*

Syntax

```
result = ASIND (x)
```

x	(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results

The result type and kind are the same as *x* and are expressed in degrees. The value lies in the range −90 to 90 degrees.

Specific Name	Argument Type	Result Type
ASIND	REAL(4)	REAL(4)
DASIND	REAL(8)	REAL(8)
QASIND	REAL(16)	REAL(16)

Example

ASIND (0.2467590) has the value 14.28581.

ASINH

Elemental Intrinsic Function (Generic): *Produces the hyperbolic arcsine of x.*

Syntax

```
result = ASINH (x)
```

x (Input) Must be of type real or complex.

Results

The result type and kind are the same as *x*.

If the result is complex, the imaginary part is expressed in radians and lies in the range - pi/ 2 <= AIMAG (ASINH (x)) <= pi / 2.

Specific Name	Argument Type	Result Type
ASINH	REAL(4)	REAL(4)
DASINH	REAL(8)	REAL(8)
QASINH	REAL(16)	REAL(16)

Example

ASINH (1.0) has the value -0.88137.

ASINH (180.0) has the value 5.88611.

ASSIGN - Label Assignment

Statement: *Assigns a statement label value to an integer variable. This feature has been deleted in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.*

Syntax

```
ASSIGN label TO var
```

label Is the label of a branch target or FORMAT statement in the same scoping unit as the ASSIGN statement.

var Is a scalar integer variable.

When an ASSIGN statement is executed, the statement label is assigned to the integer variable. The variable is then undefined as an integer variable and can only be used as a label (unless it is later redefined with an integer value).

The ASSIGN statement must be executed before the statements in which the assigned variable is used.

Indirect branching through integer variables makes program flow difficult to read, especially if the integer variable is also used in arithmetic operations. Using these statements permits inconsistent usage of the integer variable, and can be an obscure source of error. The ASSIGN statement was used to simulate internal procedures, which now can be coded directly.

Example

The value of a label is not the same as its number; instead, the label is identified by a number assigned by the compiler. In the following example, 400 is the label number (not the value) of IVBL:

```
ASSIGN 400 TO IVBL
```

Variables used in ASSIGN statements are not defined as integers. If you want to use a variable defined by an ASSIGN statement in an arithmetic expression, you must first define the variable by a computational assignment statement or by a READ statement, as in the following example:

```
IVBL = 400
```

The following example shows ASSIGN statements:

```
INTEGER ERROR
...
ASSIGN 10 TO NSTART
ASSIGN 99999 TO KSTOP
ASSIGN 250 TO ERROR
```

Note that NSTART and KSTOP are integer variables implicitly, but ERROR must be previously declared as an integer variable.

The following statement associates the variable NUMBER with the statement label 100:

```
ASSIGN 100 TO NUMBER
```

If an arithmetic operation is subsequently performed on variable NUMBER (such as follows), the run-time behavior is unpredictable:

```
NUMBER = NUMBER + 1
```

To return NUMBER to the status of an integer variable, you can use the following statement:

```
NUMBER = 10
```

This statement dissociates NUMBER from statement 100 and assigns it an integer value of 10. Once NUMBER is returned to its integer variable status, it can no longer be used in an assigned GO TO statement.

See Also

Assignment: intrinsic
Obsolescent Language Features in the Fortran Standard

Assignment(=) - Defined Assignment

Statement: *An interface block that defines generic assignment. The only procedures allowed in the interface block are subroutines that can be referenced as defined assignments.*

Syntax

The initial line for such an interface block takes the following form:

```
INTERFACE ASSIGNMENT (=)
```

Description

The subroutines within the interface block must have two nonoptional arguments, the first with intent OUT or INOUT, and the second with intent IN and/or attribute VALUE.

A defined assignment is treated as a reference to a subroutine. The left side of the assignment corresponds to the first dummy argument of the subroutine; the right side of the assignment corresponds to the second argument.

The ASSIGNMENT keyword extends or redefines an assignment operation if both sides of the equal sign are of the same derived type.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

Any procedure reference involving generic assignment must be resolvable to one specific procedure; it must be unambiguous. For more information, see Unambiguous Generic Procedure References.

Example

The following is an example of a procedure interface block defining assignment:

```
INTERFACE ASSIGNMENT (=)
  SUBROUTINE BIT_TO_NUMERIC (NUM, BIT)
    INTEGER, INTENT(OUT) :: NUM
    LOGICAL, INTENT(IN)  :: BIT(:)
  END SUBROUTINE BIT_TO_NUMERIC
  SUBROUTINE CHAR_TO_STRING (STR, CHAR)
    USE STRING_MODULE                   ! Contains definition of type STRING
    TYPE(STRING), INTENT(OUT) :: STR    ! A variable-length string
    CHARACTER(*), INTENT(IN)  :: CHAR
  END SUBROUTINE  CHAR_TO_STRING
END  INTERFACE
```

The following example shows two equivalent ways to reference subroutine BIT_TO_NUMERIC:

```
  CALL BIT_TO_NUMERIC(X, (NUM(I:J)))
  X = NUM(I:J)
```

The following example shows two equivalent ways to reference subroutine CHAR_TO_STRING:

```
  CALL CHAR_TO_STRING(CH, '432C')
  CH = '432C'
```

The following is an example of a declaration and reference to a defined assignment:

```
!Converting circle data to interval data.
module mod1
TYPE CIRCLE
        REAL radius, center_point(2)
END TYPE CIRCLE
TYPE INTERVAL
        REAL lower_bound, upper_bound
END TYPE INTERVAL
CONTAINS
        SUBROUTINE circle_to_interval(I,C)
          type (interval),INTENT(OUT)::I
          type (circle),INTENT(IN)::C
!Project circle center onto the x=-axis
!Note: the length of the interval is the diameter of the circle
```

```
                I%lower_bound = C%center_point(1) - C%radius
                I%upper_bound = C%center_point(1) + C%radius
        END SUBROUTINE circle_to_interval
end module mod1

PROGRAM assign
use mod1
TYPE(CIRCLE) circle1
TYPE(INTERVAL) interval1
INTERFACE ASSIGNMENT(=)
        module procedure circle_to_interval
END INTERFACE
!Begin executable part of program
  circle1%radius = 2.5
  circle1%center_point = (/3.0,5.0/)
  interval1 = circle1
. . .
END PROGRAM
```

See Also

INTERFACE

Assignment Statements

Assignment - Intrinsic Computational

Statement: *Assigns a value to a nonpointer variable. In the case of pointers, intrinsic assignment is used to assign a value to the target associated with the pointer variable. The value assigned to the variable (or target) is determined by evaluation of the expression to the right of the equal sign.*

Syntax

variable=expression

variable	Is the name of a scalar or array of intrinsic or derived type (with no defined assignment). The array cannot be an assumed-size array, and neither the scalar nor the array can be declared with the PARAMETER or INTENT(IN) attribute.
expression	Is of intrinsic type or the same derived type as *variable*. Its shape must conform with *variable*. If necessary, it is converted to the same type and kind as *variable*.

Description

Before a value is assigned to the variable, the expression part of the assignment statement and any expressions within the variable are evaluated. No definition of expressions in the variable can affect or be affected by the evaluation of the expression part of the assignment statement.

NOTE
When the run-time system assigns a value to a scalar integer or character variable and the variable is shorter than the value being assigned, the assigned value may be truncated and significant bits (or characters) lost. This truncation can occur without warning, and can cause the run-time system to pass incorrect information back to the program.

If the variable is a pointer, it must be associated with a definable target. The shape of the target and expression must conform and their type and kind parameters must match.

If the !DIR$ NOSTRICT compiler directive (the default) is in effect, then you can assign a character expression to a noncharacter variable, and a noncharacter variable or array element (but not an expression) to a character variable.

Example

```
    REAL a, b, c
    LOGICAL abigger
    CHARACTER(16) assertion
    c = .01
    a = SQRT (c)
    b = c**2
    assertion = 'a > b'
    abigger = (a .GT. b)
    WRITE (*, 100) a, b
100 FORMAT (' a =', F7.4, ' b =', F7.4)
    IF (abigger) THEN
            WRITE (*, *) assertion, ' is true.'
    ELSE
            WRITE (*, *) assertion, ' is false.'
    END IF
    END
! The program above has the following output:
!     a =    .1000     b =   .0001         a > b is true.
! The following code shows legal and illegal
! assignment statements:
!
    INTEGER i, j
    REAL rone(4), rtwo(4), x, y
    COMPLEX z
    CHARACTER name6(6), name8(8)
    i         = 4
    x         = 2.0
    z         = (3.0, 4.0)
    rone(1) = 4.0
    rone(2) = 3.0
    rone(3) = 2.0
    rone(4) = 1.0
    name8   = 'Hello,'

! The following assignment statements are legal:
    i     = rone(2); j = rone(i); j = x
    y     = x; y = z; y = rone(3); rtwo = rone; rtwo = 4.7
    name6 = name8

! The following assignment statements are illegal:
    name6 = x + 1.0; int = name8//'test'; y = rone
    END
```

See Also

Assignment: defined
NOSTRICT directive

ASSOCIATE

Statement: *Marks the beginning of an ASSOCIATE construct. The ASSOCIATE construct creates a temporary association between a named entity and a variable or the value of an expression. The association lasts for the duration of the block.*

Syntax

```
[name:] ASSOCIATE (assoc-entity[, assoc-entity]...)
   block
END ASSOCIATE [name]
```

name — (Optional) Is the name of the ASSOCIATE construct.

assoc-entity — Is *associate-name => selector*

- *associate-name* — Is an identifier that becomes associated with the selector. It becomes the associating entity. The identifier name must be unique within the construct.
- *selector* — Is an expression or variable. It becomes the associated entity.

block — Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified at the beginning of an ASSOCIATE statement, the same name must appear in the corresponding END ASSOCIATE statement. The same construct name must not be used for different named constructs in the same scoping unit. If no name is specified at the beginning of an ASSOCIATE statement, you cannot specify one following the END ASSOCIATE statement.

During execution of the block within the construct, each *associate-name* identifies an entity, which is associated with the corresponding *selector*. The associating entity assumes the declared type and type parameters of the selector.

You can only branch to an END ASSOCIATE statement from within its ASSOCIATE construct.

This construct is useful when you want to simplify multiple accesses to a variable that has a lengthy description; for example, if the variable contains multiple subscripts and component names.

Example

The following shows an expression as a selector:

```
ASSOCIATE (O => (A-F)**2 + (B+G)**2)
  PRINT *, SQRT (O)
END ASSOCIATE
```

The following shows association with an array section:

```
ASSOCIATE (ARRAY => AB % D (I, :) % X)
  ARRAY (3) = ARRAY (1) + ARRAY (2)
END ASSOCIATE
```

Without the ASSOCIATE construct, this is what you would need to write:

```
AB % D (I, 3) % X = AB % D (I, 1) % X + AB % D (I, 2) % X
```

See Also

Construct Association
Additional Attributes Of Associate Names

ASSOCIATED

Inquiry Intrinsic Function (Generic): *Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.*

Syntax

`result = ASSOCIATED (pointer [, target])`

pointer	(Input) Must be a pointer. It can be of any data type. The pointer association status must be defined.
target	(Input; optional) Must be a pointer or target. If it is a pointer, the pointer association status must be defined.

Results

The result is a scalar of type default logical. The setting of compiler options specifying integer size can affect this function.

If only *pointer* appears, the result is true if it is currently associated with a target; otherwise, the result is false.

If *target* also appears and is a target, the result is true if *pointer* is currently associated with *target*; otherwise, the result is false.

If *target* is a pointer, the result is true if both *pointer* and *target* are currently associated with the same target; otherwise, the result is false. (If either *pointer* or *target* is disassociated, the result is false.)

Example

```
REAL C (:), D(:), E(5)
POINTER C, D
TARGET E
LOGICAL STATUS
C => E                        ! pointer assignment
D => E                        ! pointer assignment
STATUS = ASSOCIATED(C)        ! returns TRUE; C is associated
STATUS = ASSOCIATED(C, E)     ! returns TRUE; C is associated with E
STATUS = ASSOCIATED (C, D)    ! returns TRUE; C and D are associated
                              !   with the same target
```

Consider the following:

```
   REAL, TARGET, DIMENSION (0:50) :: TAR
   REAL, POINTER, DIMENSION (:) :: PTR
   PTR => TAR
   PRINT *, ASSOCIATED (PTR, TAR)      ! Returns the value true
```

The subscript range for PTR is 0:50. Consider the following pointer assignment statements:

```
   (1) PTR => TAR (:)
   (2) PTR => TAR (0:50)
   (3) PTR => TAR (0:49)
```

For statements 1 and 2, ASSOCIATED (PTR, TAR) is true because TAR has not changed (the subscript range for PTR in both cases is 1:51, following the rules for deferred-shape arrays). For statement 3, ASSOCIATED (PTR, TAR) is false because the upper bound of TAR has changed.

Consider the following:

```
REAL, POINTER, DIMENSION (:) :: PTR2, PTR3
ALLOCATE (PTR2 (0:15))
PTR3 => PTR2
PRINT *, ASSOCIATED (PTR2, PTR3)       ! Returns the value true
...
NULLIFY (PTR2)
NULLIFY (PTR3)
PRINT *, ASSOCIATED (PTR2, PTR3)       ! Returns the value false
```

See Also

ALLOCATED
POINTER
TARGET
Pointer Assignments

ASSUME

General Compiler Directive: *Provides heuristic information to the compiler optimizer.*

Syntax

`!DIR$ ASSUME (`*`scalar-Boolean-expression`*`)`

`scalar-Boolean-expression` Is any expression that evaluates to .TRUE. or .FALSE. at run-time.

The *scalar-Boolean-expression* is presumed to be true and may be used by the optimizer to generate better code.

If the `check assume` option is specified and *scalar-Boolean-expression* does not evaluate to .TRUE. at run-time, an error message is displayed and execution is aborted.

Example

In the example below, the compiler is told that A is aligned on a 32-byte boundary using the ASSUME_ALIGNED directive. The ASSUME directive says that the length of the first dimension of A is a multiple of 8. Therefore the optimizer knows that A(I,J+1) and A(I,J-1) are 0 mod 64 bytes away from A(I,J) and are therefore also aligned on 32-byte boundaries. This information helps the optimizer in generating efficiently vectorized code for these loops.

```
SUBROUTINE F (A, NX,NY,I1,I2,J1,J2)
REAL (8) :: A (NX,NY)
!DIR$ ASSUME_ALIGNED A:32
!DIR$ ASSUME (MOD(NX,8) .EQ. 0)
! ensure that the first array access in the loop is aligned
!DIR$ ASSUME (MOD(I1,8) .EQ. 1)
DO J=J1,J2
  DO I=I1,I2
     A(I,J) = A(I,J) + A(I,J+1) + A(I,J-1)
  ENDDO
ENDDO
END SUBROUTINE F
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
ASSUME_ALIGNED directive
`check` compiler option (setting assume)

ASSUME_ALIGNED

General Compiler Directive: *Specifies that an entity in memory is aligned.*

Syntax

```
!DIR$ ASSUME_ALIGNED address1:n1 [, address2:n2]...
```

address

An array variable. It can be of any data type, kind, or rank > 0. It can be an array component of a variable of derived type or a record field reference, host or use associated, or have the ALLOCATABLE or POINTER attribute.

It cannot be any of the following:

- An entity in COMMON (or an entity EQUIVALENCEd to something in COMMON)
- A component of a variable of derived type or a record field reference
- An entity accessed by use or host association

If it is a module variable, that address is silently ignored.

n

A positive integer constant expression. Its value must be a power of 2 between 1 and 256, that is, 1, 2, 4, 8, 16, 32, 64, 128, 256. It specifies the memory alignment in bytes of *address*.

The ASSUME_ALIGNED directive must appear after the specification statements section or inside the executable statements section.

If you specify more than one *address*:*n* item, they must be separated by a comma.

If *address* is a Cray POINTER or it has the POINTER attribute, it is the POINTER and not the pointee or the TARGET that is assumed aligned.

If the check assume option is specified and *address* is not aligned on an *n*-byte boundary at run-time, an error message is displayed and execution is aborted.

For more information, see the example in the description of the ASSUME directive.

Example

The following example shows the correct placement and usage of the ASSUME_ALIGNED directive:

```
SUBROUTINE F(A, N)
  TYPE NODE
    REAL(KIND=8), POINTER :: A(:,:)
  END TYPE NODE

  TYPE(NODE), POINTER :: NODES

  ALLOCATE(NODES)
  ALLOCATE(NODES%A(1000,1000))
```

```
!DIR$ ASSUME_ALIGNED NODES%A(1,1) : 16
  DO I=1,N
    NODES%A(1,I) = NODES%A(1,I)+1
  ENDDO
…
END
```

It is illegal to place ASSUME_ALIGNED inside a type definition; for example:

```
TYPE S
!DIR$ ASSUME_ALIGNED T : 16           ! this is an error
  REAL(8), ALLOCATABLE :: T(:)
END TYPE S
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
ATTRIBUTES ALIGN
ASSUME directive
`check` compiler option

ASYNCHRONOUS

Statement and Attribute: *Specifies that a variable can be used for asynchronous input and output.*

Syntax

The ASYNCHRONOUS attribute can be specified in a type declaration statement or an ASYNCHRONOUS statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] ASYNCHRONOUS [, *att-ls*] :: *var* [, *var*] ...

Statement:

ASYNCHRONOUS [::] *var* [, *var*] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
var	Is the name of a variable.

Description

Asynchronous I/O, or non-blocking I/O, allows a program to continue processing data while the I/O operation is performed in the background.

A variable can have the ASYNCHRONOUS attribute in a particular scoping unit without necessarily having it in other scoping units. If an object has the ASYNCHRONOUS attribute, then all of its subobjects also have the ASYNCHRONOUS attribute.

The ASYNCHRONOUS attribute can also be implied by use of a variable in an asynchronous READ or WRITE statement.

You can specify variables that are used for asynchronous communication, such as with Message Passing Interface Standard (MPI). Asynchronous communication has the following restrictions:

- For input, a pending communication affector must not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed.
- For output, a pending communication affector must not be redefined, become undefined, or have its pointer association status changed.

Examples

The following example shows how the ASYNCHRONOUS attribute can be applied in an OPEN and READ statement.

```
program test
integer, asynchronous, dimension(100) :: array
open (unit=1,file='asynch.dat',asynchronous='YES',&
  form='unformatted')
write (1) (i,i=1,100)
rewind (1)
read (1,asynchronous='YES') array
wait(1)
write (*,*) array(1:10)
end
```

See Also

Type Declarations
Compatible attributes

ATAN

Elemental Intrinsic Function (Generic): *Produces the arctangent of an argument.*

Syntax

`result = ATAN (x)`

`result = ATAN (y,x)`

y	(Input) Must be of type real.
x	(Input) If *y* appears, *x* must be of type real with the same kind type parameter as *y*. If *y* has the value zero, *x* must not have the value zero. If *y* does not appear, *x* must be of type real or complex.

Results

The result type and kind are the same as *x*.

If *y* appears, the result is the same as the result of ATAN2 (y, x).

If *y* does not appear, the real part of the result is expressed in radians and lies in the range -pi/2 <= ATAN (x) <= pi/2.

Specific Name	Argument Type	Result Type
ATAN	REAL(4)	REAL(4)
DATAN	REAL(8)	REAL(8)
QATAN	REAL(16)	REAL(16)

Example

ATAN (1.5874993) has the value 1.008666.

ATAN (2.679676, 1.0) has the value 1.213623.

ATAN2

Elemental Intrinsic Function (Generic): *Produces an arctangent (inverse tangent). The result is the principal value of the argument of the nonzero complex number (x, y).*

Syntax

```
result = ATAN2 (y,x)
```

y	(Input) Must be of type real.
x	(Input) Must have the same type and kind parameters as *y*. If *y* has the value zero, *x* cannot have the value zero.

Results

The result type and kind are the same as *x* and are expressed in radians. The value lies in the range -pi <= ATAN2 (*y*, *x*) <= pi.

If *x* is not zero, the result is approximately equal to the value of arctan (*y*/ *x*).

If *y* > zero, the result is positive.

If *y* < zero, the result is negative.

If *y* is zero and *x* > zero, the result is *y* (so for x>0, ATAN2 ((+0.0), x) is +0.0 and ATAN2 ((-0.0), x) is -0.0).

If *y* is a positive real zero and *x* < zero, the result is pi.

If *y* is a negative real zero and *x* < zero, the result is -pi.

If *x* is a positive real zero, the result is pi/2.

If *y* is a negative real zero, the result is -pi/2.

Specific Name	Argument Type	Result Type
ATAN2	REAL(4)	REAL(4)
DATAN2	REAL(8)	REAL(8)
QATAN2	REAL(16)	REAL(16)

Example

ATAN2 (2.679676, 1.0) has the value 1.213623.

If Y is an array that has the value

```
  [  1   1 ]
  [ -1  -1 ]
```

and X is an array that has the value

```
  [ -1  1 ]
  [ -1  1 ],
```

then ATAN2 (Y, X) is

$$\begin{bmatrix} \frac{3pi}{4} & \frac{pi}{4} \\ \frac{-3pi}{4} & \frac{-pi}{4} \end{bmatrix}$$

ATAN2D

Elemental Intrinsic Function (Generic): *Produces an arctangent. The result is the principal value of the argument of the nonzero complex number (x, y).*

Syntax

```
result = ATAN2D (y,x)
```

y	(Input) Must be of type real.
x	(Input) Must have the same type and kind parameters as *y*. If *y* has the value zero, *x* cannot have the value zero.

Results

The result type and kind are the same as *x* and are expressed in degrees. The value lies in the range -180 degrees to 180 degrees. If *x* zero, the result is approximately equal to the value of arctan (*y*/ *x*).

If *y* > zero, the result is positive.

If *y* < zero, the result is negative.

If *y* = zero, the result is zero (if *x* > zero) or 180 degrees (if *x* < zero).

If *x* = zero, the absolute value of the result is 90 degrees.

Specific Name	Argument Type	Result Type
ATAN2D	REAL(4)	REAL(4)
DATAN2D	REAL(8)	REAL(8)
QATAN2D	REAL(16)	REAL(16)

Example

ATAN2D (2.679676, 1.0) has the value 69.53546.

ATAND

Elemental Intrinsic Function (Generic): *Produces the arctangent of x.*

Syntax

```
result = ATAND (x)
```

x	(Input) Must be of type real and must be greater than or equal to zero.

Results

The result type and kind are the same as *x* and are expressed in degrees.

Specific Name	Argument Type	Result Type
ATAND	REAL(4)	REAL(4)
DATAND	REAL(8)	REAL(8)
QATAND	REAL(16)	REAL(16)

Example

ATAND (0.0874679) has the value 4.998819.

ATANH

Elemental Intrinsic Function (Generic): *Produces the hyperbolic arctangent of x.*

Syntax

```
result = ATANH (x)
```

x	(Input) Must be of type real, where | *x* | is less than or equal to 1, or of type complex.

Results

The result type and kind are the same as *x*.

If the result is real, it lies in the range -1.0 < ATANH (x) < 1.0.

If the result is complex, the imaginary part is expressed in radians and lies in the range –pi/2 <= AIMAG (ATANH (x)) <= pi/2.

Specific Name	Argument Type	Result Type
ATANH	REAL(4)	REAL(4)
DATANH	REAL(8)	REAL(8)
QATANH	REAL(16)	REAL(16)

Example

ATANH (-0.77) has the value -1.02033.

ATANH (0.5) has the value 0.549306.

ATOMIC

OpenMP* Fortran Compiler Directive: *Ensures that a specific memory location is updated atomically; this prevents the possibility of multiple threads simultaneously reading and writing the specific memory location.*

Syntax

```
!$OMP ATOMIC [type-clause[[,] clause]]
-or- !$OMP ATOMIC [clause[[,] type-clause]]
   block
!$OMP END ATOMIC
```

type-clause

(Optional) Is one of the following:

- CAPTURE

  If !$OMP ATOMIC CAPTURE is specified, *block* is either one *update-statement* and one *capture-statement* in either order, or one *capture-statement* followed by one *write-statement* in that order.
- READ

  If !$OMP ATOMIC READ is specified, *block* is a *capture-statement*.
- WRITE

  If !$OMP ATOMIC WRITE is specified, *block* is a *write-statement*.
- UPDATE

  If !$OMP ATOMIC UPDATE is specified, *block* is an *update-statement*.

For details on the effects of these clauses, see the table in the Description section.

clause

(Optional) Is the following:

- SEQ_CST

  Forces the atomically performed operation to include an implicit flush operation without a list. For details on the effects of this clause, see the table in the Description section.

capture-statement

Is an expression in the form *v* = *x*.

write-statement

Is an expression in the form *x* = *expr*.

update-statement

Is an expression with one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr-list)
x = intrinsic (expr-list, x)
```

The following rules apply:

- Operators in *expr* must have precedence equal to or greater than the precedence of operator.
- *xoperatorexpr* must be mathematically equivalent to *x* operator (*expr*). This requirement is satisfied if the *operator*s in *expr* have precedence greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.

	• *exproperatorx* must be mathematically equivalent to (*expr*) operator *x*. This requirement is satisfied if the *operator*s in *expr* have precedence equal to or greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*. • All assignments must be intrinsic assignments.
x, v	Are scalar variables of intrinsic type. During execution of an atomic region, all references to storage location *x* must specify the same storage location. *v* must not access the same storage location as *x*.
expr, expr-list	The *expr* is a scalar expression. The *expr-list* is a comma-separated, non-empty list of scalar expressions. They must not access the same storage location as *x*. If *intrinsic* is IAND, IOR, or IEOR, only one expression can appear in *expr-list*.
operator	Is one of the following intrinsic operators: +, *, -, /, ,AND., ,OR., .EQV., or .NEQV..
intrinsic	Is one of the following intrinsic procedures: MAX, MIN, IAND, IOR, or IEOR.

The binding thread set for an ATOMIC construct is all threads. Atomic regions enforce exclusive access with respect to other atomic regions that access the same storage location *x* among all the threads in the program without regard to the teams to which the threads belong.

If !$OMP ATOMIC is specified with no *type-clause* or no *clause*, it is the same as specifying !$OMP ATOMIC UPDATE.

If !$OMP ATOMIC CAPTURE is specified, you must include an !$OMP END ATOMIC directive following the *block*. Otherwise, the !$OMP END ATOMIC directive is optional.

Note that the following restriction applies to the ATOMIC directive:

- All atomic accesses to the storage locations designated by *x* throughout the program must have the same type and type parameters.

The following table describes what happens when you specify one of the clauses in an ATOMIC construct.

Clause	Result
CAPTURE	Causes an atomic update of the location designated by *x* using the designated operator or intrinsic while also capturing the original or final value of the location designated by *x* in *v*. The following rules also apply: • The original or final value of the location designated by *x* is written in the location designated by *v*, depending on the form of the ATOMIC construct, structured block, or statements, following the usual language semantics. • Only the read and write of the location designated by x are performed mutually atomically.

Clause	Result
	• The evaluation of *expr* or *expr-list*, and the write to the location designated by *v* do not need to be atomic with respect to the read or write of the location designated by *x*. No task scheduling points are allowed between the read and the write of the location designated by *x*.
READ	Causes an atomic read of the location designated by *x* regardless of the native machine word size.
SEQ_CST	Causes the atomically performed operation to include an implicit flush operation without a list. If this clause is specified, the construct is a *sequentially consistent atomic construct*. Unlike non-sequentially consistent atomic constructs, sequentially consistent atomic constructs preserve the interleaving (sequentially consistent) behavior of correct, data-race-free programs. However, they are not designed to replace the FLUSH directive as a mechanism to enforce ordering for non-sequentially consistent atomic constructs, and attempts to do so require extreme caution. For example, a sequentially consistent atomic write construct may appear to be reordered with a subsequent non-sequentially consistent atomic write construct, since such reordering would not be observable by a correct program if the second write were outside an ATOMIC directive.
UPDATE	Causes an atomic update of the location designated by *x* using the designated operator or intrinsic. The following rules also apply: • The evaluation of *expr* or *expr-list* need not be atomic with respect to the read or write of the location designated by *x*. • No task scheduling points are allowed between the read and the write of the location designated by *x*.
WRITE	Causes an atomic write of the location designated by *x* regardless of the native machine word size.

Any combination of two or more of these atomic constructs enforces mutually exclusive access to the locations designated by *x*.

A race condition exists when two unsynchronized threads access the same shared variable with at least one thread modifying the variable; this can cause unpredictable results. To avoid race conditions, all accesses of the locations designated by *x* that could potentially occur in parallel must be protected with an ATOMIC construct.

Atomic regions do not guarantee exclusive access with respect to any accesses outside of atomic regions to the same storage location x even if those accesses occur during a CRITICAL or ORDERED region, while an OpenMP lock is owned by the executing task, or during the execution of a REDUCTION clause.

However, other OpenMP* synchronization can ensure the desired exclusive access. For example, a BARRIER directive following a series of atomic updates to x guarantees that subsequent accesses do not form a race condition with the atomic accesses.

Example

The following example shows a way to avoid race conditions by using ATOMIC to protect all simultaneous updates of the location by multiple threads.

Since the ATOMIC directive below applies only to the statement immediately following it, elements of Y are not updated atomically.

```
   REAL FUNCTION FOO1(I)
      INTEGER I
      FOO1 = 1.0 * I
      RETURN
   END FUNCTION FOO1

   REAL FUNCTION FOO2(I)
      INTEGER I
      FOO2 = 2.0 * I
      RETURN
   END FUNCTION FOO2

   SUBROUTINE SUB(X, Y, INDEX, N)
      REAL X(*), Y(*)
      INTEGER INDEX(*), N
      INTEGER I
!$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
      DO I=1,N
!$OMP ATOMIC UPDATE
         X(INDEX(I)) = X(INDEX(I)) + FOO1(I)
         Y(I) = Y(I) + FOO2(I)
      ENDDO
   END SUBROUTINE SUB

   PROGRAM ATOMIC_DEMO
      REAL X(1000), Y(10000)
      INTEGER INDEX(10000)
      INTEGER I
      DO I=1,10000
         INDEX(I) = MOD(I, 1000) + 1
         Y(I) = 0.0
      ENDDO
      DO I = 1,1000
         X(I) = 0.0
      ENDDO
      CALL SUB(X, Y, INDEX, 10000)
   END PROGRAM ATOMIC_DEMO
```

The following non-conforming example demonstrates the restriction on the ATOMIC construct:

```
   SUBROUTINE ATOMIC_INCORRECT()
      INTEGER:: I
```

```
      REAL:: R
      EQUIVALENCE(I,R)
!$OMP PARALLEL
!$OMP ATOMIC UPDATE
      I = I + 1
!$OMP ATOMIC UPDATE
      R = R + 1.0
! The above is incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL
   END SUBROUTINE ATOMIC_INCORRECT
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
CRITICAL
ORDERED
BARRIER
FLUSH
REDUCTION Clause
Parallel Processing Model for information about Binding Sets

ATOMIC_ADD

Atomic Intrinsic Subroutine (Generic): *Performs atomic addition.*

Syntax

`CALL ATOMIC_ADD (atom, value [, stat])`

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of *atom* + *value* if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* and *value* + *atom* must be representable as integers with kind ATOMIC_INT_KIND.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_ADD (N[12], 7)
```

If N on image 12 is 4 when this operation is initiated, the value of N on image 12 is defined with the value 11 when the operation is complete and no error occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_AND

Atomic Intrinsic Subroutine (Generic): *Performs atomic bitwise AND.*

Syntax

`CALL ATOMIC_AND (atom, value [, stat])`

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of `iand` (atom, int (value, ATOMIC_INT_KIND)) if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* must be representable as an integer with kind ATOMIC_INT_KIND.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_AND (N[4], 22)
```

If the value of N on image 4 is 29 when this operation is initiated, the value of N on image 4 is 20 when the operation is complete and no error condition occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_CAS

Atomic Intrinsic Subroutine (Generic): *Performs atomic compare and swap.*

Syntax

`CALL ATOMIC_CAS (atom, old, compare, new [, stat])`

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or type logical with kind ATOMIC_LOGICAL_KIND. *atom* becomes undefined if an error occurs. Otherwise, if *atom* is of type integer and equal to *compare*, or of type logical and equivalent to *compare*, it becomes defined with the value *new*. If *atom* is of type logical and has been assigned a value other than .true. or .false., the result is undefined.

ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are named constants in the intrinsic module ISO_FORTRAN_ENV.

old (Output) Must be a scalar of the same type as *atom*. It becomes undefined if an error occurs. Otherwise, it becomes defined with the value *atom* has at the beginning of the atomic operation.

compare (Input) Must be scalar and the same type and kind as *atom*.

new (Input) Must be scalar and the same type and kind as *atom*.

stat (Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_CAS (N[4], I, 8, 10)
```

If the value of N on image 4 is 8 when the atomic operation is initiated, N on image 4 is defined with the value 10, and I is defined with the value 8 when the operation is complete and no error occurs during the subroutine reference. If the value N on image 4 is 13 when the atomic operation is initiated, the value of N is unchanged, and the value of I is 13 when the operation completes and no error condition occurs during the procedure reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_DEFINE

Atomic Intrinsic Subroutine (Generic): *Defines a variable atomically.*

Syntax

```
CALL ATOMIC_DEFINE (atom, value [, stat])
```

atom (Output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND. If its kind is the same as that of value or its type is logical, it becomes defined with the value of *value*. Otherwise, it becomes defined with the value of INT (VALUE, ATOMIC_INT_KIND).

ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are named constants in the intrinsic module ISO_FORTRAN_ENV.

value (Input) Must be a scalar and of the same type as *atom*.

stat (Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_DEFINE (N[9], 7)
```

This causes N on image 9 to become defined with the value 7.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_FETCH_ADD

Atomic Intrinsic Subroutine (Generic): *Performs atomic fetch and addition.*

Syntax

`CALL ATOMIC_FETCH_ADD (atom, value, old [, stat])`

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of *atom* + *value* if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* and *value* + *atom* must be representable as integers with kind ATOMIC_INT_KIND.
old	(Output) Must be a scalar of the same type as *atom*. It becomes undefined if an error occurs. Otherwise, it becomes defined with the value *atom* has at the beginning of the atomic operation.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_FETCH_ADD (N[4], 8, M)
```

If the value of N on image 4 was 7 when the atomic operation is initiated, N on image 4 is defined with the value 15, and M becomes is defined with the value 7 when the operation completes and no error occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_FETCH_AND

Atomic Intrinsic Subroutine (Generic): *Performs atomic fetch and bitwise AND.*

Syntax

CALL ATOMIC_FETCH_AND (*atom*, *value*, *old* [, *stat*])

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of `iand` (atom, int (value, ATOMIC_INT_KIND)) if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* must be representable as an integer with kind ATOMIC_INT_KIND.
old	(Output) Must be a scalar of the same type as *atom*. It becomes undefined if an error occurs. Otherwise, it becomes defined with the value *atom* has at the beginning of the atomic operation.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_FETCH_AND (N[4], 29, M)
```

If the value of N on image 4 was 23 when the atomic operation is initiated, N on image 4 is defined with the value 21, and M is defined with the value 23 when the operation completes and no error occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_FETCH_OR

Atomic Intrinsic Subroutine (Generic): *Performs atomic fetch and bitwise OR.*

Syntax

CALL ATOMIC_FETCH_OR (*atom*, *value*, *old* [, *stat*])

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of `ior` (atom, int (value, ATOMIC_INT_KIND)) if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* must be representable as an integer with kind ATOMIC_INT_KIND.

old	(Output) Must be a scalar of the same type as *atom*. It becomes undefined if an error occurs. Otherwise, it becomes defined with the value *atom* has at the beginning of the atomic operation.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
ATOMIC_FETCH_OR (N[4], 9, M)
```

If the value of N on image 4 is 4 when the atomic operation is initiated, N on image 4 is defined with the value 13, and M is defined with the value 4 when the operation completes and no error occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_FETCH_XOR

Atomic Intrinsic Subroutine (Generic): *Performs atomic fetch and bitwise exclusive OR.*

Syntax

`CALL ATOMIC_FETCH_XOR (atom, value, old [, stat])`

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of `ieor` (atom, int (value, ATOMIC_INT_KIND)) if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* must be representable as an integer with kind ATOMIC_INT_KIND.
old	(Output) Must be a scalar of the same type as *atom*. It becomes undefined if an error occurs. Otherwise, it becomes defined with the value *atom* has at the beginning of the atomic operation.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_FETCH_XOR (N[4], 9, M)
```

If the value of N on image 4 was 10 when the atomic operation is initiated, N on image 4 is defined with the value 3, and M is defined with the value 10 when the operation completes and no error occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_OR

Atomic Intrinsic Subroutine (Generic): *Performs atomic bitwise OR.*

Syntax

```
CALL ATOMIC_OR (atom, value [, stat])
```

atom	(Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of `ior` (atom, int (value, ATOMIC_INT_KIND)) if no error occurs. Otherwise, it becomes undefined. ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.
value	(Input) Must be a scalar integer. The value of *value* must be representable as an integer with kind ATOMIC_INT_KIND.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_OR (N[4], 10)
```

If the value of N on image 4 is 9 when the atomic operation is initiated, the value of N on image 4 is 11 when the operation completes and no error condition occurs during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_REF

Atomic Intrinsic Subroutine (Generic): *Lets you reference a variable atomically.*

Syntax

```
CALL ATOMIC_REF (value, atom [, stat])
```

value (Output) Must be a scalar and of the same type as *atom*. If its kind is the same as that of *atom* or its type is logical, it becomes defined with the value of *atom*. Otherwise, it is defined with the value of INT(ATOM, KIND (VALUE)).

atom (Input) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND.

ATOMIC_INT_KIND and ATOMIC_LOGICAL_KIND are named constants in the intrinsic module ISO_FORTRAN_ENV.

stat (Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_REF (SOL, I [9])
```

This causes SOL to become defined with the value of I on image 9.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATOMIC_XOR

Atomic Intrinsic Subroutine (Generic): *Performs atomic bitwise exclusive OR.*

Syntax

```
CALL ATOMIC_XOR (atom, value [, stat])
```

atom (Input; output) Must be a scalar coarray or coindexed object and of type integer with kind ATOMIC_INT_KIND. It becomes defined with the value of `ieor` (atom, int (value, ATOMIC_INT_KIND)) if no error occurs. Otherwise, it becomes undefined.

ATOMIC_INT_KIND is a named constant in the intrinsic module ISO_FORTRAN_ENV.

value (Input) Must be a scalar integer. The value of *value* must be representable as an integer with kind ATOMIC_INT_KIND.

stat (Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Atomic Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.

Example

Consider the following:

```
CALL ATOMIC_XOR (N[4], 10)
```

If the value of N on image 4 is 14 when the atomic operation is initiated, the value on N on image 4 is the value 4 when the operation is complete and no error condition occurred during the subroutine reference.

See Also

Overview of Atomic Subroutines
ISO_FORTRAN_ENV Module

ATTRIBUTES

General Compiler Directive: *Declares properties for specified variables.*

Syntax

`!DIR$ ATTRIBUTES att[,att]...:: object[,object]...`

att

Is one of the following options (or properties):

ALIAS	DLLEXPORT	OPTIMIZATION_PARAMETER
ALIGN	DLLIMPORT	REFERENCE
ALLOCATABLE	EXTERN	STDCALL
ALLOW_NULL	FORCEINLINE	VALUE
C	IGNORE_LOC	VARYING
CODE_ALIGN	INLINE	VECTOR
CONCURRENCY_SAFE	MIXED_STR_LEN_ARG	
CVF	NO_ARG_CHECK	
DECORATE	NOCLONE	
DEFAULT	NOINLINE	

object

Is the name of a data object or procedure.

The following table shows which ATTRIBUTES options can be used with various objects:

Option	Variable and Array Declarations	Common Block Names [1]	Subprogram Specification and EXTERNAL Statements
ALIAS	No	Yes	Yes
ALIGN	Yes	No	No
ALLOCATABLE	Yes[2]	No	No
ALLOW_NULL	Yes	No	No
C	No	Yes	Yes
CODE_ALIGN	No	No	Yes[5, 6]

Option	Variable and Array Declarations	Common Block Names [1]	Subprogram Specification and EXTERNAL Statements
CONCURRENCY_SAFE	No	No	Yes
CVF	No	Yes	Yes
DECORATE	No	No	Yes
DEFAULT	No	Yes	Yes
DLLEXPORT	Yes[3]	Yes	Yes
DLLIMPORT	Yes	Yes	Yes
EXTERN	Yes	No	No
FASTMEM	Yes	No	No
FORCEINLINE	No	No	Yes
IGNORE_LOC	Yes[4]	No	No
INLINE	No	No	Yes
MIXED_STR_LEN_ARG	No	No	Yes
NO_ARG_CHECK	Yes	No	Yes[5]
NOCLONE	No	No	Yes
NOINLINE	No	No	Yes
OPTIMIZATION_PARAMETER	No	No	Yes[5, 6]
REFERENCE	Yes	No	Yes
STDCALL	No	Yes	Yes
VALUE	Yes	No	No
VARYING	No	No	Yes
VECTOR	No	No	Yes[5]

[1]A common block name is specified as [/]common-block-name[/]

[2]This option can only be applied to arrays.

[3]Module-level variables and arrays only.

[4]This option can only be applied to INTERFACE blocks.

[5]This option cannot be applied to EXTERNAL statements.

[6]This option can be applied to named main programs.

These options can be used in function and subroutine definitions, in type declarations, and with the INTERFACE and ENTRY statements.

Options applied to entities available through use or host association are in effect during the association. For example, consider the following:

```
MODULE MOD1
  INTERFACE
    SUBROUTINE NEW_SUB
    !DIR$ ATTRIBUTES C, ALIAS:'othername' :: NEW_SUB
    END SUBROUTINE
  END INTERFACE
  CONTAINS
    SUBROUTINE SUB2
    CALL NEW_SUB
    END SUBROUTINE
END MODULE
```

In this case, the call to NEW_SUB within SUB2 uses the C and ALIAS options specified in the interface block.

Options C, STDCALL, REFERENCE, VALUE, and VARYING affect the calling conventions of routines:

- You can specify C, STDCALL, REFERENCE, and VARYING for an entire routine.
- You can specify VALUE and REFERENCE for individual arguments.

Examples

```
INTERFACE
   SUBROUTINE For_Sub (I)
     !DIR$ ATTRIBUTES C, ALIAS:'_For_Sub' :: For_Sub
     INTEGER I
   END SUBROUTINE For_Sub
END INTERFACE
```

You can assign more than one option to multiple variables with the same compiler directive. All assigned options apply to all specified variables. For example:

```
!DIR$ ATTRIBUTES REFERENCE, VARYING, C :: A, B, C
```

In this case, the variables `A`, `B`, and `C` are assigned the REFERENCE, VARYING, and C options. The only restriction on the number of options and variables is that the entire compiler directive must fit on one line.

The identifier of the variable or procedure that is assigned one or more options must be a simple name. It cannot include initialization or array dimensions. For example, the following is not allowed:

```
!DIR$ ATTRIBUTES C :: A(10)  ! This is illegal.
```

The following shows another example:

```
SUBROUTINE ARRAYTEST(arr)
!DIR$ ATTRIBUTES DLLEXPORT :: ARRAYTEST
   REAL(4) arr(3, 7)
   INTEGER i, j
   DO i = 1, 3
     DO j = 1, 7
       arr (i, j) = 11.0 * i + j
     END DO
   END DO
END SUBROUTINE
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
Programming with Mixed Languages Overview

ATTRIBUTES ALIAS

The ATTRIBUTES directive option ALIAS specifies an alternate external name to be used when referring to external subprograms.

Syntax

```
!DIR$ ATTRIBUTES ALIAS: external-name:: subprogram
```

external-name — Is a character constant delimited by apostrophes or quotation marks. The character constant is used as is; the string is not changed to uppercase, nor are blanks removed.

subprogram — Is an external subprogram.

The ALIAS option overrides the C (and STDCALL) option. If both C and ALIAS are specified for a subprogram, the subprogram is given the C calling convention, but not the C naming convention. It instead receives the name given for ALIAS, with no modifications.

ALIAS cannot be used with internal procedures, and it cannot be applied to dummy arguments.

The following example gives the subroutine happy the name "_OtherName@4" outside this scoping unit:

```
INTERFACE
  SUBROUTINE happy(i)
    !DIR$ ATTRIBUTES STDCALL, DECORATE, ALIAS:'OtherName' :: happy
    INTEGER i
  END SUBROUTINE
END INTERFACE
```

!DIR$ ATTRIBUTES ALIAS has the same effect as the !DIR$ ALIAS directive.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives
ATTRIBUTES DECORATE

ATTRIBUTES ALIGN

The ATTRIBUTES directive option ALIGN specifies the byte alignment for variables and for allocatable or pointer components of derived types.

Syntax

```
!DIR$ ATTRIBUTES ALIGN: n:: object
```

n — Is the number of bytes for the minimum alignment boundary.

For allocatable *object*s, the boundary value must be a power of 2, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on. *n* must have a value between 1 and 2097152 == 2**21 == 2MB on Linux* and macOS* systems, and between 1 and 8192 == 2**13 == 8KB on Windows* systems.

For non-allocatable *object*s, the boundary value must be a power of 2 between 1 and 64 on Windows systems, between 1 and 65536 == 2**16 == 64KB on Linux systems, or between 1 and 4096 == 2**12 == 4KB on macOS* systems.

object Is the variable or the allocatable or pointer component of a derived type to be aligned.

Objects that can be aligned by this directive include static local variables, automatic variables, module variables, dynamically allocated arrays, allocatable array components of derived types, and the start of common blocks. This directive cannot be used to align variables within common blocks

If you specify directive !DIR$ ATTRIBUTES ALIGN on an object with the ALLOCATABLE or POINTER attribute, an ALLOCATE statement will attempt to use that alignment when the memory is allocated.

For allocatable or pointer components of derived types, the directive must appear within the derived-type TYPE…END TYPE block.

If the TYPE is an extended type, the directive cannot reference a component in the parent type.

Example

Consider the following:

```
TYPE EXAMPLE
!DIR$ ATTRIBUTES ALIGN : 64 :: R_alloc
REAL, ALLOCATABLE :: R_alloc ( : )
REAL :: R_scalar
INTEGER :: I_nonalloc(25)
END TYPE EXAMPLE

TYPE (EXAMPLE) :: MyVar

ALLOCATE (MyVar%R_alloc(1000))   ! Memory is allocated aligned at a 64-byte boundary
```

Note that it is valid to give the ALIGN:64 attribute to component R_alloc, but not to component R_scalar or to component I_nonalloc.

The following example shows that the name of a common block may optionally be enclosed in slashes:

```
!DIR$ ATTRIBUTES ALIGN: n :: /common_name/
```

See Also

ATTRIBUTES
ASSUME_ALIGNED directive
`align` compiler option (see setting array*n*byte)
Syntax Rules for Compiler Directives

ATTRIBUTES ALLOCATABLE

The ATTRIBUTES directive option ALLOCATABLE is provided for compatibility with older programs. It lets you delay allocation of storage for a particular declared entity until some point at run time when you explicitly call a routine that dynamically allocates storage for the entity.

Syntax

`!DIR$ ATTRIBUTES ALLOCATABLE :: entity`

entity Is the name of the entity that should have allocation delayed.

The recommended method for dynamically allocating storage is to use the ALLOCATABLE statement or attribute.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives

ATTRIBUTES ALLOW_NULL

The ATTRIBUTES directive option ALLOW_NULL enables a corresponding dummy argument to pass a NULL pointer (defined by a zero or the NULL intrinsic) by value for the argument.

Syntax

```
!DIR$ ATTRIBUTES ALLOW_NULL :: arg
```

arg	Is the name of the argument.

ALLOW_NULL is only valid if ATTRIBUTES REFERENCE is also specified; otherwise, it has no effect.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives

ATTRIBUTES C and STDCALL

The ATTRIBUTES directive options C and STDCALL specify procedure calling, naming, and argument passing conventions.

Syntax

```
!DIR$ ATTRIBUTES C :: object[, object] ...
!DIR$ ATTRIBUTES STDCALL :: object[, object] ...
```

object	Is the name of a data object or procedure.

On Windows* systems on IA-32 architecture, C and STDCALL have slightly different meanings; on all other platforms, STDCALL is treated as C.

When applied to a subprogram, these options define the subprogram as having a specific set of calling conventions. The effects depend on whether or not the subprogram is interoperable (has the BIND attribute).

For interoperable subprograms with the BIND attribute, ATTRIBUTES STDCALL has the following effects for subprograms in applications targeting Windows systems on IA-32 architecture:

- The STDCALL calling convention is used where the called subprogram cleans up the stack at exit.
- The external name has @*n* appended, where *n* is the number of bytes of arguments pushed on the stack.

No other effects of ATTRIBUTES STDCALL are applied for interoperable subprograms. If pass-by-value is desired for a dummy argument to an interoperable subprogram, the Fortran standard VALUE attribute should be specified for that argument.

For platforms other than Windows systems on IA-32 architecture, ATTRIBUTES STDCALL has no effect on interoperable subprograms. You should not specify ATTRIBUTES C for interoperable subprograms.

The following table and subsequent text summarizes the differences between the calling conventions for subprograms that are not interoperable:

Convention	C [1]	STDCALL [1]	Default [2]
Arguments passed by value	Yes	Yes	No

Convention	C [1]	STDCALL [1]	Default [2]
Case of external subprogram names	L*X, M*X: Lowercase W*S: Lowercase	L*X, M*X: Lowercase W*S: Lowercase	L*X, M*X: Lowercase W*S: Uppercase
L*X, M*X only:			
Trailing underscore added	No	No	Yes[3]
M*X only:			
Leading underscore added	No	No	Yes
W*S only:			
Leading underscore added	Yes	Yes	Yes[4]
Number of argument bytes added to name	No	Yes	No
Caller stack cleanup	Yes	No	Yes
Variable number of arguments	Yes	No	Yes

[1]STDCALL is treated as C on Linux*, macOS*, and on Windows* on Intel® 64 architecture.

[2]The Intel® Fortran calling convention

[3]On Linux, if there are one or more underscores in the external name, two trailing underscores are added; if there are no underscores, one is added.

[4]IA-32 architecture only

If C or STDCALL is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran conventions pass arguments by reference.

On IA-32 architecture, an underscore (_) is placed at the beginning of the external name of a subprogram. If STDCALL is specified, an at sign (@) followed by the number of argument bytes being passed is placed at the end of the name. For example, a subprogram named SUB1 that has three INTEGER(4) arguments and is defined with STDCALL is assigned the external name _sub1@12.

Character arguments are passed as follows:

- By default, hidden lengths are put at the end of the argument list.

  On Windows* systems using IA-32 architecture, you can get Compaq* Visual Fortran default behavior by specifying compiler option iface.
- If C or STDCALL (only) is specified:

  On all systems, the first character of the string is passed (and padded with zeros out to INTEGER(4) length).
- If C or STDCALL is specified, and REFERENCE is specified for the argument:

  On all systems, the string is passed with no length.
- If C or STDCALL is specified, and REFERENCE is specified for the routine (but REFERENCE is *not* specified for the argument, if any):

  On all systems, the string is passed with the length.

See Also

ATTRIBUTES
REFERENCE
BIND
Syntax Rules for Compiler Directives
iface compiler option

Compiler Reference section: *Mixed Language Programming*

ATTRIBUTES CODE_ALIGN

The ATTRIBUTES directive option CODE_ALIGN specifies the byte alignment for a procedure.

Syntax

`!DIR$ ATTRIBUTES CODE_ALIGN: n:: procedure-name`

n	Is the number of bytes for the minimum alignment boundary. It must be a power of 2 between 1 and 4096, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on. If you specify 1 for *n*, no alignment is performed. If you do not specify *n*, the default alignment is 16 bytes.
procedure-name	Is the name of a procedure.

This directive can be affected by compiler option `-falign-loops` (Linux* and macOS*) or `/Qalign-loops` (Windows*), the CODE_ALIGN directive, and the CODE_ALIGN attribute.

If code is compiled with the `-falign-loops=m` (Linux and macOS*) or `/Qalign-loops:m` (Windows) option and a procedure has the CODE_ALIGN:k attribute, the procedure is aligned on a MAX (m, k) byte boundary. If a procedure has the CODE_ALIGN:k attribute and a CODE_ALIGN:n directive precedes a loop, then both the procedure and the loop are aligned on a MAX (k, n) byte boundary.

Example

Consider the following code fragment in file `test_align.f90`:

```
FUNCTION F ()
!DIR$ ATTRIBUTES CODE_ALIGN:32 :: F
…
!DIR$ CODE_ALIGN:16
DO J = 1, N
…
END DO
…
END FUNCTION F
```

Compiling `test_align.f90` with option `-falign-loops=64` (Linux and macOS*) or `/Qalign-loops:64` (Windows) aligns the function F and the DO J loop on 64-byte boundaries.

See Also

ATTRIBUTES
CODE_ALIGN directive
falign-loops, Qalign-loops compiler option
Syntax Rules for Compiler Directives

ATTRIBUTES CONCURRENCY_SAFE

The ATTRIBUTES directive option CONCURRENCY_SAFE specifies that there are no unacceptable side effects and no illegal (or improperly synchronized) memory access interferences among multiple invocations of a routine or between an invocation of the specified routine and other statements in the program if they were executed concurrently.

Syntax

`!DIR$ ATTRIBUTES CONCURRENCY_SAFE [: clause] :: routine-name-list`

clause	Is one of the following: • `profitable` • `cost` (*int-cycle-count*)
int-cycle-count	Is an integer scalar constant expression.
routine-name-list	Is a comma-separated list of function and subroutine names.

When a CONCURRENCY_SAFE routine is called from parallelized code, you can ignore assumed cross-block or cross-iteration dependencies and side effects of calling the specified routine from parallelized code.

The `profitable` clause indicates that the loops or blocks that contain calls to the routine can be safely executed in parallel if the loop or blocks are legal to be parallelized; that is, if it is profitable to parallelize them.

The `cost` clause indicates the execution cycles of the routine where the compiler can perform parallelization profitability analysis while compiling its enclosing loops or blocks.

The attribute can appear in the declaration of the routine; for example:

```
function f(x)
!DIR$ attributes concurrency_safe :: f
```

The attribute can also appear in the code of the caller; for example:

```
main m
integer f
external f
!dir$ attributes concurrency_safe :: f   ! or it could be in an interface block describing f
...
Print *, f(x)
```

NOTE
For every routine named in *routine-name-list*, you should ensure that any possible side effects are acceptable or expected, and the memory access interferences are properly synchronized.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives

ATTRIBUTES CVF

The ATTRIBUTES direction option CVF tells the compiler to use calling conventions compatible with Compaq Visual Fortran and Microsoft Fortran PowerStation.*

Syntax

```
!DIR$ ATTRIBUTES CVF :: object[, object] ...
```

object Is the name of a data object or procedure.

The conventions that are used are as follows:

- The calling mechanism: STDCALL on Windows* systems using IA-32 architecture
- The argument passing mechanism: by reference
- Character-length argument passing: following the argument address
- The external name case: uppercase
- The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 architecture. On Windows* systems using IA-32 architecture, @n suffix where *n* is the number of bytes to be removed from the stack on exit from the procedure. No suffix on other systems.

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

ATTRIBUTES DECORATE

The ATTRIBUTES directive option DECORATE specifies that the external name used in !DIR$ ALIAS or !DIR$ ATTRIBUTES ALIAS should have the prefix and postfix decorations performed on it that are associated with the platform and calling mechanism that is in effect. These are the same decorations performed on the procedure name when ALIAS is not specified, except that, on Linux and macOS* systems, DECORATE does not add a trailing underscore signifying a Fortran procedure.*

Syntax

```
!DIR$ ATTRIBUTES DECORATE :: exname
```

exname Is an external name. It may not be the name of an internal procedure.

The case of the ALIAS external name is not modified.

If ALIAS is not specified, this option has no effect.

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

ATTRIBUTES ALIAS

The summary of prefix and postfix decorations in the description of the ATTRIBUTES options C and STDCALL

ATTRIBUTES DEFAULT

The ATTRIBUTES directive option DEFAULT overrides certain compiler options that can affect external routine and COMMON block declarations.

Syntax

```
c!DIR$ ATTRIBUTES DEFAULT :: entity
```

entity	Is an external procedure, a COMMON block, a module variable that is initialized, or a PARAMETER in a module.

It specifies that the compiler should ignore compiler options that change the default conventions for external symbol naming and argument passing for routines and COMMON blocks (such as names, assume underscore, assume 2underscores on Linux systems, and iface on Windows* systems).

This option can be combined with other ATTRIBUTES options, such as STDCALL, C, REFERENCE, ALIAS, etc. to specify properties different from the compiler defaults.

This option is useful when declaring INTERFACE blocks for external routines, since it prevents compiler options from changing calling or naming conventions.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives
iface compiler option
names compiler option
assume compiler option

ATTRIBUTES DLLEXPORT and DLLIMPORT

The ATTRIBUTES directive options DLLEXPORT and DLLIMPORT define a dynamic-link library's interface for processes that use them. The options can be assigned to module variables, COMMON blocks, and procedures. These directive options are available on Windows and macOS* systems.*

Syntax

```
!DIR$ ATTRIBUTES DLLEXPORT :: object[, object] ...
!DIR$ ATTRIBUTES DLLIMPORT :: object[, object] ...
```

object	Is the name of a module variable, COMMON block, or procedure. The name of a COMMON block must be enclosed in slashes.

DLLEXPORT and DLLIMPORT define the interface for the following dynamic-link libraries:

- DLL on Windows*
- DYLIB on macOS*

DLLEXPORT specifies that procedures or data are being exported to other applications or dynamic libraries. This causes the compiler to produce efficient code; for example, eliminating the need on Windows systems for a module definition (.def) file to export symbols.

DLLEXPORT should be specified in the routine to which it applies. If the routine's implementation is in a submodule, specify DLLEXPORT in the parent module's INTERFACE block for the routine. If MODULE PROCEDURE is used in the submodule, the DLLEXPORT attribute will be inherited; otherwise you must also specify DLLEXPORT in the submodule routine.

Symbols defined in a DLL are imported by programs that use them. On Windows*, the program must link with the DLL import library (.lib).

The DLLIMPORT option is used inside the program unit that imports the symbol. DLLIMPORT is specified in a declaration, not a definition, since you cannot define a symbol you are importing.

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

ATTRIBUTES EXTERN

The ATTRIBUTES directive option EXTERN specifies that a variable is allocated in another source file. EXTERN can be used in global variable declarations, but it must not be applied to dummy arguments.

Syntax

```
!DIR$ ATTRIBUTES EXTERN :: var
```

var	Is the variable to be allocated.

This option must be used when accessing variables declared in other languages.

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE

The ATTRIBUTES directive options INLINE, NOINLINE, and FORCEINLINE can be used to control inlining decisions made by the compiler. You should place the directive option in the procedure whose inlining you want to influence.

Syntax

The INLINE option specifies that a function or subroutine can be inlined. The inlining can be ignored by the compiler if inline heuristics determine it may have a negative impact on performance or will cause too much of an increase in code size.

```
!DIR$ ATTRIBUTES INLINE :: procedure
```

procedure	Is the function or subroutine that can be inlined.

The NOINLINE option disables inlining of a function.

```
!DIR$ ATTRIBUTES NOINLINE :: procedure
```

procedure	Is the function or subroutine that must not be inlined.

The FORCEINLINE option specifies that a function or subroutine must be inlined unless it will cause errors.

```
!DIR$ ATTRIBUTES FORCEINLINE :: procedure
```

procedure	Is the function or subroutine that must be inlined.

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

ATTRIBUTES IGNORE_LOC

The ATTRIBUTES directive option IGNORE_LOC enables %LOC to be stripped from an argument.

Syntax

```
!DIR$ ATTRIBUTES IGNORE_LOC :: arg
```

arg	Is the name of an argument.

IGNORE_LOC is only valid if ATTRIBUTES REFERENCE is also specified; otherwise, it has no effect.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives

ATTRIBUTES MIXED_STR_LEN_ARG and NOMIXED_STR_LEN_ARG

These ATTRIBUTES directive options specify where hidden lengths for character arguments and character-valued functions should be placed.
MIXED_STR_LEN_ARG specifies that hidden lengths for character arguments and character-valued functions should be placed immediately following the argument address in the argument list.
NOMIXED_STR_LEN_ARG specifies that these hidden lengths should be placed in sequential order at the end of the argument list.

Syntax

```
!DIR$ ATTRIBUTES MIXED_STR_LEN_ARG :: args
```

```
!DIR$ ATTRIBUTES NOMIXED_STR_LEN_ARG :: args
```

args	Is a list of arguments.

The default is NOMIXED_STR_LEN_ARG. However, If you specify compiler option `/iface:CVF` or `/iface:mixed_str_len_arg` (Windows*), or compiler option `-mixed-str-len-arg` (Linux* and macOS*), the default is MIXED_STR_LEN_ARG.

See Also

ATTRIBUTES directive
Syntax Rules for Compiler Directives
iface compiler option

ATTRIBUTES NO_ARG_CHECK

The ATTRIBUTES directive option NO_ARG_CHECK specifies that type and shape matching rules related to explicit interfaces are to be ignored. This permits the construction of an INTERFACE block for an external procedure or a module procedure that accepts an argument of any type or shape; for example, a memory copying routine.

Syntax

```
!DIR$ ATTRIBUTES NO_ARG_CHECK :: object
```

object Is the name of an argument or procedure.

NO_ARG_CHECK can appear only in an INTERFACE block for a non-generic procedure or in a module procedure. It can be applied to an individual dummy argument name or to the routine name, in which case the option is applied to all dummy arguments in that interface.

NO_ARG_CHECK *cannot* be used for procedures with the PURE or ELEMENTAL prefix.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives

ATTRIBUTES NOCLONE

The ATTRIBUTES directive option NOCLONE can be used to prevent a procedure from being considered for cloning, which is a mechanism performed by interprocedural constant propagation that produces specialized copies of the procedure.

Syntax

```
!DIR$ ATTRIBUTES NOCLONE :: procedure
```

procedure Is a function or subroutine that can be inlined.

Note that if you specify ATTRIBUTES NOINLINE, it does not prevent this cloning.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives
ATTRIBUTES INLINE

ATTRIBUTES OPTIMIZATION_PARAMETER

The ATTRIBUTES directive option OPTIMIZATION_PARAMETER passes certain information about a procedure or main program to the optimizer.

Syntax

```
!DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER: string::{ procedure-name | named-main-program}
```

string Is a character constant that is passed to the optimizer. The constant must be delimited by apostrophes or quotation marks, and it may have one of the following values:

- `TARGET_ARCH=` *cpu*

  Tells the compiler to generate code specialized for a particular processor. For the list of *cpu*s you can specify, see option `[Q]x`.
- `G2S` = {ON | OFF}

  Disables or enables the use of gather/scatter instructions in the specified program unit.

  ON tells the optimizer to disable the generation of gather/scatter and to transform gather/scatter into unit-strided loads/stores plus a set of shuffles wherever possible.

OFF tells the optimizer to enable the generation of gather/scatter instructions and not to transform gather/scatter into unit-strided loads/stores.

- `INLINE_MAX_PER_ROUTINE=` *n*

  Specifies the maximum number of times the inliner may inline into the procedure. The *n* is one of the following:

  - A non-negative scalar integer constant (>=0) that specifies the maximum number of times the the inliner may inline into the procedure. If you specify zero, no inlining is done into the routine.
  - The keyword UNLIMITED, which means that there is no limit to the number of times the inliner may inline into the procedure.

  For more information, see option `[Q]inline-max-per-routine`.
- `INLINE_MAX_TOTAL_SIZE=` *n*

  Specifies how much larger a routine can normally grow when inline expansion is performed. The *n* is one of the following:

  - A non-negative scalar integer constant (>=0) that specifies the permitted increase in the routine's size when inline expansion is performed. If you specify zero, no inlining is done into the routine.
  - The keyword UNLIMITED, which means that there is no limit to the size a routine may grow when inline expansion is performed.

  For more information, see option `[Q]inline-max-total-size`.

procedure-name Is the name of a procedure.

named-main-program Is the name of a main program

Description

The characters in string can appear in any combination of uppercase and lowercase. The following rules also apply to string:

- If string does not contain an equal sign (=), then the entire value of string is converted to lowercase before being passed to the optimizer.
- If string contains an equal sign, then all characters to the left of the equal sign are converted to lowercase before all of string is passed to the optimizer.

  Characters to the right of the equal sign are not converted to lowercase since their value may be case sensitive to the optimizer, for example “target_arch=AVX”.

You can specify multiple ATTRIBUTES OPTIMIZATION_PARAMETER directives for one procedure or one main program.

For the named procedure or main program, the values specified for ATTRIBUTES OPTIMIZATION_PARAMETER override any settings specified for the following compiler options:

- `[Q]x`, `-m`, and `/arch`
- `[Q]inline-max-per-routine`
- `[Q]inline-max-total-size`

Example

Consider the two attributes optimization_parameter directives in the following code:

```
function f (x)
!dir$ attributes optimization_parameter: "inline_max_per_routine=10" :: f
!dir$ attributes optimization_parameter: "inline_max_total_size=2000" :: f
real :: f, x
…
```

The two directives have the same effect as if the function F had been complied with "/Qinline-max-per-routine:10 /Qinline-max-total-size:2000" on Windows* or with "-inline-max-per-routine=10 -inline-max-total-size=2000" on Linux* or macOS*, that is, inlining will not increase the size of F by more than 2000 and the inliner will not inline routines into F more than 10 times.

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

x, Qx compiler option

inline-max-per-routine, Qinline-max-per-routine compiler option

inline-max-total-size, Qinline-max-total-size compiler option

ATTRIBUTES REFERENCE and VALUE

The ATTRIBUTES directive options REFERENCE and VALUE specify how a dummy argument is to be passed.

Syntax

!DIR$ ATTRIBUTES REFERENCE :: *arg*

!DIR$ ATTRIBUTES VALUE :: *arg*

arg Is the name of a dummy argument.

REFERENCE specifies a dummy argument's memory location is to be passed instead of the argument's value.

VALUE specifies a dummy argument's value is to be passed instead of the argument's memory location.

When VALUE is specified for a dummy argument, the actual argument passed to it can be of a different type. If necessary, type conversion is performed before the subprogram is called.

When a complex (KIND=4 or KIND=8) argument is passed by value, *two* floating-point arguments (one containing the real part, the other containing the imaginary part) are passed by immediate value.

Character values, substrings, assumed-size arrays, and adjustable arrays cannot be passed by value.

If REFERENCE (only) is specified for a character argument, the string is passed with no length.

If REFERENCE is specified for a character argument, and C (or STDCALL) has been specified for the routine, the string is passed with no length. This is true even if REFERENCE is also specified for the routine.

If REFERENCE and C (or STDCALL) are specified for a routine, but REFERENCE has *not* been specified for the argument, the string is passed with the length.

VALUE is the default if the C or STDCALL option is specified in the subprogram definition.

In the following example integer x is passed by value:

```
      SUBROUTINE Subr (x)
      INTEGER x
!DIR$ ATTRIBUTES VALUE :: x
```

See Also

C and STDCALL
ATTRIBUTES
Syntax Rules for Compiler Directives

Mixed Language Programming: Adjusting Calling Conventions in Mixed-Language Programming Overview

ATTRIBUTES VARYING

The ATTRIBUTES directive option VARYING allows a Fortran routine to call a C/C++ routine with a variable number of arguments.

Syntax

```
!DIR$ ATTRIBUTES VARYING :: var[, var] ...
```

var	Is a variable representing a C/C++ routine that takes a variable number of arguments.

This attribute can be used in an interface block to create an explicit interface for a C/C++ routine or it can be used on a variable declared EXTERN that represents a C/C++ routine. When the routine is called from the Fortran code, a variable number of arguments can be specified.

This attribute cannot be used with a Fortran routine declaration.

If ATTRIBUTES VARYING is specified, the C calling convention must also be used, either implicitly or explicitly.

All actual arguments in the routine call are passed to the called routine, regardless of the number of dummy arguments specified in the interface. If the called routine tries to access a dummy argument that has no matching actual argument, it causes a user error and the program may fail unpredictably.

See Also

ATTRIBUTES
Syntax Rules for Compiler Directives

ATTRIBUTES VECTOR

The ATTRIBUTES directive option VECTOR tells the compiler to vectorize the specified function or subroutine.

Syntax

```
!DIR$ ATTRIBUTES [att,] VECTOR [:clause] [, att]... :: routine-name
!DIR$ ATTRIBUTES [att,] VECTOR :(clause [, clause]...) [, att] :: routine-name
```

att	Is an ATTRIBUTES directive option. For a list of possible directive options, see the description of argument *att* in ATTRIBUTES.
clause	Is one or more of the following optional clauses: • LINEAR (*var*1:*step*1 [, *var*2:*step*2]...)

var	Is a scalar variable that is a dummy argument in the specified routine.
step	Is a compile-time positive, integer constant expression.

Tells the compiler that for each consecutive invocation of the routine in a serial execution, the value of *var*1 is incremented by *step*1, *var*2 is incremented by *step*2, and so on.

If more than one *step* is specified for a particular *var*, a compile-time error occurs.

Multiple LINEAR clauses are merged as a union.

- `[NO]MASK`

  Determines whether the compiler generates a masked vector version of the routine.
- PROCESSOR (cpuid)
- `UNIFORM` (*arg* [, *arg*]…)

arg	Is a scalar variable that is a dummy argument in the specified routine.

  Tells the compiler that the values of the specified arguments can be broadcasted to all iterations as a performance optimization.

  Multiple UNIFORM clauses are merged as a union.
- `VECTORLENGTH` (*n*[, *n*]…)

n	Is a vector length (VL). It must be an integer, scalar constant expression that is a power of 2; the value must be 2, 4, 8, or 16. If you specify more than one n, the compiler will choose the VL from the values specified.

  Tells the compiler that each routine invocation at the call site should execute the computation equivalent to n times the scalar function execution.

  The VECTORLENGTH and VECTORLENGTHFOR clauses are mutually exclusive. You cannot use the VECTORLENGTH clause with the VECTORLENGTHFOR clause, and vice versa.

  Multiple VECTORLENGTH clauses cause a syntax error.
- `VECTORLENGTHFOR` (*data-type*)

  data-type Is one of the following intrinsic data types:

Data Type	Fortran Intrinsic Type
INTEGER	Default INTEGER
INTEGER(1)	INTEGER (KIND=1)
INTEGER(2)	INTEGER (KIND=2)
INTEGER(4)	INTEGER (KIND=4)
INTEGER(8)	INTEGER (KIND=8)

Data Type	Fortran Intrinsic Type
REAL	Default REAL
REAL(4)	REAL (KIND=4)
REAL(8)	REAL (KIND=8)
COMPLEX	Default COMPLEX
COMPLEX(4)	COMPLEX (KIND=4)
COMPLEX(8)	COMPLEX (KIND=8)

Causes each iteration in the vector loop to execute the computation equivalent to *n* iterations of scalar loop execution where *n* is computed from `size_of_vector_register/ sizeof(data_type)`.

For example, VECTORLENGTHFOR (REAL (KIND=4)) results in *n*=4 for SSE2 to SSE4.2 targets (packed float operations available on 128-bit XMM registers) and *n*=8 for AVX target (packed float operations available on 256-bit YMM registers). VECTORLENGTHFOR(INTEGER (KIND=4)) results in *n*=4 for SSE2 to AVX targets.

The VECTORLENGTHFOR and VECTORLENGTH clauses are mutually exclusive. You cannot use the VECTORLENGTHFOR clause with the VECTORLENGTH clause, and vice versa.

Multiple VECTORLENGTHFOR clauses cause a syntax error.

Without explicit VECTORLENGTH and VECTORLENGTHFOR clauses, the compiler will choose a VECTORLENGTH using its own cost model. Misclassification of variables into PRIVATE, FIRSTPRIVATE, LASTPRIVATE, LINEAR, and REDUCTION, or the lack of appropriate classification of variables, may lead to unintended consequences such as runtime failures and/or incorrect results.

routine-name Is the name of a routine (a function or subroutine). It must be the enclosing routine or the routine immediately following the directive.

If you specify more than one clause, they must be separated by commas and enclosed in parentheses.

When you specify the ATTRIBUTES VECTOR directive, the compiler provides data parallel semantics by combining with the vectorized operations or loops at the call site. When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced. If you specify one or more clauses, they affect the data parallel semantics provided by the compiler.

If you specify the ATTRIBUTES VECTOR directive with no VECTORLENGTH clause, a default VECTORLENGTH is computed based on efficiency heuristics of the vectorizer and the following:

- The return type of the function, if the function has a return type.
- The data type of the first non-scalar argument (that is, the first argument that is specified in the scalar clause), if any.
- Default integer type, if neither of the above is supplied.

If you do not explicitly specify a VECTORLENGTH clause, the compiler will choose a VECTORLENGTH using its own cost model.

If you specify the ATTRIBUTES VECTOR directive with no clause, the compiler will generate vector code based on compiler efficiency heuristics and whatever processor compiler options are specified.

The VECTOR attribute implies the C attribute, so that when you specify the VECTOR attribute on a routine, the C attribute is automatically also set on the same routine. This changes how the routine name is decorated and how arguments are passed.

NOTE
You should ensure that any possible side effects for the specified routine-name are acceptable or expected, and the memory access interferences are properly synchronized.

The Fortran Standard keyword ELEMENTAL specifies that a procedure written with scalar arguments can be extended to conforming array arguments by processing the array elements one at a time in any order. The ATTRIBUTES VECTOR directive tells the optimizer to produce versions of the procedure *routine-name* that execute with contiguous slices of the array arguments as defined by the VECTORLENGTH clause in an "elemental" fashion. *routine-name* does not need to be defined as ELEMENTAL to be given the VECTOR attribute.

The VECTOR attribute causes the compiler to generate a short vector form of the procedure, which can perform the procedure's operation on multiple elements of its array arguments in a single invocation. The short vector version may be able to perform multiple operations as fast as the regular implementation performs a single operation by using the vector instruction set in the CPU.

In addition, when invoked from an OMP construct, the compiler may assign different copies of the elemental procedures to different threads, executing them concurrently. The end result is that your data parallel operation executes on the CPU using both the parallelism available in the multiple cores and the parallelism available in the vector instruction set. If the short vector procedure is called inside a parallel loop or an auto-parallelized loop that is vectorized, you can achieve both vector-level and thread-level parallelism.

The INTENT(OUT) or INTENT(INOUT) attribute is not allowed for arguments of a procedure with the VECTOR attribute since the VECTOR attribute forces the procedure to receive its arguments by value.

The Intel C/C++ compiler built in function __intel_simd_lane() may be helpful in removing certain performance penalties caused by non-unit stride vector access. Consider the following:

```
interface
! returns a number between 0 and vectorlength – 1 that reflects the current “lane id” within the
SIMD vector
! __intel_simd_lane() will return zero if the loop is not vectorized
    function for_simd_lane () bind (C, name = “__intel_simd_lane”)
        integer (kind=4) :: for_simd_lane
        !DEC$ attributes known_intrinsic, default :: for_simd_lane
    end function for_simd_lane
end interface
```

For more details, see the Intel C++ documentation.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Example

The ATTRIBUTES VECTOR directive must be accessible in the caller, either via an INTERFACE block or by USE association.

The following shows an example of an external function with an INTERFACE block:

```
!... function definition

  function f(x)
 !dir$ attributes vector :: f
  real :: f, x
...
 !  attribute vector explicit in calling procedure using an INTERFACE

interface
   function f(x)
  !dir$ attributes vector :: f
  real :: f, x
  end
end interface
...
do i=1,n
  z(i) = f( x(i) )
end do
```

The ATTRIBUTES VECTOR directive can be brought into the caller by USE association if the vector function is a module procedure; for example:

```
!  attribute vector in definition of module procedure

module use_vect

contains
  function f(x)
  !dir$ attributes vector :: f
    real :: f, x
...
  end function
end module use_vect

!  USE and call of f(x) from another procedure with a module USE statement

 USE use_vect     !brings in ATTRIBUTE VECTOR for f(x)
...

! now simply call f(x)
```

```
do i=1,n
  z(i) = f( x(i) )
end do
```

You can specify more than one SCALAR or LINEAR clause in an ATTRIBUTES VECTOR directive. For example, all of the following are valid:

```
!DIR$ ATTRIBUTES VECTOR:PROCESSOR(atom) :: f
!DIR$ ATTRIBUTES VECTOR:(SCALAR(a), SCALAR(b)) :: f
!DIR$ ATTRIBUTES VECTOR:(LINEAR(x:1), LINEAR(y:1)) :: f
```

The three directives above are equivalent to specifying a single, continued, directive in fixed-form source, as follows:

```
!DIR$ ATTRIBUTES VECTOR:( PROCESSOR(atom),
!DIR$& SCALAR(a, b),
!DIR$& LINEAR(x:1, y:1) ) :: f
```

See Also

ATTRIBUTES

Syntax Rules for Compiler Directives

AUTOAddArg (W*S)

***AUTO Subroutine**: Passes an argument name and value and adds the argument to the argument list data structure.*

Module

USE IFAUTO

USE IFWINTY

Syntax

CALL AUTOAddArg (*invoke_args*,*name*,*value*[,*intent_arg*][,*type*])

invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).
name	The argument's name of type CHARACTER*(*).
value	The argument's value. Must be of type INTEGER(1), INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), CHARACTER*(*), or a single dimension array of one of these types. Can also be of type VARIANT, which is defined in the IFWINTY module.
intent_arg	Indicates the intended use of the argument by the called method. Must be one of the following constants defined in the IFAUTO module: • AUTO_ARG_IN: The argument's value is read by the called method, but not written. This is the default value if *intent_arg* is not specified. • AUTO_ARG_OUT: The argument's value is written by the called method, but not read. • AUTO_ARG_INOUT: The argument's value is read and written by the called method.

When the value of *intent_arg* is AUTO_ARG_OUT or AUTO_ARG_INOUT, the variable used in the *value* argument should be declared using the VOLATILE attribute. This is because the value of the variable will be changed by the subsequent call to AUTOInvoke. The compiler's global optimizations need to know that the value can change unexpectedly.

type

The variant type of the argument. Must be one of the following constants defined in the IFWINTY module:

VARIANT Type	Value Type
VT_I1	INTEGER(1)
VT_I2	INTEGER(2)
VT_I4	INTEGER(4)
VT_R4	REAL(4)
VT_R8	REAL(8)
VT_CY	REAL(8)
VT_DATE	REAL(8)
VT_BSTR	CHARACTER*(*)
VT_DISPATCH	INTEGER(4)
VT_ERROR	INTEGER(4)
VT_BOOL	LOGICAL(2)
VT_VARIANT	TYPE(VARIANT)
VT_UNKNOWN	INTEGER(4)

Example

See the example in COMInitialize.

AUTOAllocateInvokeArgs (W*S)

***AUTO Function**: Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.*

Module

USE IFAUTO

Syntax

```
result = AUTOAllocateInvokeArgs( )
```

Results

The value returned is an argument list data structure of type INTEGER(INT_PTR_KIND()).

Example

See the example in COMInitialize.

AUTODeallocateInvokeArgs (W*S)

***AUTO Subroutine**: Deallocates an argument list data structure.*

Module

USE IFAUTO

Syntax

`CALL AUTODeallocateInvokeArgs (invoke_args)`

invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Example

See the example in COMInitialize.

AUTOGetExceptInfo (W*S)

***AUTO Subroutine**: Retrieves the exception information when a method has returned an exception status.*

Module

USE IFAUTO

Syntax

`CALL AUTOGetExceptInfo (invoke_args,code,source,description,h_file,h_context,scode)`

invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).
code	An output argument that returns the error code. Must be of type INTEGER(2).
source	An output argument that returns a human-readable name of the source of the exception. Must be of type CHARACTER*(*).
description	An output argument that returns a human-readable description of the error. Must be of type CHARACTER*(*).
h_file	An output argument that returns the fully qualified path of a Help file with more information about the error. Must be of type CHARACTER*(*).
h_context	An output argument that returns the Help context of the topic within the Help file. Must be of type INTEGER(4).
scode	An output argument that returns an SCODE describing the error. Must be of type INTEGER(4).

AUTOGetProperty (W*S)

***AUTO Function**: Passes the name or identifier of the property and gets the value of the automation object's property.*

Module

USE IFAUTO

USE IFWINTY

Syntax

```
result = AUTOGetProperty (idispatch,id,value[,type])
```

idispatch — The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

id — The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

value — An output argument that returns the argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types.

type — The variant type of the requested argument. Must be one of the following constants defined in the IFWINTY module:

VARIANT Type	Value Type
VT_I2	INTEGER(2)
VT_I4	INTEGER(4)
VT_R4	REAL(4)
VT_R8	REAL(8)
VT_CY	REAL(8)
VT_DATE	REAL(8)
VT_BSTR	CHARACTER*(*)
VT_DISPATCH	INTEGER(4)
VT_ERROR	INTEGER(4)
VT_BOOL	LOGICAL(2)
VT_UNKNOWN	INTEGER(4)

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOGetPropertyByID (W*S)

***AUTO Function**: Passes the member ID of the property and gets the value of the automation object's property into the argument list's first argument.*

Module

USE IFAUTO

Syntax

```
result = AUTOGetPropertyByID (idispatch,memid,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
memid	Member ID of the property. Must be of type INTEGER(4).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOGetPropertyInvokeArgs (W*S)

***AUTO Function**: Passes an argument list data structure and gets the value of the automation object's property specified in the argument list's first argument.*

Module

USE IFAUTO

Syntax

```
result = AUTOGetPropertyInvokeArgs (idispatch,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(INT_PTR_KIND()).

AUTOInvoke (W*S)

***AUTO Function**: Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.*

Module

USE IFAUTO

Syntax

```
result = AUTOInvoke (idispatch,id,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
id	The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

Example

See the example in COMInitialize.

AUTOMATIC

Statement and Attribute: *Controls the storage allocation of variables in subprograms (as does STATIC). Variables declared as AUTOMATIC and allocated in memory reside in the stack storage area, rather than at a static memory location.*

Syntax

The AUTOMATIC attribute can be specified in a type declaration statement or an AUTOMATIC statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] AUTOMATIC [, att-ls] :: v[, v] ...
```

Statement:

```
AUTOMATIC [::] v[, v] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
v	Is the name of a variable or an array specification. It can be of any type.

AUTOMATIC declarations only affect how data is allocated in storage.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

Automatic variables can reduce memory use because only the variables currently being used are allocated to memory.

Automatic variables allow possible recursion. With recursion, a subprogram can call itself (directly or indirectly), and resulting values are available upon a subsequent call or return to the subprogram. For recursion to occur, RECURSIVE must be specified in one of the following ways:

- As a keyword in a FUNCTION or SUBROUTINE statement
- As a compiler option
- As an option in an OPTIONS statement

By default, the compiler allocates local scalar variables on the stack. Other non-allocatable variables of non-recursive subprograms are allocated in static storage by default. This default can be changed through compiler options. Appropriate use of the SAVE attribute may be required if your program assumes that local variables retain their definition across subprogram calls.

To change the default for variables, specify them as AUTOMATIC or specify RECURSIVE (in one of the ways mentioned above).

To override any compiler option that may affect variables, explicitly specify the variables as AUTOMATIC.

NOTE
Variables that are data-initialized, and variables in COMMON and SAVE statements are always static. This is regardless of whether a compiler option specifies recursion.

A variable cannot be specified as AUTOMATIC more than once in the same scoping unit.

If the variable is a pointer, AUTOMATIC applies only to the pointer itself, not to any associated target.

Some variables cannot be specified as AUTOMATIC. The following table shows these restrictions:

Variable	AUTOMATIC
Dummy argument	No
Automatic object	No
Common block item	No
Use-associated item	No
Function result	No
Component of a derived type	No

If a variable is in a module's outer scope, it *cannot* be specified as AUTOMATIC.

Use the `heap-arrays` compiler option to avoid stack overflows at run-time by placing large automatic arrays in the heap instead of on the stack.

Example

The following example shows a type declaration statement specifying the AUTOMATIC attribute:

```
REAL, AUTOMATIC :: A, B, C
```

The following example uses an AUTOMATIC statement:

```
...
CONTAINS
 INTEGER FUNCTION REDO_FUNC
   INTEGER I, J(10), K
   REAL C, D, E(30)
   AUTOMATIC I, J, K(20)
   STATIC C, D, E
   ...
 END FUNCTION
...
C       In this example, all variables within the program unit
C       are saved, except for "var1" and "var3". These are
C       explicitly declared in an AUTOMATIC statement, and thus have
C       memory locations on the stack:
```

```
        SUBROUTINE DoIt (arg1, arg2)
        INTEGER(4) arg1, arg2
        INTEGER(4) var1, var2, var3, var4
        SAVE
        AUTOMATIC var1, var3
C       var2 and var4 are saved
```

See Also

heap-arrays compiler option
recursive compiler option
STATIC
SAVE
Type Declarations

Compatible attributes
RECURSIVE
OPTIONS
POINTER
Modules and Module Procedures

AUTOSetProperty (W*S)

***AUTO Function**: Passes the name or identifier of the property and a value, and sets the value of the automation object's property.*

Module

USE IFAUTO

USE IFWINTY

Syntax

`result = AUTOSetProperty (idispatch,id,value[,type])`

idispatch — The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

id — The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

value — The argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types.

type — The variant type of the argument. Must be one of the following constants defined in the IFWINTY module:

VARIANT Type	Value Type
VT_I2	INTEGER(2)
VT_I4	INTEGER(4)
VT_R4	REAL(4)

VARIANT Type	Value Type
VT_R8	REAL(8)
VT_CY	REAL(8)
VT_DATE	REAL(8)
VT_BSTR	CHARACTER*(*)
VT_DISPATCH	INTEGER(4)
VT_ERROR	INTEGER(4)
VT_BOOL	LOGICAL(2)
VT_UNKNOWN	INTEGER(4)

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOSetPropertyByID (W*S)

***AUTO Function**: Passes the member ID of the property and sets the value of the automation object's property into the argument list's first argument.*

Module

USE IFAUTO

Syntax

```
result = AUTOSetPropertyByID (idispatch,memid,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
memid	Member ID of the property. Must be of type INTEGER(4).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOSetPropertyInvokeArgs (W*S)

***AUTO Function**: Passes an argument list data structure and sets the value of the automation object's property specified in the argument list's first argument.*

Module

USE IFAUTO

Syntax

```
result = AUTOSetPropertyInvokeArgs (idispatch,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

BACKSPACE

Statement: *Positions a sequential file at the beginning of the preceding record, making it available for subsequent I/O processing. It takes one of the following forms:*

Syntax

```
BACKSPACE ([UNIT=]io-unit[, ERR=label] [, IOMSG=msg-var] [, IOSTAT=i-var])

BACKSPACE io-unit
```

io-unit	(Input) Is an external unit specifier.
label	Is the label of the branch target statement that receives control if an error occurs.
msg-var	(Output) Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs.
i-var	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

The I/O unit number must specify an open file on disk or magnetic tape.

Backspacing from the current record *n* is performed by rewinding to the start of the file and then performing *n* - 1 successive READs to reach the previous record.

A BACKSPACE statement must not be specified for a file that is open for direct or append access, because *n* is not available to the Fortran I/O system.

BACKSPACE cannot be used to skip over records that have been written using list-directed or namelist formatting.

If a file is already positioned at the beginning of a file, a BACKSPACE statement has no effect.

If the file is positioned between the last record and the end-of-file record, BACKSPACE positions the file at the start of the last record.

Example

```
BACKSPACE 5
BACKSPACE (5)
BACKSPACE lunit
BACKSPACE (UNIT = lunit, ERR = 30, IOSTAT = ios)
```

The following statement repositions the file connected to I/O unit 4 back to the preceding record:

```
BACKSPACE 4
```

Consider the following statement:

```
BACKSPACE (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 back to the preceding record. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

See Also

REWIND
ENDFILE
Data Transfer I/O Statements
Branch Specifiers

BADDRESS

Inquiry Intrinsic Function (Generic): *Returns the address of an argument. This function cannot be passed as an actual argument. This function can also be specified as IADDR.*

Syntax

```
result = BADDRESS (x)
```

x	Is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of a statement function. If it is a pointer, it must be defined and associated with a target.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

Example

```
PROGRAM batest
  INTEGER X(5), I
  DO I=1, 5
    PRINT *, BADDRESS(X(I))
  END DO
END
```

BARRIER

OpenMP* Fortran Compiler Directive:
Synchronizes all the threads in a team. It causes each thread to wait until all of the other threads in the team have reached the barrier.

Syntax

```
!$OMP BARRIER
```

The binding thread set for a BARRIER construct is the current team. A barrier region binds to the innermost enclosing parallel region.

Each barrier region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.

The barrier region must also be encountered in the same order by all threads in a team.

Example

```
INTEGER K
K = 17

!$OMP PARALLEL SHARED (K) NUM_THREADS (2)
IF (OMP_GET_THREAD_NUM() == 0) THEN
    X = 5
ELSE
    ! The following read access of K creates a race condition
    PRINT *,"1: THREAD# ", OMP_GET_THREAD_NUM (), "K = ", K
ENDIF

! This barrier contains implicit flushes on all threads, as well as a thread
! synchronization: this guarantees that the value 5 will be printed by
! both PRINT 2 and PRINT 3 below.

!$OMP BARRIER

IF (OMP_GET_THREAD_NUM() == 0) THEN
    PRINT *,"2: THREAD# ", OMP_GET_THREAD_NUM (), "K = ", K
ELSE
    PRINT *,"3: THREAD# ", OMP_GET_THREAD_NUM (), "K = ", K
ENDIF
!$OMP END PARALLEL
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Nesting and Binding Rules
Parallel Processing Model for information about Binding Sets

BEEPQQ

Portability Subroutine: *Sounds the speaker at the specified frequency for the specified duration in milliseconds.*

Module

USE IFPORT

Syntax

`CALL BEEPQQ (frequency,duration)`

frequency	(Input) INTEGER(4). Frequency of the tone in Hz.
duration	(Input) INTEGER(4). Length of the beep in milliseconds.

BEEPQQ does not return until the sound terminates.

Example

```
USE IFPORT
INTEGER(4) frequency, duration
frequency = 4000
duration  = 1000
CALL BEEPQQ(frequency, duration)
```

See Also

SLEEPQQ

BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

Portability Functions: *Compute the single-precision values of Bessel functions of the first and second kinds.*

Module

USE IFPORT

Syntax

`result = BESJ0 (value)`

`result = BESJ1 (value)`

`result = BESJN (n, value)`

`result = BESY0 (posvalue)`

`result = BESY1 (posvalue)`

`result = BESYN (n, value)`

value	(Input) REAL(4). Independent variable for a Bessel function.
n	(Input) INTEGER(4). Specifies the order of the selected Bessel function computation.
posvalue	(Input) REAL(4). Independent variable for a Bessel function. Must be greater than or equal to zero.

Results

BESJ0, BESJ1, and BESJN return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

BESY0, BESY1, and BESYN return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

Negative arguments cause BESY0, BESY1, and BESYN to return QNAN.

Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun. Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

See the descriptions of the BESSEL_* functions, if you need to use quad-precision (REAL(16)).

See Also

DBESJ0, DBESJ1, DBESJN

BESSEL_J0

Elemental Intrinsic Function (Generic): *Computes a Bessel function of the first kind, order 0.*

Syntax

```
result = BESSEL_J0 (x)
```

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the Bessel function of the first kind and order zero of *x*.

Example

BESSEL_J0 (1.0) has the approximate value 0.765.

BESSEL_J1

Elemental Intrinsic Function (Generic): *Computes a Bessel function of the first kind, order 1.*

Syntax

```
result = BESSEL_J1 (x)
```

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the Bessel function of the first kind and order 1 of *x*.

Example

BESSEL_J1 (1.0) has the approximate value 0.440.

BESSEL_JN

Elemental and Transformational Intrinsic Functions (Generic): *Compute Bessel functions of the first kind.*

Syntax

```
Elemental function: result = BESSEL_JN (n, x)
```

```
Transformational function: result = BESSEL_JN (n1, n2, x)
```

n, *n1*, *n2* (Input) Must be of type integer and nonnegative.

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*.

The result of BESSEL_JN (*n*, *x*) is scalar. The result value of BESSEL_JN (*n* , *x*) is a processor-dependent approximation to the Bessel function of the first kind and order *n* of *x*.

The result of BESSEL_JN (*n1*, *n2*, *x*) is a rank-one array with extent MAX (*n2* - *n1* + 1, 0). Element *i* of the result value of BESSEL_JN (*n1*, *n2*, *x*) is a processor-dependent approximation to the Bessel function of the first kind and order *n1* + *i* - 1 of *x*.

Example

BESSEL_JN (2, 1.0) has the approximate value 0.115.

Consider the following program Bessel.90:

```
  real :: z (6) = [0:5]/5.          ! 0.0 through 1.0 by 0.2
  print *, z
  print *, bessel_jn (2, 1.0)      ! scalar argument, answer about 0.115
  print *, bessel_jn (1, z)          ! elemental
  print *, bessel_jn (1, 4, 1.0)  ! orders 1 thru 4 on a scalar
  end
```

Compile bessel.f90 and execute the result:

```
> ifort Bessel.f90 -o Bessel
> bessel
```

The above commands produce the following result:

```
  0.0000000E+00  0.2000000      0.4000000      0.6000000      0.8000000      1.000000
  0.1149035
  0.0000000E+00  9.9500835E-02  0.1960266      0.2867010      0.3688421      0.4400506
  0.4400506      0.1149035      1.9563355E-02  2.4766389E-03
```

BESSEL_Y0

Elemental Intrinsic Function (Generic): *Computes a Bessel function of the second kind, order 0.*

Syntax

`result = BESSEL_Y0 (x)`

x (Input) Must be of type real with a value greater than zero.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the Bessel function of the second kind and order zero of *x*.

Example

BESSEL_Y0 (1.0) has the approximate value 0.088.

BESSEL_Y1

Elemental Intrinsic Function (Generic): *Computes a Bessel function of the second kind, order 1.*

Syntax

```
result = BESSEL_Y1 (x)
```

x	(Input) Must be of type real with a value greater than zero.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the Bessel function of the second kind and order 1 of *x*.

Example

BESSEL_Y1 (1.0) has the approximate value -0.781.

BESSEL_YN

Elemental and Transformational Intrinsic Functions (Generic): *Compute Bessel functions of the second kind.*

Syntax

```
Elemental function: result = BESSEL_YN (n, x)
```

```
Transformational function: result = BESSEL_YN (n1, n2, x)
```

n, *n1*, *n2*	(Input) Must be of type integer and nonnegative.
x	(Input) Must be of type real with a value greater than zero.

Results

The result type and kind are the same as *x*.

The result of BESSEL_YN (*n*, *x*) is scalar. The result value of BESSEL_YN (*n* , *x*) is a processor-dependent approximation to the Bessel function of the second kind and order *n* of *x*.

The result of BESSEL_YN (*n1*, *n2*, *x*) is a rank-one array with extent MAX (*n2* - *n1* + 1, 0). Element *i* of the result value of BESSEL_YN (*n1*, *n2*, *x*) is a processor-dependent approximation to the Bessel function of the second kind and order *n1* + *i* - 1 of *x*.

Example

BESSEL_YN (2, 1.0) has the approximate value -1.651.

BGE

Elemental Intrinsic Function (Generic): *Performs a bitwise greater than or equal to on its arguments.*

Syntax

```
result = BGE (i,j)
```

i	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.
j	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.

If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result is true if the sequence of bits represented by *i* is greater than or equal to the sequence of bits represented by *j*, according to the method of bit sequence comparison in Bit Sequence Comparisons; otherwise, the result is false.

The interpretation of a binary, octal, or hexadecimal literal constant as a sequence of bits is described in Binary, Octal, Hexadecimal, and Hollerith Constants.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If BIT_SIZE (J) has the value 8, BGE (Z'FF', J) has the value true for any value of J. BGE (0, -1) has the value false.

See Also

BIT_SIZE

BGT

Elemental Intrinsic Function (Generic): *Performs a bitwise greater than on its arguments.*

Syntax

```
result = BGT (i,j)
```

i	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.
j	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.

If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result is true if the sequence of bits represented by *i* is greater than the sequence of bits represented by *j*, according to the method of bit sequence comparison in Bit Sequence Comparisons; otherwise, the result is false.

The interpretation of a binary, octal, or hexadecimal literal constant as a sequence of bits is described in Binary, Octal, Hexadecimal, and Hollerith Constants.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

BGT (Z'FF', Z'FC') has the value true. BGT (0, -2) has the value false.

BIC, BIS

Portability Subroutines: *Perform a bit-level set and clear for integers.*

Module

USE IFPORT

Syntax

CALL BIC (*bitnum*, *target*)

CALL BIS (*bitnum*, *target*)

bitnum	(Input) INTEGER(4). Bit number to set. Must be in the range 0 (least significant bit) to 31 (most significant bit) if *target* is INTEGER(4). If *target* is INTEGER(8), *bitnum* must be in range 0 to 63.
target	(Input) INTEGER(4) or INTEGER(8). Variable whose bit is to be set.

BIC sets bit *bitnum* of *target* to 0; BIS sets bit *bitnum* to 1.

Example

Consider the following:

```
USE IFPORT
 integer(4) bitnum, target_i4
 integer(8) target_i8
 target_i4 = Z'AAAA'
 bitnum = 1
 call BIC(bitnum, target_i4)
 target_i8 = Z'FFFFFFFF00000000'
 bitnum = 40
 call BIC(bitnum, target_i8)
 bitnum = 0
 call BIS(bitnum, target_i4)
 bitnum = 1
 call BIS(bitnum, target_i8)
 print '(" integer*4 result ",Z)', target_i4
 print '(" integer*8 result ",Z)', target_i8
 end
```

See Also

BIT

BIND

Statement and Attribute: *Specifies that an object is interoperable with C and has external linkage.*

Syntax

The BIND attribute can be specified in a type declaration statement or a BIND statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] BIND (C [, NAME=*ext-name*]) [, *att-ls*] :: *object*

Statement:

BIND (C [, NAME=*ext-name*]) [::] *object*

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.

ext-name	Is a character scalar constant expression that can be used to construct the external name.
object	Is the name of a variable or common block. It can also be the name of an internal procedure if NAME= is not specified.

Description

If a common block is specified in a BIND statement, it must be specified with the same binding label in each scoping unit in which it is declared.

For variables and common blocks, BIND also implies the SAVE attribute, which may be explicitly confirmed with SAVE.

A variable given the BIND attribute (or declared in a BIND statement) must appear in the specification part of a module. You cannot specify BIND for a subroutine local variable or a variable in a main program.

The BIND attribute is similar to directive !DIR$ ATTRIBUTES C as follows:

- The compiler applies the same naming rules, that is, names are lowercase (unless NAME= specifies otherwise).
- The compiler applies the appropriate platform decoration, such as a leading underscore.

However, procedure argument passing differs. When BIND is specified, procedure arguments are passed by reference unless the VALUE attribute is also specified.

The BIND attribute can optionally be used in a PROCEDURE, SUBROUTINE, or FUNCTION declaration. It must be used in an ENUM declaration.

Example

The following example shows the BIND attribute used in a type declaration statement, a statement, and a SUBROUTINE statement.

```
INTEGER, BIND(C) :: SOMEVAR

BIND(C,NAME='SharedCommon') :: /SHAREDCOMMON/

! you need empty parens after the subroutine name if BIND is present
SUBROUTINE FOOBAR() BIND(C, NAME='FooBar')
...
END SUBROUTINE
```

See Also

Modules and Module Procedures

Type Declarations

Compatible attributes

Pointer Assignments

FUNCTION

SUBROUTINE

PROCEDURE

Enumerations and Enumerators (ENUM)

BIND(C)
in mixed language programming

BIT

Portability Function: *Performs a bit-level test for integers.*

Module

USE IFPORT

Syntax

`result = BIT (bitnum, source)`

bitnum	(Input) INTEGER(4). Bit number to test. Must be in the range 0 (least significant bit) to 31 (most significant bit).
source	(Input) INTEGER(4) or INTEGER(8). Variable being tested.

Results

The result type is logical. It is .TRUE. if bit *bitnum* of *source* is 1; otherwise, .FALSE..

See Also

BIC, BIS

BIT_SIZE

Inquiry Intrinsic Function (Generic): *Returns the number of bits in an integer type.*

Syntax

`result = BIT_SIZE (i)`

i	(Input) Must be of type integer or of type logical (which is treated as an integer).

Results

The result is a scalar integer with the same kind parameter as *i*. The result value is the number of bits (*s*) defined by the bit model for integers with the kind parameter of the argument. For information on the bit model, see Model for Bit Data.

Example

BIT_SIZE (1_2) has the value 16 because the KIND=2 integer type contains 16 bits.

See Also

BTEST
IBCLR
IBITS
IBSET

BLE

Elemental Intrinsic Function (Generic): *Performs a bitwise less than or equal to on its arguments.*

Syntax

```
result = BLE (i,j)
```

i	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.
j	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.

If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result is true if the sequence of bits represented by *i* is less than or equal to the sequence of bits represented by *j*, according to the method of bit sequence comparison in Bit Sequence Comparisons; otherwise, the result is false.

The interpretation of a binary, octal, or hexadecimal literal constant as a sequence of bits is described in Binary, Octal, Hexadecimal, and Hollerith Constants.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

BLE (0, J) has the value true for any value of J. BLE (-2, 0) has the value false.

BLOCK

Statement: *Marks the beginning of a BLOCK construct. The BLOCK construct executes a block of statements or constructs that can contain declarations.*

Syntax

```
[name:] BLOCK
   [specification-part]
   block
END BLOCK [name]
```

name	(Optional) Is the name of the BLOCK construct.
specification-part	(Optional) Is one or more specification statements, except for the following: • COMMON • FUNCTION (outside of an INTERFACE block) • EQUIVALENCE • IMPLICIT • INTENT (or its equivalent attribute) • MODULE

- NAMELIST
- OPTIONAL (or its equivalent attribute)
- SUBROUTINE (outside of an INTERFACE block)
- VALUE (or its equivalent attribute)
- Statement functions

block Is a sequence of zero or more statements or constructs, except for the following:

- CONTAINS (outside of a TYPE definition)
- ENTRY
- IMPORT (outside of a TYPE definition)
- Statement functions

Description

A BLOCK construct is itself a scoping unit. Entities declared in a BLOCK construct are local to the BLOCK construct and are accessible only in that construct and in any contained constructs. A local entity in a block construct hides any entity with the same name in its host scope. No transfer of control into a block from outside the block is allowed, except for the return from a procedure call. Transfers within a block or out of the block are allowed.

If a construct name is specified at the beginning of a BLOCK statement, the same name must appear in the corresponding END BLOCK statement. The same construct name must not be used for different named constructs in the same scoping unit. If no name is specified at the beginning of a BLOCK statement, you cannot specify one following the END BLOCK statement.

You can only branch to an END BLOCK statement from within its BLOCK construct.

The SAVE attribute specifies that a local variable of a BLOCK construct retains its association status, allocation status, definition status, and value after termination of the construct unless it is a pointer and its target becomes undefined. If the BLOCK construct contains a SAVE statement, the SAVE statement cannot specify the name of a common block. A SAVE statement outside a BLOCK construct does not affect variables local to the BLOCK construct, because a SAVE statement affects variables in its scoping unit which excludes nested scoping units in it.

The statements specified within the *specification-part* are evaluated in a processor-dependent order, followed by execution of block. When execution exits block, all non-SAVEd automatic and allocatable local variables are deallocated.

Example

The following shows a BLOCK construct:

```
block
  integer :: i
  real :: a(n)
   do i = 1,n
     a(i) = i
   end do
  …
end block
```

When control exits the bottom of the BLOCK, local variables `i` and `a` revert to their meaning outside the block.

The following example shows two nested BLOCK constructs where the inner BLOCK construct has the construct name INNER and the outer one does not have a name:

```
BLOCK
  ...
  INNER: BLOCK
    ...
  END BLOCK INNER
  ...
END BLOCK
```

In the following example, the appearance and the reference of the FORMAT statement are legal:

```
PROGRAM MAIN
  WRITE(6, FMT=10)
  ...
  BLOCK
10   FORMAT("Hello")
  END BLOCK
  ...
END
```

Implicit typing is not affected by BLOCK constructs. In the following example, even if NSQP only appears in the two BLOCK constructs, the scope of NSQP is the whole subroutine S:

```
SUBROUTINE S(N)
  ...
  IF (N>0) THEN
    BLOCK
      NSQP = CEILING(SQRT(DBLE(N)))
    END BLOCK
  END IF
  ...
  IF (N>0) THEN
    BLOCK
      PRINT *,NSQP
    END BLOCK
  END IF
END SUBROUTINE S
```

BLOCK DATA

Statement: *Identifies a block-data program unit, which provides initial values for variables in named common blocks. BLOCK DATA is an obsolescent language feature in Standard Fortran.*

Syntax

```
BLOCK DATA [name]
   [specification-part]
END [BLOCK DATA [name]]
```

name Is the name of the block data program unit.

specification-part Is one or more of the following statements:

COMMON	INTRINSIC	STATIC

DATA	PARAMETER	TARGET
Derived-type definition	POINTER	Type declaration [2]
DIMENSION	RECORD[1]	USE [3]
EQUIVALENCE	Record structure declaration[1]	
IMPLICIT	SAVE	

[1] For more information, see RECORD statement and record structure declarations.

[2] Can only contain attributes: DIMENSION, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC, or TARGET.

[3] Allows access to only named constants.

Description

A block data program unit need not be named, but there can only be one unnamed block data program unit in an executable program.

If a name follows the END statement, it must be the same as the name specified in the BLOCK DATA statement.

An interface block must not appear in a block data program unit and a block data program unit must not contain any executable statements.

If a DATA statement initializes any variable in a named common block, the block data program unit must have a complete set of specification statements establishing the common block. However, all of the variables in the block do not have to be initialized.

A block data program unit can establish and define initial values for more than one common block, but a given common block can appear in only one block data program unit in an executable program.

The name of a block data program unit can appear in the EXTERNAL statement of a different program unit to force a search of object libraries for the block data program unit at link time.

Example

The following shows a block data program unit:

```
BLOCK DATA BLKDAT
  INTEGER S,X
  LOGICAL T,W
  DOUBLE PRECISION U
  DIMENSION R(3)
  COMMON /AREA1/R,S,U,T /AREA2/W,X,Y
  DATA R/1.0,2*2.0/, T/.FALSE./, U/0.214537D-7/, W/.TRUE./, Y/3.5/
END
```

The following shows another example:

```
C      Main Program
       CHARACTER(LEN=10) family
       INTEGER a, b, c, d, e
       REAL X(10), Y(4)
       COMMON/Lakes/a,b,c,d,e,family/Blk2/x,y
       ...
```

```
C      The following block-data subprogram initializes
C      the named common block /Lakes/:
C
       BLOCK DATA InitLakes
       COMMON /Lakes/ erie, huron, michigan, ontario,
      +        superior, fname
       DATA erie, huron, michigan, ontario, superior /1, 2, 3, 4, 5/
       CHARACTER(LEN=10) fname/'GreatLakes'/
       INTEGER erie, huron, michigan, ontario, superior
       END
```

See Also

COMMON
DATA
EXTERNAL
Program Units and Procedures
Obsolescent Language Features in the Fortran Standard

BLOCK_LOOP and NOBLOCK_LOOP

General Compiler Directives: *Enables or disables loop blocking for the immediately following nested DO loops. BLOCK_LOOP enables loop blocking for the nested loops. NOBLOCK_LOOP disables loop blocking for the nested loops.*

Syntax

`!DIR$ BLOCK_LOOP [clause[[,] clause]...]`

`!DIR$ NOBLOCK_LOOP`

clause

Is one or more of the following:

- FACTOR(*expr*)

  expr

  Is a positive scalar constant integer expression representing the blocking factor for the specified loops.

  This clause is optional. If the FACTOR clause is not present, the blocking factor will be determined based on processor type and memory access patterns and will be applied to the specified levels in the nested loop following the directive.

  At most only one FACTOR clause can appear in a BLOCK_LOOP directive.
- LEVEL(*level* [, *level*]...])

  level

  Is specified in the form:

  const1 or *const1*:*const2*

  where *const1* is a positive integer constant *m* <= 8 representing the loop at level *m*, where the immediate following loop is level 1.

The *const2* is a positive integer constant *n* <= 8 representing the loop at level *n*, where *n* > *m*: *const1*:*const2* represents the nested loops from level *const1* through *const2*.

This clause is optional. If the LEVEL clause is not present, the specified blocking factor is applied to all levels of the immediately following nested loops.

At most only one LEVEL clause can appear in a BLOCK_LOOP directive.

The clauses can be specified in any order. If you do not specify any *clause*, the compiler chooses the best blocking factor to apply to all levels of the immediately following nested loop.

The BLOCK_LOOP directive lets you exert greater control over optimizations on a specific DO loop inside a nested DO loop.

Using a technique called loop blocking, the BLOCK_LOOP directive separates large iteration counted DO loops into smaller iteration groups. Execution of these smaller groups can increase the efficiency of cache space use and augment performance.

If there is no LEVEL and FACTOR clause, the blocking factor will be determined based on the processor's type and memory access patterns and it will apply to all the levels in the nested loops following this directive.

You can use the NOBLOCK_LOOP directive to tune the performance by disabling loop blocking for nested loops.

NOTE
The loop-carried dependence is ignored during the processing of BLOCK_LOOP directives.

Example

```
!dir$ block_loop factor(256) level(1)       ! applies blocking factor 256 to
!dir$ block_loop factor(512) level(2)       !  the top level loop in the following
                                            !  nested loop and blocking factor 512 to
                                            !  the 2nd level {1st nested} loop

!dir$ block_loop factor(256) level(2)
!dir$ block_loop factor(512) level(1)       ! levels can be specified in any order

!dir$ block_loop factor(256) level(1:2)     ! adjacent loops can be specified as a range

!dir$ block_loop factor (256)     ! the blocking factor applies to all levels of loop nest

!dir$ block_loop                  ! the blocking factor will be determined based on
                                  !  processor type and memory access patterns and will
                                  !  be applied to all the levels in the nested loop
                                  !  following the directive

!dir$ noblock_loop                ! None of the levels in the nested loop following this
                                  !  directive will have a blocking factor applied
```

Consider the following:

```
!dir$ block_loop factor(256) level(1:2)
do  j = 1,n
        f  = 0
       do  i =1,n
            f  =  f +   a (i) *  b (i)
        enddo
         c(j) = c(j) + f
enddo
```

The above code produces the following result after loop blocking:

```
do jj=1,n/256+1
   do ii = 1,n/256+1
        do j = (jj-1)*256+1, min(jj*256, n)
             f = 0
             do i = (ii-1)*256+1, min(ii*256,n)
                 f = f + a(i) * b(i)
              enddo
              c(j) = c(j) + f
         enddo
    enddo
enddo
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements
Nested DO Constructs

BLT

Elemental Intrinsic Function (Generic): *Performs a bitwise less than on its arguments.*

Syntax

`result = BLT (i,j)`

i	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.
j	(Input) Must be of type integer or a binary, octal, or hexadecimal literal constant.

If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result is true if the sequence of bits represented by *i* is less than the sequence of bits represented by *j*, according to the method of bit sequence comparison in Bit Sequence Comparisons; otherwise, the result is false.

The interpretation of a binary, octal, or hexadecimal literal constant as a sequence of bits is described in Binary, Octal, Hexadecimal, and Hollerith Constants.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

BLT (0, 2) has the value true. BLT (Z'FF', Z'FC') has the value false.

BSEARCHQQ

Portability Function: *Performs a binary search of a sorted one-dimensional array for a specified element. The array elements cannot be derived types or structures.*

Module

USE IFPORT

Syntax

`result = BSEARCHQQ (adrkey,adrarray,length,size)`

adrkey
(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Address of the variable containing the element to be found (returned by LOC).

adrarray
(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Address of the array (returned by LOC).

length
(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Number of elements in the array.

size
(Input) INTEGER(4). Positive constant less than 32,767 that specifies the kind of array to be sorted. The following constants, defined in `IFPORT.F90`, specify type and kind for numeric arrays:

Constant	Type of array
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) or equivalent
SRT$INTEGER4	INTEGER(4) or equivalent
SRT$INTEGER8	INTEGER(8) or equivalent
SRT$REAL4	REAL(4) or equivalent
SRT$REAL8	REAL(8) or equivalent
SRT$REAL16	REAL(16) or equivalent

If the value provided in *size* is not a symbolic constant and is less than 32,767, the array is assumed to be a character array with *size* characters per element.

Results

The result type is INTEGER(4). It is an array index of the matched entry, or 0 if the entry is not found.

The array must be sorted in ascending order before being searched.

Caution
The location of the array and the element to be found must both be passed by address using the LOC function. This defeats Fortran type checking, so you must make certain that the *length* and *size* arguments are correct, and that *size* is the same for the element to be found and the array searched.

If you pass invalid arguments, BSEARCHQQ attempts to search random parts of memory. If the memory it attempts to search is not allocated to the current process, the program is halted, and you receive a General Protection Violation message.

Example

```
USE IFPORT
INTEGER(4) array(10), length
INTEGER(4) result, target
length = SIZE(array)
...
result = BSEARCHQQ(LOC(target),LOC(array),length,SRT$INTEGER4)
```

See Also
SORTQQ
LOC

BTEST

Elemental Intrinsic Function (Generic): *Tests a bit of an integer argument.*

Syntax

`result = BTEST (i,pos)`

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
pos	(Input) Must be of type integer. It must not be negative and it must be less than BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.

Results

The result type is default logical.

The result is true if bit *pos* of *i* has the value 1. The result is false if *pos* has the value zero. For more information, see Bit Functions.

For information on the model for the interpretation of an integer value as a sequence of bits, see Model for Bit Data.

The setting of compiler options specifying integer size can affect this function.

Specific Name	Argument Type	Result Type
BBTEST	INTEGER(1)	LOGICAL(1)
BITEST[1]	INTEGER(2)	LOGICAL(2)
BTEST [2]	INTEGER(4)	LOGICAL(4)

Specific Name	Argument Type	Result Type
BKTEST	INTEGER(8)	LOGICAL(8)
[1]Or HTEST		
[2]Or BJTEST		

Example

BTEST (9, 3) has the value true.

If A has the value

```
  [ 1  2 ]
  [ 3  4 ],
```

the value of BTEST (A, 2) is

```
  [ false  false ]
  [ false   true ]
```

and the value of BTEST (2, A) is

```
  [ true   false ]
  [ false  false ].
```

The following shows more examples:

Function reference	Value of *i*	Result
BTEST (*i*,2)	00011100 01111000	.FALSE.
BTEST (*i*,3)	00011100 01111000	.TRUE.

The following shows another example:

```
INTEGER(1) i(2)
LOGICAL result(2)
i(1) = 2#10101010
i(2) = 2#01010101
result = BTEST(i, (/3,2/))   ! returns (.TRUE.,.TRUE.)
write(*,*) result
```

See Also

IBCLR
IBSET
IBCHNG
IOR
IEOR
IAND

BYTE

Statement: *Specifies the BYTE data type, which is equivalent to INTEGER(1).*

Example

```
BYTE count, matrix(4, 4) / 4*1, 4*2, 4*4, 4*8 /
BYTE num  / 10 /
```

See Also
INTEGER
Integer Data Types

C to D

C to D

C_ASSOCIATED

Intrinsic Module Inquiry function (Generic): *Indicates the association status of one argument, or whether two arguments are associated with the same entity.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`result = C_ASSOCIATED(c_ptr_1[, c_ptr_2])`

c_ptr_1	(Input) Is a scalar of derived type C_PTR or C_FUNPTR.
c_ptr_2	(Optional; input) Is a scalar of the same type as *c_ptr_1*.

Results

The result is a scalar of type default logical. The result value is one of the following:

- If only *c_ptr_1* is specified, the result is false if *c_ptr_1* is a C null pointer; otherwise, the result is true.
- If *c_ptr_2* is specified, the result is false if *c_ptr_1* is a C null pointer. The result is true if *c_ptr_1* is equal to *c_ptr_2*; otherwise, the result is false.

See Also
Intrinsic Modules

ISO_C_BINDING Module

C_F_POINTER

Intrinsic Module Subroutine: *Associates a data pointer with the target of a C pointer and specifies its shape.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`CALL C_F_POINTER(cptr, fptr [,shape])`

cptr	(Input) Is a scalar of derived type C_PTR. Its value is the C address of an interoperable data entity, or the result of a reference to function C_LOC with a noninteroperable argument. If the value of *cptr* is the C address of a Fortran variable, it must have the TARGET attribute.
fptr	(Output) Is a data pointer. If it is an array, *shape* must be specified.
shape	(Optional, input) Must be of type integer and rank one. Its size equals the rank of *fptr*.

If the value of *cptr* is the C address of an interoperable data entity, *fptr* must be a data pointer with type and type parameters interoperable with the type of the entity. In this case, *fptr* becomes pointer-associated with the target of *cptr*.

If *fptr* is an array, it has the shape specified by *shape* and each lower bound is 1.

If the value of *cptr* is the result of a reference to C_LOC with a noninteroperable argument *x*, the following rules apply:

- C_LOC argument *x* (or its target) must not have been deallocated or have become undefined due to the execution of a RETURN or END statement since the reference to C_LOC.
- *fptr* is a scalar pointer with the same type and type parameters as *x*. *fptr* becomes pointer-associated with *x*, or it becomes pointer-associated with its target if *x* is a pointer.

Since the resulting data pointer *fptr* could point to a target that was not allocated with an ALLOCATE statement, *fptr* cannot be freed with a DEALLOCATE statement.

See Also

Intrinsic Modules

ISO_C_BINDING Module

C_LOC

C_F_PROCPOINTER

Intrinsic Module Subroutine: *Associates a Fortran procedure pointer with the target of a C function pointer.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`CALL C_F_PROCPOINTER(cptr, fptr)`

cptr	(Input) Is a scalar of derived type C_FUNPTR. Its value is the C address of a procedure.
fptr	(Output) Is a Fortran procedure pointer. It becomes pointer-associated with the target of *cptr*.

Example

The following Fortran subroutine can be called from a C program that passes a pointer to a C function to be called:

```
SUBROUTINE CallIt (cp) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
TYPE(C_FUNPTR), INTENT(IN) :: cp
ABSTRACT INTERFACE
  SUBROUTINE Add_Int (i) BIND(C)
  IMPORT
  INTEGER(C_INT), INTENT(INOUT) :: i
  END SUBROUTINE Add_Int
END INTERFACE
PROCEDURE(Add_Int), POINTER :: fp
INTEGER(C_INT) :: j

CALL C_F_PROCPOINTER (cp, fp)
j = 1
CALL fp(j)
...
```

See Also

Intrinsic Modules

ISO_C_BINDING Module

Procedure Pointers

PROCEDURE

C_FUNLOC

Intrinsic Module Inquiry function (Generic):
Returns the C address of a function pointer.

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`result = C_FUNLOC(x)`

x	(Input) Is a procedure or a Fortran pointer of type INTEGER associated with a procedure. If *x* is a procedure pointer, it must be associated. *x* cannot be a coindexed object. If C_FUNLOC is called from a PURE procedure, *x* must be PURE.

Results

The result is a scalar of derived type C_FUNPTR. The result value represents the C address of the argument.

See Also

Intrinsic Modules

ISO_C_BINDING Module

C_LOC

Intrinsic Module Inquiry function (Generic):
Returns the C address of an argument.

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`result = C_LOC(x)`

x	(Input) Is one of the following:
	• An interoperable variable that has the TARGET attribute; if the variable is an array, it does not have to be interoperable
	• An interoperable, allocatable, variable that is allocated, has the TARGET attribute, and is not an array of size zero; if the variable is an array, it does not have to be interoperable
	• An associated, interoperable scalar pointer
	• A scalar that has no length type parameters and is one of the following:
	  • A nonallocatable, nonpointer variable that has the TARGET attribute
	  • An allocatable variable that is allocated and has the TARGET attribute
	  • An associated pointer

Results

The result is a scalar of derived type C_PTR. The result value represents the C address of the argument.

The result is a value that can be used as an actual CPTR argument in a call to procedure C_F_POINTER where *fptr* has attributes that allow the pointer assignment *fptr*=>*x*. Such a call to C_F_POINTER has the effect of the pointer assignment *fptr*=>*x*.

If *x* is a scalar, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of *x* and the pointer assignment CPTR%PX=>*x* were executed.

If *x* is an array, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of *x* and the pointer assignment CPTR%PX to the first element of *x* were executed.

See Also

Intrinsic Modules

ISO_C_BINDING Module

C_F_POINTER

C_SIZEOF

Intrinsic Module Inquiry function (Generic): *Returns the number of bytes of storage used by the argument. It cannot be passed as an actual argument.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`result = C_SIZEOF(x)`

x	(Input) Is an interoperable data entity of any type and any rank. It must not be an assumed-size array or an assumed-rank array associated with an assumed-size array.

Results

The result is a scalar of type INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. If *x* is scalar, the result value is the size of *x* in bytes. If *x* is an array, the result value is the size of a single element of *x* multiplied by the number of elements in *x*.

Example

Consider the following:

```
INTEGER(4) :: S
INTEGER(4) :: T(3)
C_SIZEOF(S) ! has the value 4
C_SIZEOF(T) ! has the value 12
```

See Also

Intrinsic Modules

ISO_C_BINDING Module

SIZEOF

CACHESIZE

Inquiry Intrinsic Function (Generic): *Returns the size of a level of the memory cache.*

Syntax

`result = CACHESIZE (n)`

n	(Input) Must be scalar and of type integer.

Results

The result type and kind are the same as *n*. The result value is the number of kilobytes in the level *n* memory cache.

n = 1 specifies the first level cache; *n* = 2 specifies the second level cache; etc. If cache level *n* does not exist, the result value is 0.

Example

CACHESIZE(1) returns 16 for a processor with a 16KB first level memory cache.

CALL

Statement: *Transfers control to a subroutine subprogram.*

Syntax

```
CALL sub[( [a-arg[,a-arg]...] )]
```

sub

Is the name of the subroutine subprogram or other external procedure, or a dummy argument associated with a subroutine subprogram or other external procedure.

a-arg

Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the subroutine. The keyword is assigned a value when the procedure is invoked.

Each actual argument must be a variable, an expression, the name of a procedure, or an alternate return specifier. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

An alternate return specifier is an asterisk (*), or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement. (An alternate return is an obsolescent feature in Standard Fortran.)

Description

When the CALL statement is executed, any expressions in the actual argument list are evaluated, then control is passed to the first executable statement or construct in the subroutine. When the subroutine finishes executing, control returns to the next executable statement following the CALL statement, or to a statement identified by an alternate return label (if any).

If an argument list appears, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see table Specific Functions Not Allowed as Actual Arguments in Intrinsic Procedures).

The procedure invoked by the CALL statement must be a subroutine subprogram and not a function. Calling a function as if it were a subroutine can cause unpredictable results.

Example

The following example shows valid CALL statements:

```
CALL CURVE(BASE,3.14159+X,Y,LIMIT,R(LT+2))
CALL PNTOUT(A,N,'ABCD')
CALL EXIT
CALL MULT(A,B,*10,*20,C)       ! The asterisks and ampersands denote
CALL SUBA(X,&30,&50,Y)         !     alternate returns
```

The following example shows a subroutine with argument keywords:

```
PROGRAM KEYWORD_EXAMPLE
  INTERFACE
    SUBROUTINE TEST_C(I, L, J, KYWD2, D, F, KYWD1)
    INTEGER I, L(20), J, KYWD1
    REAL, OPTIONAL :: D, F
    COMPLEX KYWD2
    ...
    END SUBROUTINE TEST_C
  END INTERFACE
  INTEGER I, J, K
  INTEGER L(20)
  COMPLEX Z1
  CALL TEST_C(I, L, J, KYWD1 = K, KYWD2 = Z1)
  ...
```

The first three actual arguments are associated with their corresponding dummy arguments by position. The argument keywords are associated by keyword name, so they can appear in any order.

Note that the interface to subroutine TEST has two optional arguments that have been omitted in the CALL statement.

The following shows another example of a subroutine call with argument keywords:

```
CALL TEST(X, Y, N, EQUALITIES = Q, XSTART = X0)
```

The first three arguments are associated by position.

The following shows another example:

```
!Variations on a subroutine call
      REAL S,T,X
      INTRINSIC NINT
      S=1.5
      T=2.5
      X=14.7
      !This calls SUB1 using keywords. NINT is an intrinsic function.
      CALL SUB1(B=X,C=S*T,FUNC=NINT,A=4.0)
!Here is the same call using an implicit reference
      CALL SUB1(4.0,X,S*T,NINT)
      CONTAINS
        SUBROUTINE sub1(a,b,c,func)
        INTEGER func
        REAL a,b,c
        PRINT *, a,b,c, func(b)
        END SUBROUTINE
      END
```

See Also

SUBROUTINE
CONTAINS

RECURSIVE
USE
Program Units and Procedures

CANCEL

OpenMP* Fortran Compiler Directive: *Requests cancellation of the innermost enclosing region of the construct specified, and causes the encountering implicit or explicit task to proceed to the end of the canceled construct.*

Syntax

```
!$OMP CANCEL construct-clause [[,] if-clause]
```

construct-clause	Is one of the following: • DO • PARALLEL • SECTIONS • TASKGROUP
if-clause	(Optional) IF (scalar-logical-expression)

The binding thread set of a CANCEL construct is the current team. The cancel region binds to the innermost enclosing construct of the type corresponding to the *construct-clause* specified in the directive specifying the innermost DO, PARALLEL, SECTIONS, or TASKGROUP construct.

This is a stand-alone directive, so there are some restrictions on its placement within a program:

- It can only be placed at a point where a Fortran executable statement is allowed.
- It cannot be used as the action statement in an IF statement, or as the executable statement following a label, if the label is referenced in the program.

If *construct-clause* is TASKGROUP, the CANCEL construct must be closely nested inside a TASK construct. Otherwise, the CANCEL construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-clause*.

The CANCEL construct requests cancellation of the innermost enclosing region of the type specified. The request is checked at a cancellation point. When a cancellation is observed, execution jumps to the end of the canceled region.

Cancellation points are implied at certain locations, as follows:

- Implicit barriers
- BARRIER regions
- CANCEL regions
- CANCELLATION POINT regions

When cancellation of tasks occurs with a CANCEL TASKGROUP construct, the encountering task jumps to the end of its task region and is considered complete. Any task that belongs to the innermost enclosing taskgroup and has already begun execution, must run to completion or run until a cancellation point is reached. Any task that belongs to the innermost enclosing taskgroup and has not begun execution may be discarded and considered completed.

When cancellation occurs for a PARALLEL region, each thread of the binding thread set resumes execution at the end of the canceled region and any tasks that have been created by a TASK construct and their descendants are canceled according to the above taskgroup cancellation semantics.

When cancellation occurs for a DO or SECTIONS region, each thread of the binding thread set resumes execution at the end of the canceled region but no task cancellation occurs.

A DO construct that is being canceled must not have a NOWAIT or an ORDERED clause. A SECTIONS construct that is being canceled must not have a NOWAIT clause.

The behavior for concurrent cancellation of a region and a region nested within it is unspecified.

NOTE
You must release locks and similar data structures that can cause a deadlock when a CANCEL construct is encountered; blocked threads cannot be canceled.

If the canceled construct contains a REDUCTION or LASTPRIVATE clause, the final value of the REDUCTION or LASTPRIVATE variable is undefined.

All private objects or subobjects with the ALLOCATABLE attribute that are allocated inside the canceled construct are deallocated.

See Also
OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
CANCELLATION POINT
Parallel Processing Model for information about Binding Sets

CANCELLATION POINT

OpenMP* Fortran Compiler Directive: *Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified. This construct does not implement a synchronization between threads or tasks.*

Syntax

```
!$OMP CANCELLATION POINT construct-clause
```

construct-clause	Is one of the following: • DO • PARALLEL • SECTIONS • TASKGROUP

This is a stand-alone directive, so there are some restrictions on its placement within a program:

- It can only be placed at a point where a Fortran executable statement is allowed.
- It cannot be used as the action statement in an IF statement, or as the executable statement following a label, if the label is referenced in the program.

A CANCELLATION POINT region binds to the current task region.

If *construct-clause* is TASKGROUP, the CANCELLATION POINT construct must be closely nested inside a TASK construct. Otherwise, the CANCELLATION POINT construct must be closely nested inside an OpenMP construct that matches the type specified by the *construct-clause*.

When an implicit or explicit task reaches a user-defined cancellation point, the task immediately checks for cancellation of the region specified in the clause and performs cancellation of this region if cancellation is observed. If the clause specified is TASKGROUP then the current task region is canceled.

An OpenMP program with orphaned CANCELLATION POINT constructs is non-conforming.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
CANCEL

CASE

Statement: *Marks the beginning of a CASE construct. A CASE construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement.*

Syntax

```
[name:] SELECT CASE (expr)
[CASE (case-value [, case-value] ...) [name]
   block]...
[CASE DEFAULT [name]
   block]
END SELECT [name]
```

name — Is the name of the CASE construct.

expr — Is a scalar expression of type integer, logical, or character (enclosed in parentheses). Evaluation of this expression results in a value called the *case index*.

case-value — Is one or more scalar integer, logical, or character initialization expressions enclosed in parentheses. Each *case-value* must be of the same type and kind parameter as *expr*. If the type is character, *case-value* and *expr* can be of different lengths, but their kind parameter must be the same.

Integer and character expressions can be expressed as a range of case values, taking one of the following forms:

```
low:high
low:
:high
```

Case values must not overlap.

block — Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified in a SELECT CASE statement, the same name must appear in the corresponding END SELECT statement. The same construct name can optionally appear in any CASE statement in the construct. The same construct name must not be used for different named constructs in the same scoping unit.

The case expression (*expr*) is evaluated first. The resulting case index is compared to the case values to find a matching value (there can only be one). When a match occurs, the block following the matching case value is executed and the construct terminates.

The following rules determine whether a match occurs:

- When the case value is a single value (no colon appears), a match occurs as follows:

Data Type	A Match Occurs If:
Logical	case-index .EQV. case-value
Integer or Character	case-index = = case-value

- When the case value is a range of values (a colon appears), a match depends on the range specified, as follows:

Range	A Match Occurs If:
low **:**	case-index >= low
: high	case-index <= high
low **:** high	low <= case-index <= high

The following are all valid case values:

```
CASE (1, 4, 7, 11:14, 22)      ! Individual values as specified:
                               !     1, 4, 7, 11, 12, 13, 14, 22
CASE (:-1)                     ! All values less than zero
CASE (0)                       ! Only zero
CASE (1:)                      ! All values above zero
```

If no match occurs but a CASE DEFAULT statement is present, the block following that statement is executed and the construct terminates.

If no match occurs and no CASE DEFAULT statement is present, no block is executed, the construct terminates, and control passes to the next executable statement or construct following the END SELECT statement.

The following figure shows the flow of control in a CASE construct:

Flow of Control in CASE Constructs

You cannot use branching statements to transfer control to a CASE statement. However, branching to a SELECT CASE statement is allowed. Branching to the END SELECT statement is allowed only from within the CASE construct.

Example

The following are examples of CASE constructs:

```
INTEGER FUNCTION STATUS_CODE (I)
  INTEGER I
  CHECK_STATUS: SELECT CASE (I)
  CASE (:-1)
    STATUS_CODE = -1
  CASE (0)
    STATUS_CODE = 0
  CASE (1:)
    STATUS_CODE = 1
  END SELECT CHECK_STATUS
END FUNCTION STATUS_CODE

SELECT CASE (J)
CASE (1, 3:7, 9)    ! Values: 1, 3, 4, 5, 6, 7, 9
  CALL SUB_A
```

```
CASE DEFAULT
  CALL SUB_B
END SELECT
```

The following three examples are equivalent:

```
1. SELECT CASE (ITEST .EQ. 1)
   CASE (.TRUE.)
     CALL SUB1 ()
   CASE (.FALSE.)
     CALL SUB2 ()
   END SELECT

2. SELECT CASE (ITEST)
   CASE DEFAULT
     CALL SUB2 ()
   CASE (1)
     CALL SUB1 ()
   END SELECT

3. IF (ITEST .EQ. 1) THEN
     CALL SUB1 ()
   ELSE
     CALL SUB2 ()
   END IF
```

The following shows another example:

```
CHARACTER*1 cmdchar
GET_ANSWER: SELECT CASE (cmdchar)
CASE ('0')
   WRITE (*, *) "Must retrieve one to nine files"
CASE ('1':'9')
   CALL RetrieveNumFiles (cmdchar)
CASE ('A', 'a')
   CALL AddEntry
CASE ('D', 'd')
   CALL DeleteEntry
CASE ('H', 'h')
   CALL Help
CASE DEFAULT
   WRITE (*, *) "Command not recognized; please use H for help"
END SELECT GET_ANSWER
```

See Also

Execution Control

CDFLOAT

Portability Function: *Converts a COMPLEX(4) argument to double-precision real type.*

Module

USE IFPORT

Syntax

`result = CDFLOAT (input)`

input (Input) COMPLEX(4). The value to be converted.

Results

The result type is REAL(8).

CEILING

Elemental Intrinsic Function (Generic): *Returns the smallest integer greater than or equal to its argument.*

Syntax

```
result = CEILING (a[,kind])
```

a (Input) Must be of type real.

kind (Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the smallest integer greater than or equal to *a*.

The setting of compiler options specifying integer size can affect this function.

Example

CEILING (4.8) has the value 5.

CEILING (-2.55) has the value -2.0.

The following shows another example:

```
INTEGER I, IARRAY(2)
I = CEILING(8.01) ! returns 9
I = CEILING(-8.01) ! returns -8
IARRAY = CEILING((/8.01,-5.6/)) ! returns (9, -5)
```

See Also

FLOOR

CFI_address

C function prototype: *Returns the C address of an object described by a C descriptor.*

Syntax

```
void *CFI_address (const CFI_cdesc_t *dv,
                   const CFI_index_t subscripts[]);
```

Formal Parameters:

dv	The address of a C descriptor describing the object. The object must not be an unallocated allocatable variable or a pointer that is not associated.
subscripts	A null pointer or the address of an array of type `CFI_index_t`. If the object is an array, subscripts must be the address of an array of type `CFI_index_t` with at least *n* elements, where *n* is the rank of the object. The value of subscripts [*i*] must be within the bounds of dimension *i* specified by the `dim` member of the C descriptor.

Result Value

If the object is an array of rank *n*, the result is the C address of the element of the object that the first *n* elements of the subscripts argument would specify if used as subscripts. If the object is scalar, the result is its C address.

Example

If *dv* is the address of a C descriptor for the Fortran array A declared as follows:

```
REAL(C_FLOAT) :: A(100, 100)
```

then the following code calculates the C address of A(5, 10):

```
CFI_index_t subscripts[2];
float *address;
subscripts[0] = 4;
subscripts[1] = 9;
address = (float *) CFI_address(dv, subscripts );
```

See Also

C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_allocate

C function prototype: *Allocates memory for an object described by a C descriptor.*

Syntax

```
int CFI_allocate (CFI_cdesc_t *dv,
                  const CFI_index_t lower_bounds[]),
                  const CFI_index_t upper_bounds[],
                  size_t elem_len);
```

Formal Parameters:

dv	The address of a C descriptor specifying the rank and type of the object. The `base_addr` member of the C descriptor is a null pointer. If the type is not a character type, the *elem_len* member must specify the element length. The attribute member must have a value of `CFI_attribute_allocatable` or `CFI_attribute_pointer`.
lower_bounds	The address of an array with at least dv->rank elements. The first dv->rank elements of *lower_bounds* provide the lower Fortran bounds for each corresponding dimension of the object.

upper_bounds	The address of an array with at least dv->rank elements. The first dv->rank elements of *upper_bounds* provide the upper Fortran bounds for each corresponding dimension of the object.
elem_len	If the type specified in the C descriptor type is a Fortran character type, the value of *elem_len* is the storage size in bytes of an element of the object; otherwise, *elem_len* is ignored.

Description

Successful execution of CFI_allocate allocates memory for the object described by the C descriptor with the address dv using the same mechanism as the Fortran ALLOCATE statement, and assigns the address of that memory to dv->`base_addr`.

The first dv->rank elements of the *lower_bounds* and *upper_bounds* arguments provide the lower and upper Fortran bounds, respectively, for each corresponding dimension of the object. The supplied lower and upper bounds override any current dimension information in the C descriptor. If the rank is zero, the *lower_bounds* and *upper_bounds* arguments are ignored.

If the type specified in the C descriptor is a character type, the supplied element length overrides the current element-length information in the descriptor.

If an error is detected, the C descriptor is not modified.

Result Value

The result is an error indicator.

Example

If *dv* is the address of a C descriptor for the Fortran array A declared as follows:

```
REAL, ALLOCATABLE :: A(:, :)
```

and the array is not allocated, the following code allocates it to be of shape [100, 500]:

```
CFI_index_t lower[2], upper[2];
int ind;
lower[0] = 1; lower[1] = 1;
upper[0] = 100; upper[1] = 500;
ind = CFI_allocate(dv, lower, upper, 0);
```

See Also

ALLOCATE
C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_deallocate

C function prototype: *Deallocates memory for an object described by a C descriptor.*

Syntax

```
int CFI_deallocate (CFI_cdesc_t *dv);
```

Formal Parameters:

dv — The address of a C descriptor describing the object. It must have been allocated using the same mechanism as the Fortran ALLOCATE statement. If the object is a pointer, it must be associated with a target satisfying the conditions for successful deallocation by the Fortran DEALLOCATE statement.

Description

Successful execution of CFI_deallocate deallocates memory for the object using the same mechanism as the Fortran DEALLOCATE statement, and the `base_addr` member of the C descriptor becomes a null pointer.

If an error is detected, the C descriptor is not modified.

Result Value

The result is an error indicator.

Example

If *dv* is the address of a C descriptor for the Fortran array A declared as follows:

```
REAL, ALLOCATABLE :: A(:, :)
```

and the array is allocated, the following code deallocates it:

```
int ind;
ind = CFI_deallocate(dv);
```

See Also

ALLOCATE
DEALLOCATE
C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_establish

C function prototype: *Establishes a C descriptor.*

Syntax

```
int CFI_establish(CFI_cdesc_t *dv, void *base_addr,
                  CFI_attribute_t attribute,
                  CFI_type_t type, size_t elem_len,
                  CFI_rank_t rank,
                  const CFI_index_t extents[]);
```

Formal Parameters:

dv — The address of a data object large enough to hold a C descriptor of the rank specified by *rank*. It must not have the same value as either a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument. It must not be the address of a C descriptor that describes an allocated allocatable object.

base_addr	A null pointer or the base address of the object to be described. If it is not a null pointer, it must be the address of a contiguous storage sequence that is appropriately aligned for an object of the type specified by *type*.
attribute	One of the attribute codes in Table "Macros for attribute codes" in C Typedefs and Macros for interoperability. If it is CFI_attribute_allocatable, *base_addr* must be a null pointer.
type	One of the type codes in Table "Macros for type codes" in C Typedefs and Macros for interoperability.
elem_len	If the type is CFI_type_struct, CFI_type_other, or a Fortran character type code, *elem_len* must be greater than zero and equal to the storage size in bytes of an element of the object. Otherwise, *type* is ignored.
rank	A value in the range 0 ≤ *rank* ≤ CFI_MAX_RANK. It specifies the rank of the object.
extents	This is ignored if rank is equal to zero or if *base_addr* is a null pointer. Otherwise, it must be the address of an array with rank elements; the value of each element must be nonnegative, and extents[*i*] specifies the extent of dimension *i* of the object.

Description

Successful execution of CFI_establish updates the object with the address *dv* to be an established C descriptor for a nonallocatable nonpointer data object of known shape, an unallocated allocatable object, or a data pointer.

If *base_addr* is not a null pointer, it is the address for a nonallocatable entity that is a scalar or a contiguous array. If the attribute argument has the value CFI_attribute_pointer, the lower bounds of the object described by *dv* are set to zero. If *base_addr* is a null pointer, the established C descriptor is for an unallocated allocatable, a disassociated pointer, or is a C descriptor that has the attribute CFI_attribute_other but does not describe a data object. If *base_addr* is the C address of a Fortran data object, the *type* and *elem_len* arguments must be consistent with the type and type parameters of the Fortran data object.

The remaining properties of the object are given by the other arguments.

CFI_establish is used to initialize a C descriptor declared in C with CFI_CDESC_T before passing it to any other functions as an actual argument, in order to set the rank, attribute, type and element length.

A C descriptor with attribute CFI_attribute_other and a *base_addr* that is a null pointer can be used as the argument result in calls to CFI_section or CFI_select_part, which will produce a C descriptor for a nonallocatable nonpointer data object.

If an error is detected, the object with the address *dv* is not modified.

Result Value

The result is an error indicator.

Example

The following code fragment establishes a C descriptor for an unallocated rank-one allocatable array that can be passed to Fortran for allocation there:

```
CFI_rank_t rank;
CFI_CDESC_T(1) field;
int ind;
rank = 1;
ind = CFI_establish((CFI_cdesc_t *)&field, NULL,
CFI_attribute_allocatable,
CFI_type_double, 0, rank, NULL);
```

If the following Fortran type definition is specified:

```
TYPE, BIND(C) :: T
REAL(C_DOUBLE) :: X
COMPLEX(C_DOUBLE_COMPLEX) :: Y
END TYPE
```

and a Fortran subprogram that has an assumed-shape dummy argument of type T, the following code fragment creates a descriptor `a_fortran` for an array of size 100 that can be used as the actual argument in an invocation of the subprogram from C:

```
typedef struct {double x; double _Complex y;} t;
t a_c[100];
CFI_CDESC_T(1) a_fortran;
int ind;
CFI_index_t extent[1];

extent[0] = 100;
ind = CFI_establish((CFI_cdesc_t *)&a_fortran, a_c,
CFI_attribute_other,
CFI_type_struct, sizeof(t), 1, extent);
```

See Also

C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_is_contiguous

C function prototype: *Tests contiguity of an array.*

Syntax

```
int CFI_is_contiguous(const CFI_cdesc_t *dv);
```

Formal Parameters:

dv	The address of a C descriptor describing an array. The `base_addr` member of the C descriptor must not be a null pointer.

Result Value

The value of the result is 1 if the array described by *dv* is contiguous; otherwise, 0.

Since assumed-size and allocatable arrays are always contiguous, the result of CFI_is_contiguous on a C descriptor for such an array is 1.

See Also

C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_section

C function prototype: *Updates a C descriptor for an array section for which each element is an element of a given array.*

Syntax

```
int CFI_section(CFI_cdesc_t *result, const CFI_cdesc_t *source,
                          const CFI_index_t lower_bounds[],
                          const CFI_index_t upper_bounds[],
                          const CFI_index_t strides[]);
```

Formal Parameters:

result

The address of a C descriptor with rank equal to the rank of *source* minus the number of zero *strides*. The attribute member must have the value `CFI_attribute_other` or `CFI_attribute_pointer`. If the value of result is the same as either a C formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument, the attribute member must have the value `CFI_attribute_pointer`.

Successful execution of CFI_section updates the `base_addr` and `dim` members of the C descriptor with the address result to describe the array section determined by *source*, *lower_bounds*, *upper_bounds*, and *strides*, as follows:

- The array section is equivalent to the Fortran array section SOURCE(*sectsub*1, *sectsub*2, ... *sectsub*n), where SOURCE is the array described by *source*, *n* is the rank of that array, and *sectsubi* is the subscript *loweri* if *stridesi* is zero, and the section subscript *loweri* : *upperi* : *stridei* otherwise.
- The value of *loweri* is the lower bound of dimension *i* of SOURCE if *lower_bounds* is a null pointer and *lower_bounds*[*i*] otherwise.
- The value of *upperi* is the upper bound of dimension *i* of SOURCE if *upper_bounds* is a null pointer and *upper_bounds*[*i*] otherwise.
- The value of *stridei* is 1 if *strides* is a null pointer and *strides*[*i*] otherwise. If *stridei* has the value zero, *loweri* must have the same value as *upperi*.

source

The address of a C descriptor that describes a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer. The `elem_len` and `type` members of source must have the same values as the corresponding members of result.

lower_bounds

A null pointer or the address of an array with at least source->rank elements. If it is not a null pointer, and *stridei* is zero or (*upperi* −*lower_bounds*[*i*] + *stridei*)/*stridei* > 0, the value of *lower_bounds*[*i*] must be within the bounds of dimension *i* of SOURCE.

upper_bounds

A null pointer or the address of an array with at least source->rank elements. If source describes an assumed-size array, *upper_bounds* must not be a null pointer. If it is not a null pointer and *stridei* is zero

	or (*upper_bounds*[*i*] − *loweri* + *stridei*)/*stridei* > 0, the value of *upper_bounds*[*i*] must be within the bounds of dimension *i* of SOURCE.
strides	A null pointer or the address of an array with at least source->rank elements.

If an error is detected, the C descriptor with the address result is not modified.

Result Value

The result is an error indicator.

Example

If source is already the address of a C descriptor for the rank-one Fortran array A, the lower bounds of A are equal to 1, and the lower bounds in the C descriptor are equal to 0, the following code fragment establishes a new C descriptor section and updates it to describe the array section A(3::5):

```
CFI_index_t lower[1], strides[1];
CFI_CDESC_T(1) section;
int ind;
lower[0] = 2;
strides[0] = 5;
ind = CFI_establish((CFI_cdesc_t *)&section, NULL,
CFI_attribute_other,
CFI_type_float, 0, 1, NULL);
ind = CFI_section((CFI_cdesc_t *)&section, source,
lower, NULL, strides);
```

If source is already the address of a C descriptor for a rank-two Fortran assumed-shape array A with lower bounds equal to 1, the following code fragment establishes a C descriptor and updates it to describe the rank-one array section A(:, 42):

```
CFI_index_t lower[2], upper[2], strides[2];
CFI_CDESC_T(1) section;
int ind;
lower[0] = source->dim[0].lower_bound;
upper[0] = source->dim[0].lower_bound + source->dim[0].extent - 1;
strides[0] = 1;
lower[1] = upper[1] = source->dim[1].lower_bound + 41;
strides[1] = 0;
ind = CFI_establish((CFI_cdesc_t *)&section, NULL,
CFI_attribute_other,
CFI_type_float, 0, 1, NULL);
ind = CFI_section((CFI_cdesc_t *)&section, source,
lower, upper, strides);
```

See Also

C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_select_part

C function prototype: *Updates a C descriptor for an array section for which each element is a part of the corresponding element of an array.*

Syntax

```
int CFI_select_part(CFI_cdesc_t *result, const CFI_cdesc_t *source,
                    size_t displacement, size_t elem_len);
```

Formal Parameters:

result	The address of a C descriptor; result->rank must have the same value as source->rank and result->attribute must have the value CFI_attribute_other or CFI_attribute_pointer. If the address specified by *result* is the value of a C formal parameter that corresponds to a Fortran actual argument or of a C actual argument that corresponds to a Fortran dummy argument, result->attribute must have the value CFI_attribute_pointer. The value of result->type specifies the type of the array section.
source	The address of a C descriptor for a nonallocatable nonpointer array, an allocated allocatable array, or an associated array pointer.
displacement	A value 0 ≤ *displacement* ≤ *source->elem_len* − 1, and the sum of the *displacement* and the size in bytes of an element of the array section must be less than or equal to *source->elem_len*. The address displacement bytes greater than the value of source->base_addr is the base of the array section and must be appropriately aligned for an object of the type of the array section.
elem_len	A value equal to the storage size in bytes of an element of the array section if result->type specifies a Fortran character type; otherwise, *elem_len* is ignored.

Description

Successful execution of CFI_select_part updates the base_addr, dim, and elem_len members of the C descriptor with the address result for an array section for which each element is a part of the corresponding element of the array described by the C descriptor with the address source. The part must be a component of a structure, a substring, or the real or imaginary part of a complex value.

If an error is detected, the C descriptor with the address result is not modified.

Result Value

The result is an error indicator.

Example

If source is already the address of a C descriptor for the Fortran array A declared as follows:

```
TYPE, BIND(C) :: T
REAL(C_DOUBLE) :: X
COMPLEX(C_DOUBLE_COMPLEX) :: Y
END TYPE
TYPE(T) A(100)
```

then the following code fragment establishes a C descriptor for the array A%Y:

```
typedef struct {
double x; double _Complex y;
} t;
CFI_CDESC_T(1) component;
CFI_cdesc_t * comp_cdesc = (CFI_cdesc_t *)&component;
```

```
CFI_index_t extent[] = { 100 };
(void)CFI_establish(comp_cdesc, NULL, CFI_attribute_other,
                    CFI_type_double_Complex,
                    sizeof(double _Complex), 1, extent);
(void)CFI_select_part(comp_cdesc, source, offsetof(t,y), 0);
```

See Also

C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CFI_setpointer

C function prototype: *Updates a C descriptor for a Fortran pointer to be associated with the whole of a given object or to be disassociated.*

Syntax

```
int CFI_setpointer(CFI_cdesc_t *result, CFI_cdesc_t *source,
                   const CFI_index_t lower_bounds[]);
```

Formal Parameters:

result	The address of a C descriptor for a Fortran pointer. It is updated using information from the *source* and *lower_bounds* arguments.
source	A null pointer or the address of a C descriptor for a nonallocatable nonpointer data object, an allocated allocatable object, or a data pointer object. If *source* is not a null pointer, the corresponding values of the `elem_len`, `rank`, and `type` members must be the same in the C descriptors with the addresses *source* and *result*.
lower_bounds	If source is not a null pointer and source->rank is nonzero, *lower_bounds* must be a null pointer or the address of an array with at least source->rank elements.

Description

Successful execution of `CFI_setpointer` updates the `base_addr` and `dim` members of the C descriptor with the address result as follows:

- If *source* is a null pointer or the address of a C descriptor for a disassociated pointer, the updated C descriptor describes a disassociated pointer.
- Otherwise, the C descriptor with the address result becomes a C descriptor for the object described by the C descriptor with the address source, except that if source->rank is nonzero and *lower_bounds* is not a null pointer, the lower bounds are replaced by the values of the first source->rank elements of the *lower_bounds* array.

Result Value

The result is an error indicator.

Example

If `ptr` is already the address of a C descriptor for an array pointer of rank 1, the following code updates it to be a C descriptor for a pointer to the same array with lower bound 0:

```
CFI_index_t lower_bounds[1];
int ind;
lower_bounds[0] = 0;
ind = CFI_setpointer(ptr, ptr, lower_bounds);
```

See Also

C Structures Typedefs and Macros for interoperability
Interoperating with arguments using C descriptors

CHANGEDIRQQ

Portability Function: *Makes the specified directory the current, default directory.*

Module

USE IFPORT

Syntax

`result = CHANGEDIRQQ (dir)`

dir (Input) Character*(*). Directory to be made the current directory.

Results

The result type is LOGICAL(4). It is .TRUE. if successful; otherwise, .FALSE..

If you do not specify a drive in the *dir* string, the named directory on the current drive becomes the current directory. If you specify a drive in *dir*, the named directory on the specified drive becomes the current directory.

Example

```
  USE IFPORT
  LOGICAL(4) status
  status = CHANGEDIRQQ('d:\fps90\bin'))
! We are now CCed to 'd:\fps90\bin'
  status = CHANGEDIRQQ('bessel')
! We are now CCed to 'd:\fps90\bin\bessel'
```

See Also

GETDRIVEDIRQQ
MAKEDIRQQ
DELDIRQQ
CHANGEDRIVEQQ

CHANGEDRIVEQQ

Portability Function: *Makes the specified drive the current, default drive.*

Module

USE IFPORT

Syntax

```
result = CHANGEDRIVEQQ (drive)
```

drive (Input) Character*(*). String beginning with the drive letter.

Results

The result type is LOGICAL(4). On Windows* systems, the result is .TRUE. if successful; otherwise, .FALSE. On Linux* and macOS* systems, the result is always .FALSE..

Because drives are identified by a single alphabetic character, CHANGEDRIVEQQ examines only the first character of *drive*. The drive letter can be uppercase or lowercase.

CHANGEDRIVEQQ changes only the current drive. The current directory on the specified drive becomes the new current directory. If no current directory has been established on that drive, the root directory of the specified drive becomes the new current directory.

Example

```
USE IFPORT
LOGICAL(4) status
status = CHANGEDRIVEQQ('d')
```

See Also

GETDRIVESQQ
GETDRIVESIZEQQ
GETDRIVEDIRQQ
CHANGEDIRQQ

CHAR

Elemental Intrinsic Function (Generic): *Returns the character in the specified position of the processor's character set. It is the inverse of the function ICHAR.*

Syntax

```
result = CHAR (i[,kind])
```

i (Input) Must be of type integer with a value in the range 0 to n - 1, where *n* is the number of characters in the processor's character set.

kind (Input; optional) Must be a scalar integer constant expression.

Results

The result is of type character with length 1. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default character. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is the character in position *i* of the processor's character set. ICHAR(CHAR (*i*, *kind*(*c*))) has the value I for 0 to n - 1 and CHAR(ICHAR(*c*), *kind*(*c*)) has the value *c* for any character *c* capable of representation in the processor.

Specific Name	Argument Type	Result Type
	INTEGER(1)	CHARACTER

Specific Name	Argument Type	Result Type
	INTEGER(2)	CHARACTER
CHAR [1]	INTEGER(4)	CHARACTER
	INTEGER(8)	CHARACTER

[1]This specific function cannot be passed as an actual argument.

Example

CHAR (76) has the value `'L'`.

CHAR (94) has the value `'^'`.

See Also

ACHAR
IACHAR
ICHAR
Character and Key Code Charts

CHARACTER

Statement: *Specifies the CHARACTER data type.*

Syntax

```
CHARACTER ([LEN=]len [, [KIND=]n])

CHARACTER (KIND=n [, [LEN=]len])

CHARACTER* len
```

n	Is kind 1.
len	Is a string length (not a kind). For more information, see Declaration Statements for Character Types. If no kind type parameter is specified, the kind of the constant is default character.

Example

```
C
C   Length of wt and vs is 10, city is 80, and ch is 1
C
    CHARACTER wt*10, city*80, ch
    CHARACTER (LEN = 10), PRIVATE :: vs
    CHARACTER*(*) arg !declares a dummy argument
C   name and plume are ten-element character arrays
C   of length 20
    CHARACTER name(10)*20
    CHARACTER(len=20), dimension(10):: plume
C
C   Length of susan, patty, and dotty are 2, alice is 12,
C   jane is a 79-member array of length 2
C
    CHARACTER(2) susan, patty, alice*12, dotty, jane(79)
```

See Also

Character Data Type
Character Constants
Character Substrings
C Strings
Declaration Statements for Character Types

CHDIR

Portability Function: *Changes the default directory.*

Module

USE IFPORT

Syntax

`result = CHDIR(dir_name)`

dir_name	(Input) Character*(*). Name of a directory to become the default directory.

Results

The result type is INTEGER(4). It returns zero if the directory was changed successfully; otherwise, an error code. Possible error codes are:

- ENOENT: The named directory does not exist.
- ENOTDIR: The *dir_name* parameter is not a directory.

Example

```
     use ifport
     integer(4) istatus, enoent, enotdir
     character(255)  newdir
     character(300)  prompt, errmsg

     prompt = 'Please enter directory name: '
10   write(*,*)  TRIM(prompt)
     read *, newdir
     ISTATUS = CHDIR(newdir)
     select case (istatus)
       case (2)  ! ENOENT
         errmsg = 'The directory '//TRIM(newdir)//' does not exist'
       case (20) ! ENOTDIR
         errmsg = TRIM(newdir)//' is not a directory'
       case (0) ! NO error
         goto 40
       case default
         write (errmsg,*) 'Error with code ', istatus
      end select

      write(*,*)  TRIM(errmsg)
      goto 10

40    write(*,*) 'Default directory successfully changed.'
end
```

See Also
CHANGEDIRQQ

CHMOD
Portability Function: *Changes the access mode of a file.*

Module
USE IFPORT

Syntax
```
result = CHMOD (name,mode)
```

name (Input) Character*(*). Name of the file whose access mode is to be changed. Must have a single path.

mode (Input) Character*(*). File permission: either Read, Write, or Execute. The *mode* parameter can be either symbolic or absolute. An absolute mode is specified with an octal number, consisting of any combination of the following permission bits ORed together:

Permission bit	Description	Action
4000	Set user ID on execution	W*S: Ignored; never true L*X, M*X: Settable
2000	Set group ID on execution	W*S: Ignored; never true L*X, M*X: Settable
1000	Sticky bit	W*S: Ignored; never true L*X, M*X: Settable
0400	Read by owner	W*S: Ignored; always true L*X, M*X: Settable
0200	Write by owner	Settable
0100	Execute by owner	W*S: Ignored; based on file name extension L*X, M*X: Settable
0040, 0020, 0010	Read, Write, Execute by group	W*S: Ignored; assumes owner permissions L*X, M*X: Settable

Permission bit	Description	Action
0004, 0002, 0001	Read, Write, Execute by others	W*S: Ignored; assumes owner permissions L*X, M*X: Settable

The following regular expression represents a symbolic mode:

```
[ugoa]*[+-=] [rwxXst]*
```

On Windows* systems, "[ugoa]*" is ignored. On Linux* and macOS* systems, a combination of the letters "ugoa" control which users' access to the file will be changed:

u	The user who owns the file
g	Other users in the group that owns the file
o	Other users not in the group that owns the file
a	All users

"[+ - =]" indicates the operation to carry out:

+	Add the permission
-	Remove the permission
=	Absolutely set the permission

"[rwxXst]*" indicates the permission to add, subtract, or set. On Windows systems, only "w" is significant and affects write permission; all other letters are ignored. On Linux and macOS* systems, all letters are significant.

Results

The result type is INTEGER(4). It is zero if the mode was changed successfully; otherwise, an error code. Possible error codes are:

- ENOENT: The specified file was not found.
- EINVAL: The mode argument is invalid.
- EPERM: Permission denied; the file's mode cannot be changed.

Example

```
USE IFPORT
integer(4) I,Istatus
I = ACCESS ("DATAFILE.TXT", "w")
if (i) then
   ISTATUS = CHMOD ("datafile.txt", "[+w]")
end if
I = ACCESS ("DATAFILE.TXT","w")
print *, i
```

See Also

SETFILEACCESSQQ

CLASS

Statement: *Declares a polymorphic object. It takes one of the following forms:*

Syntax

```
CLASS (name) att-list :: v-list

CLASS (*) att-list :: v-list
```

name	Is the name of the extensible derived data type.
att-list	Is one or more attribute specifiers. These are the same attribute specifiers allowed for a derived-type TYPE statement.
v-list	Is the name of one or more data objects or functions. The name can optionally be followed by any of the following: • An array specification, if the object is an array. In a function declaration, an array must be a deferred-shape array if it has the POINTER attribute; otherwise, it must be an explicit-shape array. • A character length, if the object is of type character. • A coarray specification, if the object is a coarray. • A constant expression preceded by an = or, for pointer objects, => NULL( ).

Description

A polymorphic object can have differing types during program execution.

The type of the object at a particular point during execution of a program is its dynamic type.

The declared type of a data entity is the type that it is declared to have, either explicitly or implicitly.

If CLASS (*) is specified, it denotes an unlimited polymorphic object. An unlimited polymorphic entity is not declared to have a type. It is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.

An entity declared with the CLASS keyword must be a dummy argument or have the ALLOCATABLE or POINTER attribute.

A polymorphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same declared type or any of its extensions. Even though an unlimited polymorphic entity is not considered to have a declared type, it is type compatible with all entities. An entity is said to be type compatible with a type if it is type compatible with entities of that type.

A polymorphic allocatable object can be allocated to be of any type with which it is type compatible.

During program execution, a polymorphic pointer or dummy argument can be associated with objects with which it is type compatible.

See Also

Type declarations
TYPE Statement (Derived Types)

CLEARSCREEN (W*S)

Graphics Subroutine: *Erases the target area and fills it with the current background color.*

Module

USE IFQWIN

Syntax

`CALL CLEARSCREEN (area)`

area	(Input) INTEGER(4). Identifies the target area. Must be one of the following symbolic constants (defined in `IFQWIN.F90`): • $GCLEARSCREEN - Clears the entire screen. • $GVIEWPORT - Clears only the current viewport. • $GWINDOW - Clears only the current text window (set with SETTEXTWINDOW).

All pixels in the target area are set to the color specified with SETBKCOLORRGB. The default color is black.

Example

```
USE IFQWIN
CALL CLEARSCREEN($GCLEARSCREEN)
```

See Also

GETBKCOLORRGB
SETBKCOLORRGB
SETTEXTWINDOW
SETVIEWPORT

CLEARSTATUSFPQQ

Portability Subroutine: *Clears the exception flags in the floating-point processor status word.*

Module

USE IFPORT

Syntax

`CALL CLEARSTATUSFPQQ()`

Description

The floating-point status word indicates which floating-point exception conditions have occurred. Intel® Fortran initially clears (sets to 0) all floating-point status flags, but as exceptions occur, the status flags accumulate until the program clears the flags again. CLEARSTATUSFPQQ will clear the flags.

CLEARSTATUSFPQQ is appropriate for use in applications that poll the floating-point status register as the method for detecting a floating-point exception has occurred.

For a full description of the floating-point status word, exceptions, and error handling, see *Floating-Point Operations*: Floating-Point Environment.

Example

```
! Program to demonstrate CLEARSTATUSFPQQ.
! This program uses polling to detect that a
! floating-point exception has occurred.
! So, build this console application with the default
! floating-point exception behavior, fpe3.
! You need to specify compiler option /debug or /Od (Windows)
! or -O0 (Linux) to get the correct results.
!
! PROGRAM CLEARFP

  USE IFPORT

  REAL*4 A,B,C
  INTEGER*2 STS

  A = 2.0E0
  B = 0.0E0

! Poll and display initial floating point status
  CALL GETSTATUSFPQQ(STS)
  WRITE(*,'(1X,A,Z4.4)') 'Initial fp status = ',STS

! Cause a divide-by-zero exception
! Poll and display the new floating-point status
  C = A/B
  CALL GETSTATUSFPQQ(STS)
  WRITE(*,'(1X,A,Z4.4)') 'After div-by-zero fp status = ',STS

! If a divide by zero error occurred, clear the floating-point
! status register so future exceptions can be detected.
  IF ((STS .AND. FPSW$ZERODIVIDE) > 0) THEN
    CALL CLEARSTATUSFPQQ()
    CALL GETSTATUSFPQQ(STS)
    WRITE(*,'(1X,A,Z4.4)') 'After CLEARSTATUSFPQQ fp status = ',STS
  ENDIF

  END
```

This program is available in the online samples.

See Also

GETSTATUSFPQQ
SETCONTROLFPQQ
GETCONTROLFPQQ
SIGNALQQ

CLICKMENUQQ (W*S)

QuickWin Function: *Simulates the effect of clicking or selecting a menu command. The QuickWin application responds as though the user had clicked or selected the command.*

Module

USE IFQWIN

Syntax

```
result = CLICKMENUQQ (item)
```

item — (Input) INTEGER(4). Constant that represents the command selected from the Window menu. Must be one of the following symbolic constants (defined in `IFQWIN.F90`):

- QWIN$STATUS - Status command
- QWIN$TILE - Tile command
- QWIN$CASCADE - Cascade command
- QWIN$ARRANGE - Arrange Icons command

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

See Also

REGISTERMOUSEEVENT
UNREGISTERMOUSEEVENT
WAITONMOUSEEVENT

CLOCK

Portability Function: *Converts a system time into an 8-character ASCII string.*

Module

USE IFPORT

Syntax

```
result = CLOCK( )
```

Results

The result type is character with a length of 8. The result is the current time in the form hh:mm:ss, using a 24-hour clock.

Example

```
USE IFPORT
character(8) whatimeisit
whatimeisit = CLOCK ()
print *, 'The current time is ',whatimeisit
```

See Also

DATE_AND_TIME

CLOCKX

Portability Subroutine: *Returns the processor clock in units of microseconds.*

Module

USE IFPORT

Syntax

```
CALL CLOCKX (clock)
```

clock (Input) REAL(8). The current time.

On Windows systems, this subroutine has millisecond precision, and the last three digits of the returned value are not significant.

CLOSE

Statement: *Disconnects a file from a unit.*

Syntax

```
CLOSE ( [UNIT=] io-unit[, {STATUS | DISPOSE | DISP} = p] [, ERR= label] [, IOMSG=msg-var]
[, IOSTAT=i-var])
```

io-unit (Input) an external unit specifier.

p (Input) a scalar default character expression indicating the status of the file after it is closed. It has one of the following values:

- 'KEEP' or 'SAVE' - Retains the file after the unit closes.
- 'DELETE' - Deletes the file after the unit closes (unless OPEN(READONLY) is in effect).
- 'PRINT' - Submits the file to the line print spooler, then retains it (sequential files only).
- 'PRINT/DELETE' - Submits the file to the line print spooler, then deletes it (sequential files only).
- 'SUBMIT' - Forks a process to execute the file.
- 'SUBMIT/DELETE' - Forks a process to execute the file, then deletes the file after the fork is completed.

The default is 'DELETE' for user windows in Windows* QuickWin applications and for scratch files. For all other files, the default is 'KEEP'.

Scratch files are temporary and are always deleted upon normal program termination; specifying STATUS='KEEP' for scratch files causes a run-time error.

For user windows in Windows* QuickWin applications, STATUS='KEEP' causes the child window to remain on the screen even after the unit closes. The default status is 'DELETE', which removes the child window from the screen.

label Is the label of the branch target statement that receives control if an error occurs.

msg-var (Output) Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs.

i-var (Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

The CLOSE statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT= keyword is optional if the unit specifier is the first item in the I/O control list.

The status specified in the CLOSE statement supersedes the status specified in the OPEN statement, except that a file opened as a scratch file cannot be saved, printed, or submitted, and a file opened for read-only access cannot be deleted.

If a CLOSE statement is specified for a unit that is not open, it has no effect.

You do not need to explicitly close open files. Normal program termination closes each file according to its default status. The CLOSE statement does not have to appear in the same program unit that opened the file.

Closing unit 0 automatically reconnects unit 0 to the keyboard and screen. Closing units 5 and 6 automatically reconnects those units to the keyboard or screen, respectively.Closing the asterisk (*) unit causes a compile-time error. In Windows QuickWin applications, use CLOSE with unit 0, 5, or 6 to close the default window. If all of these units have been detached from the console (through an explicit OPEN), you must close one of these units beforehand to reestablish its connection with the console. You can then close the reconnect unit to close the default window.

If a parameter of the CLOSE statement is an expression that calls a function, that function must not cause an I/O operation or the EOF intrinsic function to be executed, because the results are unpredictable.

NOTE
You may get unexpected results if you specify OPEN with a filename and a USEROPEN specifier that opens a different filename, and then use a CLOSE statement with STATUS=DELETE (or DISPOSE=DELETE). In this case, the run-time library assumes you want to delete the file named in the OPEN statement, not the one you specified in the USEROPEN function. For more information about how to use the USEROPEN specifier, see User-Supplied OPEN Procedures: USEROPEN Specifier.

Example

```
!    Close and discard file:
     CLOSE (7, STATUS = 'DELETE')
```

Consider the following statement:

```
  CLOSE (UNIT=J, STATUS='DELETE', ERR=99)
```

This statement closes the file connected to unit J and deletes it. If an error occurs, control is transferred to the statement labeled 99.

See Also

Data Transfer I/O Statements
Branch Specifiers

CMPLX

Elemental Intrinsic Function (Specific): *Converts the argument to complex type. This function cannot be passed as an actual argument.*

Syntax

This intrinsic function can take one of the following forms:

`result = CMPLX (x [,kind])`

x	(Input) Must be of type complex.
kind	(Input; optional) Must be a scalar integer constant expression.

or

```
result = CMPLX (x [,y ,kind])
```

x	(Input) Must be of type integer, real, or a binary, octal, or hexadecimal literal constant.
y	(Input; optional) Must be of type integer, real, or a binary, octal, or hexadecimal literal constant.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is complex. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default real type.

If *x* is type complex, *y* must not be specified, and the result value is CMPLX (REAL (*x*), AIMAG (*x*)), with default kind if *kind* is not present, otherwise with the kind type as specified by *kind*.

If *x* is present and not of complex type, and *y* is not present, *x* is converted into the real part of the result value and zero is assigned to the imaginary part.

If *x* and *y* are present, the complex value is produced by converting the *x* into the real part of the value, and converting the *y* into the imaginary part.

CMPLX (*x*, *y*, *kind*) has the complex value whose real part is REAL (*x*, *kind*) and whose imaginary part is REAL (*y*, *kind*).

The setting of compiler options specifying real size can affect this function.

If the argument is a binary, octal, or hexadecimal literal constant, the result is affected by the `assume old-boz` option. The default option setting, `noold-boz`, treats the argument as a bit string that represents a value of the data type of the intrinsic, that is, the bits are not converted. If setting `old-boz` is specified, the argument is treated as a signed integer and the bits are converted.

NOTE
The result values of CMPLX are defined by references to the intrinsic function REAL with the same arguments. Therefore, the padding and truncation of binary, octal, and hexadecimal literal constant arguments to CMPLX is the same as for the intrinsic function REAL.

Example

CMPLX (-3) has the value (-3.0, 0.0).

CMPLX (4.1, 2.3) has the value (4.1, 2.3).

The following shows another example:

```
COMPLEX z1, z2
COMPLEX(8) z3
z1 = CMPLX(3)     ! returns the value 3.0 + i 0.0
z2 = CMPLX(3,4)   ! returns the value 3.0 + i 4.0
z3 = CMPLX(3,4,8) ! returns a COMPLEX(8) value 3.0D0 + i 4.0D0
```

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
Model for Bit Data
DCMPLX
FLOAT
INT
IFIX

REAL
SNGL

CO_BROADCAST

Collective Intrinsic Subroutine (Generic):
Broadcasts a value to other images.

Syntax

```
CALL CO_BROADCAST (a, source_image [, stat, errmsg])
```

a	(Input; output) Must have the same shape, dynamic type, and type parameter values in corresponding references across all participating images. It cannot be a coindexed object. If an error occurs, it becomes undefined. Otherwise, *a* becomes defined as if by intrinsic assignment with the value *a* on image *source_image* on all images of the current team.
source_image	(Input) Must be a scalar integer. The value of *source_image* must the value of an image index of an image on the current team. Its value must be the same in corresponding references on all images participating in the collective operation.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Collective Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.
errmsg	(Input; output; optional) Must be a non-coindexed default character scalar variable. The semantics of *errmsg* is described in *Overview of Collective Subroutines*.

Example

Consider the following:

```
CALL CO_BROADCAST (R, 5)
```

If R is a four-element array defined with the value [10, 20, 30, 40] on image 5 when the subroutine is referenced, R becomes defined with the value [10, 20, 30, 40] on all images of the current team if no error condition occurs during the subroutine reference.

See Also

Overview of Collective Subroutines

CO_MAX

Collective Intrinsic Subroutine (Generic):
Calculates the maximum value across images.

Syntax

```
CALL CO_MAX (a [, result_image, stat, errmsg])
```

a	(Input; output) Must be of type real, integer, or charater, and have the same shape, type, and type parameter values in corresponding references across all images of the current team. It cannot be a

	coindexed object. If it is scalar, the computed value is the maximum value of *a* in all corresponding references. If it is an array, each element of the computed value is equal to the maximum value of the corresponding element of *a* in all corresponding references. If no error occurs, the computed value is assigned to *a* on all images of the current team if *result_image* is not present, or on the executing image if the executing image is the image identified by *result_image*. Otherwise, *a* becomes undefined.
result_image	(Input; optional) Must be a scalar integer. If present, it must be present with the same value in all corresponding references and be a valid image index in the current team. If *result_image* is not present, it cannot be present in any corresponding reference.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Collective Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.
errmsg	(Input; output; optional) Must be a non-coindexed default character scalar variable. The semantics of *errmsg* is described in *Overview of Collective Subroutines*.

Example

Consider the following:

```
CALL CO_MAX (R)
```

If there are two images and R is a four-element array defined with the value [5, 10, 20, 15] on image one and [10, 15, 20, 5] on image two when the subroutine is referenced, R becomes defined with the value [10, 15, 20, 15] on both images if no error occurs during the subroutine reference, and CALL CO_MAX (R, 2) causes R on image to become defined with the values [10, 15, 20, 15] on image 2; R on image 1 becomes undefined.

See Also

Overview of Collective Subroutines

CO_MIN

Collective Intrinsic Subroutine (Generic):
Calculates the minimum value across images.

Syntax

```
CALL CO_MIN (a [, result_image, stat, errmsg])
```

a	(Input; output) Must be of type real, integer, or charater, and have the same shape, type, and type parameter values in corresponding references across all images of the current team. It cannot be a coindexed object. If it is scalar, the computed value is the minimum value of *a* in all corresponding references. If it is an array, each element of the computed value is equal to the maximum value of the corresponding element of *a* in all corresponding references.

	If no error occurs, the computed value is assigned to *a* on all images of the current team if *result_image* is not present, or on the executing image if the executing image is the image identified by *result_image*. Otherwise, *a* becomes undefined.
result_image	(Input; optional) Must be a scalar integer. If present, it must be present with the same value in all corresponding references and be a valid image index in the current team. If *result_image* is not present, it cannot be present in any corresponding reference.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Collective Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.
errmsg	(Input; output; optional) Must be a non-coindexed default character scalar variable. The semantics of *errmsg* is described in *Overview of Collective Subroutines*.

Example

Consider the following:

```
CALL CO_MIN (R)
```

If there are two images, and R is a four-element array defined with the value [5, 10, 20, 15] on image one and [10, 15, 20, 5] on image two when the subroutine is referenced, R becomes defined with the value [5, 10, 20, 5] on both images if no error occurs during the subroutine reference, and CALL CO_MIN (R, 1) causes R on image 1 to become defined with the value [5, 10, 20 5]; R on image 2 becomes undefined.

See Also

Overview of Collective Subroutines

CO_REDUCE

Collective Intrinsic Subroutine (Generic):
Performs generalized reduction across images.

Syntax

```
CALL CO_REDUCE (a, operation [, result_image, stat, errmsg])
```

a	(Input; output) Must be non-polymorphic, non-coindexed, with the same shape, type and type parameters in corresponding references. It may not be a coindexed object. If *a* is scalar, the computed value is the result of the reduction of applying operation to the values of *a* in all corresponding references. If *a* is an array, each element of the computed value is equal to the result of the reduction of the reduction operation of applying operation to corresponding elements of *a* in all corresponding references. If no error occurs, the computed value is assigned to *a* on all images of the current team if *result_image* is not present, or on the executing image if the executing image is the image identified by *result_image*. Otherwise, *a* becomes undefined.

operation	Is a pure function with exactly two arguments whose result and each argument are a scalar, nonallocatable, nonpointer, nonpolymorphic data object with the same type and kind type parameters as *a*. Neither argument can be optional. If one argument has the attribute TARGET, VOLATILE, or ASYNCHRONOUS, the other argument must have that attribute. *operation* must implement a mathematical associative operation and be the same in each corresponding reference. If operation is not commutative, the computed value may depend on the order of evaluation. The computed value of a reduction operation over a set of values is the result of an iterative process. Each iteration evaluates operation (x, y) for x and y in that set, the removal of x and y from that set, and the addition of the result of operation (x, y) to that set. The process ends when the set has a single value, which is the computed value.
result_image	(Input; optional) Must be a scalar integer. If present, it must be present with the same value in all corresponding references and be a valid image index in the current team. If *result_image* is not present, it cannot be present in any corresponding reference.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Collective Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.
errmsg	(Input; output; optional) Must be a non-coindexed default character scalar variable. The semantics of *errmsg* is described in*Overview of Collective Subroutines*.

Example

The following subroutine demonstrates how CO_REDUCE can be used to create a collective version of the intrinsic function ANY.

```
SUBROUTINE CO_ANY (VALUE)
  LOGICAL,INTENT(INOUT) :: VALUE
  CALL CO_REDUCE (VALUE, COMBINER)
  CONTAINS
    PURE FUNCTION COMBINER (OPND1, OPND2)RESULT = LOGICAL_SUM
      LOGICAL                 :: LOGICAL_SUM
      LOGICAL,INTENT(IN)      :: OPND1, OPND2
      LOGICAL_SUM = OPND1 .OR. OPND2
    END FUNCTION COMBINER
END SUBROUTINE CO_ANY
```

If the number of images is two, and R is a four-element logical array with the value [.T., .T., .F., .F.] on image one and [.T., .F., .T., .F.] on image two, CALL CO_REDUCE (R, CO_ANY) causes the value of R to become defined with the value of [.T., .T., .T., .F.] on both images if no error occurs during the reference.

See Also

Overview of Collective Subroutines

CO_SUM

Collective Intrinsic Subroutine (Generic):
Performs a sum reduction across images.

Syntax

```
CALL CO_MAX (a [, result_image, stat, errmsg])
```

a	(Input; output) Must be of type real, integer, or complex, and have the same shape, type, and type parameter values in corresponding references. It may not be a coindexed object. If it is scalar, the computed value is the sum of the values of *a* in all corresponding references. If it as an array, each element of the computed value is equal to the sum of the values of the corresponding element of *a* in all corresponding references. If no error occurs, the computed value is assigned to *a* on all images of the current team if *result_image* is not present, or on the executing image if the executing image is the image identified by *result_image*. Otherwise, *a* becomes undefined.
result_image	(Input; optional) Must be a scalar integer. If present, it must be present with the same value in all corresponding references and be a valid image index in the current team. If *result_image* is not present, it may not be present in any corresponding reference.
stat	(Output; optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least four (KIND=2 or greater). The value assigned to *stat* is specified in *Overview of Collective Subroutines*. If *stat* is not present and an error condition occurs, error termination is initiated.
errmsg	(Input; output; optional) Must be a non-coindexed default character scalar variable. The semantics of errmsg is described in *Overview of Collective Subroutines*.

Example

Consider the following:

```
CALL CO_SUM (R, 2)
```

If the number of images is two and if R is a four-element array defined with the value [5, 10, 20, 15] on image one and [10, 15, 20, 5] on image two when the procedure is references, R becomes defined with the value [15, 25, 40, 20] on image two and undefined on image one if no error occurs during the subroutine reference, and CALL CO_SUM (R) causes R on both images to become defined with the value [15, 25, 40, 20].

See Also

Overview of Collective Subroutines

CODE_ALIGN

General Compiler Directive:*Specifies the byte alignment for a loop.*

Syntax

```
!DIR$ CODE_ALIGN [:n]
```

n	(Optional) A positive integer constant expression indicating the number of bytes for the minimum alignment boundary. Its value must be a power of 2 between 1 and 4096, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on. If you specify 1 for *n*, no alignment is performed. If you do not specify *n*, the default alignment is 16 bytes.

This directive must precede the loop or block of code to be aligned.

If code is compiled with the `-falign-loops=m` (Linux* and macOS*) or `/Qalign-loops:m` (Windows*) option and a "CODE_ALIGN:n" directive precedes a loop, the loop is aligned on a MAX (m, n) byte boundary. If a procedure has the "CODE_ALIGN:k" attribute and a "CODE_ALIGN:n" directive precedes a loop, then both the procedure and the loop are aligned on a MAX (k, n) byte boundary.

Example

Consider the following code fragment in file `test_code_align.f90`:

```
!DIR$ CODE_ALIGN
DO J = 1, N
…
END DO
```

Compiling `test_code_align.f90` aligns the code that begins the DO J loop on a (default) 16-byte boundary. If you do not specify the CODE_ALIGN directive, the alignment of the loop is implementation-dependent and may change from compilation to compilation.

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
falign-loops, Qalign-loops compiler option
ATTRIBUTES CODE_ALIGN directive

CODIMENSION

Statement and Attribute: *Specifies that an entity is a coarray, and specifies its corank and cobounds, if any.*

Syntax

The CODIMENSION attribute can be specified in a type declaration statement or a CODIMENSION statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] CODIMENSION [*coarray-spec*] :: *var-list* [(*array-spec*)]...

type, [*att-ls*,] *var-list* [(*array-spec*)] [*coarray-spec*]...

Statement:

CODIMENSION [::] *var-list* [*coarray-spec*]...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.

coarray-spec Is an allocatable (deferred-coshape) coarray or an explicit-coshape coarray.

var-list Is a list of variable names, separated by commas.

Description

In Intel® Fortran, the sum of the rank and corank of an entity must not exceed 31. The Fortran 2008 Standard allows a combined rank of up to 15.

A coarray must be a component or a variable that is not a function result.

A coarray must not be of type C_PTR or C_FUNPTR.

An entity whose type has a coarray ultimate component must be a nonpointer nonallocatable scalar, must not be a coarray, and must not be a function result.

A coarray or an object with a coarray ultimate component must be an associate name, a dummy argument, or have the ALLOCATABLE or SAVE attribute.

A coarray must not be a dummy argument of a procedure that has a BIND attribute.

A coarray can be a derived type with pointer or allocatable components. The target of such a pointer component is always on the same image as the pointer.

Examples

Explicit-shape coarrays that are not dummy arguments must have the SAVE attribute. Because of this, automatic coarrays are not allowed. For example, coarray TASK in the following code is not valid:

```
SUBROUTINE SUBA(I,C,D)
INTEGER :: I
REAL :: C(I)[*], D(I)
REAL :: TASK(I)[*]        ! Not permitted
```

The following lines show valid examples of CODIMENSION attribute specifications:

```
REAL, CODIMENSION[3,*] :: B(:)   ! Assumed-shape coarray
REAL R(50,50)[0:5,*]             ! Explicit-shape coarray
REAL, CODIMENSION[*] :: A        ! Scalar coarray
REAL, CODIMENSION[:],ALLOCATABLE :: C(:,:)  ! Allocatable coarray
```

COLLAPSE Clause

Parallel Directive Clause: *Specifies how many loops are associated with a loop construct.*

Syntax

`COLLAPSE (n)`

n Must be a constant positive scalar integer expression.

If *n* is greater than one, the iterations of all associated loops are collapsed into one larger iteration, which is then divided according to the SCHEDULE clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration. The associated loops must be perfectly nested, that is, there must be no intervening code or any OpenMP* directive between any two loops.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified. The integer kind used to compute the iteration count for the collapsed loop is implementation defined.

If COLLAPSE is not specified, the only loop that is associated with the loop construct is the one that immediately follows the construct.

At most one COLLAPSE clause can appear in a directive that allows the clause.

See Nested DO Constructs for restrictions on perfectly nested loops using COLLAPSE.

If more than one loop is associated with a TASKLOOP construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the specifications in the GRAINSIZE and NUM_TASKS clauses. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

COMAddObjectReference (W*S)

***COM Function**: Adds a reference to an object's interface.*

Module

USE IFCOM

Syntax

```
result = COMAddObjectReference (iunknown)
```

iunknown — An IUnKnown interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

Results

The result type is INTEGER(4). It is the object's current reference count.

See Also

IUnknown::AddRef in the Microsoft* Platform SDK

COMCLSIDFromProgID (W*S)

***COM Subroutine**: Passes a programmatic identifier and returns the corresponding class identifier.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMCLSIDFromProgID (prog_id,clsid,status)
```

prog_id — The programmatic identifier of type CHARACTER*(*).

clsid — The class identifier corresponding to the programmatic identifier. Must be of type GUID, which is defined in the IFWINTY module.

status	The status of the operation. It can be any status returned by CLSIDFromProgID. Must be of type INTEGER(4).

See Also

CLSIDFromProgID in the Microsoft* Platform SDK

COMCLSIDFromString (W*S)

***COM Subroutine**: Passes a class identifier string and returns the corresponding class identifier.*

Module

USE IFCOM

USE IFWINTY

Syntax

`CALL COMCLSIDFromString (string,clsid,status)`

string	The class identifier string of type CHARACTER*(*).
clsid	The class identifier corresponding to the identifier string. Must be of type GUID, which is defined in the IFWINTY module.
status	The status of the operation. It can be any status returned by CLSIDFromString. Must be of type INTEGER(4).

See Also

CLSIDFromString in the Microsoft* Platform SDK

COMCreateObject (W*S)

***COM Subroutine**: A generic routine that executes either COMCreateObjectByProgID or COMCreateObjectByGUID.*

Module

USE IFCOM

USE IFWINTY

Description

Your application obtains its first pointer to an object's interface by calling `COMCreateObject`. It creates a new instance of an object class and returns a pointer to it.

See Also

COMCreateObjectByGUID
COMCreateObjectByProgID

COMCreateObjectByGUID (W*S)

***COM Subroutine**: Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.*

Module

USE IFCOM

USE IFWINTY

Syntax

CALL COMCreateObjectByGUID (*clsid*,*clsctx*,*iid*,*interface*,*status*)

clsid	The class identifier of the class of object to be created. Must be of type GUID, which is defined in the IFWINTY module.
clsctx	Lets you restrict the types of servers used for the object. Must be of type INTEGER(4). Must be one of the CLSCTX_* constants defined in the IFWINTY module.
iid	The interface identifier of the interface being requested. Must be of type GUID, which is defined in the IFWINTY module.
interface	An output argument that returns the object's interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by CoCreateInstance. Must be of type INTEGER(4).

See Also

CoCreateInstance in the Microsoft* Platform SDK

Using the Intel(R) Fortran Module Wizard (COM Client) in Key Features: Fortran Language Extensions

COMCreateObjectByProgID (W*S)

***COM Subroutine**: Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.*

Module

USE IFCOM

Syntax

CALL COMCreateObjectByProgID (*prog_id*,*idispatch*,*status*)

prog_id	The programmatic identifier of type CHARACTER*(*).
idispatch	An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by CLSIDFromProgID or CoCreateInstance. Must be of type INTEGER(4).

See Also

COMCLSIDFromProgID

CoCreateInstance in the OLE section of the Microsoft* Platform SDK

Using the Intel(R) Fortran Module Wizard (COM Client) in Key Features: Fortran Language Extensions

COMGetActiveObjectByGUID (W*S)

***COM Subroutine**: Passes a class identifier and returns a pointer to the interface of a currently active object.*

Module

USE IFCOM

USE IFWINTY

Syntax

`CALL COMGetActiveObjectByGUID (clsid,iid,interface,status)`

clsid	The class identifier of the class of object to be found. Must be of type GUID, which is defined in the IFWINTY module.
iid	The interface identifier of the interface being requested. Must be of type GUID, which is defined in the IFWINTY module.
interface	An output argument that returns the object's interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by GetActiveObject. Must be of type INTEGER(4).

See Also

GetActiveObject in the Microsoft* Platform SDK

COMGetActiveObjectByProgID (W*S)

***COM Subroutine**: Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.*

Module

USE IFCOM

Syntax

`CALL COMGetActiveObjectByProgID (prog_id,idispatch,status)`

prog_id	The programmatic identifier of type CHARACTER*(*).
idispatch	An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by CLSIDFromProgID or GetActiveObject. Must be of type INTEGER(4).

Example

See the example in COMInitialize.

See Also

CLSIDFromProgID and GetActiveObject in the Microsoft* Platform SDK

COMGetFileObject (W*S)

***COM Subroutine**: Passes a file name and returns a pointer to the IDispatch interface of an automation object that can manipulate the file.*

Module

USE IFCOM

Syntax

CALL COMGetFileObject (*filename*,*idispatch*,*status*)

filename	The path of the file of type CHARACTER*(*).
idispatch	An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by the CreateBindCtx or MkParseDisplayName routines, or the IMoniker::BindToObject method. Must be of type INTEGER(4).

See Also

CreateBindCtx, MkParseDisplayName, and IMonker::BindToObject in the Microsoft* Platform SDK

COMInitialize (W*S)

***COM Subroutine**: Initializes the COM library.*

Module

USE IFCOM

Syntax

CALL COMInitialize (*status*)

status	The status of the operation. It can be any status returned by OleInitialize. Must be of type INTEGER(4).

You must use this routine to initialize the COM library before calling any other COM or AUTO routine.

Example

Consider the following:

```
program COMInitExample

  use ifwin
  use ifcom
  use ifauto

      implicit none

      ! Variables
  integer(4) word_app
  integer(4) status
  integer(INT_PTR_KIND()) invoke_args

  call COMInitialize(status)
```

```
      ! Call GetActiveObject to get a reference to a running MS WORD application
      call COMGetActiveObjectByProgID("Word.Application", word_app, status)
  if (status >= 0) then
              ! Print the active document
              invoke_args = AutoAllocateInvokeArgs()
              call AutoAddArg(invoke_args, "Copies", 2)
              status = AutoInvoke(word_app, "PrintOut", invoke_args)
              call AutoDeallocateInvokeArgs(invoke_args)
      ! Release the reference
      status = COMReleaseObject(word_app)
  end if

  call COMUninitialize()

end program
```

See Also

OleInitialize in the Microsoft* Platform SDK

COMIsEqualGUID (W*S)

***COM Function**: Determines whether two globally unique identifiers (GUIDs) are the same.*

Module

USE IFCOM

USE IFWINTY

Syntax

`result = COMIsEqualGUID (guid1,guid2)`

guid1	The first GUID. Must be of type GUID, which is defined in the IFWINTY module. It can be any type of GUID, including a class identifier (CLSID), or an interface identifier (IID).
guid2	The second GUID, which will be compared to *guid1*. It must be the same type of GUID as *guid1*. For example, if *guid1* is a CLSID, *guid2* must also be a CLSID.

Results

The result type is LOGICAL(4). The result is .TRUE. if the two GUIDs are the same; otherwise, .FALSE.

See Also

IsEqualGUID in the Microsoft* Platform SDK

COMMAND_ARGUMENT_COUNT

***Inquiry Intrinsic Function (Generic):** Returns the number of command arguments.*

Syntax

`result = COMMAND_ARGUMENT_COUNT ()`

Results

The result is a scalar of type default integer. The result value is equal to the number of command arguments available. If there are no command arguments available, the result is 0. The command name does not count as one of the command arguments.

Example

Consider the following:

```
program echo_command_line
integer i, cnt, len, status
character c*30, b*100

call get_command (b, len, status)
if (status .ne. 0) then
   write (*,*) 'get_command failed with status = ', status
   stop
end if
write (*,*) 'command line = ', b (1:len)

call get_command_argument (0, c, len, status)
if (status .ne. 0) then
   write (*,*) 'Getting command name failed with status = ', status
   stop
end if
write (*,*) 'command name = ', c (1:len)

cnt = command_argument_count ()
write (*,*) 'number of command arguments = ', cnt

do i = 1, cnt
   call get_command_argument (i, c, len, status)
   if (status .ne. 0) then
     write (*,*) 'get_command_argument failed: status = ', status, ' arg = ', i
     stop
   end if
   write (*,*) 'command arg ', i, ' = ', c (1:len)
end do

write (*,*) 'command line processed'
end
```

If the above program is invoked with the command line " `echo_command_line.exe -o 42 -a hello b`", the following is displayed:

```
command line = echo_command_line.exe -o 42 -a hello b
command name = echo_command_line.exe
number of command arguments = 5
command arg 1 = -o
command arg 2= 42
command arg 3 = -a
command arg 4 = hello
command arg 5 = b
command line processed
```

See Also

GETARG
NARGS
IARGC

GET_COMMAND
GET_COMMAND_ARGUMENT

COMMITQQ

Run-Time Function: *Forces the operating system to execute any pending write operations for the file associated with a specified unit to the file's physical device.*

Module

USE IFCORE

Syntax

`result = COMMITQQ (unit)`

unit	(Input) INTEGER(4). A Fortran logical unit attached to a file to be flushed from cache memory to a physical device.

Results

The result type is LOGICAL(4). If an open unit number is supplied, .TRUE. is returned and uncommitted records (if any) are written. If an unopened unit number is supplied, .FALSE. is returned.

Data written to files on physical devices is often initally written into operating-system buffers and then written to the device when the operating system is ready. Data in the buffer is automatically flushed to disk when the file is closed. However, if the program or the computer crashes before the data is transferred from buffers, the data can be lost. COMMITQQ tells the operating system to write any cached data intended for a file on a physical device to that device immediately. This is called flushing the file.

COMMITQQ is most useful when you want to be certain that no loss of data occurs at a critical point in your program; for example, after a long calculation has concluded and you have written the results to a file, or after the user has entered a group of data items, or if you are on a network with more than one program sharing the same file. Flushing a file to disk provides the benefits of closing and reopening the file without the delay.

Example

```
USE IFCORE
INTEGER unit / 10 /
INTEGER len
CHARACTER(80) stuff
OPEN(unit, FILE='COMMITQQ.TST', ACCESS='Sequential')
DO WHILE (.TRUE.)
  WRITE (*, '(A, \)') 'Enter some data (Hit RETURN to &
                       exit): '
  len = GETSTRQQ (stuff)
  IF (len .EQ. 0) EXIT
  WRITE (unit, *) stuff
  IF (.NOT. COMMITQQ(unit)) WRITE (*,*) 'Failed'
END DO
CLOSE (unit)
END
```

See Also

PRINT
WRITE

COMMON

Statement: *Defines one or more contiguous areas, or blocks, of physical storage (called common blocks) that can be accessed by any of the scoping units in an executable program. COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items. COMMON is an obsolescent language feature in Standard Fortran.*

Syntax

COMMON [/[*cname*]/] *var-list*[[,] /[*cname*]/ *var-list*]...

cname — (Optional) Is the name of the common block. The name can be omitted for blank common (//).

var-list — Is a list of variable names, separated by commas.

The variable must not be a dummy argument, allocatable array, automatic object, function, function result, a variable with the BIND attribute,or entry to a procedure. It must not have the PARAMETER attribute. If an object of derived type is specified, it must be a sequence type or a type with the BIND attribute.

Description

Common blocks can be named or unnamed (a *blank common*).

A common block is a global entity, and must not have the same name as any other global entity in the program, such as a subroutine or function.

Any common block name (or blank common) can appear more than once in one or more COMMON statements in a program unit. The list following each successive appearance of the same common block name is treated as a continuation of the list for the block associated with that name. Consider the following COMMON statements:

```
COMMON /glenn/ lovell, armstrong, aldrin
COMMON /     / shepard, grissom, carpenter
COMMON /glenn/ borman, anders
COMMON /young/ mcdivitt, white, conrad
COMMON schirra, cooper
```

They are equivalent to these COMMON statements:

```
COMMON /glenn/ lovell, armstrong, aldrin, borman, anders
COMMON         shepard, grissom, carpenter, schirra, cooper
COMMON /young/ mcdivitt, white, conrad
```

A variable can appear in only one common block within a scoping unit.

A common block object must *not* be one of the following:

- A dummy argument
- A result variable
- An allocatable variable
- A derived-type object with an ultimate component that is allocatable
- A procedure pointer
- An automatic object
- A variable with the BIND attribute

- An unlimited polymorphic pointer
- A coarray

If an array is specified, it can be followed by an explicit-shape array specification, each bound of which must be a constant specification expression. Such an array must not have the POINTER attribute.

A pointer can only be associated with pointers of the same type and kind parameters, and rank.

An object with the TARGET attribute can only be associated with another object with the TARGET attribute and the same type and kind parameters.

A nonpointer can only be associated with another nonpointer, but association depends on their types, as follows:

Type of Variable	Type of Associated Variable
Intrinsic numeric [1]or numeric sequence [2]	Can be of any of these types
Default character or character sequence [2]	Can be of either of these types
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

[1]Default integer, default real, double precision real, default complex, double complex, or default logical.

[2]If an object of numeric sequence or character sequence type appears in a common block, it is as if the individual components were enumerated directly in the common list.

So, variables can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
COMMON /QUANTA/ A, Y
```

When common blocks from different program units have the same name, they share the same storage area when the units are combined into an executable program.

Entities are assigned storage in common blocks on a one-for-one basis. So, the data type of entities assigned by a COMMON statement in one program unit should agree with the data type of entities placed in a common block by another program unit. For example:

Program Unit A	Program Unit B
`COMMON CENTS`	`INTEGER(2) MONEY`
. . .	`COMMON MONEY`
	. . .

When these program units are combined into an executable program, incorrect results can occur if the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

Named common blocks must be declared to have the same size in each program unit. Blank common can have different lengths in different program units.

NOTE
If a common block is initialized by a DATA statement, the module containing the initialization must declare the common block to be its maximum defined length.

This limitation does not apply if you compile all source modules together.

Example

```
PROGRAM MyProg
COMMON i, j, x, k(10)
COMMON /mycom/ a(3)
...
END
SUBROUTINE MySub
COMMON pe, mn, z, idum(10)
COMMON /mycom/ a(3)
...
END
```

In the following example, the COMMON statement in the main program puts HEAT and X in blank common, and KILO and Q in a named common block, BLK1:

Main Program	Subprogram
`COMMON HEAT,X /BLK1/KILO,Q`	`SUBROUTINE FIGURE`
`...`	`COMMON /BLK1/LIMA,R / /ALFA,BET`
	`...`
`CALL FIGURE`	
`...`	`RETURN`
	`END`

The COMMON statement in the subroutine makes ALFA and BET share the same storage location as HEAT and X in blank common. It makes LIMA and R share the same storage location as KILO and Q in BLK1.

The following example shows how a COMMON statement can be used to declare arrays:

```
COMMON / MIXED / SPOTTED(100), STRIPED(50,50)
```

The following example shows a valid association between subroutines in different program units. The object lists agree in number, type, and kind of data objects:

```
SUBROUTINE unit1
REAL(8)      x(5)
INTEGER      J
CHARACTER    str*12
TYPE(member) club(50)
COMMON / blocka / x, j, str, club
...
SUBROUTINE unit2
REAL(8)      z(5)
INTEGER      m
CHARACTER    chr*12
TYPE(member) myclub(50)
COMMON / blocka / z, m, chr, myclub
...
```

See also the example for BLOCK DATA.

See Also

BLOCK DATA
DATA

MODULE
EQUIVALENCE
Specification expressions
Storage association
Interaction between COMMON and EQUIVALENCE Statements
Obsolescent Language Features in the Fortran Standard

COMPILER_OPTIONS

Module Intrinsic Inquiry Function: *Returns a string containing the compiler options that were used for compilation.*

Module

USE ISO_FORTRAN_ENV

Syntax

```
result = COMPILER_OPTIONS( )
```

Results

The result is a scalar of type default character of processor-defined length.

The return value is a list of the compiler options that were used for compilation.

Example

Consider the following file named `t.f90`:

```
use ISO_FORTRAN_ENV

character (len = :), allocatable :: res

res = compiler_options ()
print *, "len of res is: ", len (res)
print *, "('<<', A, '>>')") res

deallocate (res)
end
```

The following is the output:

On Windows:

```
Win>ifort /exe:t /warn:alignments /assume:writeable-strings t.f90
Intel(R) Visual Fortran Compiler for applications running on architecture,
Version version Built date-and-time by user on platform in directory
Copyright (C) 1985-2016 Intel Corporation.  All rights reserved.

Microsoft (R) Incremental Linker Version version
Copyright (C) Microsoft Corporation.  All rights reserved.

-out:t.exe
-subsystem:console
t.obj

Win> t.exe
len of compiler_options is:           48
<</exe:t /warn:alignments /assume:writeable-strings>>
```

On Linux:

```
Lin$ ifort -o t.out -warn alignments -assume writeable-strings t.f90
Lin$ ./t.out
len of res is:           51
<<-o t.out -warn alignments -assume writeable-strings>>
```

See Also

COMPILER_VERSION
ISO_FORTRAN_ENV Module

COMPILER_VERSION

Module Intrinsic Inquiry Function: *Returns a string containing the name and version number of the compiler used for compilation.*

Module

USE ISO_FORTRAN_ENV

Syntax

```
result = COMPILER_VERSION( )
```

Results

The result is a scalar of type default character of processor-defined length.

The return value contains the name and version number of the compiler used for compilation.

Example

Consider the following file named `t.f90`:

```
use ISO_FORTRAN_ENV

character (len = :), allocatable :: res

res = compiler_version ()
print *, "len of res is: ", len (res)
print *, "('<<', A, '>>')", res

deallocate (res)
close (1)
end
```

The following is the output:

On Windows, with generic labels like *user* so the length displayed is not accurate:

```
Win>ifort /exe:t /warn:alignments /assume:writeable-strings t.f90
Intel(R) Visual Fortran Compiler for applications running on architecture,
Version version Built date-and-time by user on platform in directory
Copyright (C) 1985-2016 Intel Corporation.  All rights reserved.

Microsoft (R) Incremental Linker Version version
Copyright (C) Microsoft Corporation.  All rights reserved.

-out:t.exe
-subsystem:console
t.obj
```

```
Win> t.exe
len of res is:          184

<<Intel(R) Visual Fortran Compiler for applications running on architecture,
Version version Built date-and-time by user on platform in directory>>

Win>
```

On Linux, with generic labels like *user* so the length displayed is not accurate:

```
Lin$ ifort -o t.out -warn alignments -assume writeable-strings t.f90
Lin$ ./t.out
len of res is:          202

<<Intel(R) Fortran Intel(R) architecture Compiler for applications running
on Intel(R) architecture, Version version
Built date-and-time by user on platform in directory>>

Lin$
```

See Also

COMPILER_OPTIONS
ISO_FORTRAN_ENV Module

COMPLEX Statement

Statement: *Specifies the COMPLEX data type.*

Syntax

`COMPLEX`

`COMPLEX([KIND=] n)`

`COMPLEX*s`

`DOUBLE COMPLEX`

n	Is kind 4, 8, or 16.
s	Is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type default complex.

DOUBLE COMPLEX is COMPLEX(8). No kind parameter is permitted for data declared with type DOUBLE COMPLEX.

Example

```
COMPLEX ch
COMPLEX (KIND=4),PRIVATE :: zz, yy   !equivalent to COMPLEX*8 zz, yy
COMPLEX(8) ax, by        !equivalent to COMPLEX*16 ax, by
COMPLEX (kind(4)) y(10)
complex (kind=8)  x, z(10)
```

See Also

DOUBLE COMPLEX
Complex Data Type

COMPLEX(4) Constants
COMPLEX(8) or DOUBLE COMPLEX Constants
Data Types, Constants, and Variables

COMPLINT, COMPLREAL, COMPLLOG

Portability Functions: *Return a BIT-WISE complement or logical .NOT. of the argument.*

Module

USE IFPORT

Syntax

`result = COMPLINT (intval)`

`result = COMPREAL (realval)`

`result = COMPLLOG (logval)`

intval	(Input) INTEGER(4).
realval	(Input) REAL(4).
logval	(Input) LOGICAL(4).

Results

The result is INTEGER(4) for COMPLINT, REAL(4) for COMPLREAL and LOGICAL(4) for COMPLLOG with a value that is the bitwise complement of the argument.

COMQueryInterface (W*S)

COM Subroutine*: Passes an interface identifier and returns a pointer to an object's interface.*

Module

USE IFCOM

USE IFWINTY

Syntax

`CALL COMQueryInterface (iunknown,iid,interface,status)`

iunknown	An IUnknown interface pointer. Must be of type INTEGER(4).
iid	The interface identifier of the interface being requested. Must be of type GUID, which is defined in the IFWINTY module.
interface	An output argument that returns the object's interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by the IUnknown method QueryInterface. Must be of type INTEGER(4).

See Also

IUnknown::QueryInterface in the Microsoft* Platform SDK

Using the Intel(R) Fortran Module Wizard (COM Client) in Key Features: Fortran Language Extensions

COMReleaseObject (W*S)

***COM Function**: Indicates that the program is done with a reference to an object's interface.*

Module

USE IFCOM

Syntax

`result = COMReleaseObject (iunknown)`

iunknown	An IUnknown interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

Results

The result type is INTEGER(4). It is the object's current reference count.

Example

See the example in COMInitialize.

COMStringFromGUID (W*S)

***COM Subroutine**: Passes a globally unique identifier (GUID) and returns a string of printable characters.*

Module

USE IFCOM

USE IFWINTY

Syntax

`CALL COMStringFromGUID (guid,string,status)`

guid	The GUID to be converted. Must be of type GUID, which is defined in the IFWINTY module. It can be any type of GUID, including a class identifier (CLSID), or an interface identifier (IID).
string	A character variable of type CHARACTER*(*) that receives the string representation of the GUID. The length of the character variable should be at least 38.
status	The status of the operation. If the string is too small to contain the string representation of the GUID, the value is zero. Otherwise, the value is the number of characters in the string representation of the GUID. Must be of type INTEGER(4).

The string representation of a GUID has a format like that of the following:

[c200e360-38c5-11ce-ae62-08002b2b79ef]

where the successive fields break the GUID into the form DWORD-WORD-WORD-WORD-WORD.DWORD covering the 128-bit GUID. The string includes enclosing braces, which are an OLE convention.

See Also

StringFromGUID2 in the Microsoft* Platform SDK

COMUninitialize (W*S)

***COM Subroutine**: Uninitializes the COM library.*

Module

USE IFCOM

Syntax

```
CALL COMUninitialize( )
```

When using COM routines, this must be the last routine called.

Example

See the example in COMInitialize.

CONJG

***Elemental Intrinsic Function (Generic):** Calculates the conjugate of a complex number.*

Syntax

```
result = CONJG (z)
```

z (Input) Must be of type complex.

Results

The result type and kind are the same as *z*. If *z* has the value (x, y), the result has the value (x, -y).

Specific Name	Argument Type	Result Type
CONJG	COMPLEX(4)	COMPLEX(4)
DCONJG	COMPLEX(8)	COMPLEX(8)
QCONJG	COMPLEX(16)	COMPLEX(16)

Example

CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

CONJG ((1.0, -4.2)) has the value (1.0, 4.2).

The following shows another example:

```
COMPLEX z1
COMPLEX(8) z2
z1 = CONJG((3.0, 5.6))       ! returns (3.0, -5.6)
z2 = DCONJG((3.0D0, 5.6D0))  ! returns (3.0D0, -5.6D0)
```

See Also

AIMAG

CONTAINS

Statement: *Separates the body of a main program, module, submodule, or external subprogram from any internal or module procedures it may contain, or it introduces the type-bound procedure part of a derived-type definition. It is not executable.*

Syntax

```
CONTAINS
```

Any number of internal procedures can follow a CONTAINS statement, but a CONTAINS statement cannot appear in the internal procedures themselves.

An empty CONTAINS section is allowed.

Example

```
PROGRAM OUTER
   REAL, DIMENSION(10) :: A
   . . .
   CALL INNER (A)
CONTAINS
   SUBROUTINE INNER (B)
   REAL, DIMENSION(10) :: B
   . . .
   END SUBROUTINE INNER
END PROGRAM OUTER
```

See Also

Internal Procedures
Modules and Module Procedures
Main Program

CONTIGUOUS

Statement and Attribute: *Specifies that the target of a pointer or an assumed-sized array is contiguous.*

Syntax

The CONTIGUOUS attribute can be specified in a type declaration statement or an CONTIGUOUS statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] CONTIGUOUS [, *att-ls*] :: *object* [, *object*] ...

Statement:

CONTIGUOUS [::] *object* [, *object*] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
object	Is an assumed-shape array or an array pointer.

Description

This attribute explicitly indicates that an assumed-shape array is contiguous or that a pointer will only be associated with a contiguous object.

An entity can be contiguous even if CONTIGUOUS is not specified. An object is contiguous if it is one of the following:

- An object with the CONTIGUOUS attribute
- A nonpointer whole array that is not assumed-shape
- An assumed-shape array that is argument associated with an array that is contiguous
- An array allocated by an ALLOCATE statement
- An pointer associated with a contiguous target
- A nonzero-sized array section in which the following is true:
  - Its base object is contiguous.
  - It does not have a vector subscript.
  - The elements of the section, in array element order, are a subset of the base object elements that are consecutive in array element order.
  - If the array is of type character and a substring-range appears, the substring-range specifies all of the characters of the parent-string.
  - Only its final reference to a structure component, if any, has nonzero rank
  - It is not the real or imaginary part of an array of type complex.

An object is *not* contiguous if it is an array subobject, and all of the following are true:

- The object has two or more elements.
- The elements of the object in array element order are not consecutive in the elements of the base object.
- The object is not of type character with length zero.
- The object is not of a derived type that has no ultimate components other than zero-sized arrays and characters with length zero.

The CONTIGUOUS attribute can make it easier to enable optimizations that rely on the memory layout of an object occupying a contiguous block of memory.

Examples

The following examples show valid CONTIGUOUS statements:

```
REAL, CONTIGUOUS, DIMENSION(:,:) :: A
REAL, POINTER, CONTIGUOUS :: MY_POINTER(:)
```

See Also

Type Declarations
Compatible attributes

CONTINUE

Statement: *Primarily used to terminate a labeled DO construct when the construct would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement.*

Syntax

```
CONTINUE
```

The statement by itself does nothing and has no effect on program results or execution sequence.

Example

The following example shows a CONTINUE statement:

```
    DO 150 I = 1,40
40  Y = Y + 1
    Z = COS(Y)
    PRINT *, Z
    IF (Y .LT. 30) GO TO 150
    GO TO 40
150 CONTINUE
```

The following shows another example:

```
     DIMENSION narray(10)
     DO 100 n = 1, 10
     narray(n) = 120
100  CONTINUE
```

See Also

END DO

DO

Execution Control

COPYIN Clause

Parallel Directive Clause: *Specifies that the data in the master thread of the team is to be copied to the thread private copies of the common block at the beginning of the parallel region.*

Syntax

`COPYIN (list)`

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

The COPYIN clause applies only to common blocks declared as THREADPRIVATE.

You do not need to specify the whole THREADPRIVATE common block, you can specify named variables within the common block.

COPYPRIVATE Clause

Parallel Directive Clause: *Uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. The COPYPRIVATE clause can only appear in the END SINGLE directive.*

Syntax

`COPYPRIVATE (list)`

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables in the list must not appear in a PRIVATE or FIRSTPRIVATE clause for the SINGLE directive construct. A dummy argument that is a pointer with the INTENT (IN) attribute must not appear in a COPYPRIVATE clause.

If the directive is encountered in the dynamic extent of a parallel region, variables in the list must be private in the enclosing context.

If a common block is specified, it must be declared as THREADPRIVATE; the effect is the same as if the variable names in its common block object list were specified.

The effect of the COPYPRIVATE clause on the variables in its list occurs after the execution of the code enclosed within the SINGLE construct, and before any threads in the team have left the barrier at the end of the construct.

COS

Elemental Intrinsic Function (Generic): *Produces the cosine of an argument.*

Syntax

```
result = COS (x)
```

x	(Input) Must be of type real or complex. It must be in radians and is treated as modulo 2*pi.

Results

The result type and kind are the same as *x*.

If *x* is of type real, the result is a value in radians.

If *x* is of type complex, the real part of the result is a value in radians.

Specific Name	Argument Type	Result Type
COS	REAL(4)	REAL(4)
DCOS	REAL(8)	REAL(8)
QCOS	REAL(16)	REAL(16)
CCOS [1]	COMPLEX(4)	COMPLEX(4)
CDCOS [2]	COMPLEX(8)	COMPLEX(8)
CQCOS	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CCOS.

[2]This function can also be specified as ZCOS.

Example

COS (2.0) has the value -0.4161468.

COS (0.567745) has the value 0.8431157.

COSD

Elemental Intrinsic Function (Generic): *Produces the cosine of x.*

Syntax

```
result = COSD (x)
```

x (Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type and kind are the same as *x*.

Specific Name	Argument Type	Result Type
COSD	REAL(4)	REAL(4)
DCOSD	REAL(8)	REAL(8)
QCOSD	REAL(16)	REAL(16)

Example

COSD (2.0) has the value 0.9993908.

COSD (30.4) has the value 0.8625137.

COSH

Elemental Intrinsic Function (Generic): *Produces a hyperbolic cosine.*

Syntax

```
result = COSH (x)
```

x (Input) Must be of type real or complex.

Results

The result type and kind are the same as *x*.

If *x* is of type complex, the imaginary part of the result is in radians.

Specific Name	Argument Type	Result Type
COSH	REAL(4)	REAL(4)
DCOSH	REAL(8)	REAL(8)
QCOSH	REAL(16)	REAL(16)

Example

COSH (2.0) has the value 3.762196.

COSH (0.65893) has the value 1.225064.

COSHAPE

Inquiry Intrinsic Function (Generic): *Returns the sizes of codimensions of a coarray.*

Syntax

`result = COSHAPE (coarray [, kind])`

coarray	(Input) Is a coarray. It may be of any data type. It must not be an unallocated allocatable coarray.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is an integer array of rank one whose size is equal to the corank of *coarray*. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer.

The result has a value whose *i*th element is equal to the size of the *i*th codimension of *coarray* as provided by UCOBOUND(*coarray*, *i*) − LCOBOUND(*coarray*, *i*) + 1.

The setting of compiler options specifying integer size can affect this function.

Example

Consider the following coarray declaration:

```
REAL :: X (10, 20) [10, -1:8, 0:*]
```

It NUM_IMAGES() is 200, it has these properties:

- RANK (X) == 2
- corank of X == 3
- SHAPE (X) == [10,20]
- COSHAPE (X) == [10,10,2]
- LCOBOUND (X) == [1, -1, 0]
- UCOBOUND (X) == [10, 8, 1]

See Also

Coarrays
Using Coarrays
CODIMENSION
LCOBOUND
UCOBOUND

COTAN

Elemental Intrinsic Function (Generic): *Produces the cotangent of x.*

Syntax

`result = COTAN (x)`

x	(Input) Must be of type real; it cannot be zero. It must be in radians and is treated as modulo 2*pi.

Results

The result type and kind are the same as *x*.

Specific Name	Argument Type	Result Type
COTAN	REAL(4)	REAL(4)
DCOTAN	REAL(8)	REAL(8)
QCOTAN	REAL(16)	REAL(16)

Example

COTAN (2.0) has the value -4.576575E-01.

COTAN (0.6) has the value 1.461696.

COTAND

Elemental Intrinsic Function (Generic): *Produces the cotangent of x.*

Syntax

```
result = COTAND (x)
```

x (Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type and kind are the same as *x*.

Specific Name	Argument Type	Result Type
COTAND	REAL(4)	REAL(4)
DCOTAND	REAL(8)	REAL(8)
QCOTAND	REAL(16)	REAL(16)

Example

COTAND (2.0) has the value 0.2863625E+02.

COTAND (0.6) has the value 0.9548947E+02.

COUNT

Transformational Intrinsic Function (Generic): *Counts the number of true elements in an entire array or in a specified dimension of an array.*

Syntax

```
result = COUNT (mask[,dim][, kind])
```

mask (Input) Must be a logical array.

dim	(Input; optional) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *mask*.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is an array or a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result has a value equal to the number of true elements of *mask*. If *mask* has size zero, the result is zero.

An array result has a rank that is one less than *mask*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of *mask*.

Each element in an array result equals the number of elements that are true in the one dimensional array defined by *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n).

Example

COUNT ((/.TRUE., .FALSE., .TRUE./)) has the value 2 because two elements are true.

COUNT ((/.TRUE., .TRUE., .TRUE./)) has the value 3 because three elements are true.

A is the array

```
  [ 1  5  7 ]
  [ 3  6  8 ]
```

and B is the array

```
  [ 0  5  7 ]
  [ 2  6  9 ].
```

COUNT (A .NE. B, DIM=1) tests to see how many elements in each column of A are not equal to the elements in the corresponding column of B. The result has the value (2, 0, 1) because:

- The first column of A and B have 2 elements that are not equal.
- The second column of A and B have 0 elements that are not equal.
- The third column of A and B have 1 element that is not equal.

COUNT (A .NE. B, DIM=2) tests to see how many elements in each row of A are not equal to the elements in the corresponding row of B. The result has the value (1, 2) because:

- The first row of A and B have 1 element that is not equal.
- The second row of A and B have 2 elements that are not equal.

The following shows another example:

```
LOGICAL mask (2, 3)
INTEGER AR1(3), AR2(2), I
mask = RESHAPE((/.TRUE., .TRUE., .FALSE., .TRUE., &
                 .FALSE., .FALSE./),(/2,3/))
!
! mask is the array   true false false
!                     true true false
AR1 = COUNT(mask,DIM=1) ! counts true elements by
                        ! column yielding [2 1 0]
AR2 = COUNT(mask,DIM=2) ! counts true elements by row
                        ! yielding [1 2]
I = COUNT( mask)        ! returns 3
```

See Also

ALL
ANY

CPU_TIME

Intrinsic Subroutine (Generic): *Returns a processor-dependent approximation of the processor time in seconds. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL CPU_TIME (time)
```

time (Output) Must be scalar and of type real.

The time returned is summed over all active threads. The result is the sum (in units of seconds) of the current process's user time and the user and system time of all its child processes, if any.

If a meaningful time cannot be returned, a processor-dependent negative value is returned.

NOTE
If you want to estimate performance or scaling of multithreaded applications, you should use intrinsic subroutine SYSTEM_CLOCK or portability function DCLOCK. Both of these routines return the elapsed time from a single clock.

Example

Consider the following:

```
REAL time_begin, time_end
...
CALL CPU_TIME ( time_begin )
!
! task to be timed
!
CALL CPU_TIME ( time_end )
WRITE (*,*) 'Time of operation was ', time_end - time_begin, ' seconds'
```

See Also

DCLOCK

SYSTEM_CLOCK

CRITICAL Directive

OpenMP* Fortran Compiler Directive: *Restricts access to a block of code to only one thread at a time.*

Syntax

```
!$OMP CRITICAL [(name)]
   block
!$OMP END CRITICAL [(name)]
```

name	Is the name of the critical section.
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

The binding thread set for a CRITICAL construct is all threads in the contention group. Region execution is restricted to a single thread at a time among all threads in the contention group, without regard to the teams to which the threads belong.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name. All unnamed CRITICAL directives map to the same name.

If a name is specified in the CRITICAL directive, the same name must appear in the corresponding END CRITICAL directive. If no name appears in the CRITICAL directive, no name can appear in the corresponding END CRITICAL directive.

Critical section names are global entities of the program. If the name specified conflicts with any other entity, the behavior of the program is undefined.

Example

The following example shows a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation is placed in a critical section.

Because there are two independent queues in this example, each queue is protected by CRITICAL directives having different names, XAXIS and YAXIS, respectively:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
!$OMP CRITICAL(XAXIS)
      CALL DEQUEUE(IX_NEXT, X)
!$OMP END   CRITICAL(XAXIS)
      CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(YAXIS)
      CALL DEQUEUE(IY_NEXT,Y)
!$OMP END   CRITICAL(YAXIS)
      CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

CRITICAL Statement

Statement: *Marks the beginning of a CRITICAL construct. A CRITICAL construct limits execution of a block to one image at a time.*

Syntax

```
[name:] CRITICAL [([STAT=stat-var] [, ERRMSG=err-var])]
   block
END CRITICAL [name]
```

name	(Optional) Is the name of the CRITICAL construct.
stat-var	Is a scalar integer variable with an exponent range of at least 4 (KIND=2 or greater).

err-var	Is a scalar default character variable.
block	Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified in a CRITICAL statement, the same name must appear in the corresponding END CRITICAL statement. If no name is specified at the beginning of a CRITICAL statement, you cannot specify one following the END CRITICAL statement. The same construct name must not be used for different named constructs in the same scoping unit.

The block of a CRITICAL construct must not contain a RETURN statement or an image control statement.

A branch within a CRITICAL construct must not have a branch target that is outside the construct. A branch to the END CRITICAL statement is permitted from within the construct.

STAT= and ERRMSG= specifiers can appear in any order. Each may appear at most once.

Execution of the CRITICAL construct is completed when execution of its block is completed. A procedure that is invoked, directly or indirectly, from a CRITICAL construct must not execute an image control statement.

If no error condition occurs during the execution of the construct, *stat-var* becomes defined with the value zero. If the image that previously entered the construct failed while executing the construct, *stat-var* becomes defined with the value STAT_FAILED_IMAGE defined in the intrinsic module ISO_FORTRAN_ENV. If any other error occurs and STAT= is specified, *stat-var* becomes defined with a positive integer value other than that of STAT_FAILED_IMAGE. Otherwise, if an error occurs, error termination is initiated.

If ERRMSG= is specified and an error condition occurs during execution of the construct, *err-var* becomes defined with a descriptive message describing the nature of the error.

The processor ensures that once an image has commenced executing *block*, no other image can start executing *block* until this image has completed executing *block*. The image must not execute an image control statement during the execution of *block*. The sequence of executed statements is therefore a segment. If image S is the next to execute the construct after image N, the segment on image N precedes the segment on image S.

If more than one image executes the block of a CRITICAL construct, its execution by one image always precedes or succeeds its execution by another image. Normally no other statement ordering is needed.

Example

Consider the following example:

```
CONA: CRITICAL
  MY_COUNTER[1] = MY_COUNTER[1] + 1
END CRITICAL CONA
```

The definition of MY_COUNTER [1] by a particular image will always precede the reference to the same variable by the next image to execute the block.

The following example shows a way to share a large number of tasks among images:

```
INTEGER :: NUMBER_TASKS[*], TASK
IF (THIS_IMAGE() == 1) READ(*,*) NUMBER_TASKS
SYNC ALL
DO
   CRITICAL
      TASK = NUMBER_TASKS[1]
      NUMBER_TASKS[1] = TASK - 1
   END CRITICAL
IF (TASK > 0) THEN
ELSE
   EXIT
```

```
END IF
END DO
SYNC ALL
```

See Also

Image Control Statements
Coarrays
Using Coarrays
ISO_FORTRAN_ENV Module

CSHIFT

Transformational Intrinsic Function (Generic):
Performs a circular shift on a rank-one array, or performs circular shifts on all the complete rank-one sections (vectors) along a given dimension of an array of rank two or greater.

Syntax

Elements shifted off one end are inserted at the other end. Different sections can be shifted by different amounts and in different directions.

`result = CSHIFT (array,shift [,dim])`

array	(Input) Array whose elements are to be shifted. It can be of any data type.
shift	(Input) The number of positions shifted. Must be a scalar integer or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
dim	(Input; optional) Optional dimension along which to perform the shift. Must be a scalar integer with a value in the range 1 to n, where n is the rank of array. If *dim* is omitted, it is assumed to be 1.

Results

The result is an array with the same type and kind parameters, and shape as *array*.

If *array* has rank one, element i of the result is *array*(1 + MODULO (i + *shift*- 1, SIZE (*array*))). (The same shift is applied to each element.)

If *array* has rank greater than one, each section (s_1,s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) of the result is shifted as follows:

- By the value of *shift*, if *shift* is scalar
- According to the corresponding value in *shift*(s_1, s_2,..., s_{dim-1}, s_{dim+1},..., s_n), if *shift* is an array

The value of *shift* determines the amount and direction of the circular shift. A positive *shift* value causes a shift to the left (in rows) or up (in columns). A negative *shift* value causes a shift to the right (in rows) or down (in columns). A zero *shift* value causes no shift.

Example

V is the array (1, 2, 3, 4, 5, 6).

CSHIFT (V, SHIFT=2) shifts the elements in V circularly to the *left* by 2 positions, producing the value (3, 4, 5, 6, 1, 2). 1 and 2 are shifted off the beginning and inserted at the end.

CSHIFT (V, SHIFT= -2) shifts the elements in V circularly to the *right* by 2 positions, producing the value (5, 6, 1, 2, 3, 4). 5 and 6 are shifted off the end and inserted at the beginning.

M is the array

```
  [ 1  2  3 ]
  [ 4  5  6 ]
  [ 7  8  9 ].
```

CSHIFT (M, SHIFT = 1, DIM = 2) produces the result

```
  [ 2  3  1 ]
  [ 5  6  4 ]
  [ 8  9  7 ].
```

Each element in rows 1, 2, and 3 is shifted to the *left* by 1 position. The elements shifted off the beginning are inserted at the end.

CSHIFT (M, SHIFT = -1, DIM = 1) produces the result

```
  [ 7  8  9 ]
  [ 1  2  3 ]
  [ 4  5  6 ].
```

Each element in columns 1, 2, and 3 is shifted down by 1 position. The elements shifted off the end are inserted at the beginning.

CSHIFT (M, SHIFT = (/1, -1, 0/), DIM = 2) produces the result

```
  [ 2  3  1 ]
  [ 6  4  5 ]
  [ 7  8  9 ].
```

Each element in row 1 is shifted to the *left* by 1 position; each element in row 2 is shifted to the *right* by 1 position; no element in row 3 is shifted at all.

The following shows another example:

```
INTEGER array (3, 3), AR1(3, 3), AR2 (3, 3)
DATA array /1, 4, 7, 2, 5, 8, 3, 6, 9/
!
! array is   1 2 3
!            4 5 6
!            7 8 9
!AR1 = CSHIFT(array, 1, DIM = 1) ! shifts all columns
                                 ! by 1 yielding
                                 !         4 5 6
                                 !         7 8 9
                                 !         1 2 3
                                 !
AR2=CSHIFT(array,shift=(/-1, 1, 0/),DIM=2) ! shifts
                                 ! each row separately
                                 ! by the amount in
                                 ! shift yielding
                                 !         3 1 2
                                 !         5 6 4
                                 !         7 8 9
```

See Also

EOSHIFT
ISHFT
ISHFTC

CSMG

Portability Function: *Performs an effective BIT-WISE store under mask.*

Module

USE IFPORT

Syntax

`result = CSMG (x,y,z)`

x, *y*, *z*	(Input) INTEGER(4).

Results

The result type is INTEGER(4). The result is equal to the following expression:

```
(x & z) | (y & ~z )
```

where "&" is a bitwise AND operation, | - bitwise OR, ~ - bitwise NOT.

The function returns the value based on the following rule: when a bit in *z* is 1, the output bit is taken from *x*. When a bit in *z* is zero, the corresponding output bit is taken from *y*.

CTIME

Portability Function: *Converts a system time into a 24-character ASCII string.*

Module

USE IFPORT

Syntax

`result = CTIME (stime)`

stime	(Input) INTEGER(4). An elapsed time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.

Results

The result is a value in the form Mon Jan 31 04:37:23 1994. Hours are expressed using a 24-hour clock.

The value of *stime* can be determined by calling the TIME function. CTIME(TIME( )) returns the current time and date.

Example

```
USE IFPORT
character (24) systime
systime = CTIME (TIME( ))
print *, 'Current date and time is ',systime
```

See Also

DATE_AND_TIME

CYCLE

Statement: *Interrupts the current execution cycle of the innermost (or named) DO construct.*

Syntax

```
CYCLE [name]
```

name	(Optional) Is the name of the DO construct.

Description

When a CYCLE statement is executed, the following occurs:

1. The current execution cycle of the named (or innermost) DO construct is terminated.

   If a DO construct name is specified, the CYCLE statement must be within the range of that construct.
2. The iteration count (if any) is decremented by 1.
3. The DO variable (if any) is incremented by the value of the increment parameter (if any).
4. A new iteration cycle of the DO construct begins.

Any executable statements following the CYCLE statement (including a labeled terminal statement) are not executed.

A CYCLE statement can be labeled, but it cannot be used to terminate a DO construct.

Execution of a CYCLE statement that belongs to a DO CONCURRENT construct completes execution of that iteration of the construct.

Example

The following example shows a CYCLE statement:

```
DO I =1, 10
  A(I) = C + D(I)
  IF (D(I) < 0) CYCLE    ! If true, the next statement is omitted
  A(I) = 0               ! from the loop and the loop is tested again.
END DO
```

The following shows another example:

```
      sample_loop: do i = 1, 5
           print *,i
             if( i .gt. 3 ) cycle sample_loop
           print *,i
           end do sample_loop
           print *,'done!'
!output:
!     1
!     1
!     2
!     2
!     3
!     3
!     4
!     5
!     done!
```

See Also

DO
DO WHILE

Execution Control

DATA

Statement: *Assigns initial values to variables before program execution.*

Syntax

DATA *var-list /clist/* [[,] *var-list /clist/*]...

var-list

Is a list of variables or implied-DO lists, separated by commas. *var* cannot be a coarray, a dummy argument, accessed by use or host association, a function name, a function result name, an automatic variable, or a allocatable variable.

Subscript expressions, section expressions, and substring expressions must be constant expressions.

An implied-DO list in a DATA statement takes the following form:

(*do-list*, *do-var*= *expr1*, *expr2*[, *expr3*])

do-list

Is a list of one or more array elements, substrings, scalar structure components, or implied-DO lists, separated by commas. Any array elements or scalar structure components must not have a constant parent.

do-var

Is the name of a scalar integer variable (the implied-DO variable). It cannot be a coarray.

expr

Implied-DO limits must be scalar constant expressions. They may contain implied-DO variables from outer nested implied-DO lists. For more details, see Iteration Loop Control.

clist

Is a list of scalar integer constant expressions (or names of constants), constant structure constructors, or, for pointer objects, NULL (), separated by commas. If the constant is a binary, octal, or hexadecimal literal, the corresponding *var* must be of type INTEGER.

A constant can be specified in the form *r**constant, where *r* is a repeat specification. *r* is a nonnegative scalar integer constant (with no kind parameter). If *r* is a named constant, it must have been declared previously in the scoping unit or made accessible through use or host association. If *r* is omitted, it is assumed to be 1.

Description

A variable can be initialized only once in an executable program. A variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration which may change the implicit typing.

The number of constants in *c-list* must equal the number of variables in *var-list*. The constants are assigned to the variables in the order in which they appear (from left to right).

The following objects cannot be initialized in a DATA statement:

- A dummy argument
- A function
- A function result
- An automatic object
- An allocatable array

- A variable that is accessible by use or host association
- A variable in a named common block (unless the DATA statement is in a block data program unit)
- A variable in blank common

Except for variables in named COMMON blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement. You can confirm this property by specifying the variable in a SAVE statement or a type declaration statement containing the SAVE attribute.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array in the order of subscript progression. If the associated constant list does not contain enough values to fill the array, a warning is issued and the remaining array elements become undefined.

Array element values can be initialized in three ways: by name, by element, or by an implied-DO list (interpreted in the same way as a DO construct).

The following conversion rules and restrictions apply to variable and constant list items:

- If the constant and the variable are both of numeric type, the following conversion occurs:
  - The constant value is converted to the data type of the variable being initialized, if necessary.
  - When a binary, octal, or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the data item. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeros. If the constant contains more digits than can be stored, the constant is truncated on the left. An error results if any nonzero digits are truncated.
- If the constant and the variable are both of character type, the following conversion occurs:
  - If the length of the constant is less than the length of the variable, the rightmost character positions of the variable are initialized with blank characters.
  - If the length of the constant is greater than the length of the variable, the character constant is truncated on the right.
- If the constant is of numeric type and the variable is of character type, the following restrictions apply:
  - The character variable must have a length of one character.
  - The constant must be an integer, binary, octal, or hexadecimal constant, and must have a value in the range 0 through 255.

  When the constant and variable conform to these restrictions, the variable is initialized with the character that has the ASCII code specified by the constant. (This lets you initialize a character object to any 8-bit ASCII code.)
- If the constant is a Hollerith or character constant, and the variable is a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the data item.

  If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with blank characters. If the constant contains more characters than can be stored, the constant is truncated on the right.

Example

The following example shows the three ways that DATA statements can initialize array element values:

```
DIMENSION A(10,10)
DATA A/100*1.0/    ! initialization by name
DATA A(1,1), A(10,1), A(3,3) /2*2.5, 2.0/ ! initialization by element
DATA ((A(I,J), I=1,5,2), J=1,5) /15*1.0/  ! initialization by implied-DO list
```

The following example shows DATA statements containing structure components:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

```
 TYPE(EMPLOYEE) MAN_NAME, CON_NAME
 DATA MAN_NAME / EMPLOYEE(417, 'Henry Adams') /
 DATA CON_NAME%ID, CON_NAME%NAME /891, "David James"/
```

In the following example, the first DATA statement assigns zero to all 10 elements of array A, and four asterisks followed by two blanks to the character variable STARS:

```
 INTEGER A(10), B(10)
 CHARACTER BELL, TAB, LF, FF, STARS*6
 DATA A,STARS /10*0,'****'/
 DATA BELL,TAB,LF,FF /7,9,10,12/
 DATA (B(I), I=1,10,2) /5*1/
```

In this case, the second DATA statement assigns ASCII control character codes to the character variables BELL, TAB, LF, and FF. The last DATA statement uses an implied-DO list to assign the value 1 to the odd-numbered elements in the array B.

The following shows another example:

```
   INTEGER n, order, alpha, list(100)
   REAL coef(4), eps(2),
   pi(5), x(5,5)
   CHARACTER*12 help
   COMPLEX*8 cstuff
   DATA  n /0/, order /3/
   DATA  alpha /'A'/
   DATA  coef /1.0, 2*3.0, 1.0/, eps(1) /.00001/
   DATA  cstuff /(-1.0, -1.0)/
!    The following example initializes diagonal and below in
!    a 5x5 matrix:
   DATA  ((x(j,i), i=1,j), j=1,5) / 15*1.0 /
   DATA  pi / 5*3.14159 /
   DATA  list / 100*0 /
   DATA  help(1:4), help(5:8), help(9:12) /3*'HELP'/
```

Consider the following:

```
CHARACTER (LEN = 10) NAME
INTEGER, DIMENSION (0:9) :: MILES
REAL, DIMENSION (100, 100) :: SKEW
TYPE (MEMBER) MYNAME, YOURS
DATA NAME / 'JOHN DOE' /, miles / 10*0 /
DATA ((SKEW (k, j), j = 1, k), k = 1, 100) / 5050*0.0 /
DATA ((SKEW (k, j), j = k + 1, 100), k = 1, 99) / 4950*1.0 /
DATA MYNAME / MEMBER (21, 'JOHN SMITH') /
DATA YOURS % age, YOURS % name / 35, 'FRED BROWN' /
```

In this example, the character variable NAME is initialized with the value JOHN DOE with two trailing blanks to fill out the declared length of the variable. The ten elements of MILES are initialized to zero. The two-dimensional array SKEW is initialized so that its lower triangle is zero and its upper triangle is one. The structures MYNAME and YOURS are declared using the derived type MEMBER. The derived-type variable MYNAME is initialized by a structure constructor. The derived-type variable YOURS is initialized by supplying a separate value for each component.

The first DATA statement in the previous example could also be written as:

```
DATA name / 'JOHN DOE' /
DATA miles / 10*0 /
```

A pointer can be initialized as disassociated by using a DATA statement. For example:

```
INTEGER, POINTER :: P
DATA P/NULL( )/
END
```

The implied-DO limits can be any constant expressions in a DATA statement. For example:

```
DATA (A(I),I=LBOUND(A),UBOUND(A)) /10*4.0/
```

See Also

CHARACTER
INTEGER
REAL
COMPLEX
COMMON
Data Types, Constants, and Variables
I/O Lists
Derived Data Types
Allocating Common Blocks in section: Fortran Language Extensions

DATE Intrinsic Procedure

Intrinsic Subroutine (Generic): *Returns the current date as set within the system. DATE can be used as an intrinsic subroutine or as a portability routine. It is an intrinsic procedure unless you specify USE IFPORT. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL DATE (buf)`

buf	(Output) Is a variable, array, or array element of any data type, or a character substring. It must contain at least nine bytes of storage.

The date is returned as a 9-byte ASCII character string taking the form dd-mmm-yy, where:

dd	is the 2-digit date
mmm	is the 3-letter month
yy	is the last two digits of the year

If *buf* is of numeric type and smaller than 9 bytes, data corruption can occur.

If *buf* is of character type, its associated length is passed to the subroutine. If *buf* is smaller than 9 bytes, the subroutine truncates the date to fit in the specified length. If an array of type character is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

Caution
The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

```
CHARACTER*1 DAY(9)
…
CALL DATE (DAY)
```

The length of the first array element in CHARACTER array DAY is passed to the DATE subroutine. The subroutine then truncates the date to fit into the 1-character element, producing an incorrect result.

See Also

DATE_AND_TIME

DATE portability routine

DATE Portability Routine

Portability Function or Subroutine: *Returns the current system date. DATE can be used as a portability routine or as an intrinsic procedure. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

Function Syntax:

```
result = DATE( )
```

Subroutine Syntax:

```
CALL DATE (dstring)
```

dstring	(Output) CHARACTER. Is a variable or array containing at least nine bytes of storage.

DATE in its function form returns a CHARACTER string of length 8 in the form mm/dd/yy, where mm, dd, and yy are two-digit representations of the month, day, and year, respectively.

DATE in its subroutine form returns *dstring* in the form dd-mmm-yy, where dd is a two-digit representation of the current day of the month, mmm is a three-character abbreviation for the current month (for example, Jan) and yy are the last two digits of the current year.

Caution
The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

```
USE IFPORT
!If today's date is March 02, 2000, the following
!code prints "02-Mar-00"
CHARACTER(9) TODAY
CALL DATE(TODAY)
PRINT *, TODAY
!The next line prints "03/02/00"
PRINT *, DATE( )
```

See Also

DATE_AND_TIME

DATE intrinsic procedure

DATE4

Portability Subroutine: *Returns the current system date.*

Module

USE IFPORT

Syntax

`CALL DATE4 (datestr)`

datestr (Output) CHARACTER. Is a variable or array containing at least eleven bytes of storage.

This subroutine returns *datestr* in the form dd-mmm-yyyy, where dd is a two-digit representation of the current day of the month, mmm is a three-character abbreviation for the current month (for example, Jan) and yyyy are the four digits of the current year.

DATE_AND_TIME

Intrinsic Subroutine (Generic): *Returns character and binary data on the real-time clock and date. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL DATE_AND_TIME ([date,time,zone,values])`

date (Output; optional) Must be scalar and of type default character; its length must be at least 8 to contain the complete value. Its leftmost 8 characters are set to a value of the form CCYYMMDD, where:

CC	Is the century
YY	Is the year within the century
MM	Is the month within the year
DD	Is the day within the month

time (Output; optional) Must be scalar and of type default character; its length must be at least 10 to contain the complete value. Its leftmost 10 characters are set to a value of the form hhmmss.sss, where:

hh	Is the hour of the day
mm	Is the minutes of the hour

ss.sss	Is the seconds and milliseconds of the minute

zone

(Output; optional) Must be scalar and of type default character; its length must be at least 5 to contain the complete value. Its leftmost 5 characters are set to a value of the form +hhmm or -hhmm, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC) in hours and parts of an hour expressed in minutes, respectively.

UTC is also known as Greenwich Mean Time.

values

(Output; optional) Must be of type integer. One-dimensional array with size of at least 8. The values returned in *values* are as follows:

values(1)	Is the 4-digit year
values(2)	Is the month of the year
values(3)	Is the day of the month
values(4)	Is the time difference with respect to Coordinated Universal Time (UTC) in minutes
values(5)	Is the hour of the day (range 0 to 23) - local time
values(6)	Is the minutes of the hour (range 0 to 59) - local time
values(7)	Is the seconds of the minute (range 0 to 59) - local time
values(8)	Is the milliseconds of the second (range 0 to 999) - local time

Example

Consider the following example executed on 2000 March 28 at 11:04:14.5:

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 12) REAL_CLOCK (3)
CALL DATE_AND_TIME (REAL_CLOCK (1), REAL_CLOCK (2), &
                    REAL_CLOCK (3), DATE_TIME)
```

This assigns the value "20000328" to REAL_CLOCK (1), the value "110414.500" to REAL_CLOCK (2), and the value "-0500" to REAL_CLOCK (3). The following values are assigned to DATE_TIME: 2000, 3, 28, -300, 11, 4, 14, and 500.

The following shows another example:

```
CHARACTER(10) t
CHARACTER(5) z
CALL DATE_AND_TIME(TIME = t, ZONE = z)
```

See Also

GETDAT

GETTIM

IDATE intrinsic procedure

FDATE

TIME intrinsic procedure

ITIME

RTC

CLOCK

DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN

Portability Functions: *Compute the double-precision values of Bessel functions of the first and second kinds.*

Module

USE IFPORT

Syntax

```
result = DBESJ0 (value)
result = DBESJ1 (value)
result = DBESJN (n, value)
result = DBESY0 (posvalue)
result = DBESY1 (posvalue)
result = DBESYN (n, posvalue)
```

value	(Input) REAL(8). Independent variable for a Bessel function.
n	(Input) INTEGER(4). Specifies the order of the selected Bessel function computation.
posvalue	(Input) REAL(8). Independent variable for a Bessel function. Must be greater than or equal to zero.

Results

DBESJ0, DBESJ1, and DBESJN return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable *value*.

DBESY0, DBESY1, and DBESYN return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

Negative arguments cause DBESY0, DBESY1, and DBESYN to return a huge negative value.

Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions*(Abramowitz and Stegun. Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

See the descriptions of the BESSEL_* functions, if you need to use quad-precision (REAL(16)).

Example

```
    USE IFPORT
    real(8) besnum, besout
10  read *, besnum
    besout = dbesj0(besnum)
    print *, 'result is ',besout
    goto 10
    end
```

See Also

BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

DBLE

Elemental Intrinsic Function (Generic): *Converts a number to double-precision real type.*

Syntax

```
result = DBLE (a)
```

a	(Input) Must be of type integer, real, or complex, or a binary, octal, or hexadecimal literal constant.

Results

The result type is double precision real (by default, REAL(8) or REAL*8). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If *a* is of type double precision, the result is the value of the *a* with no conversion (DBLE(*a*) = *a*).

If *a* is of type integer or real, the result has as much precision of the significant part of *a* as a double precision value can contain.

If *a* is of type complex, the result has as much precision of the significant part of the real part of *a* as a double precision value can contain.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
	INTEGER(2)	REAL(8)
	INTEGER(4)	REAL(8)
	INTEGER(8)	REAL(8)
DBLE[2]	REAL(4)	REAL(8)
	REAL(8)	REAL(8)
DBLEQ	REAL(16)	REAL(8)
	COMPLEX(4)	REAL(8)
	COMPLEX(8)	REAL(8)
	COMPLEX(16)	REAL(8)

[1]These specific functions cannot be passed as actual arguments.

Specific Name [1]	Argument Type	Result Type

[2] The setting of compiler options specifying double size can affect DBLE.

If the argument is a binary, octal, or hexadecimal literal constant, the result is affected by the `assume old-boz` option. The default option setting, `noold-boz`, treats the argument as a bit string that represents a value of the data type of the intrinsic, that is, the bits are not converted. If setting `old-boz` is specified, the argument is treated as a signed integer and the bits are converted.

NOTE
The result values of DBLE are defined by references to the intrinsic function REAL with the same arguments. Therefore, the padding and truncation of binary, octal, and hexadecimal literal constant arguments to DBLE is the same as for the intrinsic function REAL.

Example

DBLE (4) has the value 4.0.

DBLE ((3.4, 2.0)) has the value 3.4.

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
Model for Bit Data
FLOAT
SNGL
REAL
CMPLX

DCLOCK

Portability Function: *Returns the elapsed time in seconds since the start of the current process.*

Module

USE IFPORT

Syntax

```
result = DCLOCK( )
```

Results

The result type is REAL(8). This routine provides accurate timing to the nearest millisecond (Windows*) or to the nearest microsecond (Linux* and macOS*), taking into account the frequency of the processor where the current process is running.

Note that the first call to DCLOCK performs calibration.

Example

```
USE IFPORT
DOUBLE PRECISION START_TIME, STOP_TIME
START_TIME = DCLOCK()
CALL FOO()
STOP_TIME = DCLOCK()
PRINT *, 'foo took:', STOP_TIME - START_TIME, 'seconds.'
```

See Also

DATE_AND_TIME
CPU_TIME

DCMPLX

Elemental Intrinsic Function (Specific): *Converts the argument to double complex type. This function cannot be passed as an actual argument.*

Syntax

```
result = DCMPLX (x[,y])
```

x	(Input) Must be of type integer, real, or complex.
y	(Input; optional) Must be of type integer or real. It must not be present if *x* is of type complex.

Results

The result type is double complex (COMPLEX(8) or COMPLEX*16).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX(REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

DCMPLX(*x*, *y*) has the complex value whose real part is REAL(*x*, KIND=8) and whose imaginary part is REAL(*y*, KIND=8).

Example

DCMPLX (-3) has the value (-3.0, 0.0).

DCMPLX (4.1, 2.3) has the value (4.1, 2.3).

See Also

CMPLX
FLOAT
INT
IFIX
REAL
SNGL

DEALLOCATE

Statement: *Frees the storage allocated for allocatable variables and nonprocedure pointer targets (and causes the pointers to become disassociated).*

Syntax

```
DEALLOCATE (object[,object]...[, dealloc-opt])
```

object	Is a structure component or the name of a variable, and must be a pointer or allocatable variable.

dealloc-opt	(Output) Is one of the following:	
	STAT=*stat-var*	*stat-var* is a scalar integer variable in which the status of the deallocation is stored.
	ERRMSG=*err-var*	*err-var* is a scalar default character value in which an error condition is stored if such a condition occurs.

Description

If a STAT= variable or ERRMSG= variable is specified, it must not be deallocated in the DEALLOCATE statement in which it appears. If the deallocation is successful, the STAT= variable is set to zero and the ERRMSG= variable is unchanged. If the deallocation is not successful, an error condition occurs, the STAT= variable is set to a positive integer value (representing the run-time error), and the ERRMSG= variable is defined with a descriptive message about the error condition. If no STAT= variable is specified and an error condition occurs, error termination is initiated.

If an ALLOCATE or DEALLOCATE statement with a coarray allocatable object is executed when one or more images has initiated termination of execution, the STAT= variable becomes defined with the processor-dependent positive integer value of the constant STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. Otherwise, if an allocatable object is a coarray and one or more images of the current team has failed, the STAT= variable becomes defined with the processor-dependent positive integer value of the constant STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV.

If any other error condition occurs during execution of the ALLOCATE or DEALLOCATE statement, the STAT= variable becomes defined with a processor-dependent positive integer value different from STAT_STOPPED_IMAGE or STAT_FAILED_IMAGE.

If an ALLOCATE or DEALLOCATE statement with a coarray allocatable object is executed when one or more images of the current team has failed, each allocatable object is successfully allocated or deallocated on the active images of the current team. If any other error occurs, the allocation status of each allocatable object is processor dependent:

- Successfully allocated allocatable objects have the allocation status of allocated, or associated if the allocate object is has the POINTER attribute.
- Successfully deallocated allocatable objects have the allocation status of deallocated, or disassociated if the allocatable object has the POINTER attribute.
- An allocatable object that was not successfully allocated or deallocated has its previous allocation status, or its previous association status if it has the POINTER attribute.

It is recommended that all explicitly allocated storage be explicitly deallocated when it is no longer needed.

To disassociate a pointer that was not associated with the ALLOCATE statement, use the NULLIFY statement.

For a list of run-time errors, see Error Handling in the *Compiler Reference*.

Example

The following example shows deallocation of an allocatable array:

```
INTEGER ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE (A(10), B(-2:8,1:5))
...
DEALLOCATE(A, B, STAT = ALLOC_ERR)
```

The following shows another example:

```
INTEGER, ALLOCATABLE :: dataset(:,:,:)
INTEGER reactor, level, points, error
DATA reactor, level, points / 10, 50, 10 /
ALLOCATE (dataset(1:reactor,1:level,1:points), STAT = error)
DEALLOCATE (dataset, STAT = error)
```

See Also

ALLOCATE
NULLIFY
Arrays
Dynamic Allocation
ISO_FORTRAN_ENV Module

DECLARE and NODECLARE

General Compiler Directives: *DECLARE generates warnings for variables that have been used but have not been declared (like the IMPLICIT NONE statement). NODECLARE (the default) disables these warnings.*

Syntax

```
!DIR$ DECLARE
```

```
!DIR$ NODECLARE
```

The DECLARE directive is primarily a debugging tool that locates variables that have not been properly initialized, or that have been defined but never used.

See Also

IMPLICIT
General Compiler Directives
Syntax Rules for Compiler Directives
Equivalent Compiler Options

DECLARE REDUCTION

OpenMP* Fortran Compiler Directive: *Declares user-defined reductions which are identified by a reduction-identifier that can be used in a reduction clause of other directives.*

Syntax

`!$OMP DECLARE REDUCTION (`*reduction-identifier* `:` *type-list* `:` *combiner*`) [`*initializer-clause*`]`

reduction-identifier	Is a Fortran identifier, or a defined or extended operator.
type-list	Is a comma-separated list of one or more type identifiers. These may be intrinsic types or accessible derived types. You cannot specify polymorphic and parameterized derived types, and coarrays.

type-list cannot contain a type which has previously been specified in a DECLARE REDUCTION directive with the same *reduction-identifier* if the *reduction-identifier*/type pair is accessible by use or host association.

If more than one type is specified, it is as if there is a separate DECLARE REDUCTION directive for each type.

combiner

Is an assignment statement, or a subroutine name followed by an argument list. It indicates how partial results are combined into a single value.

There are two special identifiers that are allowed in the *combiner*:

- `omp_out`

  This identifier refers to the storage that holds the resulting combined value following execution of the combiner.
- `omp_in`

The above identifiers refer to variables that are the type of the reduction variables specified in *type-list* for the *reduction-identifier*. They denote values to be combined by executing the *combiner*.

No other identifiers are permitted in the *combiner*. Any number of literal or named constants can appear in the *combiner*.

If *combiner* is a subroutine name followed by an argument list, it is evaluated by calling the subroutine with the specified argument list. If *combiner* is an assignment statement, *combiner* is evaluated by executing the assignment statement.

The number of times the *combiner* is executed, and the order of these executions, is unspecified.

initializer-clause

Is `initializer` (*initializer-expression*).

At most one *initializer-clause* can be specified.

Only the identifiers `omp_priv` and `omp_orig` are allowed in the *initializer-clause*. `omp_orig` refers to the storage of the original reduction variable that appears in the list in the REDUCTION clause that specifies *reduction-identifier*. If `omp_orig` is modified in the *initializer-clause*, the behavior is unspecified.

No other identifiers are allowed in *initializer-clause*. Any number of literal or named constants are permitted.

initializer-expression

Is one of the following identifiers:

- `omp_priv` = *expression*

  This identifier refers to the storage to be initialized.
- subroutine-name (*argument-list*)

If *initializer-expression* is a subroutine name and an argument list, the `initializer` is evaluated by executing a call to the subroutine with the specified argument list. If `initializer` is an assignment statement, it is evaluated by executing the assignment.

If *initializer-expression* is a subroutine name and an argument list, one of the arguments must be `omp_priv`, and it must be associated with an INTENT(OUT) dummy argument of the subroutine.

The number of times *initializer-expression* is evaluated and the order of the evaluations is unspecified.

The DECLARE REDUCTION directive is a specification directive. It can appear in a specification part of a subroutine, function, main program, module, or block construct.

User-defined (custom) reductions can be defined using the DECLARE REDUCTION directive. The reduction is identified by the *reduction-identifier* and the associated type from *type-list*. The *reduction-identifier* can be used in a REDUCTION clause in another OpenMP* directive anywhere it is accessible by use or host association.

A DECLARE REDUCTION directive cannot redefine a predefined *reduction-identifier* (see the table of implicitly defined reduction identifiers in the REDUCTION clause section).

If a type in *type-list* has deferred or assumed-length type parameters, the *reduction-identifier* can be used in a REDUCTION clause with a variable of the same type and kind type parameter as type, regardless of the length parameter with which the variable is declared. The length parameter of a character type must be a constant, colon, or asterisk. An accessible *reduction-identifier* defined with a deferred or assumed-length character type cannot appear in another DECLARE REDUCTION directive with a *type-list* item of type character with the same kind type parameter.

The accessibility of a *reduction-identifier* is determined by the same rules as for other Fortran entities; it can be declared PUBLIC or PRIVATE, be made accessible or blocked by a USE or IMPORT statement, and it can be renamed. If the *reduction-identifier* is the same as a generic name that is also the name of a derived type, the accessibility of the *reduction-identifier* is the same as that of the generic name.

If a subroutine or function used in *initializer-expression* or *combiner* is not an intrinsic procedure, it must have an accessible interface. Defined operators and defined assignments used in `initializer` or *combiner* must have accessible interfaces. All subroutines, functions, defined operators and defined assignments used in `initializer` or *combiner* must have accessible interfaces in the subprogram in which the corresponding REDUCTION clause appears. Procedures referenced in *combiner* and `initializer` cannot be alternate return subprograms.

The initial value of a user-defined reduction is not known before it is specified. The *initializer-clause* can be used to specify an initial value for the reduction variable. The *initializer-clause* will be executed to establish initial values for the private copies of reduction list items indicated in a REDUCTION clause that specifies the *reduction-identifier*.

If `initializer` is not specified, private reduction variables are initialized as follows:

- If the reduction variable is type COMPLEX, REAL, or INTEGER, the default initializer is the value zero.
- If the reduction variable specified in list of the REDUCTION clause is LOGICAL, the default initializer is the value .FALSE..
- If the reduction variable is of a default initialized derived type, the default initializer value is used.
- Otherwise, the initial value is unspecified.

If `initializer` is used in a target region, then a DECLARE TARGET construct (Linux* only) must be specified for any procedures that are executed during the execution of *combiner* or `initializer`.

If the execution of *combiner* or `initializer` results in the execution of an OpenMP* construct or an OpenMP* API call, the behavior is undefined. If the variable `omp_orig` is defined during execution of `initializer`, the behavior is unspecified.

Example

Consider that a DECLARE REDUCTION directive is used to declare a sum reduction for an integer component of a type `my_type` that is identified by the *reduction-identifier* '+'. It is then used in a REDUCTION clause of a parallel region to produce the sum of the thread numbers (numbered 1 thru 4) of the region:

```
module types
  type my_type
    integer :: component
  end type
  interface operator(+)
    module procedure :: my_add
  end interface
!$omp declare reduction (+ : my_type : omp_out = omp_out + omp_in) initializer (omp_priv =
my_type (/0/))

  contains
    function my_add (a1, a2)
      type(my_type),intent(in) :: a1, a2
      type(my_type)            :: my_add
      my_add%component = a1%component + a2%component
      return
    end function my_add
end module types

program main
  use types
  use omp_lib
  type(my_type) :: my_var

! Initialize the reduction variable before entering the OpenMP region
  my_var%component = 0

!$omp parallel reduction (+ : my_var) num_threads(4)
  my_var%component = omp_get_thread_num() + 1
!$omp end parallel

  print *, "sum of thread numbers is ", my_var%component
end program
```

The output of the program follows:

```
sum of thread numbers is 10
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
DECLARE TARGET
REDUCTION clause

DECLARE SIMD

OpenMP* Fortran Compiler Directive: *Creates a version of a function that can process multiple arguments using Single Instruction Multiple Data (SIMD) instructions from a single invocation from a SIMD loop.*

Syntax

```
!$OMP DECLARE SIMD(routine-name) [clause[[,] clause]... ]
```

routine-name Is the name of a routine (a function or subroutine). It cannot be a generic name; it must be a specific name. It also cannot be a procedure pointer or an entry name.

clause Is an optional vectorization clause. It can be one or more of the following:

- ALIGNED (list [:n])
- INBRANCH | NOTINBRANCH

  The INBRANCH clause specifies that the routine must always be called from inside a conditional statement of a SIMD loop.

  The NOTINBRANCH clause specifies that the routine must never be called from inside a conditional statement of a SIMD loop.

  If neither clause is specified, then the routine may or may not be called from inside a conditional statement of a SIMD loop.

  You can only specify INBRANCH or NOTINBRANCH; you cannot specify both.
- LINEAR (linear-list[: linear-step])
- PROCESSOR (cpuid) (an Intel® language extension)
- SIMDLEN(*n*)

  Specifies the number of concurrent arguments (*n*) for the SIMD version of *routine-name*. The *n* must be a constant positive integer expression.

  If SIMDLEN is not specified, the number of concurrent arguments for the *routine-name* is implementation defined.

  Only one SIMDLEN clause can appear in a DECLARE SIMD directive.
- UNIFORM(*list*)

  Tells the compiler that the values of the specified arguments have an invariant value for all concurrent invocations of the routine in the execution of a single SIMD loop.

  The *list* is one or more scalar variables that are dummy arguments in the specified routine.

  Multiple UNIFORM clauses are merged as a union.

The DECLARE SIMD construct enables the creation of SIMD versions of the specified subroutine or function. You can use multiple DECLARE SIMD constructs in a single procedure to produce more than one SIMD version of a procedure. These versions can be used to process multiple arguments from a single invocation from a SIMD loop concurrently.

When *routine-name* is executed, it cannot have any side-effects that would change its execution for concurrent iterations of a SIMD chunk. When the routine is called from a SIMD loop, it cannot cause the execution of any OpenMP* Fortran construct.

If a DECLARE SIMD directive is specified for a routine name with explicit interface and for the definition of the routine, they must match. Otherwise, the result is unspecified.

You cannot use procedure pointers to access routines created by the DECLARE SIMD directive.

You can only specify a particular variable in at most one instance of a UNIFORM or LINEAR clause.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

DECLARE TARGET

OpenMP* Fortran Compiler Directive: *Specifies that named variables, common blocks, functions, and subroutines are mapped to a device.*

Syntax

It takes one of the following forms:

```
!$OMP DECLARE TARGET [(extended-list)]
```

```
!$OMP DECLARE TARGET[clause[[,]clause]...]
```

extended-list

Is a list of one or more variables, functions, subroutines, data pointers, procedure pointers, or common blocks. If you specify more than one *extended-list* item, they must be separated by commas. A common block name must appear between slashes (/ /); you cannot specify a blank common block. The specified *extended-list* items can be used inside a target region that executes on the device.

If the *extended-list* item is a function or subroutine, it must not be a generic name or entry name. A device-specific version of the routine is created that can be called from a target region.

If the *extended-list* item is a variable:

- It is mapped to a corresponding variable in the device data environment. If the variable is initialized, the corresponding variable in the device data environment is initialized with the same value.
- It can only appear in the scope in which it is declared.
- It must be declared in the Fortran scope of a module, or it must have the SAVE attribute (explicitly or implicitly).

You cannot specify the following variables in the DECLARE TARGET directive:

- A THREADPRIVATE variable
- A variable that is part of another variable (for example, an element in an array or a field of a structure)
- A variable that is an element of a common block
- A variable that appears in an EQUIVALENCE statement

If the *extended-list* item is a common block:

- It must be declared to be a common block in the same scoping unit in which the DECLARE TARGET directive appears.
- If the DECLARE TARGET directive specifying the common block name appears in one program unit, a DECLARE TARGET directive must also appear in every other program unit that contains a COMMON statement specifying the same common block name. The directive must appear after the last relevant COMMON statement in the program unit.

clause

Is one of the following:

- TO (*extended-list*)

  Is a comma-separated collection of one or more *list* items or procedures.

  If a *list* item is a routine then a device-specific version of the routine is created that can be called from a target region.

  If a *list* item is a variable then the original variable is mapped to a corresponding variable in the device data environment as if it had appeared in a MAP clause with the *map-type* TO on the implicit TARGET DATA construct for each device.

  The *list* item is never removed from those device data environments.
- LINK (*list*)

  The *list* items of a LINK clause are not mapped by the DECLARE TARGET directive. Instead, their mapping occurs only when they are mapped by TARGET DATA or TARGET constructs.

If you specify *list*, this directive can only appear in a specification part of a subroutine, function, program, or module.

If you do not specify *list*, the directive must appear in the specification part of the relevant subroutine, function, or interface block.

If a DECLARE TARGET directive is specified in an interface block for a procedure, it must match a DECLARE TARGET directive in the definition of the procedure.

If a procedure is declared in a procedure declaration statement, any DECLARE TARGET directive containing the procedure name must appear in the same specification part.

The following additional rules apply to variables and common blocks:

- The DECLARE TARGET directive must appear in the declaration section of a scoping unit in which the common block or variable is declared.
- If a variable or common block is declared with the BIND attribute, the corresponding C entities must also be specified in a DECLARE TARGET directive in the C program.

The same *list* item must not appear multiple times in clauses on the same directive.

The same *list* item must not appear in both a TO clause on one DECLARE TARGET directive and a LINK clause on another DECLARE TARGET directive.

Variables with static storage and procedures used in an OMP TARGET region are implicitly treated as OMP DECLARE TARGET:

```
MODULE VARS
  INTEGER X
END MODULE

REAL FUNCTION FOO()
END FUNCTION

!$OMP TARGET
  X = FOO()       ! X and FOO are implicitly DECLARE TARGET
!$OMP END TARGET
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

DECODE

Statement: *Translates data from character to internal form. It is comparable to using internal files in formatted sequential READ statements.*

Syntax

```
DECODE (c,f,b[, IOSTAT=i-var] [, ERR=label]) [io-list]
```

c	Is a scalar integer expression. It is the number of characters to be translated to internal form.
f	Is a format identifier. An error occurs if more than one record is specified.
b	Is a scalar or array reference. If *b* is an array reference, its elements are processed in the order of subscript progression. *b* contains the characters to be translated to internal form.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and as zero if no error occurs (see I/O Status Specifier).
label	Is the label of an executable statement that receives control if an error occurs.
io-list	Is an I/O list. An I/O list is either an implied-DO list or a simple list of variables (except for assumed-size arrays). The list receives the data after translation to internal form. The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

The number of characters that the DECODE statement can translate depends on the data type of *b*. For example, an INTEGER(2) array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array.

The maximum number of characters a character variable or character array element can contain is the length of the character variable or character array element.

The maximum number of characters a character array can contain is the length of each element multiplied by the number of elements.

Example

In the following example, the DECODE statement translates the 12 characters in A to integer form (as specified by the FORMAT statement):

```
     DIMENSION K(3)
     CHARACTER*12 A,B
     DATA A/'123456789012'/
     DECODE(12,100,A) K
100  FORMAT(3I4)
```

The 12 characters are stored in array K:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

See Also

READ
WRITE
ENCODE

DEFAULT Clause

Parallel Directive Clause: *Lets you specify a scope for all variables in the lexical extent of a parallel region.*

Syntax

```
DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE )
```

The specifications have the following effects:

- PRIVATE - Makes all named objects in the lexical extent of the parallel region, including common block variables but excluding THREADPRIVATE variables, private to a thread as if you explicitly listed each variable in a PRIVATE clause.
- FIRSTPRIVATE - Makes all variables in the construct that have implicitly determined data-sharing attributes firstprivate as if you explicitly listed each variable in a FIRSTPRIVATE clause.
- SHARED - Makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if you explicitly listed each variable in a SHARED clause. If you do not specify a DEFAULT clause, this is the default.
- NONE - Specifies that there is no implicit default as to whether variables are PRIVATE or SHARED. In this case, you must specify the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION property of each variable you use in the lexical extent of the parallel region.

You can specify only one DEFAULT clause in a PARALLEL directive. You can exclude variables from a defined default by using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clauses.

Variables in THREADPRIVATE common blocks are not affected by this clause.

DEFINE and UNDEFINE

General Compiler Directives: *DEFINE creates a symbolic variable whose existence or value can be tested during conditional compilation. UNDEFINE removes a defined symbol.*

Syntax

```
!DIR$ DEFINE name[ = val]
```

```
!DIR$ UNDEFINE name
```

name	Is the name of the variable.
val	INTEGER(4). The value assigned to *name*.

DEFINE creates and UNDEFINE removes symbols for use with the IF (or IF DEFINED) compiler directive. Symbols defined with DEFINE directive are local to the directive. They cannot be declared in the Fortran program.

Because Fortran programs cannot access the named variables, the names can duplicate Fortran keywords, intrinsic functions, or user-defined names without conflict.

To test whether a symbol has been defined, use the IF DEFINED (*name*) directive. You can assign an integer value to a defined symbol. To test the assigned value of *name*, use the IF directive. IF test expressions can contain most logical and arithmetic operators.

Attempting to undefine a symbol that has not been defined produces a compiler warning.

The DEFINE and UNDEFINE directives can appear anywhere in a program, enabling and disabling symbol definitions.

Example

```
!DIR$ DEFINE testflag
!DIR$ IF DEFINED (testflag)
   write (*,*) 'Compiling first line'
!DIR$ ELSE
   write (*,*) 'Compiling second line'
!DIR$ ENDIF
!DIR$ UNDEFINE testflag
```

See Also

IF Directive Construct
General Compiler Directives
Syntax Rules for Compiler Directives
D compiler option
Equivalent Compiler Options

DEFINE FILE

Statement: *Establishes the size and structure of files with relative organization and associates them with a logical unit number.*

Syntax

`DEFINE FILE u(m,n,U,asv) [,u(m,n,U,asv)] ...`

u	Is a scalar 32-bit integer constant or variable that specifies the logical unit number.
m	Is a scalar integer constant or variable that specifies the number of records in the file.
n	Is a scalar integer constant or variable that specifies the length of each record in 16-bit words (2 bytes). For files with record lengths greater than 2^{32} -1, the OPEN statement should be used.
U	Specifies that the file is unformatted (binary); this is the only acceptable entry in this position.
asv	Is a scalar integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher numbered record in the file is assigned to *asv*; *asv* must not be a dummy argument.

The DEFINE FILE statement is comparable to the OPEN statement. In situations where you can use the OPEN statement, OPEN is the preferable mechanism for creating and opening files.

The DEFINE FILE statement specifies that a file containing *m* fixed-length records, each composed of *n*16-bit words, exists (or will exist) on the specified logical unit. The records in the file are numbered sequentially from 1 through *m*.

A DEFINE FILE statement does not itself open a file. However, the statement must be executed before the first direct access I/O statement referring to the specified file. The file is opened when the I/O statement is executed.

If this I/O statement is a WRITE statement, a direct access sequential file is opened, or created if necessary.

If the I/O statement is a READ or FIND statement, an existing file is opened, unless the specified file does not exist. If a file does not exist, an error occurs.

The DEFINE FILE statement establishes the variable *asv* as the associated variable of a file. At the end of each direct access I/O operation, the Fortran I/O system places in *asv* the record number of the record immediately following the one just read or written.

The associated variable always points to the next sequential record in the file (unless the associated variable is redefined by an assignment, input, or FIND statement). So, direct access I/O statements can perform sequential processing on the file by using the associated variable of the file as the record number specifier.

Example

```
DEFINE FILE 3(1000,48,U,NREC)
```

In this example, the DEFINE FILE statement specifies that the logical unit 3 is to be connected to a file of 1000 fixed-length records; each record is forty-eight 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted.

After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the record just processed.

See Also

OPEN

DELDIRQQ

Portability Function: *Deletes a specified directory.*

Module

USE IFPORT

Syntax

`result = DELDIRQQ (dir)`

dir	(Input) Character*(*). String containing the path of the directory to be deleted.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The directory to be deleted must be empty. It cannot be the current directory, the root directory, or a directory currently in use by another process.

Example

See the example for GETDRIVEDIRQQ.

See Also

GETDRIVEDIRQQ
GETDRIVEDIRQQ
MAKEDIRQQ
CHANGEDIRQQ
CHANGEDRIVEQQ
UNLINK

DELETE

Statement: *Deletes a record from a relative file.*

Syntax

```
DELETE ([UNIT=] io-unit[, REC=r][, ERR=label] [, IOSTAT=i-var])
```

io-unit	Is an external unit specifier.
r	Is a scalar numeric expression indicating the record number to be deleted.
label	Is the label of the branch target statement that receives control if an error occurs.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

In a relative file, the DELETE statement deletes the direct access record specified by *r*. If REC= *r* is omitted, the current record is deleted. When the direct access record is deleted, any associated variable is set to the next record number.

The DELETE statement logically removes the appropriate record from the specified file by locating the record and marking it as a deleted record. It then frees the position formerly occupied by the deleted record so that a new record can be written at that position.

> **NOTE**
> You must use compiler option vms for READs to detect that a record has been deleted.

Example

The following statement deletes the fifth record in the file connected to I/O unit 10:

```
DELETE (10, REC=5)
```

Suppose the following statement is specified:

```
DELETE (UNIT=9, REC=10, IOSTAT=IOS, ERR=20)
```

The tenth record in the file connected to unit 9 is deleted. If an error occurs, control is transferred to the statement labeled 20, and a positive integer is stored in the variable IOS.

See Also

Data Transfer I/O Statements
Branch Specifiers
vms compiler option

DELETEMENUQQ (W*S)

QuickWin Function: *Deletes a menu item from a QuickWin menu.*

Module

USE IFQWIN

Syntax

```
result = DELETEMENUQQ (menuID, itemID)
```

menuID	(Input) INTEGER(4). Identifies the menu that contains the menu item to be deleted, starting with 1 as the leftmost menu.
itemID	(Input) INTEGER(4). Identifies the menu item to be deleted, starting with 0 as the top menu item.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Example

```
USE IFQWIN
LOGICAL(4) result
CHARACTER(25) str
str = 'Add to EDIT Menu'C   ! Append to 2nd menu
result = APPENDMENUQQ(2, $MENUENABLED, str, WINSTATUS)
! Delete third item (EXIT) from menu 1 (FILE)
result = DELETEMENUQQ(1, 3)
! Delete entire fifth menu (WINDOW)
result = DELETEMENUQQ(5,0)
END
```

See Also

APPENDMENUQQ
INSERTMENUQQ
MODIFYMENUFLAGSQQ
MODIFYMENUROUTINEQQ
MODIFYMENUSTRINGQQ

DELFILESQQ

Portability Function: *Deletes all files matching the name specification, which can contain wildcards (* and ?).*

Module

USE IFPORT

Syntax

```
result = DELFILESQQ (files)
```

files	(Input) Character*(*). Files to be deleted. Can contain wildcards (* and ?).

Results

The result type is INTEGER(2). The result is the number of files deleted.

You can use wildcards to delete more than one file at a time. DELFILESQQ does not delete directories or system, hidden, or read-only files. Use this function with caution because it can delete many files at once. If a file is in use by another process (for example, if it is open in another process), it cannot be deleted.

Example

```
USE IFPORT
USE IFCORE
INTEGER(4) len, count
CHARACTER(80) file
CHARACTER(1) ch
WRITE(*,*) "Enter names of files to delete: "
len = GETSTRQQ(file)
IF (file(1:len) .EQ. '*.*') THEN
   WRITE(*,*) "Are you sure (Y/N)?"
   ch = GETCHARQQ()
   IF ((ch .NE. 'Y') .AND. (ch .NE. 'y')) STOP
END IF
count = DELFILESQQ(file)
WRITE(*,*) "Deleted ", count, " files."
END
```

See Also

FINDFILEQQ

DEPEND Clause

Parallel Directive Clause: *Enforces additional constraints on the scheduling of a task by enabling dependences between sibling tasks in the task region.*

Syntax

It takes one of the following forms:

In Target Control Directives !$OMP TARGET, etc., and TASKING Directive !$OMP TASK:

`DEPEND (`*`dependence-type`*` : `*`list`*`)`

In Synchronization Directive !$OMP ORDERED

`DEPEND (SOURCE)` *-or-*

`DEPEND (SINK:`*`vec`*`)`

dependence-type

Can be any one of the following clauses: IN, OUT, or INOUT.

For IN, the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an OUT or INOUT clause.

For OUT and INOUT, the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an IN, OUT, or INOUT clause.

list

Is one or more variables or non-zero length array sections. Any *list* items used in a DEPEND clause of the same task or sibling tasks must indicate identical storage or disjoint storage. The list items that appear in the DEPEND clause may include array sections.

Note that this enforced task dependence establishes a synchronization of accesses to each *list* item performed by a dependent task, with respect to accesses to the same *list* item performed by any previous tasks. You must properly synchronize access with respect to other concurrent accesses to each *list* item.

SOURCE	Specifies the satisfaction of cross-iteration dependences that arise from the current iteration.
SINK	Specifies a cross-iteration dependence, where the iteration vector *vec* indicates the iteration that satisfies the dependence.
vec	Is the iteration vector. It has the form: $x_1[\pm d_1], x_2[\pm d_2],\ldots, x_n[\pm d_n]$ where *n* is the value specified by the ORDERED clause in the DO loop directive, x_i denotes the loop iteration variable of the *i*-th nested loop associated with the loop directive, and d_i is a non-negative integer scalar constant. If *vec* does not occur in the iteration space, the DEPEND clause is ignored. Note that if *vec* does not indicate a lexicographically earlier iteration, it can cause a deadlock. For a *vec* element form of $x_i + d_i$ or $x_i - d_i$, the expression $x_i + d_i$ or $x_i - d_i$ for any value of the integer loop iteration variable x_i that can encounter the ordered construct must be computable in the loop iteration variable's type without overflow.

If a DEPEND clause appears in a TARGET or TARGET UPDATE directive, it is treated as if it had appeared on the implicit task construct that encloses the TARGET construct. These directives are only available on Linux* systems.

DEVICE Clause

Parallel Directive Clause: *Specifies the target device for a processor control directive like TARGET.*

Syntax

```
DEVICE (integer-expression)
```

integer-expression	Is an integer expression. It must evaluate to a positive scalar integer value.

At most one DEVICE clause can appear in a directive that allows the clause.

If DEVICE is not specified, the default device is determined by the internal control variable (ICV) named `device-num-var`.

DFLOAT

Elemental Intrinsic Function (Generic): *Converts an integer to double-precision real type.*

Syntax

```
result = DFLOAT (a)
```

a	(Input) Must be of type integer.

Results

The result type is double-precision real (by default, REAL(8) or REAL*8). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

Specific Name [1]	Argument Type	Result Type [2]
	INTEGER(1)	REAL(8)
DFLOTI	INTEGER(2)	REAL(8)
DFLOTJ	INTEGER(4)	REAL(8)
DFLOTK	INTEGER(8)	REAL(8)

[1]These specific functions cannot be passed as actual arguments.

[2]The setting of compiler options specifying double size can affect DFLOAT.

Example

DFLOAT (-4) has the value -4.0.

See Also

REAL

DFLOATI, DFLOATJ, DFLOATK

Portability Functions: *Convert an integer to double-precision real type.*

Module

USE IFPORT

Syntax

```
result = DFLOATI (i)
result = DFLOATJ (j)
result = DFLOATK (k)
```

i (Input) Must be of type INTEGER(2).

j (Input) Must be of type INTEGER(4).

k (Input) Must be of type INTEGER(8).

Results

The result type is double-precision real (REAL(8) or REAL*8).

See Also

DFLOAT

DIGITS

Inquiry Intrinsic Function (Generic): *Returns the number of significant digits for numbers of the same type and kind parameters as the argument.*

Syntax

```
result = DIGITS (x)
```

x	(Input) Must be of type integer or real; it can be scalar or array valued.

Results

The result is a scalar of type default integer.

The result has the value *q* if *x* is of type integer; it has the value *p* if *x* is of type real. Integer parameter *q* is defined in Model for Integer Data; real parameter *p* is defined in Model for Real Data.

Example

If *x* is of type REAL(4), DIGITS(x) has the value 24.

See Also

EXPONENT
RADIX
FRACTION
Data Representation Models

DIM

Elemental Intrinsic Function (Generic): *Returns the difference between two numbers (if the difference is positive).*

Syntax

```
result = DIM (x, y)
```

x	(Input) Must be of type integer or real.
y	(Input) Must have the same type and kind parameters as *x*.

Results

The result type and kind are the same as *x*. The value of the result is *x*- *y* if *x* is greater than *y*; otherwise, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Specific Name	Argument type	Result Type
BDIM	INTEGER(1)	INTEGER(1)
IIDIM[1]	INTEGER(2)	INTEGER(2)
IDIM [2]	INTEGER(4)	INTEGER(4)
KIDIM	INTEGER(8)	INTEGER(8)
DIM	REAL(4)	REAL(4)
DDIM	REAL(8)	REAL(8)
QDIM	REAL(16)	REAL(16)

[1]Or HDIM.

[2]Or JIDIM. For compatibility, IDIM can also be specified as a generic function for integer types.

Example

DIM (6, 2) has the value 4.

DIM (-4.0, 3.0) has the value 0.0.

The following shows another example:

```
INTEGER i
REAL r
REAL(8) d
i = IDIM(10, 5)            ! returns 5
r = DIM (-5.1, 3.7)        ! returns 0.0
d = DDIM (10.0D0, -5.0D0)  ! returns 15.0D0
```

See Also

Argument Keywords in Intrinsic Procedures

DIMENSION

Statement and Attribute: *Specifies that an object is an array, and defines the shape of the array.*

Syntax

The DIMENSION attribute can be specified in a type declaration statement or a DIMENSION statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] DIMENSION (*a-spec*) [, *att-ls*] :: *a*[(*a-spec*)][, *a*[(*a-spec*)]] ...

Statement:

DIMENSION [::]*a*(*a-spec*) [, *a*(*a-spec*)] ...

type
Is a data type specifier.

att-ls
Is an optional list of attribute specifiers.

a-spec
Is an array specification. It can be any of the following:

- An explicit-shape specification; for example, a(10,10)
- An assumed-shape specification; for example, a(:)
- A deferred-shape specification; for example, a(:,:)
- An assumed-size specification; for example, a(10,*)
- An assumed-rank specification; for example, a(..)
- An implicit-shape specification; for example, a(*)

For more information on array specifications, see Declaration Statements for Arrays.

In a type declaration statement, any array specification following an array overrides any array specification following DIMENSION.

a
Is the name of the array being declared.

Description

The DIMENSION attribute allocates a number of storage elements to each array named, one storage element to each array element in each dimension. The size of each storage element is determined by the data type of the array.

The total number of storage elements assigned to an array is equal to the number produced by multiplying together the number of elements in each dimension in the array specification. For example, the following statement defines ARRAY as having 16 real elements of 4 bytes each and defines MATRIX as having 125 integer elements of 4 bytes each:

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

An array can also be declared in the following statements: ALLOCATABLE, AUTOMATIC, COMMON, POINTER, STATIC, TARGET.

Example

The following examples show type declaration statements specifying the DIMENSION attribute:

```
REAL, DIMENSION(10, 10) :: A, B, C(10, 15)  ! Specification following C
                                            ! overrides the one following
                                            ! DIMENSION
REAL(8), DIMENSION(5,-2:2) :: A,B,C
```

The following are examples of the DIMENSION statement:

```
DIMENSION BOTTOM(12,24,10)
DIMENSION X(5,5,5), Y(4,85), Z(100)
DIMENSION MARK(4,4,4,4)
SUBROUTINE APROC(A1,A2,N1,N2,N3)
DIMENSION A1(N1:N2), A2(N3:*)
CHARACTER(LEN = 20) D
DIMENSION A(15), B(15, 40), C(-5:8, 7), D(15)
```

You can declare arrays by using type statements and ALLOCATABLE attributes and statements, for example:

```
INTEGER A(2,0:2)
COMPLEX F
ALLOCATABLE F(:,:)
REAL(8), ALLOCATABLE, DIMENSION( :, :, : ) :: E
```

You can declare an implicit-shape constant array by using a type statement and a PARAMETER attribute, for example:

```
INTEGER, PARAMETER :: R(*) = [1,2,3]
```

You can also declare arrays by using type and ALLOCATABLE statements, for example:

```
INTEGER A(2,0:2)
COMPLEX F
ALLOCATABLE F(:,:)
REAL(8), ALLOCATABLE, DIMENSION( :, :, : ) :: E
```

You can specify both the upper and lower dimension bounds. If, for example, one array contains data from experiments numbered 28 through 112, you could dimension the array as follows:

```
DIMENSION experiment(28:112)
```

Then, to refer to the data from experiment 72, you would reference experiment(72).

Array elements are stored in column-major order: the leftmost subscript is incremented first when the array is mapped into contiguous memory addresses. For example, consider the following statements:

```
INTEGER(2) a(2, 0:2)
DATA a /1, 2, 3, 4, 5, 6/
```

These are equivalent to:

```
INTEGER(2) a
DIMENSION a(2, 0:2)
DATA a /1, 2, 3, 4, 5, 6/
```

If `a` is placed at location decimal 1000 in memory, the preceding DATA statement produces the following mapping.

Array element	Address (decimal)	Value
a(1,0)	1000	1
a(2,0)	1002	2
a(1,1)	1004	3
a(2,1)	1006	4
a(1,2)	1008	5
a(2,2)	100A	6

The following DIMENSION statement defines an assumed-size array in a subprogram:

```
DIMENSION data (19,*)
```

At execution time, the array data is given the size of the corresponding array in the calling program.

The following program fragment dimensions two arrays:

```
 ...
 SUBROUTINE Subr (matrix, rows, vector)
 REAL MATRIX, VECTOR
 INTEGER ROWS
 DIMENSION MATRIX (ROWS,*), VECTOR (10),
+  LOCAL (2,4,8)
 MATRIX (1,1) = VECTOR (5)
 ...
```

See Also

ALLOCATE
Declaration Statements for Arrays
Arrays

DISPLAYCURSOR (W*S)

Graphics Function: *Controls cursor visibility.*

Module

USE IFQWIN

Syntax

`result = DISPLAYCURSOR (toggle)`

toggle	(Input) INTEGER(2). Constant that defines the cursor state. Has two possible values:

- $GCURSOROFF - Makes the cursor invisible regardless of its current shape and mode.
- $GCURSORON - Makes the cursor always visible in graphics mode.

Results

The result type is INTEGER(2). The result is the previous value of *toggle*.

Cursor settings hold only for the currently active child window. You need to call DISPLAYCURSOR for each window in which you want the cursor to be visible.

A call to SETWINDOWCONFIG turns off the cursor.

See Also

SETTEXTCURSOR
SETWINDOWCONFIG

DISTRIBUTE

OpenMP* Fortran Compiler Directive: *Specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a teams construct.*

Syntax

```
!$OMP DISTRIBUTE [clause[[,] clause]... ]
   do-loop
[!$OMP END DISTRIBUTE]
```

clause

Is one of the following:

- COLLAPSE (n)
- DIST_SCHEDULE (*kind* [, *chunk-size*])

  Specifies how iterations are divided.

  The *kind* must be STATIC (see SCHEDULE in DO Directive).

  The *chunk-size* must be a positive scalar integer expression. If specified, iterations are divided into chunks of size *chunk-size*. Chunks are assigned round-robin to the teams of the parallel region in the order of the team numbers.

  When no *chunk-size* is specified, iterations are divided into chunks that are approximately equal in size. In this case, at most one chunk is distributed to each team of the parallel region.
- FIRSTPRIVATE (list)
- LASTPRIVATE ([CONDITIONAL:] list)
- PRIVATE (list)

do-loop

Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

If an END DO directive follows a DO construct in which several loop statements share a DO termination statement, then the directive can only be specified for the outermost of these DO statements. The DISTRIBUTE construct inherits the restrictions of the loop construct.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

If more than one loop is associated with the DISTRIBUTE construct, then the iterations of all associated loops are collapsed into one larger iteration space. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

You cannot branch out of a DO loop associated with a DISTRIBUTE directive.

The binding thread set for a DISTRIBUTE construct is the set of master threads created by a TEAMS construct. A DISTRIBUTE region binds to the innermost enclosing team's parallel region. Only the threads that execute the binding team's parallel region participate in the execution of the loop iterations. A DISTRIBUTE construct must be closely nested in a team's region.

If used, the END DISTRIBUTE directive must appear immediately after the end of the loop. If you do not specify an END DISTRIBUTE directive, an END DISTRIBUTE directive is assumed at the end of the *do-loop*.

The DISTRIBUTE construct is associated with loop iterations that follow the directive. The iterations are distributed across the master threads of all teams that execute the team's parallel region to which the DISTRIBUTE region binds.

A list item may appear in a FIRSTPRIVATE or LASTPRIVATE clause but not both.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
DO Directive
TEAMS
Parallel Processing Model for information about Binding Sets

DISTRIBUTE PARALLEL DO

OpenMP* Fortran Compiler Directive: *Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.*

Syntax

```
!$OMP DISTRIBUTE PARALLEL DO [clause[[,] clause]... ]
   do-loop
[!$OMP END DISTRIBUTE PARALLEL DO]
```

clause
Can be any of the clauses accepted by the DISTRIBUTE or PARALLEL DO directives with identical meanings and restrictions except for ORDERED and LINEAR .

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

If the END DISTRIBUTE PARALLEL DO directive is not specified, an END DISTRIBUTE PARALLEL DO directive is assumed at the end of *do-loop*.

This directive specifies a composite construct.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

DISTRIBUTE PARALLEL DO SIMD

OpenMP* Fortran Compiler Directive: *Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP DISTRIBUTE PARALLEL DO SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END DISTRIBUTE PARALLEL DO SIMD]
```

clause — Can be any of the clauses accepted by the DISTRIBUTE or PARALLEL DO SIMD directives with identical meanings and restrictions.

do-loop — Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

If the END DISTRIBUTE PARALLEL DO SIMD directive is not specified, an END DISTRIBUTE PARALLEL DO SIMD directive is assumed at the end of *do-loop*.

This directive specifies a composite construct.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

DISTRIBUTE POINT

General Compiler Directive: *Specifies loop distribution. It suggests a location at which a DO loop can be split.*

Syntax

```
!DIR$ DISTRIBUTE POINT
```

Loop distribution causes large loops to be distributed (split) into smaller ones. The resulting loops contain a subset of the instructions from the initial loop. Loop distribution can enable software pipelining to be applied to more loops. It can also reduce register pressure and improve both instruction and data cache use.

If the directive is placed before a loop, the compiler will determine where to distribute; data dependencies are observed.

If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependencies are ignored. Currently only one distribute directive is supported if the directive is placed inside the loop.

Example

```
!DIR$ DISTRIBUTE POINT
   do i =1, m
     b(i) = a(i) +1
     ....
     c(i) = a(i) + b(i) ! Compiler will decide
     ! where to distribute.
     ! Data dependencies are
     ! observed
     ....
     d(i) = c(i) + 1
   enddo
   do i =1, m
     b(i) = a(i) +1
     ....
!DIR$ DISTRIBUTE POINT
     call sub(a, n)! Distribution will start here,
     ! ignoring all loop-carried
     ! depedencies
     c(i) = a(i) + b(i)
     ....
     d(i) = c(i) + 1
   enddo
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops

DISTRIBUTE SIMD

OpenMP* Fortran Compiler Directive: *Specifies a loop that will be distributed across the master threads of the teams region. It will be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP DISTRIBUTE SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END DISTRIBUTE SIMD]
```

clause	Can be any of the clauses accepted by the DISTRIBUTE or SIMD directives with identical meanings and restrictions.
do-loop	Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer. All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops. The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

If the END DISTRIBUTE SIMD directive is not specified, an END DISTRIBUTE SIMD directive is assumed at the end of *do-loop*.

This directive specifies a composite construct.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

DLGEXIT (W*S)

Dialog Subroutine: *Closes an open dialog box.*

Module

USE IFLOGM

Syntax

`CALL DLGEXIT (dlg)`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

If you want to exit a dialog box on a condition other than the user selecting the OK or Cancel button, you need to include a call to DLGEXIT from within your callback routine. DLGEXIT saves the data associated with the dialog box controls and then closes the dialog box. The dialog box is exited after DLGEXIT has returned control back to the dialog manager, not immediately after the call to DLGEXIT.

Example

```
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE IFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
...
  CALL DLGEXIT (dlg)
```

See Also

DLGSETRETURN
DLGINIT
DLGMODAL

DLGMODELESS

DLGFLUSH (W*S)

Dialog Subroutine: *Updates the display of a dialog box.*

Module

USE IFLOGM

Syntax

CALL DLGFLUSH (*dlg*[,*flushall*])

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
flushall	(Input; optional) Logical. If .FALSE. (the default), then only the controls that the dialog routines have marked as changed are updated. If .TRUE., all controls are updated with the state of the controls as known by the dialog routines. Normally, you would not set *flushall* to .TRUE..

When your application calls DLGSET to change a property of a control in a dialog box, the change is not immediately reflected in the displayed dialog box. Changes are applied when the dialog box is first displayed, and then after every dialog callback to the user's code.

This design expects that, after a call to DLGMODAL or DLGMODELESS, every call to DLGSET will be made from within a callback routine, and that the callback routine finishes quickly. This is true most of the time.

However, there may be cases where you want to change a control outside of a dialog callback, or from within a loop in a dialog callback.

In these cases, DLGFLUSH is required, but is not always sufficient, to update the dialog display. DLGFLUSH sends pending Windows* system messages to the dialog box and the controls that it contains. However, many display changes do not appear until after the program reads and processes these messages. A loop that processes the pending messages may be required; for example:

```
use IFWINTY
use USER32
use IFLOGM
logical lNotQuit, lret
integer iret
TYPE (T_MSG) mesg
lNotQuit = .TRUE.
do while (lNotQuit .AND. (PeekMessage(mesg, 0, 0, 0, PM_NOREMOVE) <> 0))
  lNotQuit = GetMessage(mesg, NULL, 0, 0)
    if (lNotQuit) then
      if (DLGISDLGMESSAGE(mesg) .EQV. .FALSE) then
         lret = TranslateMessage(mesg)
         iret = DispatchMessage(mesg)
      end if
    end if
end do
```

See Also

DLGINIT
DLGMODAL

DLGMODELESS
DLGSET
DLGSETSUB

DLGGET, DLGGETINT, DLGGETLOG, DLGGETCHAR (W*S)

Dialog Functions: *Return the state of the dialog control variable.*

Module

USE IFLOGM

Syntax

```
result = DLGGET (dlg,controlid,value[,index])
result = DLGGETINT (dlg,controlid,value[,index])
result = DLGGETLOG (dlg,controlid,value[,index])
result = DLGGETCHAR (dlg,controlid,value[,index])
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).
value	(Output) Integer, logical, or character. The value of the control's variable.
index	(Input; optional) Integer. Specifies the control variable whose value is retrieved. Necessary if the control has more than one variable of the same data type and you do not want to get the value of the default for that type.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the DLGGET functions to retrieve the values of variables associated with your dialog box controls. Each control has at least one of the integer, logical, or character variable associated with it, but not necessarily all. For information about the location of a document that contains lists of index variables for each control type, see Additional Documentation: Creating Fortran Applications that Use Windows* Features.

You can use DLGGET to retrieve the value of any variable. You can also use DLGGETINT to retrieve an integer value, or DLGGETLOG and DLGGETCHAR to retrieve logical and character values, respectively. If you use DLGGET, you do not have to worry about matching the function to the variable type. If you use the wrong function type for a variable or try to retrieve a variable type that is not available, the DLGGET functions return .FALSE..

If two or more controls have the same *controlid*, you cannot use these controls in a DLGGET operation. In this case the function returns .FALSE..

The dialog box does not need to be open to access its control variables.

Example

```
USE IFLOGM
INCLUDE "THISDLG.FD"
TYPE (DIALOG)  dlg
INTEGER        val
LOGICAL        retlog, is_checked
CHARACTER(256) text
...
retlog = DLGGET (dlg, IDC_CHECKBOX1, is_checked, dlg_status)
retlog = DLGGET (dlg, IDC_SCROLLBAR2, val, dlg_range)
retlog = DLGGET (dlg, IDC_STATIC_TEXT1, text, dlg_title)
...
```

See Also

DLGSET
DLGSETSUB
DLGINIT
DLGMODAL
DLGMODELESS

DLGINIT, DLGINITWITHRESOURCEHANDLE (W*S)

Dialog Functions: *Initialize a dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGINIT (id,dlg)`

`result = DLGINITWITHRESOURCEHANDLE (id,hinst,dlg)`

id	(Input) INTEGER(4). Dialog identifier. Can be either the symbolic name for the dialog or the identifier number, both listed in the Include file (with extension .FD).
dlg	(Output) Derived type `dialog`. Contains dialog box parameters.
hinst	(Input) INTEGER(HANDLE). Handle of the module instance in which the dialog resource can be found. INTEGER(HANDLE) is INTEGER(4) on IA-32 architecture and INTEGER(8) on Intel® 64 architecture.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

DLGINIT must be called to initialize a dialog box before it can be used with DLGMODAL, DLGMODELESS, or any other dialog function.

DLGINIT will only search for the dialog box resource in the main application. For example, it will not find a dialog box resource that has been built into a dynamic link library.

DLGINITWITHRESOURCEHANDLE can be used when the dialog resource is not in the main application. If the dialog resource is in a dynamic link library (DLL), *hinst* must be the value passed as the first argument to the DLLMAIN procedure.

Dialogs can be used from any application, including console, QuickWin, and Windows* applications.

Example

```
USE IFLOGM
INCLUDE 'DLG1.FD'
LOGICAL retlog
TYPE (DIALOG) thisdlg
...
retlog = DLGINIT (IDD_DLG3, thisdlg)
IF (.not. retlog) THEN
  WRITE (*,*) 'ERROR: dialog not found'
ELSE
...
```

See Also

DLGEXIT
DLGMODAL
DLGMODELESS
DLGUNINIT

DLGISDLGMESSAGE, DLGISDLGMESSAGEWITHDLG (W*S)

Dialog Functions: *Determine whether the specified message is intended for one of the currently displayed modeless dialog boxes, or a specific dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGISDLGMESSAGE (mesg)`

`result = DLGISDLGMESSAGEWITHDLG (mesg, dlg)`

mesg	(Input) Derived type `T_MSG`. Contains a Windows message.
dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

Results

The result type is LOGICAL(4). The result is .TRUE. if the message is processed by the dialog box. Otherwise, the result is .FALSE. and the message should be further processed.

DLGISDLGMESSAGE must be called in the message loop of Windows applications that display a modeless dialog box using DLGMODELESS. DLGISDGMESSAGE determines whether the message is intended for one of the currently displayed modeless dialog boxes. If it is, it passes the message to the dialog box to be processed.

DLGISDLGMESSAGEWITHDLG specifies a particular dialog box to check. Use DLGISDLGMESSAGEWITHDLG when the message loop is in a main application and the currently active modeless dialog box was created by a DLL.

Example

```
use IFLOGM
include 'resource.fd'
type (DIALOG) dlg
type (T_MSG)  mesg
```

```
integer*4 ret
logical*4 lret
...
! Create the main dialog box and set up the controls and callbacks
lret = DlgInit(IDD_THERM_DIALOG, dlg)
lret = DlgSetSub(dlg, IDD_THERM_DIALOG, ThermSub)
...
lret = DlgModeless(dlg, nCmdShow)
...
! Read and process messsages
do while( GetMessage (mesg, NULL, 0, 0) /= 0 )
   ! Note that DlgIsDlgMessage must be called in order to give
   ! the dialog box first chance at the message.
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
      lret = TranslateMessage( mesg )
      ret  = DispatchMessage( mesg )
   end if
end do
! Cleanup dialog box memory and exit the application
call DlgUninit(dlg)
WinMain = mesg%wParam
return
```

See Also

DLGMODELESS

DLGMODAL, DLGMODALWITHPARENT (W*S)

Dialog Functions: *Display a dialog box and process user control selections made within the box.*

Module

USE IFLOGM

Syntax

`result = DLGMODAL (dlg)`

`result = DLGMODAL (dlg, hwndParent)`

dlg

(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

hwndParent

(Input) Integer. Specifies the parent window for the dialog box. If omitted, the value is determined in this order:

1. If DLGMODAL is called from the callback of a modal or modeless dialog box, then that dialog box is the parent window.
2. If it is a QuickWin or Standard Graphics application, then the frame window is the parent window.
3. The Windows* desktop window is the parent window.

Results

The result type is INTEGER(4). By default, if successful, it returns the identifier of the control that caused the dialog to exit; otherwise, it returns -1. The return value can be changed with the DLGSETRETURN subroutine.

During execution, DLGMODAL displays a dialog box and then waits for user control selections. When a control selection is made, the callback routine, if any, of the selected control (set with DLGSETSUB) is called.

The dialog remains active until an exit control is executed: either the default exit associated with the OK and Cancel buttons, or DLGEXIT within your own control callbacks. DLGMODAL does not return a value until the dialog box is exited.

The default return value for DLGMODAL is the identifier of the control that caused it to exit (for example, IDOK for the OK button and IDCANCEL for the Cancel button). You can specify your own return value with DLGSETRETURN from within one of your dialog control callback routines. You should not specify -1 as your return value, because this is the error value DLGMODAL returns if it cannot open the dialog.

Use DLGMODALWITHPARENT when you want the parent window to be other than the default value (see argument *hwndParent* above). In particular, in an SDI or MDI Windows application, you may want the parent window to be the main application window. The parent window is disabled for user input while the modal dialog box is displayed.

Example

```
USE IFLOGM
INCLUDE "MYDLG.FD"
INTEGER return
TYPE (DIALOG) mydialog
...
return = DLGMODAL (mydialog)
...
```

See Also

DLGSETRETURN
DLGSETSUB
DLGINIT
DLGEXIT

DLGMODELESS (W*S)

Dialog Function: *Displays a modeless dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGMODELESS (dlg[,nCmdShow,hwndParent])`

dlg
(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user. The variable passed to this function must remain in memory for the duration of the dialog box, that is from the DLGINIT call through the DLGUNINIT call.

The variable can be declared as global data in a module, as a variable with the STATIC attribute, or in a calling procedure that is active for the duration of the dialog box. It must not be an AUTOMATIC variable in the procedure that calls DLGMODELESS.

nCmdShow
(Input) Integer. Specifies how the dialog box is to be shown. It must be one of the following values:

Value	Description
SW_HIDE	Hides the dialog box.
SW_MINIMIZE	Minimizes the dialog box.
SW_RESTORE	Activates and displays the dialog box. If the dialog box is minimized or maximized, the Windows system restores it to its original size and position.
SW_SHOW	Activates the dialog box and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates the dialog box and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates the dialog box and displays it as an icon.
SW_SHOWMINNOACTIVE	Displays the dialog box as an icon. The window that is currently active remains active.
SW_SHOWNA	Displays the dialog box in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	Displays the dialog box in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays the dialog box. If the dialog box is minimized or maximized, the Windows* system restores it to its original size and position.

The default value is SW_SHOWNORMAL.

hwndParent

(Input) Integer. Specifies the parent window for the dialog box. The default value is determined in this order:

1. If DLGMODELESS is called from a callback of a modeless dialog box, then that dialog box is the parent window.
2. The Windows desktop window is the parent window.

Results

The result type is LOGICAL(4). The value is .TRUE. if the function successfully displays the dialog box. Otherwise the result is .FALSE..

During execution, DLGMODELESS displays a modeless dialog box and returns control to the calling application. The dialog box remains active until DLGEXIT is called, either explicitly or as the result of the invocation of a default button callback.

DLGMODELESS is typically used in a Windows application. The application must contain a message loop that processes Windows messages. The message loop must call DLGISDLGMESSAGE for each message. See the example below in the Example section. Multiple modeless dialog boxes can be displayed at the same time. A modal dialog box can be displayed from a modeless dialog box by calling DLGMODAL from a modeless dialog callback. However, DLGMODELESS cannot be called from a modal dialog box callback.

DLGMODELESS also can be used in a Console, DLL, or LIB project. However, the requirements remain that the application must contain a message loop and must call DLGISDLGMESSAGE for each message. For an example of calling DLGMODELESS in a DLL project, see the Dllprgrs sample in the `...\SAMPLES\DIALOG`folder.

Use the DLG_INIT callback with DLGSETSUB to perform processing immediately after the dialog box is created and before it is displayed, and to perform processing immediately before the dialog box is destroyed.

Example

```
use IFLOGM
include 'resource.fd'
type (DIALOG)   dlg
type (T_MSG)    mesg
integer*4   ret
logical*4   lret
...
! Create the main dialog box and set up the controls and callbacks
lret = DlgInit(IDD_THERM_DIALOG, dlg)
lret = DlgSetSub(dlg, IDD_THERM_DIALOG, ThermSub)
...
lret = DlgModeless(dlg, nCmdShow)
...
! Read and process messsages
do while( GetMessage (mesg, NULL, 0, 0) )
   ! Note that DlgIsDlgMessage must be called in order to give
   ! the dialog box first chance at the message.
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
     lret = TranslateMessage( mesg )
     ret  = DispatchMessage( mesg )
   end if
end do
   ! Cleanup dialog box memory and exit the application
call DlgUninit(dlg)
WinMain = mesg%wParam
return
```

See Also

DLGSETSUB
DLGINIT
DLGEXIT
DLGISDLGMESSAGE

DLGSENDCTRLMESSAGE (W*S)

Dialog Function: *Sends a Windows message to a dialog box control.*

Module

USE IFLOGM

Syntax

result = DLGSENDCTRLMESSAGE (*dlg*,*controlid*,*msg*,*wparam*,*lparam*)

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of the control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).
msg	(Input) Integer. Derived type `T_MSG`. Specifies the message to be sent.
wparam	(Input) Integer. Specifies additional message specific information.
lparam	(Input) Integer. Specifies additional message specific information.

Results

The result type is INTEGER(4). The value specifies the result of the message processing and depends upon the message sent.

The dialog box must be currently active by a call to DLGMODAL or DLGMODELESS. This function does not return until the message has been processed by the control.

Example

```
use IFLOGM
include 'resource.fd'
type (dialog)    dlg
integer          callbacktype
integer          cref
integer          iret

if (callbacktype == dlg_init) then
    ! Change the color of the Progress bar to red
    ! NOTE:  The following message succeeds only if Internet Explorer 4.0
    !        or later is installed
        cref = Z'FF'    ! Red
        iret = DlgSendCtrlMessage(dlg, IDC_PROGRESS1, PBM_SETBARCOLOR, 0, cref)
endif
```

See Also

DLGINIT
DLGSETSUB
DLGMODAL
DLGMODELESS

DLGSET, DLGSETINT, DLGSETLOG, DLGSETCHAR (W*S)

Dialog Functions: *Set the values of dialog control variables.*

Module

USE IFLOGM

Syntax

```
result = DLGSET (dlg,controlid,value[,index])
result = DLGSETINT (dlg,controlid,value[,index])
result = DLGSETLOG (dlg,controlid,value[,index])
result = DLGSETCHAR (dlg,controlid,value[,index])
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).
value	(Input) Integer, logical, or character. The value of the control's variable.
index	(Input; optional) Integer. Specifies the control variable whose value is set. Necessary if the control has more than one variable of the same data type and you do not want to set the value of the default for that type.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the DLGSET functions to set the values of variables associated with your dialog box controls. Each control has at least one of the integer, logical, or character variables associated with it, but not necessarily all. For information about the location of a document that contains lists of index variables for each control type, see Additional Documentation: Creating Fortran Applications that Use Windows* OS Features.

You can use DLGSET to set any control variable. You can also use DLGSETINT to set an integer variable, or DLGSETLOG and DLGSETCHAR to set logical and character values, respectively. If you use DLGSET, you do not have to worry about matching the function to the variable type. If you use the wrong function type for a variable or try to set a variable type that is not available, the DLGSET functions return .FALSE..

Calling DLGSET does not cause a callback routine to be called for the changing value of a control. In particular, when inside a callback, performing a DLGSET on a control does not cause the associated callback for that control to be called. Callbacks are invoked automatically only by user action on the controls in the dialog box. If the callback routine needs to be called, you can call it manually after the DLGSET is executed.

If two or more controls have the same *controlid*, you cannot use these controls in a DLGSET operation. In this case the function returns .FALSE..

Example

```
USE IFLOGM
INCLUDE "DLGRADAR.FD"
TYPE (DIALOG)  dlg
LOGICAL        retlog
...
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 400, dlg_range)
```

```
retlog = DLGSET (dlg, IDC_CHECKBOX1, .FALSE., dlg_status)
retlog = DLGSET (dlg, IDC_RADIOBUTTON1, "Hot Button", dlg_title)
...
```

See Also

DLGSETSUB
DLGGET

DLGSETCTRLEVENTHANDLER (W*S)

Dialog Function: *Assigns user-written event handlers to ActiveX controls in a dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGSETCTRLEVENTHANDLER (dlg,controlid,handler,dispid[,iid])`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be the symbolic name for the control or the identifier number, both listed in the include (with extension .FD) file.
handler	(Input) Name of the routine to be called when the event occurs. It must be declared EXTERNAL.
dispid	(Input) Integer. Specifies the member id of the method in the event interface that identifies the event.
iid	(Input; optional) Derived type `GUID`, which is defined in the IFWINTY module. Specifies the interface identifier of the source (event) interface. If omitted, the default source interface of the ActiveX control is used.

Results

The result type is INTEGER(4). The result is an HRESULT describing the status of the operation.

When the ActiveX control event occurs, the handler associated with the event is called. You call DLGSETCTRLEVENTHANDLER to specify the handler to be called.

The events supported by an ActiveX control and the interfaces of the handlers are determined by the ActiveX control.

You can find this information in one of the following ways:

- By reading the documentation of the ActiveX control.
- By using a tool that lets you examine the type information of the ActiveX control;, such as the OLE-COM Object Viewer.
- By using the Fortran Module Wizard to generate a module that contains Fortran interfaces to the ActiveX control, and examining the generated module.

The handler that you define in your application must have the interface that the ActiveX control expects, including calling convention and parameter passing mechanisms. Otherwise, your application will likely crash in unexpected ways because of the application's stack getting corrupted.

Note that an object is always the first parameter in an event handler. This object value is a pointer to the control's source (event) interface, *not* the IDispatch pointer of the control. You can use DLGGET with the DLG_IDISPATCH index to retrieve the control's IDispatch pointer.

Example

```
USE IFLOGM
ret = DlgSetCtrlEventHandler(            &
      dlg,                          &
      IDC_ACTIVEMOVIECONTROL1,      & ! Identifies the control
      ReadyStateChange,             & ! Name of the event handling routine
      -609,                         & ! Member id of the ActiveMovie's
                                    & !   control ReadyStateChange event.
      IID_DActiveMovieEvents2 )           ! Identifer of the source (event)
                                          !    interface.
```

See Also

DLGINIT
DLGGET
DLGMODAL
DLGMODELESS
DLGSETSUB

DLGSETRETURN (W*S)

Dialog Subroutine: *Sets the return value for the DLGMODAL function from within a callback subroutine.*

Module

USE IFLOGM

Syntax

`CALL DLGSETRETURN (dlg,retval)`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
retval	(Input) Integer. Specifies the return value for DLGMODAL upon exiting.

DLGSETRETURN overrides the default return value with *retval*. You can set your own value as a means of determining the condition under which the dialog box was closed. The default return value for an error condition is -1, so you should not use -1 as your return value.

DLGSETRETURN should be called from within a callback routine, and is generally used with DLGEXIT, which causes the dialog box to be exited from a control callback rather than the user selecting the OK or Cancel button.

Example

```
SUBROUTINE SETRETSUB (dlg, button_id, callbacktype)
USE IFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG) dlg
LOGICAL     is_checked, retlog
INTEGER     return, button_id, callbacktype
...
```

```
retlog = DLGGET(dlg, IDC_CHECKBOX4, is_checked, dlg_state)
IF (is_checked) THEN
   return = 999
ELSE
   return = -999
END IF
CALL DLGSETRETURN (dlg, return)
CALL DLGEXIT (dlg)
END SUBROUTINE SETRETSUB
```

See Also

DLGEXIT
DLGMODAL

DLGSETSUB (W*S)

Dialog Function: *Assigns your own callback subroutines to dialog controls and to the dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGSETSUB (dlg,controlid,value[,index])`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be the symbolic name for the control or the identifier number, both listed in the include (with extension .FD) file, or it can be the identifier of the dialog box.
value	(Input) EXTERNAL. Name of the routine to be called when the callback event occurs.
index	(Input; optional) Integer. Specifies which callback routine is executed when the callback event occurs. Necessary if the control has more than one callback routine.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

When a callback event occurs (for example, when you select a check box), the callback routine associated with that callback event is called. You use DLGSETSUB to specify the subroutine to be called. All callback routines should have the following interface:

SUBROUTINE *callbackname*(`dlg`, *control_id*, *callbacktype*)

!DIR$ ATTRIBUTES DEFAULT :: *callbackname*

callbackname	Is the name of the callback routine.
dlg	Refers to the dialog box and allows the callback to change values of the dialog controls.

control_id	Is the name of the control that caused the callback.
callbacktype	(Input; optional) Integer. Specifies which callback routine is executed when the callback event occurs. Necessary if the control has more than one callback routine.

The *control_id* and *callbacktype* parameters let you write a single subroutine that can be used with multiple callbacks from more than one control. Typically, you do this for controls comprising a logical group. You can also associate more than one callback routine with the same control, but you must use then use *index* parameter to indicate which callback routine to use.

The *control_id* can also be the identifier of the dialog box. The dialog box supports two *callbacktype*s, DLG_INIT and DLG_SIZECHANGE. The DLG_INIT callback is executed immediately after the dialog box is created with *callbacktype* DLG_INIT, and immediately before the dialog box is destroyed with *callbacktype* DLG_DESTROY. DLG_SIZECHANGE is called when the size of a dialog is changed.

Callback routines for a control are called after the value of the control has been updated based on the user's action.

If two or more controls have the same *controlid*, you cannot use these controls in a DLGSETSUB operation. In this case, the function returns .FALSE..

Example

```
USE IFLOGM
INCLUDE "MYDLG.FD"
TYPE (dialog) mydialog
LOGICAL retlog
INTEGER return
EXTERNAL RADIOSUB
retlog = DLGINIT(IDD_mydlg, mydialog)
retlog = DLGSETSUB (mydialog, IDC_RADIO_BUTTON1, RADIOSUB)
retlog = DLGSETSUB (mydialog, IDC_RADIO_BUTTON2, RADIOSUB)
return = DLGMODAL(mydialog)
END
SUBROUTINE RADIOSUB( dlg, id, callbacktype )
!DIR$ ATTRIBUTES DEFAULT :: radiosub
  USE IFLOGM
  TYPE (dialog) dlg
  INTEGER id, callbacktype
  INCLUDE 'MYDLG.FD'
  CHARACTER(256) text
  LOGICAL retlog
  SELECT CASE (id)
    CASE (IDC_RADIO_BUTTON1)
    ! Radio button 1 selected by user so
    ! change text accordingly
      text = 'Statistics Package A'
      retlog = DLGSET( dlg, IDC_STATICTEXT1, text )
    CASE (IDC_RADIO_BUTTON2)
    ! Radio button 2 selected by user so
    ! change text accordingly
      text = 'Statistics Package B'
      retlog = DLGSET( dlg, IDC_STATICTEXT1, text )
  END SELECT
END SUBROUTINE RADIOSUB
```

See Also

DLGSET

DLGGET

DLGSETTITLE (W*S)

Dialog Subroutine: *Sets the title of a dialog box.*

Module

USE IFLOGM

Syntax

```
CALL DLGSETTITLE (dlg,title)
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
title	(Input) Character*(*). Specifies text to be the title of the dialog box.

Use this routine when you want to specify the title for a dialog box.

Example

```
USE IFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG) mydialog
LOGICAL retlog
...
retlog = DLGINIT(IDD_mydlg, mydialog)
...
CALL DLGSETTITLE(mydialog, "New Title")
...
```

See Also

DLGINIT
DLGMODAL
DLGMODELESS

DLGUNINIT (W*S)

Dialog Subroutine: *Deallocates memory associated with an initialized dialog.*

Module

USE IFLOGM

Syntax

```
CALL DLGUNINIT (dlg)
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

You should call DLGUNINIT when a dialog that was successfully initialized by DLGINIT is no longer needed. DLGUNINIT should only be called on a dialog initialized with DLGINIT. If it is called on an uninitialized dialog or one that has already been deallocated with DLGUNINIT, the result is undefined.

Example

```
USE IFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG)  mydialog
LOGICAL        retlog
...
retlog = DLGINIT(IDD_mydlg, mydialog)
...
CALL DLGUNINIT (mydialog)
END
```

See Also

DLGINIT
DLGMODAL
DLGMODELESS
DLGEXIT

DNUM

Elemental Intrinsic Function (Specific): *Converts a character string to a REAL(8) value. This function cannot be passed as an actual argument.*

Syntax

```
result = DNUM (i)
```

i (Input) Must be of type character.

Results

The result type is REAL(8). The result value is the double-precision real value represented by the character string *i*.

Example

DNUM ("3.14159") has the value 3.14159 of type REAL(8).

The following sets x to 311.0:

```
CHARACTER(3) i
DOUBLE PRECISION x
i = "311"
x = DNUM(i)
```

DO Directive

OpenMP* Fortran Compiler Directive: *Specifies that the iterations of the immediately following DO loop must be executed in parallel.*

Syntax

```
!$OMP DO [clause[[,] clause] ... ]
   do_loop
[!$OMP END DO [NOWAIT]]
```

clause

Is one of the following:

- COLLAPSE (n)
- FIRSTPRIVATE (list)
- LASTPRIVATE ([CONDITIONAL:] list)
- LINEAR (var-list[: linear-step])
- ORDERED [(*n*)]

  Must be used if ordered sections are contained in the dynamic extent of the DO directive. For more information about ordered sections, see the ORDERED directive.

  If *n* is specified, it must be a positive scalar integer constant expression.
- PRIVATE (list)
- REDUCTION (reduction-identifier : list)
- SCHEDULE ([*modifier* [, *modifier*]:] *kind*[, *chunk_size*])

  Specifies how iterations of the DO loop are divided among the threads of the team. *chunk_size* must be a loop invariant positive scalar integer expression. The value of *chunk_size* must be the same for all threads in the team. The following *kind*s are permitted, only some of which allow the optional parameter *chunk_size*:

Kinds	Effect
STATIC	Divides iterations into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins. If *chunk_size* is specified, iterations are divided into pieces of a size specified by *chunk_size*. The pieces are statically dispatched to threads in the team in a round-robin fashion in the order of the thread number.
DYNAMIC	Can be used to get a set of iterations dynamically. It defaults to 1 unless *chunk_size* is specified. If *chunk_size* is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations.

Kinds	Effect
GUIDED	Can be used to specify a minimum number of iterations. It defaults to 1 unless *chunk_size* is specified. If *chunk_size* is specified, the chunk size is reduced exponentially with each succeeding dispatch. The *chunk_size* specifies the minimum number of iterations to dispatch each time. If there are less than *chunk_size* iterations remaining, the rest are dispatched.
AUTO[1]	Delegates the scheduling decision until compile time or run time. The schedule is processor dependent. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.
RUNTIME[1]	Defers the scheduling decision until run time. You can choose a schedule type and chunk size at run time by using the environment variable OMP_SCHEDULE.

[1]No *chunk_size* is permitted for this type.

At most one SCHEDULE clause can appear. If the SCHEDULE clause is not used, the default schedule type is STATIC.

modifier can be one of the following:

Modifier	Effect
MONOTONIC	Each thread executes the chunks that it is assigned in increasing logical iteration order.
NONMONOTONIC[1]	Chunks are assigned to threads in any order and the behavior of an application that depends on any execution order of the chunks is unspecified.

Modifier	Effect
SIMD	When *do_loop* is associated with an OMP SIMD construct, the *chunk_size* for all chunks except the first and last chunks is: *new_chunk_size* = (*chunk_size* / *simd_width*)**simd_width* where *simd_width* is an implementation-defined value. The first chunk will have at least *new_chunk_size* iterations unless it is also the last chunk. The last chunk may have fewer iterations than *new_chunk_size*. If SIMD is specified and the loop is not associated with an OMP SIMD construct, the modifier is ignored.

[1]NONMONOTONIC can only be specified with SCHEDULE(DYNAMIC) or SCHEDULE(GUIDED).

If the schedule kind is STATIC or if the ORDERED clause appears, and if MONOTONIC does not appear, the effect will be as if MONOTONIC was specified. NONMONOTONIC cannot be specified if the ORDERED clause appears. Either MONOTONIC or NONMONTONIC can appear but not both.

modifier cannot appear if the LINEAR clause appears.

The SIMD *modifier* can be used with MONOTONIC or NONMONOTONIC in either order. The SIMD *modifier* and the MONOTONIC *modifier* can be used with all *kind*s.

do_loop

Is a DO iteration (an iterative DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

You cannot branch out of a DO loop associated with a DO directive.

The binding thread set for a DO construct is the current team. A DO loop region binds to the innermost enclosing parallel region.

If used, the END DO directive must appear immediately after the end of the loop. If you do not specify an END DO directive, an END DO directive is assumed at the end of the DO loop.

If you specify NOWAIT in the END DO directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight to the instruction following the loop without waiting for the other members of the team to finish the DO directive.

Parallel DO loop control variables are block-level entities within the DO loop. If the loop control variable also appears in the LASTPRIVATE list of the parallel DO, it is copied out to a variable of the same name in the enclosing PARALLEL region. The variable in the enclosing PARALLEL region must be SHARED if it is specified in the LASTPRIVATE list of a DO directive.

Only a single SCHEDULE, COLLAPSE, or ORDERED clause can appear in a DO directive.

ORDERED (*n*) specifies how many loops are associated with the DO directive and it specifies that those associated loops form a doacross loop nest. *n* does not affect how the logical iteration space is divided.

If you specify COLLAPSE (M) ORDERED (N) for loops nested K deep, the following rules apply:

- If either M > K or N > K, the behavior is unspecified.
- N must be greater than M

A LINEAR clause or an ORDERED (*n*) clause can be specified on a DO directive but not both.

DO directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

In the following example, the loop iteration variable is private by default, and it is not necessary to explicitly declare it. The END DO directive is optional:

```
!$OMP PARALLEL
!$OMP DO
      DO I=1,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

If there are multiple independent loops within a parallel region, you can use the NOWAIT option to avoid the implied BARRIER at the end of the DO directive, as follows:

```
!$OMP PARALLEL
!$OMP DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO I=1,M
        Y(I) = SQRT(Z(I))
      END DO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially, as follows:

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
      DO I=1,N
        A(I) = B(I) + C(I)
      END DO
!$OMP END PARALLEL
      CALL REVERSE(I)
```

In this case, the value of I at the end of the parallel region equals N+1, as in the sequential case.

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
  !$OMP DO ORDERED SCHEDULE(DYNAMIC)
        DO I=LB,UB,ST
          CALL WORK(I)
        END DO
        ...
        SUBROUTINE WORK(K)
  !$OMP ORDERED
        WRITE(*,*) K
  !$OMP END ORDERED
```

In the next example, the loops over J1 and J2 are collapsed and their iteration space is executed by all threads of the current team:

```
!$OMP DO COLLAPSE(2) PRIVATE(J1, J2, J3)
    DO J1 = J1_L, J1_U, J1_S
        DO J2 = J2_L, J2_U, J2_S
            DO J3 = J3_L, J3_U, J3_S
                CALL BAR(A, J1, J2, J3)
            ENDDO
        ENDDO
    ENDDO
!$OMP END DO
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Parallel Processing Model for information about Binding Sets

DO Statement

Statement: *Marks the beginning of a DO construct. The DO construct controls the repeated execution of a block of statements or constructs. This repeated execution is called a loop.*

Syntax

A DO construct takes one of the following forms:

Block Form:

```
[name:] DO [label[, ] ] [loop-control]
   block
[label] term-stmt
```

Non-block Form:

```
DO label [,] [loop-control]
   block
[label] ex-term-stmt
```

name	(Optional) Is the name of the DO construct.
label	(Optional) Is a statement label identifying the terminal statement.

loop-control	Is one of the following: • a loop iteration (see Iteration Loop Control) • WHILE (see the DO WHILE statement) • CONCURRENT (see the DO CONCURRENT statement)
block	Is a sequence of zero or more statements or constructs that make up the DO range.
term-stmt	Is the terminal statement for the block form of the construct.
ex-term-stmt	Is the terminal statement for the non-block form of the construct.

Description

The terminal statement (*term-stmt*) for a block DO construct is an END DO or CONTINUE statement. If the block DO statement contains a label, the terminal statement must be identified with the same label. If no label appears, the terminal statement must be an END DO statement.

If a construct name is specified in a block DO statement, the same name must appear in the terminal END DO statement. If no construct name is specified in the block DO statement, no name can appear in the terminal END DO statement.

The terminal statement (*ex-term-stmt*) for a non-block DO construct is an executable statement (or construct) that is identified by the label specified in the non-block DO statement. A non-block DO construct can share a terminal statement with another non-block DO construct. A block DO construct cannot share a terminal statement.

The following cannot be terminal statements for non-block DO constructs:

- CONTINUE (allowed if it is a shared terminal statement)
- CYCLE
- END (for a program or subprogram)
- EXIT
- GO TO (unconditional or assigned)
- Arithmetic IF
- RETURN
- STOP

The non-block DO construct is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.

The labeled form of a DO loop is an obsolescent feature in the Fortran Standard.

Example

The following example shows a simple block DO construct (contains no iteration count or DO WHILE statement):

```
DO
  READ *, N
  IF (N == 0) STOP
  CALL SUBN
END DO
```

The DO block executes repeatedly until the value of zero is read. Then the DO construct terminates.

The following example shows a named block DO construct:

```
LOOP_1: DO I = 1, N
          A(I) = C * B(I)
        END DO LOOP_1
```

The following example shows a nonblock DO construct with a shared terminal statement:

```
    DO 20 I = 1, N
    DO 20 J = 1 + I, N
20 RESULT(I,J) = 1.0 / REAL(I + J)
```

The following two program fragments are also examples of DO statements:

```
C   Initialize the even elements of a 20-element real array
    DIMENSION array(20)
    DO j = 2, 20, 2
      array(j) = 12.0
    END DO
C
C   Perform a function 11 times
    DO k = -30, -60, -3
      int = j / 3
      isb = -9 - k
      array(isb) = MyFunc (int)
    END DO
```

The following shows the final value of a DO variable (in this case 11):

```
   DO j = 1, 10
     WRITE (*, '(I5)') j
   END DO
   WRITE (*, '(I5)') j
```

See Also

CONTINUE
CYCLE
EXIT
DO WHILE
DO CONCURRENT
Execution Control

DO CONCURRENT

Statement: *Specifies that there are no data dependencies between the iterations of a DO loop.*

Syntax

The DO CONCURRENT statement takes the following form:

```
[name:] DO [label [,]] CONCURRENT concurrent-header [locality-spec]
   block
[END DO [name]]
```

name	(Optional) Is the name of the DO CONCURRENT construct.
concurrent-header	Is ([*type* ::] *concurrent-spec* [, *mask-expr*])
type	(Optional) Is an integer data type.
concurrent-spec	Is an assignment using a triplet specification in the form *index-name* = *concurrent-limit* : *concurrent-limit* [: *concurrent-step*]

index-name — Is a named scalar variable of type integer. It becomes defined when the index-name value set is evaluated. It has the scope of the construct.

concurrent-limit — Is a scalar integer expression.

concurrent-step — (Optional) Is a scalar integer expression.

mask-expr — (Optional) Is a masked expression that is scalar and of type logical.

Any procedure referenced in *mask-expr* must be pure, including one referenced by a defined operator. *index-name* can appear in *mask-expr*. The set of index values to be executed is the set of all *index-name* values for which *mask-expr* is true.

label — (Optional) Is a label specifying an executable statement in the same program unit.

locality-spec — (Optional) Can be any of the following:

- LOCAL (*variable-name-list*)
- LOCAL_INIT (*variable-name-list*)
- SHARED (*variable-name-list*)
- DEFAULT (NONE)

  You can specify DEFAULT (NONE) only once in a DO CONCURRENT statement. If specified, any variable or construct entity that is accessible in the scope containing the DO CONCURRENT statement that appears in the block of the DO CONCURRENT construct must have its locality explicitly specified.

You can specify LOCAL, LOCAL_INIT, SHARED, and DEFAULT (NONE) in the same DO CONCURRENT statement.

You can specify more than one of the following in the same DO CONCURRENT statement: LOCAL, LOCAL_INIT, and SHARED.

The following are rules for *variable-name* in a *locality-spec*:

- A *variable-name* must be the name of a variable that is accessible in the innermost construct containing the DO CONCURRENT statement. *variable-name* can appear at most once in any *locality-spec* of a DO CONCURRENT statement. It cannot be the same as *index-name* of the same DO CONCURRENT statement.
- A *variable-name* in a LOCAL or LOCAL_INIT *locality-spec* cannot have the OPTIONAL, ALLOCATABLE, or INTENT(IN) attribute, it cannot be a non-pointer polymorphic dummy argument, a coarray or an assumed-size array, or be of a type that is finalizable.
- *variable-name* is not permitted in a LOCAL or LOCAL_INIT *locality-spec* if it is not permitted in a variable-definition context.

block — Is a sequence of zero or more statements or constructs that make up the DO range.

A variable that appears in a *mask-expr*, *concurrent-step*, or *concurrent-limit* of a *concurrent-header*, cannot appear in a LOCAL *locality-spec* in the same DO CONCURRENT statement.

If a construct name is specified in a DO CONCURRENT statement, the same name must appear in a terminal END DO statement. If no construct name is specified in the DO CONCURRENT statement, no name can appear in the terminal END DO statement, if one is specified.

See the DO statement for the semantics of labeled and block forms of DO loops.

The DO CONCURRENT range is executed for every active combination of the *index-name* values.

Each execution of the range is an iteration. The executions can occur in any order.

If END DO is specified, it terminates the construct. If END DO is not specified, when all of the iterations have completed execution, the loop terminates, and the DO construct becomes inactive. You can branch to the END DO statement only from within the construct.

When the DO CONCURRENT construct terminates, a variable that is defined or becomes undefined during more than one iteration of the construct becomes undefined.

Execution of a CYCLE statement that belongs to a DO CONCURRENT construct completes execution of that iteration of the construct.

A branch within a DO CONCURRENT construct must not have a branch target that is outside the construct.

If *type* appears, the *index-name* has the specified type and type parameters. Otherwise, it has the type and type parameters that it would have if it were the name of a variable in the innermost executable construct or scoping unit.

If *type* does not appear, the *index-name* must not be the same as a local identifier, an accessible global identifier, or an identifier of an outer construct entity, except for a common block name or a scalar variable name.

The *index-name* of a contained FORALL or DO CONCURRENT construct must not be the same as an *index-name* of any of its containing FORALL or DO CONCURRENT constructs.

The following cannot appear in a DO CONCURRENT construct:

- A RETURN statement
- An image control statement
- A branch to a target outside the construct block
- A statement that may result in the deallocation of a polymorphic variable
- An input/output statement with an ADVANCE= specifier
- A reference to a nonpure procedure
- A reference to module IEEE_EXCEPTIONS procedure IEEE_GET_FLAG, IEEE_SET_HALTING_MODE, or IEEE_GET_HALTING_MODE

An EXIT statement must not appear within a DO CONCURRENT construct if it belongs to that construct or an outer construct.

A construct or statement entity with the SAVE attribute and with unspecified locality in a DO CONCURRENT construct has SHARED locality. If it does not have the SAVE attribute, it is a different entity in each iteration of the construct.

A variable with LOCAL or LOCAL_INIT locality is a construct entity with the same type, type parameters, and rank as variable with the same name in the innermost construct or scope containing the DO CONCURRENT construct. The variable outside the construct is inaccessible by that name inside the DO CONCURRENT construct. The construct entity does not have the BIND, SAVE, VALUE, PROTECTED, or INTENT attribute, even if the variable with the same name outside the construct has the attribute. The construct entity does have the VOLATILE, CONTIGUOUS, POINTER, TARGET or ASYCHRONOUS attribute if the variable outside the construct with the same name has the attribute. If it is a non-pointer, it has the same bounds as the variable outside the construct.

At the beginning of each iteration, a variable with LOCAL locality that is a pointer has pointer association status of undefined; otherwise, it is undefined except for any subobjects that are default initialized. A variable with LOCAL_INIT locality has the definition status and pointer association status of the variable outside the construct. The variable outside the construct cannot be an undefined nonallocatable nonpointer variable, or an undefined pointer.

A pointer that becomes associated with a LOCAL or LOCAL_INIT TARGET variable becomes undefined at the end of the iteration. If a LOCAL or LOCAL_INIT variable appears in an input/output statement, the input/output operation must complete before the iteration completes.

References in a DO CONCURRENT construct to a SHARED variable are references to the variable in the innermost construct or scope containing the DO CONCURRENT construct. If the variable is defined or becomes undefined in one iteration, it cannot be referenced, defined, or become undefined in another iteration. If it becomes pointer assigned, allocated, deallocated, or nullified in an iteration, its dynamic type, allocation or allocations status, shape, bounds, or a deferred-type parameter cannot be inquired about in another iteration. A SHARED noncontiguous array cannot be an actual argument associated with a contiguous INTENT(INOUT) dummy argument.

The following are rules for variables with unspecified locality in DO CONCURRENT constructs:

- A variable that is referenced in an iteration must be previously defined during that iteration, or it must not be defined or become undefined during any other iteration.

  A variable that is defined or becomes undefined by more than one iteration becomes undefined when the loop terminates.
- An allocatable object that is allocated in more than one iteration must be subsequently deallocated during the same iteration in which it was allocated.

  An object that is allocated or deallocated in only one iteration must not be referenced, allocated, deallocated, defined, or become undefined in a different iteration.
- A pointer that is referenced in an iteration must have been pointer associated previously during that iteration, or it must not have its pointer association changed during any iteration.

  A pointer that has its pointer association changed in more than one iteration has an association status of undefined when the construct terminates.
- An input/output statement must not write data to a file record or position in one iteration and read from the same record or position in a different iteration.
- Records written by output statements in the range of the loop to a sequential-access file appear in the file in an indeterminate order.

The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct also apply to any procedure invoked within the loop.

These restrictions ensure no interdependencies occur that might affect code optimizations.

Note that if compiler option `-qopenmp` (Linux* and macOS*) or `/Qopenmp` (Windows*) is specified, the compiler will attempt to parallelize the construct.

The labeled form of a DO CONCURRENT loop is an obsolescent feature in the Fortran Standard.

Example

The following shows a DO CONCURRENT construct with a *mask-expr* and locality specified for variables:

```
INTEGER,DIMENSION(N) :: J, K
INTEGER              :: I, M
M = 10
I = 15
DO CONCURRENT (I = 1:N, J(I)> 0)LOCAL (M) SHARED (J, K)
   M =  MOD (K(I), J(I))
   K(I) = K(I) - M
END DO
PRINT *, I, M             ! Prints 15 10
```

DO SIMD

OpenMP* Fortran Compiler Directive: *Specifies that the iterations of the loop will be distributed across threads in the team. Iterations executed by each thread can also be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP DO SIMD [clause[[,] clause] ... ]
      do-loop
[!$OMP END DO SIMD[NOWAIT]]
```

clause
: Can be any of the clauses accepted by the DO or SIMD directives.

do-loop
: Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

  All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

  The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

  You cannot branch out of a DO loop associated with a DO SIMD directive.

If the END DO SIMD directive is not specified, an END DO SIMD directive is assumed at the end of *do-loop*.

You can specify the NOWAIT clause to avoid the implied barrier at the end of a loop construct.

The DO SIMD construct converts the associated DO loop to a SIMD loop in a way that is consistent with any clauses that apply to the SIMD construct. The resulting SIMD chunks and any remaining iterations will be distributed across the implicit tasks of the parallel region in a way that is consistent with any clauses that apply to the DO construct.

This directive specifies a composite construct.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

DO WHILE

Statement: *Executes the range of a DO construct while a specified condition remains true.*

Syntax

```
[name:] DO [label[, ]] WHILE (expr)
   block
[END DO [name]]
```

name	(Optional) Is the name of the DO WHILE construct.
label	(Optional) Is a label specifying an executable statement in the same program unit.
expr	Is a scalar logical (test) expression enclosed in parentheses.
block	Is a sequence of zero or more statements or constructs that make up the DO range.

Description

If a construct name is specified in a DO WHILE statement, the same name must appear in a terminal END DO statement. If no construct name is specified in the DO WHILE statement, no name can appear in the terminal END DO statement, if one is specified.

Before each execution of the DO range, the logical expression is evaluated. If it is true, the statements in the body of the loop are executed. If it is false, the DO construct terminates and control transfers to the statement following the loop.

If END DO is specified, it terminates the construct. If END DO is not specified, when all of the iterations have completed execution, the loop terminates, and the DO construct becomes inactive.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement. See the description of the DO statement for the semantics of labeled and block forms of DO loops.

You can transfer control out of a DO WHILE loop but not into a loop from elsewhere in the program.

Terminating a DO WHILE loop with an executable statement other than a DO WHILE or a CONTINUE statement is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.

The labeled form of a DO WHILE loop is an obsolescent feature in the Fortran Standard.

Example

The following example shows a DO WHILE statement:

```
CHARACTER*132 LINE
...
I = 1
DO WHILE (LINE(I:I) .EQ. ' ')
  I = I + 1
END DO
```

The following examples show required and optional END DO statements:

```
RequiredOptional
DO WHILE (I .GT. J)          DO 10 WHILE (I .GT. J)
  ARRAY(I,J) = 1.0             ARRAY(I,J) = 1.0
  I = I - 1                    I = I - 1
END DO                    10 END DO
```

The following shows another example:

```
CHARACTER(1) input
input = ' '
DO WHILE ((input .NE. 'n') .AND. (input .NE. 'y'))
   WRITE (*, '(A)') 'Enter y or n: '
   READ (*, '(A)') input
END DO
```

See Also
CONTINUE
CYCLE
EXIT
DO statement
Execution Control

DOT_PRODUCT
Transformational Intrinsic Function (Generic): *Performs dot-product multiplication of numeric or logical vectors (rank-one arrays).*

Syntax
```
result = DOT_PRODUCT (vector_a,vector_b)
```

vector_a	(Input) Must be a rank-one array of numeric (integer, real, or complex) or logical type.
vector_b	(Input) Must be a rank-one array of numeric type if *vector_a* is of numeric type, or of logical type if *vector_a* is of logical type. It must be the same size as *vector_a*.

Results
The result is a scalar whose type depends on the types of *vector_a* and *vector_b*.

If *vector_a* is of type integer or real, the result value is SUM (*vector_a** *vector_b*).

If *vector_a* is of type complex, the result value is SUM (CONJG (*vector_a*)* *vector_b*).

If *vector_a* is of type logical, the result has the value ANY (*vector_a*.AND. *vector_b*).

If either rank-one array has size zero, the result is zero if the array is of numeric type, and false if the array is of logical type.

Example
DOT_PRODUCT ((/1, 2, 3/), (/3, 4, 5/)) has the value 26, calculated as follows:

```
((1 x 3) + (2 x 4) + (3 x 5)) = 26
```

DOT_PRODUCT ((/ (1.0, 2.0), (2.0, 3.0) /), (/ (1.0, 1.0), (1.0, 4.0) /)) has the value (17.0, 4.0).

DOT_PRODUCT ((/ .TRUE., .FALSE. /), (/ .FALSE., .TRUE. /)) has the value false.

The following shows another example:

```
  I = DOT_PRODUCT((/1,2,3/), (/4,5,6/))  ! returns the value 32
```

See Also
PRODUCT
MATMUL
TRANSPOSE

DOUBLE COMPLEX
Statement: *Specifies the DOUBLE COMPLEX data type.*

Syntax

```
DOUBLE COMPLEX
```

A COMPLEX(8) or DOUBLE COMPLEX constant is a pair of constants that represents a complex number. One of the pair must be a double-precision real constant, the other can be an integer, single-precision real, or double-precision real constant.

A COMPLEX(8) or DOUBLE COMPLEX constant occupies 16 bytes of memory and is interpreted as a complex number.

The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(KIND=8) or DOUBLE COMPLEX constants. (For more information, see REAL and DOUBLE PRECISION.)

The DOUBLE PRECISION constants in a COMPLEX(8) or DOUBLE COMPLEX constant have IEEE* binary64 format.

Example

```
DOUBLE COMPLEX vector, arrays(7,29)
DOUBLE COMPLEX pi, pi2 /3.141592654,6.283185308/
```

Valid COMPLEX(8) or DOUBLE COMPLEX constants

(547.3E0_8,-1.44_8)
(1.7039E0,-1.7039D0)
(+12739D3,0.D0)

Invalid COMPLEX(8) or DOUBLE COMPLEX constants

(1.23D0,)	Second constant missing.
(1D1,2H12)	Hollerith constants not allowed.
(1,1.2)	Neither constant is DOUBLE PRECISION; this is a valid single-precision real constant.

See Also

General Rules for Complex Constants
COMPLEX Statement
Complex Data Types
DOUBLE PRECISION
REAL

DOUBLE PRECISION

Statement: *Specifies the DOUBLE PRECISION data type.*

Syntax

```
DOUBLE PRECISION
```

A REAL(8) or DOUBLE PRECISION constant has more than twice the accuracy of a REAL(4) number, and greater range.

A REAL(8) or DOUBLE PRECISION constant occupies eight bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 15 digits are significant.

IEEE* binary64 format is used.

For more information, see General Rules for Real Constants.

Example

```
DOUBLE PRECISION varnam
DOUBLE PRECISION,PRIVATE :: zz
```

Valid REAL(8) or DOUBLE PRECISION constants
123456789D+5
123456789E+5_8
+2.7843D00
-.522D-12
2E200_8
2.3_8
3.4E7_8

Invalid REAL(8) or DOUBLE PRECISION constants	
-.25D0_2	2 is not a valid kind type for reals.
+2.7182812846182	No D exponent designator is present; this is a valid single-precision constant.
123456789.D400	Too large for any double-precision format.
123456789.D-400	Too small for any double-precision format.

See Also

REAL Statement
REAL(8) or DOUBLE PRECISION Constants
Data Types, Constants, and Variables
Real Data Types

DPROD

Elemental Intrinsic Function (Specific): *Produces a higher precision product. This is a specific function that has no generic function associated with it.*

Syntax

```
result = DPROD (x, y)
```

x	(Input) Must be of type REAL(4) or REAL(8).
y	(Input) Must have the same type and kind parameters as *x*.

Results

If *x* and *y* are of type REAL(4), the result type is double-precision real (REAL(8) or REAL*8). If *x* and *y* are of type REAL(8), the result type is REAL(16). The result value is equal to *x** *y*.

The setting of compiler options specifying real size can affect this function.

Example

DPROD (2.0, -4.0) has the value -8.00D0.

DPROD (5.0D0, 3.0D0) has the value 15.00Q0.

The following shows another example:

```
REAL(4) e
REAL(8) d
e = 123456.7
d = 123456.7D0
! DPROD (e,e) returns 15241557546.4944

! DPROD (d,d) returns 15241556774.8899992813874268904328
```

DRAND, DRANDM

Portability Functions: *Return double-precision random numbers in the range 0.0 to 1.0, not including the end points.*

Module

USE IFPORT

Syntax

`result = DRAND (iflag)`

`result = DRANDM (iflag)`

iflag (Input) INTEGER(4). Controls the way the random number is selected.

Results

The result type is REAL(8). DRAND and DRANDM return random numbers in the range 0.0 to 1.0, not including the end points.

Value of *iflag*	Selection process
1	The generator is restarted and the first random value is selected.
0	The next random number in the sequence is selected.
Otherwise	The generator is reseeded using *iflag*, then restarted, and the first random value is selected.

There is no difference between DRAND and DRANDM. Both functions are included to insure portability of existing code that references one or both of them.

The intrinsic functions RANDOM_NUMBER and RANDOM_SEED provide the same functionality and they are the recommended functions to use when writing programs to generate random numbers.

Example

```
USE IFPORT
REAL(8) num
INTEGER(4) f
f=1
CALL print_rand
```

```
f=0
CALL print_rand
f=22
CALL print_rand
CONTAINS
  SUBROUTINE print_rand
    num = drand(f)
    print *, 'f= ',f,':',num
  END SUBROUTINE
END
```

See Also
RANDOM_NUMBER
RANDOM_SEED

DRANSET
Portability Subroutine: *Sets the seed for the random number generator.*

Module
USE IFPORT

Syntax
`CALL DRANSET (seed)`

seed	(Input) REAL(8). The reset value for the seed.

See Also
RANGET

DREAL
Elemental Intrinsic Function (Specific): *Converts the real part of a double complex argument to double-precision type. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.*

Syntax
`result = DREAL (a)`

a	(Input) Must be of type double complex (COMPLEX(8) or COMPLEX*16).

Results
The result type is double precision real (REAL(8) or REAL*8).

Example
DREAL ((2.0d0, 3.0d0)) has the value 2.0d0.

See Also
REAL

DSHIFTL

Elemental Intrinsic Function (Specific): *Selects the left 64 bits after shifting a 128-bit integer value to the left. This function cannot be passed as an actual argument.*

Syntax

```
result = DSHIFTL (i,j,shift)
```

i	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.
j	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant. If both *i* and *j* are of type integer, they must have the same kind type parameter. *i* and *j* must not both be binary, octal, or hexadecimal literal constants.
shift	(Input) Integer. Must be nonnegative and less than or equal to 64. This is the shift count.

Results

The result type is integer. The result value is the 64-bit value starting at bit 128 - *shift* of the 128-bit concatenation of the values of *i* and *j*.

If either *i* or *j* is a binary, octal, or hexadecimal literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The rightmost *shift* bits of the result value are the same as the leftmost bits of *i*, and the remaining bits of the result value are the same as the rightmost bits of *j*. This is equal to IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT SIZE (J) - SHIFT)).

Example

Consider the following:

```
INTEGER(8) ILEFT / Z'111122221111222' /
INTEGER(8) IRIGHT / Z'FFFFFFFFFFFFF' /
PRINT *, DSHIFTL (ILEFT, IRIGHT, 16_8)  ! prints 1306643199093243919
```

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
Model for Bit Data

DSHIFTR

Elemental Intrinsic Function (Specific): *Selects the left 64 bits after shifting a 128-bit integer value to the right. This function cannot be passed as an actual argument.*

Syntax

```
result = DSHIFTR (i,j,shift)
```

i	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.

j	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.
shift	(Input) Integer. Must be nonnegative and less than or equal to 64. This is the shift count.

Results

The result type is integer. The result value is the 64-bit value starting at bit 64 + *shift* of the 128-bit concatenation of the values of *i* and *j*.

If either *i* or *j* is a binary, octal, or hexadecimal literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other. The leftmost *shift* bits of the result value are the same as the rightmost bits of *i*, and the remaining bits of the result value are the same as the leftmost bits of *j*. This is equal to IOR (SHIFTL (I, BIT SIZE (I) - SHIFT), SHIFTR (J, SHIFT)).

Example

Consider the following:

```
INTEGER(8) ILEFT / Z'111122221111222' /
INTEGER(8) IRIGHT / Z'FFFFFFFFFFFFF' /
PRINT *, DSHIFTR (ILEFT, IRIGHT, 16_8)  ! prints 1306606910610341887
```

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
Model for Bit Data

DTIME

Portability Function: *Returns the elapsed CPU time since the start of program execution when first called, and the elapsed execution time since the last call to DTIME thereafter.*

Module

USE IFPORT

Syntax

`result = DTIME (tarray)`

tarray	(Output) REAL(4). A rank one array with two elements: • *tarray*(1) - Elapsed user time, which is time spent executing user code. This value includes time running protected Windows subsystem code. • *tarray*(2) - Elapsed system time, which is time spent executing privileged code (code in the Windows Executive).

Results

The result type is REAL(4). The result is the total CPU time, which is the sum of *tarray*(1) and *tarray*(2). If an error occurs, -1 is returned.

Example

```
USE IFPORT
REAL(4) I, TA(2)
I = DTIME(TA)
```

```
 write(*,*) 'Program has been running for', I, 'seconds.'
 write(*,*) ' This includes', TA(1), 'seconds of user time and', &
& TA(2), 'seconds of system time.'
```

See Also

DATE_AND_TIME
CPU_TIME

E to F

E to F

ELEMENTAL

Keyword: *Asserts that a user-defined procedure is defined on scalar arguments that may be called with array arguments. An elemental procedure may be a pure procedure or an impure procedure. An elemental procedure is an elemental intrinsic procedure, an intrinsic module procedure that is specified to be elemental, a user-defined procedure that is specified to be elemental, or a type-bound procedure that is bound to an elemental procedure. A procedure pointer or a dummy procedure can not be specified to be elemental.*

Description

To specify an elemental procedure, use this keyword in a FUNCTION or SUBROUTINE statement.

An explicit interface must be visible to the caller of an ELEMENTAL procedure.

An elemental procedure can be passed an array, which is acted upon one element at a time.

For functions, the result must be scalar; it cannot have the POINTER or ALLOCATABLE attribute.

Dummy arguments in ELEMENTAL procedures may appear in specification expressions in the procedure.

Dummy arguments have the following restrictions:

- They must be scalar.
- They cannot have the POINTER or ALLOCATABLE attribute.
- They cannot be an alternate return specifier (*).
- They cannot be dummy procedures.

If the actual arguments are all scalar, the result is scalar. If the actual arguments are array valued, the values of the elements (if any) of the result are the same as if the function or subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

Elemental procedures are pure procedures and all rules that apply to pure procedures also apply to elemental procedures, unless you specify that the elemental procedure is IMPURE. In that case, the rules for pure procedures do not apply.

Example

Consider the following:

```
 MIN (A, 0, B)             ! A and B are arrays of shape (S, T)
```

In this case, the elemental reference to the MIN intrinsic function is an array expression whose elements have the following values:

```
MIN (A(I,J), 0, B(I,J)), I = 1, 2, ..., S, J = 1, 2, ..., T
```

Consider the following example:

```
ELEMENTAL REAL FUNCTION F (A, B, ORDER)
REAL, INTENT (IN)       :: A, B
INTEGER, INTENT (IN)    :: ORDER
REAL                    :: TEMP (ORDER)
```

In the above, the size of TEMP depends on the specification expression that is the value of the ORDER dummy argument.

See Also

FUNCTION
SUBROUTINE
Determining When Procedures Require Explicit Interfaces
Optional Arguments

ELLIPSE, ELLIPSE_W (W*S)

Graphics Functions: *Draw a circle or an ellipse using the current graphics color.*

Module

USE IFQWIN

Syntax

`result = ELLIPSE (control,x1,y1,x2,y2)`

`result = ELLIPSE_W (control,wx1,wy1,wx2,wy2)`

control	(Input) INTEGER(2). Fill flag. Can be one of the following symbolic constants: • $GFILLINTERIOR - Fills the figure using the current color and fill mask. • $GBORDER - Does not fill the figure.
x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the ellipse is clipped or partially out of bounds, the ellipse is considered successfully drawn, and the return is 1. If the ellipse is drawn completely out of bounds, the return is 0.

The border is drawn in the current color and line style.

When you use ELLIPSE, the center of the ellipse is the center of the bounding rectangle defined by the viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). When you use ELLIPSE_W, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*). If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

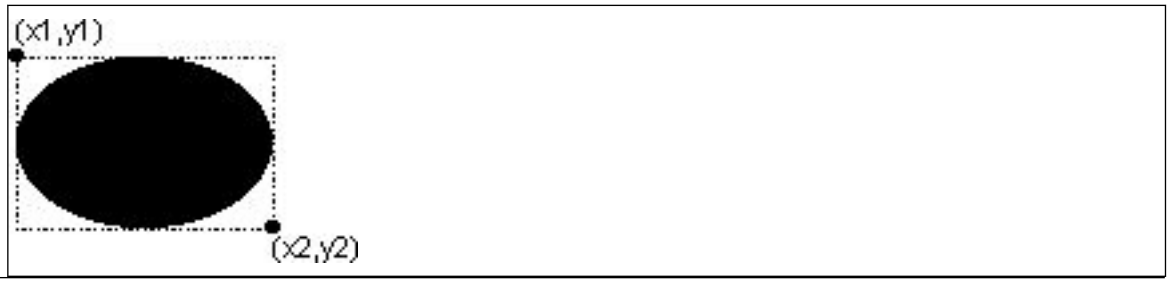
The control option given by $GFILLINTERIOR is equivalent to a subsequent call to the FLOODFILLRGB function using the center of the ellipse as the start point and the current color (set by SETCOLORRGB) as the boundary color.

NOTE
The ELLIPSE routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Ellipse routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Ellipse.

Example

The following program draws the shape shown below.

```
! compile as QuickWin or Standard Graphics application
 USE IFQWIN
 INTEGER(2) dummy, x1, y1, x2, y2
 x1 = 80;  y1 = 50
 x2 = 240; y2 = 150
 dummy = ELLIPSE( $GFILLINTERIOR, x1, y1, x2, y2 )
 END
```



See Also

ARC
FLOODFILLRGB
GRSTATUS
LINETO
PIE
POLYGON
RECTANGLE
SETCOLORRGB
SETFILLMASK

ELSE Directive

Statement: *Marks an optional branch in an IF Directive Construct.*

See Also

See IF Directive Construct.

ELSE Statement

Statement: *Marks an optional branch in an IF Construct.*

See Also

See IF Construct.

ELSEIF Directive

Statement: *Marks an optional branch in an IF Directive Construct.*

See Also

See IF Directive Construct.

ELSE IF

Statement: *Marks an optional branch in an IF Construct.*

See Also

See IF Construct.

ELSE WHERE

Statement: *Marks the beginning of an ELSE WHERE block within a WHERE construct.*

Syntax

```
[name:]WHERE (mask-expr1)
   [where-body-stmt]...
[ELSE WHERE(mask-expr2) [name]
   [where-body-stmt]...]
[ELSE WHERE[name]
   [where-body-stmt]...]
END WHERE [name]
```

name
: Is the name of the WHERE construct.

mask-expr1, mask-expr2
: Are logical array expressions (called mask expressions).

where-body-stmt
: Is one of the following:

- An assignment statement of the form: array variable = array expression.

  The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental.
- A WHERE statement or construct

Description

Every assignment statement following the ELSE WHERE is executed as if it were a WHERE statement with ".NOT. *mask-expr1*". If ELSE WHERE specifies "*mask-expr2*", it is executed as "(.NOT. *mask-expr1*) .AND. *mask-expr2*" during the processing of the ELSE WHERE statement.

Example

```
WHERE (pressure <= 1.0)
  pressure = pressure + inc_pressure
  temp = temp - 5.0
ELSEWHERE
  raining = .TRUE.
END WHERE
```

The variables `temp`, `pressure`, and `raining` are all arrays.

See Also

WHERE

ENCODE

Statement: *Translates data from internal (binary) form to character form. It is comparable to using internal files in formatted sequential WRITE statements.*

Syntax

`ENCODE (c,f,b[, IOSTAT=i-var] [, ERR=label]) [io-list]`

c	Is a scalar integer expression. It is the number of characters to be translated to internal form.
f	Is a format identifier. An error occurs if more than one record is specified.
b	Is a scalar or array reference. If *b* is an array reference, its elements are processed in the order of subscript progression. *b* contains the characters to be translated to internal form.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and as zero if no error occurs (see I/O Status Specifier).
label	Is the label of an executable statement that receives control if an error occurs.
io-list	Is an I/O list. An I/O list is either an implied-DO list or a simple list of variables (except for assumed-size arrays). The list contains the data to be translated to character form. The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

The number of characters that the ENCODE statement can translate depends on the data type of *b*. For example, an INTEGER(2) array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array.

The maximum number of characters a character variable or character array element can contain is the length of the character variable or character array element.

The maximum number of characters a character array can contain is the length of each element multiplied by the number of elements.

Example

Consider the following:

```
     DIMENSION K(3)
     CHARACTER*12 A,B
     DATA A/'123456789012'/
     DECODE(12,100,A) K
100  FORMAT(3I4)
     ENCODE(12,100,B) K(3), K(2), K(1)
```

The DECODE statement stores the 12 characters into array K:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

The ENCODE statement translates the values K(3), K(2), and K(1) to character form and stores the characters in the character variable B.:

```
B = '901256781234'
```

See Also

READ
WRITE
DECODE

END

Statement: *Marks the end of a program unit. It takes one of the following forms:*

Syntax

`END [PROGRAM [`*`program-name`*`]]`

`END [FUNCTION [`*`function-name`*`]]`

`END [SUBROUTINE [`*`subroutine-name`*`]]`

`END [MODULE [`*`module-name`*`]]`

`END [SUBMODULE [`*`module-name`*`]]`

`END [BLOCK DATA [`*`block-data-name`*`]]`

In main programs, function subprograms, and subroutine subprograms, END statements are executable and can be branch target statements. If control reaches the END statement in these program units, the following occurs:

- In a main program, execution of the END statement initiates normal termination of the image that executes it.
- In a function or subroutine subprogram, a RETURN statement is implicitly executed.

The END statement cannot be continued in a program unit, and no other statement in the program unit can have an initial line that appears to be the program unit END statement.

The END statements in a module or block data program unit are nonexecutable.

Example

```
C   An END statement must be the last statement in a program
C   unit:
    PROGRAM MyProg
    WRITE (*, '("Hello, world!")')
    END
C
C   An example of a named subroutine
C
    SUBROUTINE EXT1 (X,Y,Z)
      Real, Dimension (100,100) :: X, Y, Z
    END SUBROUTINE EXT1
```

See Also

Program Units and Procedures
Branch Statements
Program Termination

END DO

Statement: *Marks the end of a DO or DO WHILE loop.*

Syntax

```
END DO
```

Description

There must be a matching END DO statement for every DO or DO WHILE statement that does not contain a label reference.

An END DO statement can terminate only one DO or DO WHILE statement. If you name the DO or DO WHILE statement, the END DO statement can specify the same name.

Example

The following examples both produce the same output:

```
  DO ivar = 1, 10
    PRINT ivar
  END DO
  ivar = 0
do2: DO WHILE (ivar .LT. 10)
        ivar = ivar + 1
        PRINT ivar
     END DO do2
```

See Also

DO
DO WHILE
CONTINUE

ENDIF Directive

Statement: *Marks the end of an IF Directive Construct.*

See Also
See IF Directive Construct.

END IF
Statement: *Marks the end of an IF Construct.*

See Also
See IF Construct.

ENDFILE
Statement: *For sequential files, writes an end-of-file record to the file and positions the file after this record (the terminal point). For direct access files, truncates the file after the current record.*

Syntax
It can have either of the following forms:

```
ENDFILE ([UNIT=] io-unit[, ERR=label] [, IOMSG=msg-var] [, IOSTAT=i-var])

ENDFILE io-unit
```

io-unit	(Input) Is an external unit specifier.
label	Is the label of the branch target statement that receives control if an error occurs.
msg-var	(Output) Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs.
i-var	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description
If the unit specified in the ENDFILE statement is not open, the default file is opened for unformatted output.

An end-of-file record can be written only to files with sequential organization that are accessed as formatted-sequential or unformatted-segmented sequential files. An ENDFILE performed on a direct access file always truncates the file.

End-of-file records should not be written in files that are read by programs written in a language other than Fortran.

> **NOTE**
> If you use compiler option vms and an ENDFILE is performed on a sequential unit, an actual one byte record containing a CTRL+Zis written to the file. If this option is not specified, an internal ENDFILE flag is set and the file is truncated. The option does not affect ENDFILE on relative files; such files are truncated.

If a parameter of the ENDFILE statement is an expression that calls a function, that function must not cause an I/O statement or the EOF intrinsic functionto be executed, because unpredictable results can occur.

Example

The following statement writes an end-of-file record to I/O unit 2:

```
ENDFILE 2
```

Suppose the following statement is specified:

```
ENDFILE (UNIT=9, IOSTAT=IOS, ERR=10)
```

An end-of-file record is written to the file connected to unit 9. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable `IOS`.

The following shows another example:

```
WRITE (6, *) x
ENDFILE 6
REWIND 6
READ (6, *) y
```

See Also

BACKSPACE
REWIND
Data Transfer I/O Statements
Branch Specifiers

END FORALL

Statement: *Marks the end of a FORALL construct. The FORALL construct is an obsolescent language feature in Fortran 2018.*

See Also

See FORALL.

END INTERFACE

Statement: *Marks the end of an INTERFACE block.*

See Also

See INTERFACE.

END TYPE

Statement: *Specifies the end of a derived type TYPE statement.*

See Also

See TYPE Statement (Derived Types).

END WHERE

Statement: *Marks the end of a WHERE construct.*

Example

```
WHERE (pressure <= 1.0)
  pressure = pressure + inc_pressure
  temp = temp - 5.0
```

```
 ELSEWHERE
   raining = .TRUE.
 END WHERE
```

Note that the variables `temp`, `pressure`, and `raining` are all arrays.

See Also

See WHERE.

ENTRY

Statement: *Provides one or more entry points within a subprogram. It is not executable and must precede any CONTAINS statement (if any) within the subprogram.*

Syntax

```
ENTRY name[ ( [d-arg[,d-arg]...] ) [RESULT (r-name)] ]
```

name	Is the name of an entry point. If RESULT is specified, this entry name must not appear in any specification statement in the scoping unit of the function subprogram.
d-arg	(Optional) Is a dummy argument. The dummy argument can be an alternate return indicator (*) if the ENTRY statement is within a subroutine subprogram.
r-name	(Optional) Is the name of a function result. This name must not be the same as the name of the entry point, or the name of any other function or function result. This parameter can only be specified for function subprograms.

Description

ENTRY statements can only appear in external procedures or module procedures.

An ENTRY statement must not appear in an executable construct.

When the ENTRY statement appears in a subroutine subprogram, it is referenced by a CALL statement. When the ENTRY statement appears in a function subprogram, it is referenced by a function reference.

An entry name within a function subprogram can appear in a type declaration statement.

Within the subprogram containing the ENTRY statement, the entry name must not appear as a dummy argument in the FUNCTION or SUBROUTINE statement, and it must not appear in an EXTERNAL or INTRINSIC statement. For example, neither of the following are valid:

```
(1)  SUBROUTINE SUB(E)
     ENTRY E
     ...

(2)  SUBROUTINE SUB
     EXTERNAL E
     ENTRY E
     ...
```

The procedure defined by an ENTRY statement can reference itself if the function or subroutine was defined as RECURSIVE.

Dummy arguments can be used in ENTRY statements even if they differ in order, number, type and kind parameters, and name from the dummy arguments used in the FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

Dummy arguments can be referred to only in executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified. If a dummy argument is not currently associated with an actual argument, the dummy argument is undefined and cannot be referenced. Arguments do not retain their association from one reference of a subprogram to another.

Example

```
C  This fragment writes a message indicating
C  whether num is positive or negative
   IF (num .GE. 0) THEN
     CALL Sign
   ELSE
     CALL Negative
   END IF
   ...
   END
SUBROUTINE Sign
   WRITE (*, *) 'It''s positive.'
   RETURN
   ENTRY Negative
   WRITE (*, *) 'It''s negative.'
   RETURN
END SUBROUTINE
```

See Also

Program Units and Procedures
ENTRY Statements in Function Subprograms
ENTRY Statements in Subroutine Subprograms

EOF

Inquiry Intrinsic Function (Generic): *Checks whether a file is at or beyond the end-of-file record.*

Syntax

`result = EOF (unit)`

unit	(Input) Must be of type integer. It represents a unit specifier corresponding to an open file. It cannot be zero unless you have reconnected unit zero to a unit other than the screen or keyboard.

Results

The result type is default logical. The value of the result is .TRUE. if the file connected to *unit* is at or beyond the end-of-file record; otherwise, .FALSE..

Example

```
!  Creates a file of random numbers, reads them back
       REAL x, total
       INTEGER count
       OPEN (1, FILE = 'TEST.DAT')
       DO I = 1, 20
```

```
        CALL RANDOM_NUMBER(x)
        WRITE (1, '(F6.3)') x * 100.0
      END DO
      CLOSE(1)
      OPEN (1, FILE = 'TEST.DAT')
      count = 0
      total = 0.0
      DO WHILE (.NOT. EOF(1))
        count = count + 1
        READ (1, *) value
        total = total + value
      END DO
100   IF ( count .GT. 0) THEN
        WRITE (*,*) 'Average is: ', total / count
      ELSE
        WRITE (*,*) 'Input file is empty '
      END IF
       STOP
      END
```

See Also

ENDFILE
READ Statement for details on an END specifier in a READ statement
BACKSPACE
REWIND

EOSHIFT

Transformational Intrinsic Function (Generic): *Performs an end-off shift on a rank-one array, or performs end-off shifts on all the complete rank-one sections along a given dimension of an array of rank two or greater. Elements are shifted off at one end of a section and copies of a boundary value are filled in at the other end. Different sections can have different boundary values and can be shifted by different amounts and in different directions.*

Syntax

`result = EOSHIFT (`*`array`*`,`*`shift`*` [,`*`boundary`*`][,`*`dim`*`])`

array	(Input) Must be an array (of any data type).
shift	(Input) Must be a scalar integer or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
boundary	(Input; optional) Must have the same type and kind parameters as *array*. It must be a scalar or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n). The *boundary* specifies a value to replace spaces left by the shifting procedure. If *boundary* is not specified, it is assumed to have the following default values (depending on the data type of *array*):

array **Type**	*boundary* **Value**
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character(*len*)	*len* blanks

dim (Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. If *dim* is omitted, it is assumed to be 1.

Results

The result is an array with the same type and kind parameters, and shape as *array*.

If *array* has rank one, the same shift is applied to each element. If an element is shifted off one end of the array, the *boundary* value is placed at the other end the array.

If *array* has rank greater than one, each section (s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) of the result is shifted as follows:

- By the value of *shift*, if *shift* is scalar
- According to the corresponding value in *shift*(s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n), if *shift* is an array

If an element is shifted off one end of a section, the *boundary* value is placed at the other end of the section.

The value of *shift* determines the amount and direction of the end- off shift. A positive *shift* value causes a shift to the left (in rows) or up (in columns). A negative *shift* value causes a shift to the right (in rows) or down (in columns).

Example

V is the integer array (1, 2, 3, 4, 5, 6).

EOSHIFT (V, SHIFT=2) shifts the elements in V to the *left* by 2 positions, producing the value (3, 4, 5, 6, 0, 0). 1 and 2 are shifted off the beginning and two elements with the default BOUNDARY value are placed at the end.

EOSHIFT (V, SHIFT= -3, BOUNDARY= 99) shifts the elements in V to the *right* by 3 positions, producing the value (99, 99, 99, 1, 2, 3). 4, 5, and 6 are shifted off the end and three elements with BOUNDARY value 99 are placed at the beginning.

M is the CHARACTER(LEN=1) array

```
[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ].
```

EOSHIFT (M, SHIFT = 1, BOUNDARY = '*', DIM = 2) produces the result

```
[ 2  3  * ]
[ 5  6  * ]
[ 8  9  * ].
```

Each element in rows 1, 2, and 3 is shifted to the *left* by 1 position. This causes the first element in each row to be shifted off the beginning, and the BOUNDARY value to be placed at the end.

EOSHIFT (M, SHIFT = -1, DIM = 1) produces the result

```
  [ ^  ^  ^ ]
  [ 1  2  3 ]
  [ 4  5  6 ].
```

Each element in columns 1, 2, and 3 is shifted *down* by 1 position. This causes the last element in each column to be shifted off the end and the BOUNDARY value (the default character boundary value <space>, represented by "^") to be placed at the beginning.

EOSHIFT (M, SHIFT = (/1, -1, 0/), BOUNDARY = (/ '*', '?', '/' /), DIM = 2) produces the result

```
  [ 2  3  * ]
  [ ?  4  5 ]
  [ 7  8  9 ].
```

Each element in row 1 is shifted to the *left* by 1 position, causing the first element to be shifted off the beginning and the BOUNDARY value * to be placed at the end. Each element in row 2 is shifted to the *right* by 1 position, causing the last element to be shifted off the end and the BOUNDARY value ? to be placed at the beginning. No element in row 3 is shifted at all, so the specified BOUNDARY value is not used.

The following shows another example:

```
INTEGER shift(3)
CHARACTER(1) array(3, 3), AR1(3, 3)
array = RESHAPE ((/'A', 'D', 'G', 'B', 'E', 'H', &
                   'C', 'F', 'I'/), (/3,3/))
!        array is A B C
!                 D E F
!                 G H I
shift = (/-1, 1, 0/)
AR1 = EOSHIFT (array, shift, BOUNDARY = (/'*','?','#'/), DIM= 2)
! returns         * A B
!                  E F ?
!                  G H I
```

See Also

CSHIFT
ISHFT
ISHFTC
TRANSPOSE

EPSILON

Inquiry Intrinsic Function (Generic): *Returns a positive model number that is almost negligible compared to unity in the model representing real numbers.*

Syntax

`result = EPSILON (x)`

x	(Input) Must be of type real; it can be scalar or array valued.

Results

The result is a scalar of the same type and kind parameters as *x*. The result has the value b^{1-p}. Parameters *b* and *p* are defined in Model for Real Data.

EPSILON makes it easy to select a *delta* for algorithms (such as root locators) that search until the calculation is within *delta* of an estimate. If *delta* is too small (smaller than the decimal resolution of the data type), the algorithm might never halt. By scaling the value returned by EPSILON to the estimate, you obtain a *delta* that ensures search termination.

Example

If *x* is of type REAL(4), EPSILON (X) has the value 2^{-23}.

See Also

PRECISION
TINY
Data Representation Models

EQUIVALENCE

Statement: *Specifies that a storage area is shared by two or more objects in a program unit. This causes total or partial storage association of the objects that share the storage area. EQUIVALENCE is an obsolescent language feature in Standard Fortran.*

Syntax

```
EQUIVALENCE (equiv-list) [,(equiv-list)]...
```

equiv-list

Is a list of two or more variable names, array elements, or substrings, separated by commas (also called an equivalence set). If an object of derived type is specified, it must be a sequence type. Objects cannot have the TARGET attribute.

Each expression in a subscript or a substring reference must be an integer constant expression. A substring must not have a length of zero.

Description

The following objects cannot be specified in EQUIVALENCE statements:

- A dummy argument
- An allocatable variable
- An automatic object
- A pointer
- An object of nonsequence derived type
- A derived-type object that has an allocatable or pointer component at any level
- A component of a derived-type object
- A function, entry, or result name
- A named constant
- A structure component
- A subobject of any of the above objects
- A coarray
- An object with either the DLLIMPORT or DLLEXPORT attribute
- A variable with the BIND attribute
- A variable in a common block that has the BIND attribute

The EQUIVALENCE statement causes all of the entities in one parenthesized list to be allocated storage beginning at the same storage location.

If an equivalence object has the PROTECTED attribute, all of the objects in the equivalence set must have the PROTECTED attribute.

Association of objects depends on their types, as follows:

Type of Object	Type of Associated Object
Intrinsic numeric [1]or numeric sequence	Can be of any of these types
Default character or character sequence	Can be of either of these types [2]
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

[1]Default integer, default real, double precision real, default complex, double complex, or default logical.

[2]The lengths do not have to be equal.

So, objects can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
EQUIVALENCE(A, Y)
```

Objects of default character do not need to have the same length. The following example associates character variable D with the last 4 (of the 6) characters of character array F:

```
CHARACTER(LEN=4) D
CHARACTER(LEN=3) F(2)
EQUIVALENCE(D, F(1)(3:))
```

Entities having different data types can be associated because multiple components of one data type can share storage with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

The same storage unit cannot occur more than once in a storage sequence, and consecutive storage units cannot be specified in a way that would make them nonconsecutive.

Intel® Fortran lets you associate character and noncharacter entities, for example:

```
CHARACTER*1 char1(10)
REAL reala, realb
EQUIVALENCE (reala, char1(1))
EQUIVALENCE (realb, char1(2))
```

EQUIVALENCE statements require only the first subscript of a multidimensional array (unless the STRICT compiler directive is in effect). For example, the array declaration var(3,3), var(4) could appear in an EQUIVALENCE statement. The reference is to the fourth element of the array (var(1,2)), not to the beginning of the fourth row or column.

If you use the STRICT directive, the following rules apply to the kinds of variables and arrays that you can associate:

- If an EQUIVALENCE object is default integer, default real, double-precision real, default complex, default logical, or a sequenced derived type of all numeric or logical components, all objects in the EQUIVALENCE statement must be one of these types, though it is not necessary that they be the same type.
- If an EQUIVALENCE object is default character or a sequenced derived type of all character components, all objects in the EQUIVALENCE statement must be one of these types. The lengths do not need to be the same.
- If an EQUIVALENCE object is a sequenced derived type that is not purely numeric or purely character, all objects in the EQUIVALENCE statement must be the same derived type.
- If an EQUIVALENCE object is an intrinsic type other than the default (for example, INTEGER(1)), all objects in the EQUIVALENCE statement must be the same type and kind.

Example

The following EQUIVALENCE statement is invalid because it specifies the same storage unit for X(1) and X(2):

```
REAL, DIMENSION(2) :: X
REAL :: Y
EQUIVALENCE(X(1), Y), (X(2), Y)
```

The following EQUIVALENCE statement is invalid because A(1) and A(2) will not be consecutive:

```
REAL A(2)
DOUBLE PRECISION D(2)
EQUIVALENCE(A(1), D(1)), (A(2), D(2))
```

In the following example, the EQUIVALENCE statement causes the four elements of the integer array IARR to share the same storage as that of the double-precision variable DVAR:

```
DOUBLE PRECISION DVAR
INTEGER(KIND=2) IARR(4)
EQUIVALENCE(DVAR, IARR(1))
```

In the following example, the EQUIVALENCE statement causes the first character of the character variables KEY and STAR to share the same storage location. The character variable STAR is equivalent to the substring KEY(1:10).

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE(KEY, STAR)
```

The following shows another example:

```
CHARACTER name, first, middle, last
DIMENSION name(60), first(20), middle(20), last(20)
EQUIVALENCE (name(1), first(1)), (name(21), middle(1))
EQUIVALENCE (name(41), last(1))
```

Consider the following:

```
CHARACTER (LEN = 4) :: a, b
CHARACTER (LEN = 3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```

This causes the following alignment:

```
  1          2          3          4          5          6          7
a(1:1)     a(2:2)     a(3:3)     a(4:4)
                                 b(1:1)     b(2:2)     b(3:3)     b(4:4)
c(1)(1:1)  c(1)(2:2)  c(1)(3:3)  c(2)(1:1)  c(2)(2:2)  c(2)(3:3)
```

Note that the fourth element of a, the first element of b, and the first element of c(2) share the same storage unit.

See Also

Initialization Expressions
Derived Data Types
Storage Association
STRICT Directive
Obsolescent Language Features in the Fortran Standard

ERF

Elemental Intrinsic Function (Generic): *Returns the error function of an argument.*

Syntax

```
result = ERF (x)
```

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*. The result is in the range -1 to 1.

ERF returns the error function of *x* defined as follows:

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

Specific Name	Argument Type	Result Type
ERF	REAL(4)	REAL(4)
DERF	REAL(8)	REAL(8)
QERF	REAL(16)	REAL(16)

Example

ERF (1.0) has the value 0.842700794.

See Also

ERFC

ERFC

Elemental Intrinsic Function (Generic): *Returns the complementary error function of an argument.*

Syntax

```
result = ERFC (x)
```

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*. The result is in the range 0 to 2.

ERFC returns 1 - ERF(x) and is defined as follows:

$$\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt$$

ERFC is provided because of the extreme loss of relative accuracy if ERF(x) is called for large *x* and the result is subtracted from 1.

Specific Name	Argument Type	Result Type
ERFC	REAL(4)	REAL(4)
DERFC	REAL(8)	REAL(8)
QERFC	REAL(16)	REAL(16)

Example

ERFC (1.0) has the value 0.1572992057.

See Also

ERF

ERFC_SCALED

Elemental Intrinsic Function (Generic): *Returns the scaled complementary error function of an argument.*

Syntax

```
result = ERFC_SCALED (x)
```

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the exponentially-scaled complementary error function of

$$\exp\left(X^2\right)\frac{2}{\sqrt{\pi}}\int_x^\infty \exp\left(-t^2\right)dt$$

NOTE

The complementary error function is asymptotic to $\exp\left(-X^2\right)/\left(X\sqrt{\pi}\right)$. As such it underflows for $X >\approx 9$ when using IEEE single precision arithmetic. The exponentially-scaled complementary error function is asymptotic to $1/\left(X\sqrt{\pi}\right)$. As such it does not underflow until $X > HUGE(X)/\sqrt{\pi}$.

Example

ERFC_SCALED (20.0) has the approximate value 0.02817434874.

See Also

ERFC

ERRSNS

Intrinsic Subroutine (Generic): *Returns information about the most recently detected I/O system error condition. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL ERRSNS ([io_err] [,sys_err] [,stat] [,unit] [,cond])
```

io_err
(Output; Optional) Is an integer variable or array element that stores the most recent Run-Time Library error number that occurred during program execution. For a listing of error numbers, see Compiler Reference: Error Handling.

A zero indicates no error has occurred since the last call to ERRSNS or since the start of program execution.

sys_err
(Output; Optional) Is an integer variable or array element that stores the most recent system error number associated with *io_err*. This code is one of the following:

- On Windows* systems, it is the value returned by GETLASTERROR( ) at the time of the error.
- On Linux* and macOS* systems, it is an `errno` value. (See `errno(2)`.)

stat
(Output; Optional) Is an integer variable or array element that stores a status value that occurred during program execution. This value is always set to zero.

unit
(Output; Optional) Is an integer variable or array element that stores the logical unit number, if the last error was an I/O error.

cond
(Output; Optional) Is an integer variable or array element that stores the actual processor value. This value is always set to zero.

If you specify INTEGER(2) arguments, only the low-order 16 bits of information are returned or adjacent data can be overwritten. Because of this, it is best to use INTEGER(4) arguments.

The saved error information is set to zero after each call to ERRSNS.

Example

Any of the arguments can be omitted. For example, the following is valid:

```
CALL ERRSNS (SYS_ERR=I1, STAT=I2, UNIT=I4)
```

ESTABLISHQQ

Portability Function: *Lets you specify a function to handle errors detected by the Run-Time Library (RTL). It lets you take appropriate steps in addition to the RTL's error-handling behavior, or it lets you replace that behavior.*

Module

USE IFESTABLISH

Syntax

```
result = ESTABLISHQQ (handler_routine,context,prev_handler,prev_context)
```

handler_routine	(Input) Is of type "procedure(establishqq_handler)", which is defined in module IFESTABLISH. This is the function that will handle errors detected by the RTL.
context	(Input) INTEGER(INT_PTR_KIND()). This is the way you pass information to the handler function for use when it is called. It can be data or a pointer to a block of data.
prev_handler	(Optional; output) Is of type "procedure(establishqq_handler), pointer, intent(out), optional", which is defined in module IFESTABLISH. This is the previous handler function, if any.
prev_context	(Optional; output) INTEGER(INT_PTR_KIND()). This is the context specified for the previous handler function, if any; otherwise, zero.

Results

The result type for ESTABLISHQQ is LOGICAL(4). It indicates whether the handler was successfully established. .TRUE. means it was established successfully; .FALSE. means it was not.

The handler function is called when an error occurs. The result for the handler is set by your code in the handler. .TRUE. means that the error has been handled and the application should not issue an error message. .FALSE. means that the application should issue an error message and continue as if there had never been a handler.

After you use ESTABLISHQQ to specify an error handler function and an error occurs, the handler function is called with the following input arguments. They are set up by the RTL when it calls the handler function:

Handler Function Syntax:

result = Handler_Function (*error_code*, *continuable*, *message_string*, *context*)

Handler_Function is a function you supply that has a compatible interface. It must use the Intel® Fortran compiler defaults for calling mechanism and argument passing. When the Intel Fortran Run-Time Library detects an error, it first calls your handler function with the following arguments:

error_code	(Input) INTEGER(4). This is the number of the error that occurred and it is the value that will be set in an IOSTAT= or STAT= specifier variable. A list of run-time error codes can be found at List of Run-Time Error Messages.
continuable	(Input) LOGICAL(4). If execution can be continued after handling this error, this argument is passed the value .TRUE., otherwise, it is passed the value .FALSE.. Do not compare this value using arithmetic equality operators; use logical data type tests or .EQV.. If an error is not continuable, the program exits after processing the error.
message_string	(Input) CHARACTER(*). This is the text of the error message as it would otherwise be displayed to the user.

context	(Input) INTEGER(INT_PTR_KIND()). This is the value passed for the context argument to ESTABLISHQQ. Your handler function can use this for any purpose.

The function result of the handler function is type LOGICAL(4). The handler function should return .TRUE. if it successfully handled the error; otherwise, .FALSE..

If the handler function returns .TRUE. and the error is continuable, execution of the program continues. If the handler function returns .FALSE., normal error processing is performed, such as message output to the user and possible program termination.

The handler function can be written in any language, but it must follow the Intel Fortran conventions. Note that for argument *message_string*, an address-sized length is passed by value after argument *context*.

If you want to handle errors using a C/C++ handler, use the ISO_C_BINDINGS module features to call the C/C++ routine.

Example

```
! Compile with "-fpe0 -check bounds".
!
program example
use ifestablish
implicit none

    procedure(establishqq_handler), pointer :: old_handler_1
    procedure(establishqq_handler), pointer :: old_handler_2
    procedure(establishqq_handler), pointer :: old_handler_3
    procedure(establishqq_handler)          :: my_handler_1
    procedure(establishqq_handler)          :: my_handler_2
    procedure(establishqq_handler)          :: my_handler_3

    logical                 :: ret
    integer(INT_PTR_KIND()) :: old_context, my_context = 0
    real,    volatile       :: x,y,z
    integer, volatile       :: i, eleven, a(10)

    my_context = 1
    eleven = 11

    ! Test that handlers can be established and restored.
    !
    old_handler_1 => null()
    print *, "== Establish first handler"
    ret = ESTABLISHQQ(my_handler_1, my_context, old_handler_1, old_context)

    if (associated(old_handler_1)) then
        print *, "** Unexpected old handler on first ESTABLISH **"
        ret =  old_handler_1(100, .true., "call number one", 1 )
        print *, "back from call of old handler with", ret
    else
        print *,"== Got expected NULL old handler"
    end if

    print *,"== Violate array bounds; expect first handler"
    i = a(eleven)
```

```
    ! Establish second handler
    !
    old_handler_2 => null()
    print *, "== Establish second handler"
    ret = ESTABLISHQQ(my_handler_2, my_context, old_handler_2, old_context)

    if (associated(old_handler_2)) then
        print *, "== Expect first handler as old handler"
        ret =  old_handler_2(100, .true., "call number one", 1 )
    else
        print *,"** Unexpectedly didn't get first handler as old handler **"
    end if

    print *,"== Violate array bounds; expect second handler"
    i = a(eleven)

    ! Establish third handler
    !
    old_handler_3 => null()
    print *, "== Establish third handler"
    ret = ESTABLISHQQ(my_handler_3, my_context, old_handler_3, old_context)
    !print *, "Got return value ", ret, "old context", old_context

    if (associated(old_handler_3)) then
        print *, "== Expect second handler as old handler"
        ret =  old_handler_3(100, .true., "call number one", 1 )
        !print *, "back from call of old handler with", ret

    else
        print *,"** Unexpectedly didn't get second handler as old handler **"
    end if

    print *,"== Violate array bounds; expect third handler"
    i = a(eleven)

    ! Put back old handlers in stack-wise order, testing.
    !
   ret = ESTABLISHQQ(old_handler_3, old_context)
    print *,"== Violate array bounds; expect second handler"
    i = a(eleven)

    ret = ESTABLISHQQ(old_handler_2, old_context)
    print *,"== Violate array bounds; expect first handler"
    i = a(eleven)

    ret = ESTABLISHQQ(old_handler_1, old_context)
    print *,"== Violate array bounds; expect no handler and exit"
    i = a(eleven)
end

function my_handler_1 (error_code, continuable, message_string, context)
    use, intrinsic :: iso_c_binding
    implicit none
    logical :: my_handler_1
    !DEC$ ATTRIBUTES DEFAULT :: my_handler_1

    ! Arguments
    !
```

```
    integer, intent(in) :: error_code               ! RTL error code from IOSTAT table
    logical, intent(in) :: continuable              ! True if condition is continuable
    character(*), intent(in) :: message_string      ! Formatted message string a la ERRMSG/IOMSG
    integer(INT_PTR_KIND()), intent(in) :: context    ! Address-sized integer passed in to call
                                                      ! ESTABLISHQQ, for whatever purpose
                                                      ! the programmer desires

    my_handler_1 = .TRUE. ! Continue by default

    if (context == 1) then
        print *, "    Handler 1, continue"

    else if (context == 2) then
        print *, "    Handler 1, continue"

    else if (context == 3) then
        print *, "    Handler 1, code should be 73: ", error_code
        if (continuable) then
            print *,"        - is continuable (** an error! **)"
        else
            print *,"        - not continuable"
  end if
        ! We will return .TRUE., asking to continue, but because this is not
        ! a continuable error, the application will exit.

     else
        print *, "    ** Error -- wrong context value"
     end if

     return

end function my_handler_1

function my_handler_2 (error_code, continuable, message_string, context)
    use, intrinsic :: iso_c_binding
    implicit none
    logical :: my_handler_2
    !DEC$ ATTRIBUTES DEFAULT :: my_handler_2

    ! Arguments
    !
    integer, intent(in) :: error_code             ! RTL error code from IOSTAT table
    logical, intent(in) :: continuable            ! True if condition is continuable
    character(*), intent(in) :: message_string    ! Formatted message string a la ERRMSG/IOMSG
    integer(INT_PTR_KIND()), intent(in) :: context    ! Address-sized integer passed in to call
                                                      ! ESTABLISHQQ, for whatever purpose
                                                      ! the programmer desires
    if (context == 1) then
        print *,"    Handler 2, continue"
        my_handler_2 = .TRUE. ! Continue

    else if (context == 2) then
        print *, "    Handler 2, exit"
        my_handler_2 = .FALSE. ! Exit
    end if

    return
end function my_handler_2
```

```
function my_handler_3 (error_code, continuable, message_string, context)
     use, intrinsic :: iso_c_binding
     implicit none
     logical :: my_handler_3
     !DEC$ ATTRIBUTES DEFAULT :: my_handler_3

     ! Arguments
     !
     logical :: my_handler_3
     integer, intent(in) :: error_code             ! RTL error code from IOSTAT table
     logical, intent(in) :: continuable            ! True if condition is continuable
     character(*), intent(in) :: message_string    ! Formatted message string a la ERRMSG/IOMSG
     integer(INT_PTR_KIND()), intent(in) :: context    ! Address-sized integer passed in to call
                                                       ! ESTABLISHQQ, for whatever purpose
                                                       ! the programmer desires
     if (context == 1) then
         print *,"    Handler 3, continue"
     else if (context == 2) then
         print *, "    Handler 3, error is ", error_code, message_string
     end if

     my_handler_3 = .TRUE. ! Continue
     return

end function my_handler_3
```

See Also

Understanding Run-Time Errors
List of Run-Time Error Messages

ETIME

Portability Function: *On single processor systems, returns the elapsed CPU time, in seconds, of the process that calls it. On multi-core or multi-processor systems, returns the elapsed wall-clock time, in seconds.*

Module

USE IFPORT

Syntax

`result = ETIME (array)`

array

(Output) REAL(4). Must be a rank one array with two elements:

- *array(1)* – Elapsed user time, which is time spent executing user code. This value includes time running protected Windows subsystem code. On single processors, ETIME returns the elapsed CPU time, in seconds, of the process that calls it. On multiple processors, ETIME returns the elapsed wall-clock time, in seconds.
- *array(2)* – Elapsed system time, which is time spent executing privileged code (code in the Windows Executive) on single processors; on multiple processors, this value is zero.

Results

The result type is REAL(4). The result is the total CPU time, which is the sum of *array(1)* and *array(2)*.

Example

```
USE IFPORT
REAL(4) I, TA(2)
I = ETIME(TA)
write(*,*) 'Program has used', I, 'seconds of CPU time.'
write(*,*) '  This includes', TA(1), 'seconds of user time and',  &
& TA(2), 'seconds of system time.'
```

See Also

DATE_AND_TIME

EVENT POST and EVENT WAIT

Statements: *The EVENT POST statement allows an image to notify another image that it can proceed to work on tasks that use common resources. The EVENT WAIT statement allows an image to wait on events posted by other images. They take the following forms:*

Syntax

`EVENT POST (`*`event-var`* `[, `*`sync-stat-list`*`])`

`EVENT WAIT (`*`event-var`* `[, `*`wait-spec-list`*`])`

event-var	Is a scalar variable of type EVENT_TYPE. For more information, see intrinsic module ISO_FORTRAN_ENV. It must not depend on the value of *stat-var* or *err-var*. It cannot be a coindexed variable in an EVENT WAIT statement.
sync-stat-list	Is `STAT=`*stat-var* or `ERRMSG=`*err-var*
stat-var	Is a scalar integer variable in which the status of the synchronization is stored.
err-var	Is a scalar default character variable in which explanatory text is stored if an error occurs.
wait-spec-list	Is *until-spec* or *sync-stat-list*
until-spec	Is `UNTIL_COUNT=` scalar-integer-expression.

Each specifier is optional and may appear at most once in a *sync-stat-list* or in a *wait-spec-list*.

Description

An EVENT POST statement atomically increments the value of *event-var* by one. Its value is processor dependent if an error condition occurs during the execution of an EVENT POST statement. The completion of an EVENT POST statement does not in any way depend on execution of a corresponding EVEN WAIT statement.

The following actions occur during the execution of an EVENT WAIT statement:

- If UNTIL_COUNT is not specified, the threshold value is one; otherwise, it is the maximum value of one and the specified scalar-integer-expression.
- The image executing the EVENT WAIT statement waits until the value of *event-var* is equal to or greater than the threshold value; otherwise, an error condition occurs.
- If no error condition has occurred, *event-var* is atomically decremented by the threshold value.

The value of *event-var* is processor dependent if an error condition occurs during the execution of an EVENT WAIT statement.

An EVEN POST statement execution is initially unsatisfied. The successful execution of an EVENT WAIT statement with a threshold value of *n* satisfies the first *n* unsatisfied EVENT POST statement executions for the specified *event-var*. The EVENT WAIT statement delays execution of the segment following the EVENT WAIT statement to execute after the segments which precede the first *n* EVENT POST statement executions for the specified *event-var*.

Example

The following example shows the use of EVENT POST and EVENT WAIT statements to synchronize each image with its left and right neighbor. Image 1 and image NUM_IMAGES() treat each other as left and right neighbors, conceptually laying the images out in a circle.

```
USE, INTRINSIC    :: ISO_FORTRAN_ENV
TYPE(EVENT_TYPE)  :: EVENT[*]
INTEGER           :: LEFT, RIGHT
...
IF ((THIS_IMAGE().NE. 1) .AND. (THIS_IMAGE() .NE. NUM_IMAGES()) THEN
  LEFT  = THIS_IMAGE() - 1
  RIGHT = THIS_IMAGE() + 1
ELSE IF (THIS_IMAGE() == 1) THEN
  LEFT  = NUM_IMAGES()
  RIGHT = 2
ELSE IF (THIS_IMAGE()== NUM_IMAGES()) THEN
  LEFT  = NUM_IMAGES() - 1
  RIGHT = 1
END IF

EVENT POST (EVENT[LEFT])            ! Signal left neighbor you got here
EVENT POST (EVENT[RIGHT])           ! Signal right neighbor you got here
EVENT WAIT (EVENT, UNTIL_COUNT = 2) ! Wait until your neighbors have both reached this point also
```

See Also

ISO_FORTRAN_ENV
Image Control Statements
Coarrays
Using Coarrays

EVENT_QUERY

Intrinsic Subroutine (Generic): *Queries the event count of an event variable.*

Syntax

```
CALL EVENT_QUERY (event, count [, stat])
```

event (Input) Must be an event variable of type EVENT_TYPE, and must not be coindexed. The event variable is accessed atomically with respect to the execution of EVENT POST statements in unordered segments, in exact analogy to atomic subroutines.

count (Output) Must be an integer scalar with a decimal range no smaller than that of default integer. If no error condition occurs, count is assigned the value of the count of *event*, otherwise it is assigned the value -1.

stat (Output; optional) Must be an integer scalar with a decimal exponent range of at least four. It must not be coindexed. If the *stat* argument is present, it is assigned a processor-dependent positive value if an error condition occurs, or zero if no error occurs. If an error occurs and *stat* is not present, error termination is initiated.

Example

Consider the following example:

```
CALL EVENT_QUERY (EVENT, COUNT)
```

If there have been six successful posts to EVENT, and 3 successful waits that did not specify UNTIL_COUNT= in the preceding segments, the variable COUNT will have the value 3. If there have been no successful posts or waits in the preceding segments, COUNT will have the value 0.

See Also

EVENT POST and EVENT WAIT

EXECUTE_COMMAND_LINE

Intrinsic Subroutine: *Executes a command line.*

Syntax

```
CALL EXECUTE_COMMAND_LINE (command [,wait,exitstat,cmdstat,cmdmsg])
```

command (Input) Must be a scalar of type default character. It is the command line to be executed. The interpretation is processor dependent.

wait (Input; optional) Must be a scalar of type logical. If *wait* is present with the value FALSE and the processor supports asynchronous execution of the command, the command is executed asynchronously; otherwise it is executed synchronously.

exitstat (Inout; optional) Must be a scalar of type integer. If the command is executed synchronously, *exitstat* is assigned the value of the processor-dependent exit status of the executed command. Otherwise, the value of *exitstat* is unchanged.

cmdstat (Output; optional) Must be a scalar of type integer. It is assigned one of the following values:

- -1 if the processor does not support command line execution
- A processor-dependent positive value if an error condition occurs
- -2 if no error condition occurs but *wait* is present with the value FALSE and the processor does not support asynchronous execution.
- 0 otherwise

cmdmsg	(Inout; optional) Must be a scalar of type default character. If an error condition occurs, *cmdmsg* is assigned a processor-dependent explanatory message. Otherwise, *cmdmsg* is unchanged.

Description

When *command* is executed synchronously, EXECUTE_COMMAND_LINE returns after the command line has completed execution. Otherwise, EXECUTE_COMMAND_LINE returns without waiting.

If a condition occurs that would assign a nonzero value to *cmdstat* but *cmdstat* is not present, execution is terminated.

NOTE
If the application has a standard console window, command output will appear in that window. On Windows* systems, if the application does not have a console window, including QuickWin applications, command output will not be shown.

Example

```
INTEGER :: CSTAT, ESTAT
CHARACTER(100) :: CMSG
CALL EXECUTE_COMMAND_LINE ("dir > dir.txt", EXITSTAT=ESTAT, &
           CMDSTAT=CSTAT, CMDMSG=CMSG)
IF (CSTAT > 0) THEN
  PRINT *, "Command execution failed with error ", TRIM(CMSG)
ELSE IF (CSTAT < 0) THEN
  PRINT *, "Command execution not supported"
ELSE
  PRINT *, "Command completed with status ", ESTAT
END IF
END
```

In the above example, EXECUTE_COMMAND_LINE is called to execute a "dir" command with output redirected to a file. Since the WAIT argument was omitted, the call waits for the command to complete and the command's exit status is returned in ESTAT. If the command cannot be executed, the error message returned in CMSG is displayed, but note that this is not based on the success or failure of the command itself.

EXIT Statement

Statement: *Terminates execution of a DO loop or a named construct.*

Syntax

```
EXIT [name]
```

name	(Optional) Is the name of the DO loop or construct.

Description

The EXIT statement causes execution of a DO loop or a named construct to be terminated.

If *name* is specified, the EXIT statement must be within the range of that named construct. Otherwise, the EXIT statement must be within a DO loop and it exits the innermost DO within which it appears.

If a DO loop is terminated, any inner DO loops are also terminated and the DO control variables retain their last value. If a non-DO construct is terminated, any DO loops inside that construct are also terminated.

An EXIT statement must not appear within a CRITICAL or DO CONCURRENT construct if it belongs to that construct or an outer construct.

An EXIT statement can appear in any of the following constructs:

- ASSOCIATE
- BLOCK
- DO
- IF
- SELECT CASE
- SELECT RANK
- SELECT TYPE

An EXIT statement cannot appear in a WHERE or FORALL construct.

Example

The following examples demonstrate EXIT statements.

Example 1:

```
LOOP_A : DO I = 1, 15
  N = N + 1
  IF (N > I) EXIT LOOP_A
END DO LOOP_A
```

Example 2:

```
    INTEGER numpoints, point
    REAL datarray(1000), sum
    sum = 0.0
    DO point = 1, 1000
      sum = sum + datarray(point)
      IF (datarray(point+1) .EQ. 0.0) EXIT
    END DO
```

Example 3:

```
DO I=1,N
   MyBlock:  BLOCK
    REAL :: T
    T = A(I) + B(I)
    IF (T == 0.0) EXIT MyBlock
    C(I) = T + SQRT(T)
    END BLOCK
END DO
```

Example 4:

```
DO CONCURRENT (I = 1:N)
   MyBlock:  BLOCK
     REAL :: T
    T = A(I) + B(I)
    IF (T == 0.0) EXIT MyBlock
    C(I) = T + SQRT(T)
    END BLOCK
END DO
```

The following example shows illegal EXIT statements in DO CONCURRENT and CRITICAL:

```
LOOP_1 : DO CONCURRENT (I = 1:N)
  N = N + 1
  IF (N > I) EXIT LOOP_1   ! can’t EXIT DO CONCURRENT or CRITICAL
```

```
END DO LOOP_1

LOOP_2 : DO I = 1, 15
  CRITICAL
    N = N + 1
    IF (N > I) EXIT LOOP_2      ! can't EXIT outer construct from inside
  END CRITICAL                  !   DO CONCURRENT or CRITICAL
END DO LOOP_2
```

See Also

DO
DO WHILE

EXIT Subroutine

Intrinsic Subroutine (Generic): *Terminates program execution, closes all files, and returns control to the operating system. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL EXIT [( [status] )]`

status	(Output; optional) Is an integer argument you can use to specify the image exit-status value.

The exit-status value may not be accessible after program termination in some application environments.

Example

```
    INTEGER(4) exvalue
! all is well, exit with 1
    exvalue = 1
    CALL EXIT(exvalue)
! all is not well, exit with diagnostic -4
    exvalue = -4
    CALL EXIT(exvalue)
! give no diagnostic, just exit
    CALL EXIT ( )
```

See Also

END
ABORT

EXP

Elemental Intrinsic Function (Generic): *Computes an exponential value.*

Syntax

`result = EXP (x)`

x	(Input) Must be of type real or complex.

Results

The result type and kind are the same as *x*. The value of the result is e ^{x}. If *x* is of type complex, its imaginary part is regarded as a value in radians.

Specific Name	Argument Type	Result Type
EXP	REAL(4)	REAL(4)
DEXP	REAL(8)	REAL(8)
QEXP	REAL(16)	REAL(16)
CEXP [1]	COMPLEX(4)	COMPLEX(4)
CDEXP[2]	COMPLEX(8)	COMPLEX(8)
CQEXP	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CEXP.

[2]This function can also be specified as ZEXP.

Example

EXP (2.0) has the value 7.389056.

EXP (1.3) has the value 3.669297.

The following shows another example:

```
!  Given initial size and growth rate,
!  calculates the size of a colony at a given time.
     REAL sizei, sizef, time, rate
     sizei = 10000.0
     time = 40.5
     rate = 0.0875
     sizef = sizei * EXP (rate * time)
     WRITE (*, 100) sizef
100  FORMAT (' The final size is ', E12.6)
     END
```

See Also

EXP10
LOG

EXP10

Elemental Intrinsic Function (Generic): *Computes a base 10 exponential value.*

Syntax

```
result = EXP10 (x)
```

x (Input) Must be of type real or complex.

Results

The result type and kind are the same as *x*. The value is 10 raised to the power of *x*. If *x* is of type complex, its imaginary part is regarded as a value in radians.

Specific Name	Argument Type	Result Type
EXP10	REAL(4)	REAL(4)
DEXP10	REAL(8)	REAL(8)
QEXP10	REAL(16)	REAL(16)
CEXP10 [1]	COMPLEX(4)	COMPLEX(4)
CDEXP10	COMPLEX(8)	COMPLEX(8)
CQEXP10	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CEXP10.

Example

EXP10 (2.0) has the value 100.00.

EXP10 (1.3) has the value 19.95262.

See Also

LOG10
EXP

EXPONENT

Elemental Intrinsic Function (Generic): *Returns the exponent part of the argument when represented as a model number.*

Syntax

`result = EXPONENT (x)`

x	(Input) must be of type real.

Results

The result type is default integer. If *x* is not equal to zero, the result value is the exponent part of *x*. The exponent must be within default integer range; otherwise, the result is undefined.

If *x* is zero, the exponent of *x* is zero. For more information on the exponent part (*e*) in the real model, see Model for Real Data.

Example

EXPONENT (2.0) has the value 2.

If 4.1 is a REAL(4) value, EXPONENT (4.1) has the value 3.

The following shows another example:

```
REAL(4) r1, r2
REAL(8) r3, r4
r1 = 1.0
r2 = 123456.7
r3 = 1.0D0
r4 = 123456789123456.7
write(*,*) EXPONENT(r1) ! prints 1
write(*,*) EXPONENT(r2) ! prints 17
```

```
write(*,*) EXPONENT(r3) ! prints 1
write(*,*) EXPONENT(r4) ! prints 47
END
```

See Also

DIGITS
RADIX
FRACTION
MAXEXPONENT
MINEXPONENT
Data Representation Models

EXTENDS_TYPE_OF

Inquiry Intrinsic Function (Generic): *Inquires whether the dynamic type of an object is an extension type of the dynamic type of another object.*

Syntax

`result = EXTENDS_TYPE_OF (a , mold)`

a	(Input) Is an object of extensible type. If it is a pointer, it must not have an undefined association status.
mold	(Input) Is an object of extensible type. If it is a pointer, it must not have an undefined association status.

Results

The result type is default logical scalar.

The following determines the result value:

- If *mold* is unlimited polymorphic and is a disassociated pointer or an unallocated allocatable, the result is .TRUE..
- If *a* is unlimited polymorphic and is a disassociated pointer or an unallocated allocatable, the result is .FALSE..
- If the dynamic type of *a* or *mold* is extensible, the result is true only if the dynamic type of *a* is an extension type of the dynamic type of *mold*.

Otherwise, the result is processor dependent.

EXTERNAL

Statement and Attribute: *Allows an external procedure, a dummy procedure, a procedure pointer, or a block data subprogram to be used as an actual argument. (To specify intrinsic procedures as actual arguments, use the INTRINSIC attribute.)*

Syntax

The EXTERNAL attribute can be specified in a type declaration statement or an EXTERNAL statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] EXTERNAL [, *att-ls*] :: *ex-pro*[, *ex-pro*]...

Statement:

```
EXTERNAL [::]ex-pro[, ex-pro]...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ex-pro	Is the name of an external (user-supplied) procedure, a dummy procedure, a procedure pointer, or block data subprogram.

Description

In a type declaration statement, only functions can be declared EXTERNAL. However, you can use the EXTERNAL statement to declare subroutines and block data program units, as well as functions, to be external.

The name declared EXTERNAL is assumed to be the name of an external procedure, even if the name is the same as that of an intrinsic procedure. For example, if SIN is declared with the EXTERNAL attribute, all subsequent references to SIN are to a user-supplied function named SIN, not to the intrinsic function of the same name.

You can include the name of a block data program unit in the EXTERNAL statement to force a search of the object module libraries for the block data program unit at link time. However, the name of the block data program unit must not be used in a type declaration statement.

If you want to describe a routine with greater detail, use the INTERFACE statement. This statement automatically declares a routine as EXTERNAL, and provides information on result types and argument types.

A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

Example

The following example shows type declaration statements specifying the EXTERNAL attribute:

```
PROGRAM TEST
...
INTEGER, EXTERNAL :: BETA
LOGICAL, EXTERNAL :: COS
...
CALL SUB(BETA)       ! External function BETA is an actual argument
```

You can use a name specified in an EXTERNAL statement as an actual argument to a subprogram, and the subprogram can then use the corresponding dummy argument in a function reference or a CALL statement; for example:

```
EXTERNAL FACET
CALL BAR(FACET)
SUBROUTINE BAR(F)
EXTERNAL F
CALL F(2)
```

Used as an argument, a complete function reference represents a value, not a subprogram; for example, FUNC(B) represents a value in the following statement:

```
CALL SUBR(A, FUNC(B), C)
```

The following shows another example:

```
EXTERNAL MyFunc, MySub
C   MyFunc and MySub are arguments to Calc
    CALL Calc (MyFunc, MySub)
C   Example of a user-defined function replacing an
```

```
C   intrinsic
    EXTERNAL SIN
    x = SIN (a, 4.2, 37)
```

See Also

INTRINSIC
Program Units and Procedures
Type Declarations
INTRINSIC
Compatible attributes
FORTRAN 66 Interpretation of the External Statement

FAIL IMAGE

Statement: *Causes the image that executes it to stop participating in program execution, without initiating termination.*

Syntax

```
FAIL IMAGE
```

This statement allows you to simulate image failure and to test and debug image failure recovery in a program, without requiring an actual image failure.

After execution of a FAIL IMAGE statement, no additional statements are executed by the image.

A FAIL IMAGE statement is not an image control statement.

> **NOTE**
> The detection of a failed image may happen at different times in the execution of other images; for more information, see FAILED_IMAGES.

Example

If an image calls the following procedure at regular time intervals, it has a one in ten thousand chance of failure in each time step:

```
SUBROUTINE RANDOM_FAILURE ()
   REAL RANDOM
   CALL RANDOM_NUMBER (RANDOM)
   IF (RANDOM > 0.9999) FAIL IMAGE
   RETURN
END SUBROUTINE RANDOM_FAILURE
```

See Also

FAILED_IMAGES
IMAGE_STATUS

FAILED_IMAGES

Transformational Intrinsic Function (Generic): *Returns an array of index images that have failed.*

Syntax

```
result = FAILED_IMAGES ([kind])
```

kind	(Input; optional) Must be a scalar integer expression with a value that is a valid INTEGER kind type parameter.

Results

The result is a rank-one integer array with the same type kind parameters as *kind* if present; otherwise, default integer. The size of the array is equal to the number of images in the current (initial) team that are known to have failed.

The result array elements are the image index values of images on the current team that are known to have failed. The indices are arranged in increasing numeric order.

If the image executing the FAILED_IMAGES reference previously executed a collective subroutine whose STAT argument returned the value STAT_FAILED_IMAGES defined in the intrinsic module ISO_FORTRAN_ENV, or if the image executed an image control statement whose STAT= specifier returned the value STAT_FAILED_IMAGE, at least one image in the team executing the collective or image control statement is known to have failed.

Failed images may lead to unavoidable hangs

Coarray programs are parallel programs, so between synchronization points the relative ordering of events in different images is unknown and undefined. The failure of an image or the execution of the FAIL IMAGE statement is such an event. It happens at a definite time in the image that fails, but the other images will discover the failure at different points in their execution. Also, because a failed image does not participate in synchronization points, it is possible for the discovery to happen before a synchronization point in one image and after it in another.

This means that when other images synchronize (such as with a SYNC ALL), it is possible that some will know that that image has failed and some will not. In this case, the images that don't know will attempt to synchronize with the failed image and the application will hang, making no progress.

There is no certain way to prevent a hang when an image fails. However, if you structure your program so that synchronizations points are infrequent, the chance of a failure happening just before a synchronization point is lower. If images frequently do coarray loads and stores, or check image status, they are more likely to discover a failed image sooner. The FAILED_IMAGES intrinsic will check for failed images, but other images might not get the same result from that call.

NOTE
FAILED_IMAGES argument *team* has not yet been implemented. It will be added in a future release.

Example

If image 5 and 12 of the current team are known to have failed, the result of FAILED_IMAGES ( ) is an array of default integer type with size 2 defined with the value [5, 12]. If no images in the current team are known to have failed, the result of FAILED_IMAGES ( ) is a zero-sized array.

See Also

IMAGE_STATUS
STOPPED_IMAGES
ISO_FORTRAN_ENV Module

FDATE

Portability Function and Subroutine: *Returns the current date and time as an ASCII string.*

Module

USE IFPORT

Syntax

Function Syntax

```
result = FDATE()
```

Subroutine Syntax:

```
CALL FDATE ( [string] )
```

string (Output; optional) Character*(*). It is returned as a 24-character string in the form:

```
Mon Jan 31 04:37:23 2001
```

Any value in *string* before the call is destroyed.

Results

The result of the function FDATE and the value of *string* returned by the subroutine FDATE(*string*) are identical. Newline and NULL are not included in the string.

When you use FDATE as a function, declare it as:

```
CHARACTER*24 FDATE
```

Example

```
USE IFPORT
CHARACTER*24 today
!
CALL FDATE(today)
write (*,*), 'Today is ', today
!
write (*,*), 'Today is ', fdate()
```

See Also

DATE_AND_TIME

FGETC

Portability Function: *Reads the next available character from a file specified by a Fortran unit number.*

Module

USE IFPORT

Syntax

```
result = FGETC (lunit,char)
```

lunit (Input) INTEGER(4). Unit number of a file. Must be currently connected to a file when the function is called.

char (Output) CHARACTER*1. Next available character in the file. If *lunit* is connected to a console device, then no characters are returned until the Enter key is pressed.

Results

The result type is INTEGER(4). The result is zero if the read is successful, or -1 if an end-of-file is detected. A positive value is either a system error code or a Fortran I/O error code, such as:

EINVAL: The specified unit is invalid (either not already open, or an invalid unit number).

If you use WRITE, READ, or any other Fortran I/O statements with *lunit*, be sure to read Portability Routines.

Example

```
USE IFPORT
CHARACTER inchar
INTEGER istatus
istatus = FGETC(5,inchar)
PRINT *, inchar
END
```

See Also

GETCHARQQ
READ

FINAL Clause

Parallel Directive Clause: *Specifies that the generated task will be a final task.*

Syntax

`FINAL (`*`scalar-logical-expression`*`)`

scalar-logical-expression	Is a scalar logical expression. When the expression evaluates to .TRUE., it specifies that the generated task will be a final task.

All task constructs encountered during execution of a final task will generate included tasks.

Note that if a variable is used in a FINAL clause expression of a directive construct, it causes an implicit reference to the variable in all enclosing constructs.

Only a single FINAL clause can appear in the directive.

FINAL Statement

Statement: *Denotes a finalization procedure that defines one or more final subroutines that are bound to a derived type.*

Syntax

The FINAL statement takes the following form:

`FINAL [::]` *`sub1`* `[,` *`sub2`* `]...`

sub1, sub2, ...	Is a final subroutine, which is a module procedure with exactly one dummy argument of the derived type. The dummy argument cannot be INTENT(OUT), it cannot be optional, and it cannot be a pointer or allocatable. All array shape and length type parameters are assumed.

Description

A final subroutine must not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of the type.

You cannot specify a subroutine that was previously specified as a final subroutine for the derived type.

This statement is used in derived-type type-bound procedures.

A derived type is finalizable only if it has a final subroutine or a nonpointer, nonallocatable component of finalizable type. A nonpointer data entity is finalizable only if it is of finalizable type.

When an entity is finalized, the following occurs:

1. If the dynamic type of the entity has a final subroutine whose dummy argument has the same kind type parameters and rank as the entity being finalized, it is called with the entity as an actual argument
2. Otherwise, if there is an elemental final subroutine whose dummy argument has the same kind type parameters as the entity being finalized, it is called with the entity as an actual argument.
3. Otherwise, no subroutine is called.

Effects of Finalization

If the entity being finalized is an array, each finalizable component of each element of that array is finalized separately.

When a pointer is deallocated, its target is finalized. When an allocatable entity is deallocated, it is finalized.

A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized immediately before it would become undefined due to execution of a RETURN or END statement.

If the entity is of extended type and the parent type is finalizable, the parent component is finalized.

A nonpointer, nonallocatable local variable of a BLOCK construct is finalized immediately before it would become undefined due to termination of the BLOCK construct.

The following rules also apply:

- If an executable construct references a function, the result is finalized after execution of the innermost executable construct containing the reference.
- If an executable construct references a structure constructor or array constructor, the entity created by the constructor is finalized after execution of the innermost executable construct containing the reference.
- If a specification expression in a scoping unit references a function, the result is finalized before execution of the executable constructs in the scoping unit.
- If a specification expression in a scoping unit references a structure constructor or array constructor, the entity created by the constructor is finalized before execution of the executable constructs in the scoping unit.
- If image execution is terminated, either by an error or by execution of a ERROR STOP, STOP, or END PROGRAM statement, any entities that exist immediately before termination are not finalized.

Example

The following example declares two module subroutines to be final:

```
TYPE MY_TYPE
...   ! Component declarations
CONTAINS
  FINAL :: CLEAN1, CLEAN2
END TYPE MY_TYPE
```

See Also

Type-Bound Procedures

FIND

Statement: *Positions a direct access file at a particular record and sets the associated variable of the file to that record number. It is comparable to a direct access READ statement with no I/O list, and it can open an existing file. No data transfer takes place.*

Syntax

```
FIND ([UNIT=] io-unit, REC= r[, ERR= label] [, IOSTAT= i-var])
FIND (io-unit 'r [, ERR=label] [, IOSTAT=i-var])
```

io-unit	Is a logical unit number. It must refer to a relative organization file (see Unit Specifier).
r	Is the direct access record number. It cannot be less than one or greater than the number of records defined for the file (see Record Specifier).
label	Is the label of the executable statement that receives control if an error occurs.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs, and as zero if no error occurs (see I/O Status Specifier).

Example

In the following example, the FIND statement positions logical unit 1 at the first record in the file. The file's associated variable is set to one:

```
FIND(1, REC=1)
```

In the following example, the FIND statement positions the file at the record identified by the content of INDX. The file's associated variable is set to the value of INDX:

```
FIND(4, REC=INDX)
```

See Also

Forms for Direct-Access READ Statements
I/O Control List

FINDLOC

Transformational Intrinsic Function (Generic):
Finds the location of a specified value in an array.

Syntax

```
result = FINDLOC (array, value, dim [, mask, kind, back])
result = FINDLOC (array, value[, mask, kind, back])
```

array	(Input) Must be an array of intrinsic type.
value	(Input) Must be scalar and in type conformance with *array*.

dim	(Input) Must be a scalar integer with a value in the range 1 <= *dim* <= *n*, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.
mask	(Input; optional) Must be of type logical and conformable with *array*.
kind	(Input; optional) Must be a scalar integer constant expression.
back	(Input; optional) Must be a scalar of type logical.

Results

The result is integer. If *kind* is present, the kind type parameter is that specified by the value of *kind*. Otherwise, the kind type parameter is that of default integer type. If *dim* does not appear, the result is an array of rank one and of size equal to the rank of *array*; otherwise, the result is of rank *n* - 1 and shape (d_1, d_2,...d_{dim-1}, d_{dim+1},...d_n), where (d_1, d_2,...d_n) is the shape of *array*.

The result of FINDLOC (*array*, *value*) is a rank-one array whose element values are the values of the subscripts of an element of *array* whose value matches *value*. If there is such a value, the *i*th subscript returned is in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array*. If no elements match *value* or if *array* has size zero, all elements of the result are zero.

The result of FINDLOC (*array*, *value*, MASK = *mask*) is a rank-one array whose element values are the values of the subscripts of an element of *array*, corresponding to a true element of *mask*, whose value matches *value*. If there is such a value, the *i*th subscript returned is in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array*. If no elements match *value*, or *array* has size zero, or every element of *mask* has the value .FALSE., all elements of the result are zero.

If ARRAY has rank one, the result of FINDLOC (*array*, *value*, DIM = *dim* [, MASK = *mask*]) is a scalar whose value is equal to that of the first element of FINDLOC (*array*, *value* [, MASK = *mask*]). Otherwise, the value of element (s_1, s_2,...s_{dim-1}, s_{dim+1},...s_n) of the result is equal to FINDLOC (*array* (s_1, s_2,...$s_{valuedim-1}$, :, s_{dim+1},...s_n), *value*, DIM = 1 [, MASK = *mask* (s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n)]).

If both *array* and *value* are of type logical, the comparison is performed with the .EQV. operator; otherwise, the comparison is performed with the == operator. If the value of the comparison is true, that element of *array* matches *value*.

If only one element matches *value*, that element's subscripts are returned. If more than one element matches *value* and *back* is absent or present with the value .FALSE., the element whose subscripts are returned is the first element, taken in array element order. If *back* is present with the value .TRUE., the element whose subscripts are returned is the last element, taken in array element order.

The setting of compiler options specifying integer size can affect this function.

Example

The value of FINDLOC ([2, 6, 4, 6], VALUE = 6) is [2].

The value of FINDLOC ([2, 6, 4, 6], VALUE = 6, BACK = .TRUE.) is [4].

If A has the value

```
[ 0  -5   7   7 ]
[ 3   4  -1   2 ]
[ 1   5   6   7 ]
```

and M has the value

```
[ T   T   F   T ]
[ T   T   F   T ]
[ T   T   F   T ]
```

then FINDLOC (A, 7, MASK = M) is [1, 4] and FINDLOC (A, 7, MASK = M, BACK = .TRUE.) is [3, 4].

This is independent of the declared lower bounds for A.

The value of FINDLOC ([2, 6, 4], VALUE = 6, DIM = 1) is 2.

If B has the value

```
[ 1   2   -9 ]
[ 2   3    6 ]
```

then FINDLOC (B, VALUE = 2, DIM = 1) is [2, 1, 0] and FINDLOC (B, VALUE = 2, DIM = 2) is [2, 1].

This is independent of the declared lower bounds for B.

See Also

MAXLOC
MINLOC

FINDFILEQQ

Portability Function: *Searches for a specified file in the directories listed in the path contained in the environment variable.*

Module

USE IFPORT

Syntax

`result = FINDFILEQQ (filename,varname,pathbuf)`

filename	(Input) Character*(*). Name of the file to be found.
varname	(Input) Character*(*). Name of an environment variable containing the path to be searched.
pathbuf	(Output) Character*(*). Buffer to receive the full path of the file found.

Results

The result type is INTEGER(4). The result is the length of the string containing the full path of the found file returned in *pathbuf*, or 0 if no file is found.

Example

```
USE IFPORT
CHARACTER(256) pathname
INTEGER(4) pathlen
pathlen = FINDFILEQQ("libfmt.lib", "LIB", pathname)
WRITE (*,*) pathname
END
```

See Also

FULLPATHQQ
GETFILEINFOQQ
SPLITPATHQQ

FIRSTPRIVATE

Parallel Directive Clause: *Provides a superset of the functionality provided by the PRIVATE clause. It declares one or more variables to be private to each thread in a team, and initializes each of them with the value of the corresponding original variable.*

Syntax

`FIRSTPRIVATE (list)`

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a FIRSTPRIVATE list are subject to PRIVATE clause semantics. In addition, private (local) copies of each variable in the different threads are initialized to the value the variable had upon entering the parallel region.

To avoid race conditions, which are caused by unintended sharing of data, concurrent updates of the original variable must be synchronized with the read of the original variable that occurs as a result of the FIRSTPRIVATE clause.

If the original variable has the POINTER attribute, the new variable receives the same association status of the original variable as if by pointer assignment.

If the original variable does not have the POINTER attribute, initialization of the new variable occurs as if by intrinsic assignment, unless the original variable has the allocation status of "not currently allocated". In this case, the new variable also has the status of " not currently un allocated".

The following are restrictions for the FIRSTPRIVATE clause:

- A variable that is part of another variable (as an array or structure element) must not appear in a FIRSTPRIVATE clause.
- A variable that is private within a parallel region must not appear in a FIRSTPRIVATE clause in a worksharing construct if any of the worksharing regions arising from the worksharing construct ever bind to any of the parallel regions arising from the parallel construct.
- A variable that appears in a REDUCTION clause of a PARALLEL construct must not appear in a FIRSTPRIVATE clause in a worksharing or task construct if any of the worksharing or task regions arising from the worksharing or task construct ever bind to any of the parallel regions arising from the parallel construct.
- A variable that appears in a REDUCTION clause in a WORKSHARE construct must not appear in a FIRSTPRIVATE clause in a task construct encountered during execution of any of the worksharing regions arising from the worksharing construct.
- Assumed-size arrays must not appear in a PRIVATE clause.
- Variables that appear in NAMELIST statements, in variable format expressions, and in expressions for statement function definitions, must not appear in a PRIVATE clause.
- If a list item appears in both the FIRSTPRIVATE and LASTPRIVATE clauses, the update required for LASTPRIVATE occurs after all of the initializations for FIRSTPRIVATE.

NOTE
If a variable appears in both FIRSTPRIVATE and LASTPRIVATE clauses, the update required for LASTPRIVATE occurs after all initializations for FIRSTPRIVATE..

Example

Consider the following:

```
A = 3
B = 4
!$OMP PARALLEL PRIVATE(A) FIRSTPRIVATE(B)
```

In this case, variable A has an undefined value at the beginning of the parallel region. However, variable B has the value 4, which was specified in the serial region preceding the parallel region.

See Also

PRIVATE clause
LASTPRIVATE clause

FIXEDFORMLINESIZE

General Compiler Directive: *Sets the line length for fixed-form Fortran source code.*

Syntax

```
!DIR$ FIXEDFORMLINESIZE:{72 | 80 | 132}
```

You can set FIXEDFORMLINESIZE to 72 (the default), 80, or 132 characters. The FIXEDFORMLINESIZE setting remains in effect until the end of the file, or until it is reset.

The FIXEDFORMLINESIZE directive sets the source-code line length in include files, but not in USE modules, which are compiled separately. If an include file resets the line length, the change does not affect the host file.

This directive has no effect on free-form source code.

Example

```
!DIR$ NOFREEFORM
!DIR$ FIXEDFORMLINESIZE:132
      WRITE (*,*) 'Sentence that goes beyond the 72nd column without continuation.'
```

See Also

FREEFORM and NOFREEFORM
Source Forms
General Compiler Directives
Syntax Rules for Compiler Directives
Equivalent Compiler Options

FLOAT

Elemental Intrinsic Function (Generic): *Converts an integer to REAL(4).*

See Also

REAL

FLOODFILL, FLOODFILL_W (W*S)

Graphics Functions: *Fill an area using the current color index and fill mask.*

Module

USE IFQWIN

Syntax

```
result = FLOODFILL (x,y,bcolor)
result = FLOODFILL_W (wx,wy,bcolor)
```

x, y	(Input) INTEGER(2). Viewport coordinates for fill starting point.
bcolor	(Input) INTEGER(2). Color index of the boundary color.
wx, wy	(Input) REAL(8). Window coordinates for fill starting point.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *bcolor*, or if the starting point lies outside the clipping region).

FLOODFILL begins filling at the viewport-coordinate point (*x*, *y*). FLOODFILL_W begins filling at the window-coordinate point (*wx*, *wy*). The fill color used by FLOODFILL and FLOODFILL_W is set by SETCOLOR. You can obtain the current fill color index by calling GETCOLOR. These functions allow access only to the colors in the palette (256 or less). To access all available colors on a VGA (262,144 colors) or a true color system, use the RGB functions FLOODFILLRGB and FLOODFILLRGB_W.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current graphics color index set by SETCOLOR. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *bcolor*.

NOTE
The FLOODFILL routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the FloodFill routine by including the IFWIN module, you need to specify the routine name as MSFWIN$FloodFill.

Example

```
USE IFQWIN
INTEGER(2) status, bcolor, red, blue
INTEGER(2) x1, y1, x2, y2, xinterior, yinterior
x1 = 80; y1 = 50
x2 = 240; y2 = 150
red = 4
blue = 1
status = SETCOLOR(red)
status = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
bcolor = GETCOLOR()
status = SETCOLOR (blue)
```

```
xinterior = 160; yinterior = 100
status = FLOODFILL (xinterior, yinterior, bcolor)
END
```

See Also

FLOODFILLRGB, FLOODFILLRGB_W
ELLIPSE
GETCOLOR
GETFILLMASK
GRSTATUS
PIE
SETCLIPRGN
SETCOLOR
SETFILLMASK

FLOODFILLRGB, FLOODFILLRGB_W (W*S)

Graphics Functions: *Fill an area using the current Red-Green-Blue (RGB) color and fill mask.*

Module

USE IFQWIN

Syntax

`result = FLOODFILLRGB (x,y,color)`

`result = FLOODFILLRGB_W (wx,wy,color)`

x, y	(Input) INTEGER(2). Viewport coordinates for fill starting point.
color	(Input) INTEGER(4). RGB value of the boundary color.
wx, wy	(Input) REAL(8). Window coordinates for fill starting point.

Results

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *color*, or if the starting point lies outside the clipping region).

FLOODFILLRGB begins filling at the viewport-coordinate point (*x*, *y*). FLOODFILLRGB_W begins filling at the window-coordinate point (*wx*, *wy*). The fill color used by FLOODFILLRGB and FLOODFILLRGB_W is set by SETCOLORRGB. You can obtain the current fill color by calling GETCOLORRGB.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current color set by SETCOLORRGB. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *color*.

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) status
INTEGER(4) result, bcolor
INTEGER(2) x1, y1, x2, y2, xinterior, yinterior
x1 = 80; y1 = 50
```

```
x2 = 240; y2 = 150
result = SETCOLORRGB(Z'008080') ! red
status = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
bcolor = GETCOLORRGB( )
result = SETCOLORRGB (Z'FF0000') ! blue
xinterior = 160; yinterior = 100
result = FLOODFILLRGB (xinterior, yinterior, bcolor)
END
```

See Also

ELLIPSE
FLOODFILL
GETCOLORRGB
GETFILLMASK
GRSTATUS
PIE
SETCLIPRGN
SETCOLORRGB
SETFILLMASK

FLOOR

Elemental Intrinsic Function (Generic): *Returns the greatest integer less than or equal to its argument.*

Syntax

`result = FLOOR (a[,kind])`

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the greatest integer less than or equal to *a*.

The setting of compiler options specifying integer size can affect this function.

Example

FLOOR (4.8) has the value 4.

FLOOR (-5.6) has the value -6.

The following shows another example:

```
I = FLOOR(3.1)  ! returns 3
I = FLOOR(-3.1) ! returns -4
```

See Also

CEILING

FLUSH Directive

OpenMP* Fortran Compiler Directive: *Identifies synchronization points at which the threads in a team must provide a consistent view of memory.*

Syntax

```
!$OMP FLUSH [(list)]
```

list	Is the name of one or more variables to be flushed. Names must be separated by commas.

The binding thread set for a FLUSH construct is the encountering thread.

The FLUSH directive must appear at the precise point in the code at which the synchronization is required. To avoid flushing all variables, specify a *list*.

Thread-visible variables are written back to memory at the point at which this directive appears. Modifications to thread-visible variables are visible to all threads after this point. Subsequent reads of thread-visible variables fetch the latest copy of the data.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules)
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram
- Dummy arguments
- All pointer dereferences

The FLUSH directive is implied for the following directives (unless the NOWAIT keyword is used):

- ATOMIC and END ATOMIC
- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE
- END WORKSHARE
- ORDERED and END ORDERED
- PARALLEL and END PARALLEL
- PARALLEL DO and END PARALLEL DO
- PARALLEL SECTIONS and END PARALLEL SECTIONS
- TARGET
- TARGET DATA
- TARGET ENTER DATA on entry
- TARGET EXIT DATA on exit
- TARGET UPDATE on entry if TO is present and on exit if FROM is present

These directives are only available on Linux* systems: TARGET, TARGET DATA, TARGET ENTER DATA, TARGET EXIT DATA, TARGET UPDATE.

Example

The following example uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM( )
      ISYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK( )
C I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
      ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
C WAIT TILL NEIGHBOR IS DONE
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
      END DO
!$OMP END PARALLEL
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

FLUSH Statement

Statement: *Causes data written to a file to become available to other processes or causes data written to a file outside of Fortran to be accessible to a READ statement. It takes one of the following forms:*

Syntax

`FLUSH([UNIT=]`*`io-unit`* `[, ERR=`*`label`*`] [, IOMSG=`*`msg-var`*`] [IOSTAT=`*`i-var`*`])`

`FLUSH` *`io-unit`*

`io-unit`	(Input) Is an external unit specifier.
`label`	(Input) Is the label of the branch target statement that receives control if an error occurs.
`msg-var`	(Output) Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs.
`i-var`	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

The FLUSH statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT= keyword is optional if the unit specifier is the first item in the I/O control list.

This statement has no effect on file position.

FLUSH Subroutine

Portability Subroutine: *Flushes the contents of an external unit buffer into its associated file.*

Module

USE IFPORT

Syntax

CALL FLUSH (*lunit*)

lunit	(Input) INTEGER(4). Number of the external unit to be flushed. Must be currently connected to a file when the subroutine is called. This routine is thread-safe, and locks the associated stream before I/O is performed.

NOTE
The flush is performed in a non-blocking mode. In this mode, the command may return before the physical write is completed. If you want to use a blocking mode of FLUSH use COMMITQQ.

See Also
COMMITQQ

FMA and NOFMA

General Compiler Directives: *FMA tells the compiler to allow generation of fused multiply-add (FMA) instructions, also known as floating-point contractions. NOFMA disables the generation of FMA instructions.*

Syntax

```
!DIR$ FMA
```

```
!DIR$ NOFMA
```

These directives affect the current program unit but they also apply to subsequent program units in the source file, unless and until a program unit containing another FMA or NOFMA directive is encountered.

Once a NOFMA directive has been specified, it is in effect from that point forward. The setting impacts later routines in the source file. For example, consider that the following are in the same source file:

```
    real function fms_mul2( a,  b,  c,  d)
      implicit none
      real :: a, b, c, d
!       the default !dir$ fma leads to fma generation:
!       this is transformed into FMS(a,b,FMA(c,d,0))

!dir$ nofma
      fms_mul2 = a*b - c*d               ! no fma generation here
    end function fms_mul2

    real function fms_mul( a,  b,  c,  d)
      implicit none
      real :: a, b, c, d
      fms_mul = (a*b) - (c*d)            ! no fma generation here
    end function fms_mul
```

The NOFMA directive specified in `fms_mul2` will also impact the routine `fms_mul`. You must explicitly specify the FMA directive to override the effect of the NOFMA directive.

See Also
General Compiler Directives
Syntax Rules for Compiler Directives
`fma`, `Qfma` compiler option

`fp-model strict, fp:strict` compiler option

FOCUSQQ (W*S)

QuickWin Function: *Sets focus to the window with the specified unit number.*

Module

USE IFQWIN

Syntax

`result = FOCUSQQ (iunit)`

iunit	(Input) INTEGER(4). Unit number of the window to which the focus is set. Unit numbers 0, 5, and 6 refer to the default startup window.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

Units 0, 5, and 6 refer to the default window only if the program does not specifically open them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

Unlike SETACTIVEQQ, FOCUSQQ brings the specified unit to the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling SETACTIVEQQ.

A window has focus when it is given the focus by FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when on I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

By default IOFOCUS=.TRUE., except for child windows opened with as unit *. If IOFOCUS=.TRUE., the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

See Also

SETACTIVEQQ
INQFOCUSQQ

FOR__SET_FTN_ALLOC

Run-Time Function: *Lets you specify your own routine to dynamically allocate common blocks. This function is especially useful when you are sharing libraries.*

Syntax

`result = FOR__SET_FTN_ALLOC(alloc_routine)`

alloc_routine	(Input) Character. Is the name of a user-defined allocation routine. The routine takes the same arguments as the routine prototype _FTN_ALLOC, which is defined in module IFCORE. For more information on _FTN_ALLOC and its arguments, see Allocating Common Blocks.

Results

The result has the same type as the type of the argument. The return value is a pointer to the previous allocation routine you specified for allocation of COMMONs, or to a null pointer if you did not specify a previous allocation routine.

The caller of FOR__SET_FTN_ALLOC must include a USE IFCOMMONALLOC statement. The allocation routine should include ISO_C_BINDING so it can correctly declare the arguments.

This function takes precedence over _FTN_ALLOC.

Example

The following shows an example of a user-defined routine that can be used with FOR__SET_FTN_ALLOC.

Note that you must compile the program using option `[Q]dyncom` to name the commons you want to be dynamically allocated.

```
! User's allocation routine
!
subroutine my_Fortran_alloc_routine (mem, size, name)
    use, intrinsic               :: ISO_C_BINDING
    implicit none

    type(C_PTR),    intent(OUT)         :: mem
    integer(C_INT), intent(INOUT)       :: size
    character, dimension(*), intent(IN) :: name

    ! Users would put their allocation code here.  This example text
    ! does not contain code to allocate memory.

end subroutine my_Fortran_alloc_routine

! This routine uses module IFCOMMONALLOC to swap allocation
! routines for dynamic COMMONs.
!
subroutine swap_alloc_routines( for_old )
    use ifcommonalloc
    use, intrinsic :: ISO_C_BINDING
    implicit none

    logical for_old

    ! The routine to use, defined above.
    !
    procedure(alloc_rtn) :: my_Fortran_alloc_routine

    ! Where to save the old one.
    !
    type(C_FUNPTR) :: saved_alloc_routine

    ! Do the swap
    !
    print *, "my_Fortran_alloc_routine"
    if (for_old) then
        saved_alloc_routine = set_ftn_alloc( C_FUNLOC(my_Fortran_alloc_routine) )
    else
        saved_alloc_routine = set_ftn_alloc( saved_alloc_routine )
    end if
```

```
end subroutine swap_alloc_routines


! Routines with dynamic commons would go here


! The main program doesn't need to know about module IFCOMMONALLOC.
!
program main
    implicit none

    ! Dynamic commons in routines first called in this region will use the
    !   default allocation method.

    swap_alloc_routines( .true. )

    ! Dynamic commons in routines first called in this region will use
    !   my_Fortran_alloc_routine.

    swap_alloc_routines( .false. )

    ! Dynamic commons in routines first called in this region will use the
    !   default allocation method.

end program main
```

See Also

Allocating Common Blocks

`dyncom, Qdyncom` compiler option

ISO_C_BINDING

FOR_DESCRIPTOR_ASSIGN (W*S)

Run-Time Subroutine: *Creates an array descriptor in memory.*

Module

USE IFCORE

Syntax

`CALL FOR_DESCRIPTOR_ASSIGN (dp,base,size,reserved,rank,dims_info)`

dp	(Input) A Fortran pointer to an array; the array can be of any data type.
base	(Input) INTEGER(4) or INTEGER(8). The base address of the data being described by *dp*. Note that a Fortran pointer describes both the location and type of the data item.
size	(Input) INTEGER(4). The size of the data type; for example, 4 for INTEGER(4).

reserved

(Input) INTEGER(4). A logical bitwise OR combination of the following constants, which are defined in `IFCORE.F90`:

- FOR_DESCRIPTOR_ARRAY_DEFINED - Specifies whether the array pointed to has been allocated or associated. If the bit is set, the array has been allocated or associated.
- FOR_DESCRIPTOR_ARRAY_NODEALLOC - Specifies whether the array points to something that can be deallocated by a call to DEALLOCATE, or whether it points to something that cannot be deallocated. For example:

```
integer, pointer :: p(:)
integer, target :: t
p => t   ! t cannot be deallocated
allocate(p(10)) ! t can be deallocated
```

  If the bit is set, the array cannot be deallocated.
- FOR_DESCRIPTOR_ARRAY_CONTIGUOUS - Specifies whether the array pointed to is completely contiguous in memory or whether it is a slice that is not contiguous. If the bit is set, the array is contiguous.

rank

(Input) INTEGER(4). The rank of the array pointed to.

dims_info

(Input) An array of derived type `FOR_DIMS_INFO`; you must specify a rank for this array. The derived type `FOR_DIMS_INFO` is defined in IFCORE.F90 as follows:

```
TYPE FOR_DIMS_INFO
  INTEGER(4) LOWERBOUND  !Lower bound for the dimension
  INTEGER(4) UPPERBOUND  !Upper bound for the dimension
  INTEGER(4) STRIDE      !Stride for the dimension
END TYPE FOR_DIMS_INFO
```

The FOR_DESCRIPTOR_ASSIGN routine is similar to a Fortran pointer assignment, but gives you more control over the assignment, allowing, for example, assignment to any location in memory.

You can also use this routine to create an array that can be used from both Fortran or C.

Example

```
use IFCORE
common/c_array/ array
real(8) array(5,5)
external  init_array
external  c_print_array
real(8),pointer :: p_array(:,:)
type(FOR_DIMS_INFO) dims_info(2)
 call init_array()

 do i=1,5
   do j=1,5
      print *,i,j, array(i,j)
   end do
 end do

 dims_info(1)%LOWERBOUND = 11
 dims_info(1)%UPPERBOUND = 15
 dims_info(1)%STRIDE = 1
```

```
 dims_info(2)%LOWERBOUND = -5
 dims_info(2)%UPPERBOUND = -1
 dims_info(2)%STRIDE = 1

 call FOR_DESCRIPTOR_ASSIGN(p_array, &
   LOC(array), &
   SIZEOF(array(1,1)), &
   FOR_DESCRIPTOR_ARRAY_DEFINED .or. &
   FOR_DESCRIPTOR_ARRAY_NODEALLOC .or. &
   FOR_DESCRIPTOR_ARRAY_CONTIGUOUS,  &
   2, &
   dims_info )

p_array = p_array + 1
call c_print_array()
end
```

The following shows the C program containing init_array and c_print_array:

```
#include <stdio.h>
#if !defined(_WIN32) && !defined(_WIN64)
#define C_ARRAY c_array_
#define INIT_ARRAY init_array_
#define C_PRINT_ARRAY c_print_array_
#endif
double C_ARRAY[5][5];
void INIT_ARRAY(void);
void C_PRINT_ARRAY(void);
void INIT_ARRAY(void)
{
  int i,j;
  for(i=0;i<5;i++)
     for(j=0;j<5;j++)
       C_ARRAY[i][j] = j + 10*i;
}
void C_PRINT_ARRAY(void)
{
   int i,j;
   for(i=0;i<5;i++){
     for(j=0;j<5;j++)
         printf("%f ", C_ARRAY[i][j]);
     printf("\n");
   }
}
```

See Also

POINTER - Fortran

FOR_GET_FPE

Run-Time Function: *Returns the current settings of floating-point exception flags. This routine can be called from a C or Fortran program.*

Module

USE IFCORE

Syntax

```
result = FOR_GET_FPE( )
```

Results

The result type is INTEGER(4). The return value represents the settings of the current floating-point exception flags. The meanings of the bits are defined in the IFPORT module file.

To set floating-point exception flags after program initialization, use FOR_SET_FPE.

Example

```
USE IFCORE
INTEGER*4 FPE_FLAGS
FPE_FLAGS = FOR_GET_FPE ( )
```

See Also

FOR_SET_FPE

FOR_IFCORE_VERSION

Portability Function: *Returns the version of the Fortran run-time library (ifcore).*

Module

USE IFPORT

Syntax

```
result = FOR_IFCORE_VERSION (string)
```

string	(Output) Character*(*). The version information for the Fortran run-time library (ifcore).

Results

The result type is INTEGER(4). The result is non-zero if successful; otherwise, zero.

If *string* is not long enough to contain the version information, the result is truncated on the right. If *string* is longer than the version information, the result is blank-padded on the right. The result may contain multiple blank or ASCII tab characters.

Example

Consider the following:

```
program what_ifcore
    use ifport
    integer       :: res
    character*56  :: str

    res = for_ifcore_version( str )
    print ("(3A)"), "'", str, "'"

end program what_ifcore
```

The above example will produce a result similar to the following, depending on spacing and the actual version information:

```
'Intel Fortran RTL Core Library Vvv.m-eeeMmm dd yyyy    '
```

where:

vv	Is a major version number
m	Is a minor version number
eee	Is an edit number
Mmm	Is a three-character abbreviation for the month
dd	Is a two-digit day of the month
yyyy	Is a four-digit year

See Also

FOR_IFCORE_VERSION

FOR_IFPORT_VERSION

Portability Function: *Returns the version of the Fortran portability library (ifport).*

Module

USE IFPORT

Syntax

```
result = FOR_IFPORT_VERSION (string)
```

string	(Output) Character*(*). The version information for the Fortran portability library (ifport).

Results

The result type is INTEGER(4). The result is non-zero if successful; otherwise, zero.

If *string* is not long enough to contain the version information, the result is truncated on the right. If string is longer than the version information, the result is blank-padded on the right. The result may contain multiple blank or ASCII tab characters.

Example

Consider the following:

```
program what_ifport
    use ifport
    integer       :: res
    character*56  :: str

    res = for_ifport_version( str )
    print ("(3A)"), "'", str, "'"

end program what_ifport
```

The above example will produce a result similar to the following, depending on spacing and the actual version information:

```
'Intel Fortran portability library      Vvv.m-eeeMmm dd yyyy '
```

where:

vv	Is a major version number

m	Is a minor version number
eee	Is an edit number
Mmm	Is a three-character abbreviation for the month
dd	Is a two-digit day of the month
yyyy	Is a four-digit year

See Also

FOR_IFPORT_VERSION

FOR_LFENCE

Run-Time Subroutine: *Inserts a memory load fence instruction that ensures completion of preceding load instructions.*

Module

USE IFCORE

Syntax

```
CALL FOR_LFENCE ()
```

Using this subroutine guarantees that in program order, every load instruction that precedes the load fence instruction is globally visible before any load instruction that follows the load fence.

FOR_MFENCE

Run-Time Subroutine: *Inserts a memory fence instruction that ensures completion of all preceding load and store instructions.*

Module

USE IFCORE

Syntax

```
CALL FOR_MFENCE ()
```

Using this subroutine guarantees that in program order, every load and store instruction that precedes the memory fence instruction is globally visible before any load or store instruction that follows the memory fence can proceed.

for_rtl_finish_

Run-Time Function: *Cleans up the Fortran run-time environment; for example, flushing buffers and closing files. It also issues messages about floating-point exceptions, if any occur.*

Syntax

This routine should be called from a C main program; it is invoked by default from a Fortran main program.

```
result = for_rtl_finish_ ( )
```

Results

The result is an I/O status value. For information on these status values, see *Error Handling*: Using the IOSTAT Value and Fortran Exit Codes.

To initialize the Fortran run-time environment, use function for_rtl_init_ .

Example

Consider the following C code:

```
int io_status;
int for_rtl_finish_ ( );
io_status = for_rtl_finish_ ( );
```

See Also

for_rtl_init_

for_rtl_init_

Run-Time Subroutine: *Initializes the Fortran run-time environment and causes Fortran procedures and subroutines to behave the same as when called from a Fortran main program. On Linux* and macOS*, it also establishes handlers and floating-point exception handling.*

Syntax

This routine should be called from a C main program; it is invoked by default from a Fortran main program.

```
CALL for_rtl_init_ (argcount,actarg)
```

argcount	Is a command-line parameter describing the argument count.
actarg	Is a command-line parameter describing the actual arguments.

To clean up the Fortran run-time environment, use function for_rtl_finish_.

Example

Consider the following C code:

```
int argc;
char **argv;
void for_rtl_init_ (int *, char **);
for_rtl_init_ (&argc, argv);
```

See Also

for_rtl_finish_

FOR_SET_FPE

Run-Time Function: *Sets the floating-point exception flags. This routine can be called from a C or Fortran program.*

Module

USE IFCORE

Syntax

```
result = FOR_SET_FPE (a)
```

a	Must be of type INTEGER(4). It contains bit flags controlling floating-point exception trapping, reporting, and result handling.

Results

The result type is INTEGER(4). The return value represents the previous settings of the floating-point exception flags. The meanings of the bits are defined in the IFCORE module file.

To get the current settings of the floating-point exception flags, use FOR_GET_FPE.

Example

```
USE IFCORE
INTEGER*4 OLD_FPE_FLAGS, NEW_FPE_FLAGS
OLD_FPE_FLAGS = FOR_SET_FPE (NEW_FPE_FLAGS)
```

The following example program is compiled without any `fpe` options. However, it uses calls to `for_set_fpe` to enable the same flags as when compiling with option `fpe:0`. The new flags can be verified by compiling the program with option `-fpe:0`.

```
   program samplefpe
   use ifcore
   implicit none

   INTEGER(4) :: ORIGINAL_FPE_FLAGS, NEW_FPE_FLAGS
   INTEGER(4) :: CURRENT_FPE_FLAGS, PREVIOUS_FPE_FLAGS

   NEW_FPE_FLAGS = FPE_M_TRAP_UND + FPE_M_TRAP_OVF + FPE_M_TRAP_DIV0  &
     + FPE_M_TRAP_INV + FPE_M_ABRUPT_UND + FPE_M_ABRUPT_DMZ
   ORIGINAL_FPE_FLAGS = FOR_SET_FPE (NEW_FPE_FLAGS)
   CURRENT_FPE_FLAGS = FOR_GET_FPE ()

   print *,"The original FPE FLAGS were:"
   CALL PRINT_FPE_FLAGS(ORIGINAL_FPE_FLAGS)

   print *," "   print *,"The new FPE FLAGS are:"
   CALL PRINT_FPE_FLAGS(CURRENT_FPE_FLAGS)

!! restore the fpe flag to their original values
   PREVIOUS_FPE_FLAGS = FOR_SET_FPE (ORIGINAL_FPE_FLAGS)

   end

   subroutine PRINT_FPE_FLAGS(fpe_flags)
   use ifcore
   implicit none
   integer(4)   :: fpe_flags
   character(3) :: toggle

   print 10, fpe_flags, fpe_flags
10 format(X,'FPE FLAGS = 0X',Z8.8," B'",B32.32)

   if ( IAND(fpe_flags, FPE_M_TRAP_UND) .ne. 0 ) then
     toggle = "ON"
   else
     toggle = "OFF"
```

```
endif
write(*,*) "  FPE_TRAP_UND    :", toggle

if ( IAND(fpe_flags, FPE_M_TRAP_OVF) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_OVF    :", toggle

if ( IAND(fpe_flags, FPE_M_TRAP_DIV0) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_DIV0   :", toggle

if ( IAND(fpe_flags, FPE_M_TRAP_INV) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_INV    :", toggle

if ( IAND(fpe_flags, FPE_M_ABRUPT_UND) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_UND  :", toggle

if ( IAND(fpe_flags, FPE_M_ABRUPT_OVF) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_OVF  :", toggle

if ( IAND(fpe_flags, FPE_M_ABRUPT_DMZ) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_DIV0 :", toggle

if ( IAND(fpe_flags, FPE_M_ABRUPT_DIV0) .ne. 0 ) then
  toggle = "ON"
else
  toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_INV  :", toggle

if ( IAND(fpe_flags, FPE_M_ABRUPT_DMZ) .ne. 0 ) then ! ABRUPT_DMZ
  toggle = "ON"
else
  toggle = "OFF"
endif
```

```
   write(*,*) "  FPE_ABRUPT_DMZ  :", toggle, "  (ftz related)"

   end subroutine PRINT_FPE_FLAGS
```

The following shows the output from the above program:

```
>ifort set_fpe_sample01.f90
>set_fpe_sample01.exe
 The original FPE FLAGS were:
 FPE FLAGS = 0X00000000 B'00000000000000000000000000000000
   FPE_TRAP_UND    :OFF
   FPE_TRAP_OVF    :OFF
   FPE_TRAP_DIV0   :OFF
   FPE_TRAP_INV    :OFF
   FPE_ABRUPT_UND  :OFF
   FPE_ABRUPT_OVF  :OFF
   FPE_ABRUPT_DIV0 :OFF
   FPE_ABRUPT_INV  :OFF
   FPE_ABRUPT_DMZ  :OFF (ftz related)

 The new FPE FLAGS are:
 FPE FLAGS = 0X0011000F B'00000000000100010000000000001111
   FPE_TRAP_UND    :ON
   FPE_TRAP_OVF    :ON
   FPE_TRAP_DIV0   :ON
   FPE_TRAP_INV    :ON
   FPE_ABRUPT_UND  :ON
   FPE_ABRUPT_OVF  :OFF
   FPE_ABRUPT_DIV0 :ON
   FPE_ABRUPT_INV  :OFF
   FPE_ABRUPT_DMZ  :ON (ftz related)
```

The following example builds a library that has to have a particular setting of the fpe flags internally, and has to work with user programs built with any combination of the fpe flags.

```
    !-- file USE.F90 starts here
subroutine use_subnorms
    use, intrinsic :: ieee_arithmetic
    use ifcore

    use, intrinsic :: ieee_features, only: ieee_subnormal
    implicit none

    !--- Declaration for use in example code
        real, volatile ::     x, y
        integer  i
    !--- End declarations for example code

    integer(4) :: orig_flags, off_flags, not_flags, new_flags

    if (ieee_support_subnormal()) then
        print *, "Subnormals already supported"
    else
        orig_flags = for_get_fpe()

        off_flags = IOR(FPE_M_ABRUPT_UND, FPE_M_ABRUPT_DMZ)
        off_flags = IOR(off_flags,        FPE_M_TRAP_UND)
        off_flags = IOR(off_flags,        FPE_M_MSG_UND)
        not_flags = NOT(off_flags)        ! "INOT" is the 16-bit version!
```

```
        new_flags  = IAND(orig_flags, not_flags)
        orig_flags = for_set_fpe(new_flags)

        if (ieee_support_subnormal()) then
            print *, "Subnormals are now supported"
        else
            print *, "Error: Subnormals still not supported after FOR_SET_FPE call"
        end if
    end if

    !-- Begin example of user code using subnorms
1      FORMAT(1X,Z)
2      FORMAT("Use subnormals",1X,E40.25)

        x = 0.0
        y = tiny(x)

        ! Print as real values
        !        print 2, x, y

        ! Expect non-zero numbers
        !
        do i = 1, 20
            y = y / 2.0
            print 1,y
        enddo
    !-- End example of user code using subnorms

 end subroutine use_subnorms
    !-- end of file USE.F90

    !-- File FLUSH.F90 starts here
subroutine flush_subnorms
    use, intrinsic :: ieee_arithmetic
    use ifcore
    use, intrinsic :: ieee_features
    implicit none

    !--- Declaration for use in example code
        real, volatile ::     x, y
        integer  i
    !--- End declarations for example code

    integer(4) :: orig_flags, off_flags, new_flags

    if (ieee_support_subnormal()) then
        print *, "Subnormals already supported; turn off"
        orig_flags = for_get_fpe()

        off_flags = IOR(FPE_M_ABRUPT_UND, FPE_M_ABRUPT_DMZ)
        off_flags = IOR(off_flags,        FPE_M_TRAP_UND)
        off_flags = IOR(off_flags,        FPE_M_MSG_UND)

        new_flags  = IOR(orig_flags, off_flags)
        orig_flags = for_set_fpe(new_flags)

        if (ieee_support_subnormal()) then
```

```
            print *, "Error: Subnormals still supported after FOR_SET_FPE call"
        else
            print *, "Subnormals are now NOT supported, should flush to zero"
        end if

    else
        print *, "Subnormals already not supported"
    end if

    !-- Begin example of user code doing flush-to-zero

1      FORMAT(1X,Z)
2      FORMAT("Flush to zero",1X,E40.25)

        x = 0.0
        y = tiny(x)

        ! Print as real values
        !
        print 2, x, y

        ! Expect zeros
        !
        do i = 1, 20
            y = y / 2.0
            print 1,y
        enddo

    !-- End example of user code doing flush-to-zero

end subroutine flush_subnorms
    !-- end of file FLUSH.F90

    !-- File MAIN.F90 starts here
program example
    implicit none

    call use_subnorms      ! Will use subnorms
    call flush_subnorms    ! Will flush
    call use_subnorms      ! Will also flush, but WON'T use subnorms!

end program example
    !-- end of file MAIN.F90
```

You can specify the following lines to compile and link the above program:

```
ifort -c -fpic -no-ftz -fpe3   use.f90
ifort -c -fpic -ftz            flush.f90
ifort -c -fpic                 main.f90
ifort -o main.exe main.o use.o flush.o
```

FOR_SET_REENTRANCY

Run-Time Function: *Controls the type of reentrancy protection that the Fortran Run-Time Library (RTL) exhibits. This routine can be called from a C or Fortran program.*

Module

USE IFCORE

Syntax

```
result = FOR_SET_REENTRANCY (mode)
```

mode Must be of type INTEGER(4) and contain one of the following options:

FOR_K_REENTRANCY_NONE	Tells the Fortran RTL to perform simple locking around critical sections of RTL code. This type of reentrancy should be used when the Fortran RTL will *not* be reentered due to asynchronous system traps (ASTs) or threads within the application.
FOR_K_REENTRANCY_ASYNCH	Tells the Fortran RTL to perform simple locking and disables ASTs around critical sections of RTL code. This type of reentrancy should be used when the application contains AST handlers that call the Fortran RTL.
FOR_K_REENTRANCY_THREADED	Tells the Fortran RTL to perform thread locking. This type of reentrancy should be used in multithreaded applications.
FOR_K_REENTRANCY_INFO	Tells the Fortran RTL to return the current reentrancy mode.

Results

The result type is INTEGER(4). The return value represents the previous setting of the Fortran Run-Time Library reentrancy mode, unless the argument is FOR_K_REENTRANCY_INFO, in which case the return value represents the current setting.

You must be using an RTL that supports the level of reentrancy you desire. For example, FOR_SET_REENTRANCY ignores a request for thread protection (FOR_K_REENTRANCY_THREADED) if you do not build your program with the thread-safe RTL.

Example

```
PROGRAM SETREENT
USE IFCORE
INTEGER*4    MODE
CHARACTER*10 REENT_TXT(3) /'NONE    ','ASYNCH  ','THREADED'/
PRINT*,'Setting Reentrancy mode to ',REENT_TXT(MODE+1)
MODE = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_NONE)
PRINT*,'Previous Reentrancy mode was ',REENT_TXT(MODE+1)
MODE = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_INFO)
PRINT*,'Current Reentrancy mode is ',REENT_TXT(MODE+1)
END
```

FOR_SFENCE

Run-Time Subroutine: *Inserts a memory store fence instruction that ensures completion of preceding store instructions.*

Module

USE IFCORE

Syntax

```
CALL FOR_SFENCE ()
```

Using this subroutine guarantees that in program order, every store instruction that precedes the store fence instruction is globally visible before any store instruction that follows the store fence.

FORALL

Statement and Construct: *The FORALL statement and construct is an element-by-element generalization of the masked array assignment in the WHERE statement and construct. It allows more general array shapes to be assigned, especially in construct form. The FORALL construct is an obsolescent language feature in Fortran 2018.*

Syntax

Statement:

```
FORALL (triplet-spec[, triplet-spec] ...[, mask-expr]) assign-stmt
```

Construct:

```
[name:] FORALL (triplet-spec[, triplet-spec] ...[, mask-expr])
   forall-body-stmt
   [forall-body-stmt]...
END FORALL [name]
```

triplet-spec

Is a triplet specification with the following form:

```
[type::] subscript-name = subscript-1: subscript-2[:
stride]
```

The *type* is an optional integer data type. If *type* appears, the *subscript-name* has the specified type and type parameters. Otherwise, it has the type and type parameters that it would have if it were the name of a variable in the innermost executable construct or scoping unit.

If *type* does not appear, the *subscript-name* must not be the same as a local identifier, an accessible global identifier, or an identifier of an outer construct entity, except for a common block name or a scalar variable name.

The *subscript-name* is a scalar of type integer. It is valid only within the scope of the FORALL; its value is undefined on completion of the FORALL.

The *subscript*s and *stride* cannot contain a reference to any *subscript-name* in *triplet-spec*.

The *stride* cannot be zero. If it is omitted, the default value is 1.

Evaluation of an expression in a triplet specification must not affect the result of evaluating any other expression in another triplet specification.

mask-expr Is a logical array expression (called the mask expression). If it is omitted, the value .TRUE. is assumed. The mask expression can reference the subscript name in *triplet-spec*.

Description

If a construct name is specified in the FORALL statement, the same name must appear in the corresponding END FORALL statement.

A FORALL statement is executed by first evaluating all bounds and stride expressions in the triplet specifications, giving a set of values for each subscript name. The FORALL assignment statement is executed for all combinations of subscript name values for which the mask expression is true.

The FORALL assignment statement is executed as if all expressions (on both sides of the assignment) are completely evaluated before any part of the left side is changed. Valid values are assigned to corresponding elements of the array being assigned to. No element of an array can be assigned a value more than once.

A FORALL construct is executed as if it were multiple FORALL statements, with the same triplet specifications and mask expressions. Each statement in the FORALL body is executed completely before execution begins on the next FORALL body statement.

Any procedure referenced in the mask expression or FORALL assignment statement must be pure.

Pure functions can be used in the mask expression or called directly in a FORALL statement. Pure subroutines cannot be called directly in a FORALL statement, but can be called from other pure procedures.

Starting with Fortran 2018, the FORALL statement and construct are obsolescent features.

Example

The following example, which is not expressible using array syntax, sets diagonal elements of an array to 1:

```
REAL, DIMENSION(N, N) :: A
FORALL (I=1:N)  A(I, I) = 1
```

Consider the following:

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0) B(I, J) = 1.0 / A(I, J)
```

This statement takes the reciprocal of each nonzero element of array A(1:N, 1:N) and assigns it to the corresponding element of array B. Elements of A that are zero do not have their reciprocal taken, and no assignments are made to corresponding elements of B.

Every array assignment statement and WHERE statement can be written as a FORALL statement, but some FORALL statements cannot be written using just array syntax. For example, the preceding FORALL statement is equivalent to the following:

```
WHERE(A /= 0.0) B = 1.0 / A
```

However, the following FORALL example cannot be written using just array syntax:

```
FORALL(I = 1:N, J = 1:N) H(I, J) = 1.0/REAL(I + J - 1)
```

This statement sets array element H(I, J) to the value 1.0/REAL(I + J - 1) for values of I and J between 1 and N.

Consider the following:

```
TYPE MONARCH
  INTEGER, POINTER :: P
END TYPE MONARCH
TYPE(MONARCH), DIMENSION(8) :: PATTERN
INTEGER, DIMENSION(8), TARGET :: OBJECT
FORALL(J=1:8) PATTERN(J)%P => OBJECT(1+IEOR(J-1,2))
```

This FORALL statement causes elements 1 through 8 of array PATTERN to point to elements 3, 4, 1, 2, 7, 8, 5, and 6, respectively, of OBJECT. IEOR can be referenced here because it is pure.

The following example shows a FORALL construct:

```
FORALL(I = 3:N + 1, J = 3:N + 1)
  C(I, J) = C(I, J + 2) + C(I, J - 2) + C(I + 2, J) + C(I - 2, J)
  D(I, J) = C(I, J)
END FORALL
```

The assignment to array D uses the values of C computed in the first statement in the construct, not the values before the construct began execution.

See Also

WHERE

FORMAT

Statement: *Specifies the form of data being transferred and the data conversion (editing) required to achieve that form.*

Syntax

FORMAT (*format-list*)

format-list

Is a list of one or more of the following edit descriptors, separated by commas or slashes (/):

Data edit descriptors:	I, B, O, Z, F, E, EN, ES, D, G, L, and A.
Control edit descriptors:	T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, $, \, and Q.
String edit descriptors:	H, 'c', and "c", where *c* is a character constant.

A comma can be omitted in the following cases:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor
- Before a slash (/) edit descriptor when the optional repeat specification is not present
- After a slash (/) edit descriptor
- Before or after a colon (:) edit descriptor

Edit descriptors can be nested and a *repeat specification* can precede data edit descriptors, the slash edit descriptor, or a parenthesized list of edit descriptors.

Description

A FORMAT statement must be labeled.

Named constants are not permitted in format specifications.

If the associated I/O statement contains an I/O list, the format specification must contain at least one data edit descriptor or the control edit descriptor Q.

Blank characters can precede the initial left parenthesis, and additional blanks can appear anywhere within the format specification. These blanks have no meaning unless they are within a character string edit descriptor.

When a formatted input statement is executed, the setting of the BLANK specifier (for the relevant logical unit) determines the interpretation of blanks within the specification. If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks. (For more information on BLANK defaults, see the OPEN statement.

For formatted input, you can use the comma as an external field separator. The comma terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It can also designate null (zero-length) fields.

The first character of a record transmitted to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a blank, 0, 1, $,+, or ASCII NUL. Any other character is treated as a blank.

A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.

Whenever an edit descriptor requires an integer constant, you can specify an expression enclosed in angle brackets (< and >). For more information, see Variable Format Expressions.

The integer expression can be any valid Fortran expression, including function calls and references to dummy arguments, with the following restrictions:

- Expressions cannot be used with the H edit descriptor.
- Expressions cannot contain graphical relational operators (such as > and <).

The value of the expression is reevaluated each time an input/output item is processed during the execution of the READ, WRITE, or PRINT statement.

The following tables summarize the different kinds of edit descriptors:

Data Edit Descriptors

Code	Form [1]	Effect
A	A[w]	Transfers character or Hollerith values.
B	Bw[.m]	Transfers binary values.
D	Dw.d	Transfers real values with D exponents.
E	Ew.d[Ee]	Transfers real values with E exponents.
EN	ENw.d[Ee]	Transfers real values with engineering notation.
ES	ESw.d[Ee]	Transfers real values with scientific notation.
F	Fw.d	Transfers real values with no exponent.
G	Gw.d[Ee]	Transfers values of all intrinsic types.
I	Iw[.m]	Transfers decimal integer values.

Code	Form [1]	Effect
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
Z	Zw[.m]	Transfers hexadecimal values.

[1] *w* is the field width.

m is the minimum number of digits that must be in the field (including zeros).

d is the number of digits to the right of the decimal point.

E is the exponent field.

e is the number of digits in the exponent.

Control Edit Descriptors

Code	Form	Effect
BN	BN	Ignores embedded and trailing blanks in a numeric input field.
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.
P	kP	Interprets certain real numbers with a specified scale factor.
Q	Q	Returns the number of characters remaining in an input record.
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.
SP	SP	Writes optional plus sign (+) into numeric output fields.
SS	SS	Suppresses optional plus sign (+) in numeric output fields.
T	Tn	Tabs to specified position.
TL	TLn	Tabs left the specified number of positions.
TR	TRn	Tabs right the specified number of positions.
X	nX	Skips the specified number of positions.
$	$	Suppresses trailing carriage return during interactive I/O.

Code	Form	Effect
:	:	Terminates format control if there are no more items in the I/O list.
/	[r]/	Terminates the current record and moves to the next record.
\	\	Continues the same record; same as $.

String Edit Descriptors

Code	Form	Effect
H	nHch[ch...]	Transfers characters following the H edit descriptor to an output record.
'c' [2]	'c'	Transfers the character literal constant (between the delimiters) to an output record.

[2] These delimiters can also be quotation marks (").

Example

```
   INTEGER width, value
   width = 2
   read (*,1) width, value
!  if the input is 3123, prints 123, not 12
1  format ( i1, i<width>)
   print *, value
   END
```

See Also

I/O Formatting
Format Specifications
Variable Format Expressions
Data Edit Descriptors

FP_CLASS

Elemental Intrinsic Function (Generic): *Returns the class of an IEEE* real (binary32, binary64, and binary128) argument. This function cannot be passed as an actual argument.*

Syntax

```
result = FP_CLASS (x)
```

x (Input) Must be of type real.

Results

The result type is INTEGER(4). The return value is one of the following:

Class of Argument	Return Value
Signaling NaN	FOR_K_FP_SNAN
Quiet NaN	FOR_K_FP_QNAN
Positive Infinity	FOR_K_FP_POS_INF
Negative Infinity	FOR_K_FP_NEG_INF
Positive Normalized Number	FOR_K_FP_POS_NORM
Negative Normalized Number	FOR_K_FP_NEG_NORM
Positive Subnormal Number	FOR_K_FP_POS_DENORM
Negative Subnormal Number	FOR_K_FP_NEG_DENORM
Positive Zero	FOR_K_FP_POS_ZERO
Negative Zero	FOR_K_FP_NEG_ZERO

The preceding return values are defined in file `for_fpclass.for`.

Example

FP_CLASS (4.0_8) has the value 4 (FOR_K_FP_POS_NORM).

FPUTC

Portability Function: *Writes a character to the file specified by a Fortran external unit, bypassing normal Fortran input/output.*

Module

USE IFPORT

Syntax

`result = FPUTC (lunit,char)`

lunit	(Input) INTEGER(4). Unit number of a file.
char	(Output) Character*(*). Variable whose value is to be written to the file corresponding to *lunit*.

Results

The result type is INTEGER(4). The result is zero if the write was successful; otherwise, an error code, such as:

EINVAL - The specified unit is invalid (either not already open, or an invalid unit number)

If you use WRITE, READ, or any other Fortran I/O statements with *lunit*, be sure to read Input and Output Routines in Overview of Portability Routines.

Example

```
use IFPORT
integer*4 lunit, i4
character*26 string
character*1 char1
```

```
lunit = 1
open (lunit,file = 'fputc.dat')
do i = 1,26
  char1 = char(123-i)
  i4 = fputc(1,char1)        !make valid writes
  if (i4.ne.0) iflag = 1
enddo
rewind (1)
read (1,'(a)') string
print *, string
```

See Also

I/O Formatting

FRACTION

Elemental Intrinsic Function (Generic): *Returns the fractional part of the model representation of the argument value.*

Syntax

`result = FRACTION (x)`

x	(Input) Must be of type real.

Results

The result type and kind are the same as *x*.

The result has the value *x** b ^{e}. Parameters *b* and *e* are defined in Model for Real Data.

If *x* has the value zero, the result has the value zero.

Example

If 3.0 is a REAL(4) value, FRACTION (3.0) has the value 0.75.

The following shows another example:

```
REAL result
result = FRACTION(3.0) ! returns 0.75
result = FRACTION(1024.0) ! returns 0.5
```

See Also

DIGITS
RADIX
EXPONENT
Data Representation Models

FREE

Intrinsic Subroutine (Specific): *Frees a block of memory that is currently allocated. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL FREE (addr)`

addr	(Input) Must be of type INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. This value is the starting address of the memory block to be freed, previously allocated by MALLOC.

If the freed address was not previously allocated by MALLOC, or if an address is freed more than once, results are unpredictable.

Example

```
  INTEGER(4) SIZE
  REAL(4) STORAGE(*)
  POINTER (ADDR, STORAGE)      ! ADDR will point to STORAGE
  SIZE = 1024                  ! Size in bytes
  ADDR = MALLOC(SIZE)          ! Allocate the memory
  CALL FREE(ADDR)              ! Free it
```

FREEFORM and NOFREEFORM

General Compiler Directives: *FREEFORM specifies that source code is in free-form format. NOFREEFORM specifies that source code is in fixed-form format.*

Syntax

```
!DIR$ FREEFORM
```

```
!DIR$ NOFREEFORM
```

When the FREEFORM or NOFREEFORM directives are used, they remain in effect for the remainder of the program unit, or until the opposite directive is used. When in effect, they apply to include files, but do not affect USE modules, which are compiled separately.

Example

Consider the following:

```
      SUBROUTINE F (A, NX,NY,I1,I2,J1,J2)
! is in column 1 and fixed form
!     X is in column 7
      REAL (8) :: A (NX,NY)
!DIR$ ASSUME_ALIGNED A:32
!DIR$ FREEFORM          ! what follows is freeform
  !DIR$ ASSUME (MOD(NX,8) .EQ. 0)
  ! ensure that the first array access in the loop is aligned
  !DIR$ ASSUME (MOD(I1,8) .EQ. 1)
    DO J=J1,J2
      DO I=I1,I2
        A(I,J) = A(I,J) + A(I,J+1) + A(I,J-1)
      ENDDO
    ENDDO
  !DIR$ NOFREEFORM            ! and now back to fixed form
       END SUBROUTINE F
```

See Also

Source Forms
General Compiler Directives
Syntax Rules for Compiler Directives
free compiler option

Equivalent Compiler Options

FSEEK

Portability Function: *Repositions a file specified by a Fortran external unit.*

Module

USE IFPORT

Syntax

`result = FSEEK (lunit,offset,from)`

lunit (Input) INTEGER(4). External unit number of a file.

offset (Input) INTEGER(4) or INTEGER(8). Offset in bytes, relative to *from*, that is to be the new location of the file marker.

from (Input) INTEGER(4). A position in the file. It must be one of the following:

Value	Variable	Position
0	SEEK_SET	Positions the file relative to the beginning of the file.
1	SEEK_CUR	Positions the file relative to the current position.
2	SEEK_END	Positions the file relative to the end of the file.

Results

The result type is INTEGER(4). The result is zero if the repositioning was successful; otherwise, an error code, such as:

EINVAL: The specified unit is invalid (either not already open, or an invalid unit number), or the *from* parameter is invalid.

The file specified in *lunit* must be open.

Example

```
USE IFPORT
integer(4) istat, offset, ipos
character ichar
OPEN (unit=1,file='datfile.dat')
offset = 5
ipos = 0
istat=fseek(1,offset,ipos)
if (.NOT. stat) then
  istat=fgetc(1,ichar)
  print *, 'data is ',ichar
end if
```

FSTAT

Portability Function: *Returns detailed information about a file specified by a external unit number.*

Module

USE IFPORT

Syntax

`result = FSTAT (lunit,statb)`

lunit

(Input) INTEGER(4). External unit number of the file to examine.

statb

(Output) INTEGER(4) or INTEGER(8). One-dimensional array of size 12; where the system information is stored. The elements of *statb* contain the following values:

Element	Description	Values or Notes
statb(1)	Device the file resides on	W*S: Always 0 L*X: System dependent
statb(2)	File inode number	W*S: Always 0 L*X: System dependent
statb(3)	Access mode of the file	See the table in Results
statb(4)	Number of hard links to the file	W*S: Always 1 L*X: System dependent
statb(5)	User ID of owner	W*S: Always 1 L*X: System dependent
statb(6)	Group ID of owner	W*S: Always 1 L*X: System dependent
statb(7)	Raw device the file resides on	W*S: Always 0 L*X: System dependent
statb(8)	Size of the file	
statb(9)	Time when the file was last accessed[1]	W*S: Only available on non-FAT file systems; undefined on FAT systems

Element	Description	Values or Notes
		L*X: System dependent
statb(10)	Time when the file was last modified[1]	
statb(11)	Time of last file status change[1]	W*S: Same as stat(10) L*X: System dependent
statb(12)	Blocksize for file system I/O operations	W*S: Always 1 L*X: System dependent

[1]Times are in the same format returned by the TIME function (number of seconds since 00:00:00 Greenwich mean time, January 1, 1970).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, returns an error code equal to EINVAL (*lunit* is not a valid unit number, or is not open).

The access mode (the third element of *statb*) is a bitmap consisting of an IOR of the following constants:

Symbolic name	Constant	Description	Notes
S_IFMT	O'0170000'	Type of file	
S_IFDIR	O'0040000'	Directory	
S_IFCHR	O'0020000'	Character special	Never set on Windows* systems
S_IFBLK	O'0060000'	Block special	Never set on Windows systems
S_IFREG	O'0100000'	Regular	
S_IFLNK	O'0120000'	Symbolic link	Never set on Windows systems
S_IFSOCK	O'0140000'	Socket	Never set on Windows systems
S_ISUID	O'0004000'	Set user ID on execution	Never set on Windows systems
S_ISGID	O'0002000'	Set group ID on execution	Never set on Windows systems
S_ISVTX	O'0001000'	Save swapped text	Never set on Windows systems
S_IRWXU	O'0000700'	Owner's file permissions	

Symbolic name	Constant	Description	Notes
S_IRUSR, S_IREAD	O'0000400'	Owner's read permission	Always true on Windows systems
S_IWUSR, S_IWRITE	O'0000200'	Owner's write permission	
S_IXUSR, S_IEXEC	O'0000100'	Owner's execute permission	Based on file extension (.EXE, .COM, .CMD, or .BAT)
S_IRWXG	O'0000070'	Group's file permissions	Same as S_IRWXU on Windows systems
S_IRGRP	O'0000040'	Group's read permission	Same as S_IRUSR on Windows systems
S_IWGRP	O'0000020'	Group's write permission	Same as S_IWUSR on Windows systems
S_IXGRP	O'0000010'	Group's execute permission	Same as S_IXUSR on Windows systems
S_IRWXO	O'0000007'	Other's file permissions	Same as S_IRWXU on Windows systems
S_IROTH	O'0000004'	Other's read permission	Same as S_IRUSR on Windows systems
S_IWOTH	O'0000002'	Other's write permission	Same as S_IWUSR on Windows systems
S_IXOTH	O'0000001'	Other's execute permission	Same as S_IXUSR on Windows systems

STAT returns the same information as FSTAT, but accesses files by name instead of external unit number.

Example

```
USE IFPORT
integer(4) statarray(12), istat
OPEN (unit=1,file='datfile.dat')
ISTAT = FSTAT (1, statarray)
if (.NOT. istat) then
    print *, statarray
end if
```

See Also

INQUIRE
STAT

FTELL, FTELLI8

Portability Functions: *Return the current position of a file.*

Module

USE IFPORT

Syntax

```
result = FTELL (lunit)
result = FTELLI8 (lunit)
```

lunit	(Input) INTEGER(4). External unit number of a file.

Results

The result type is INTEGER(4) for FTELL; INTEGER(8) for FTELLI8. The result is the offset, in bytes, from the beginning of the file. A negative value indicates an error, which is the negation of the IERRNO error code. The following is an example of an error code:

EINVAL: *lunit* is not a valid unit number, or is not open.

FULLPATHQQ

Portability Function: *Returns the full path for a specified file or directory.*

Module

USE IFPORT

Syntax

```
result = FULLPATHQQ (name,pathbuf)
```

name	(Input) Character*(*). Item for which you want the full path. Can be the name of a file in the current directory, a relative directory or file name, or a network uniform naming convention (UNC) path.
pathbuf	(Output) Character*(*). Buffer to receive full path of the item specified in *name*.

Results

The result type is INTEGER(4). The result is the length of the full path in bytes, or 0 if the function fails. This function does not verify that the resulting path and file name are valid nor that they exist.

The length of the full path depends upon how deeply the directories are nested on the drive you are using. If the full path is longer than the character buffer provided to return it (*pathbuf*), FULLPATHQQ returns only that portion of the path that fits into the buffer.

Check the length of the path before using the string returned in *pathbuf*. If the longest full path you are likely to encounter does not fit into the buffer you are using, allocate a larger character buffer. You can allocate the largest possible path buffer with the following statements:

```
USE IFPORT
CHARACTER($MAXPATH) pathbuf
```

$MAXPATH is a symbolic constant defined in `IFPORT.F90` as 260.

Example

```
   USE IFPORT
   USE IFCORE
   CHARACTER($MAXPATH) buf
   CHARACTER(3)        drive
   CHARACTER(256)      dir
   CHARACTER(256)      name
```

```
      CHARACTER(256)      ext
      CHARACTER(256)      file
      INTEGER(4)          len
      DO WHILE (.TRUE.)
        WRITE (*,*)
        WRITE (*,'(A, \)') ' Enter filename (Hit &
                            RETURN to exit): '
        len = GETSTRQQ(file)
        IF (len .EQ. 0) EXIT
        len = FULLPATHQQ(file, buf)
        IF (len .GT. 0) THEN
          WRITE (*,*) buf(:len)
        ELSE
          WRITE (*,*) 'Can''t get full path'
          EXIT
        END IF
!
!       Split path
        WRITE (*,*)
        len = SPLITPATHQQ(buf, drive, dir, name, ext)
        IF (len .NE. 0) THEN
          WRITE (*, 900) ' Drive: ', drive
          WRITE (*, 900) ' Directory: ', dir(1:len)
          WRITE (*, 900) ' Name: ', name
          WRITE (*, 900) ' Extension: ', ext
        ELSE
          WRITE (*, *) 'Can''t split path'
        END IF
      END DO
900   FORMAT (A, A)
      END
```

See Also

SPLITPATHQQ

FUNCTION

Statement: *The initial statement of a function subprogram. A function subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression.*

Syntax

```
[prefix [prefix]] FUNCTION name [([d-arg-list])] [suffix]
   [specification-part]
   [execution-part]
[CONTAINS
   [internal-subprogram-part]]
END [FUNCTION [name]]
```

prefix

(Optional) Is any of the following:

- A data type specifier
- ELEMENTAL

Acts on one array element at a time. This is a restricted form of pure procedure.

- IMPURE

  Asserts that the procedure has side effects.
- MODULE

  Indicates a separate module procedure. See separate module procedures.
- NON_RECURSIVE

  Indicates the function is not recursive.
- PURE

  Asserts that the procedure has no side effects.
- RECURSIVE

  Permits direct and indirect recursion to occur. If a function is directly recursive and array valued, and RESULT is not specified, any reference to the function name in the executable part of the function is a reference to the function result variable.

At most one of each of the above can be specified. You cannot specify both NON_RECURSIVE and RECURSIVE. You cannot specify both PURE and IMPURE. You cannot specify ELEMENTAL if *lang-binding* is specified in *suffix*.

name

Is the name of the function. If RESULT is specified, the function name must not appear in any specification statement in the scoping unit of the function subprogram.

The function name can be followed by the length of the data type. The length is specified by an asterisk (*) followed by any unsigned, nonzero integer that is a valid length for the function's type. For example, REAL FUNCTION LGFUNC*8 (Y, Z) specifies the function result as REAL(8) (or REAL*8).

This optional length specification is not permitted if the length has already been specified following the keyword CHARACTER.

d-arg-list

(Optional) Is a list of one or more dummy arguments.

If there are no dummy arguments, no suffix, and no RESULT variable, the parentheses can be omitted. For example, the following is valid:

```
FUNCTION F
```

suffix

(Optional) Takes one of the following forms:

[RESULT (*r-name*)] *lang-binding*

lang-binding [RESULT (*r-name*)]

r-name

(Optional) Is the name of the function result. This name must not be the same as the function name.

lang-binding

Takes the following form:

BIND (C [, NAME=*ext-name*])

ext-name	Is a character scalar constant expression that can be used to construct the external name.

specification-part	Is one or more specification statements.
execution-part	Is one or more executable constructs or statements.
internal-subprogram-part	Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Description

The type and kind parameters (if any) of the function's result can be defined in the FUNCTION statement or in a type declaration statement within the function subprogram, but not both. If no type is specified, the type is determined by implicit typing rules in effect for the function subprogram.

Execution begins with the first executable construct or statement following the FUNCTION statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

If you specify CHARACTER(LEN=*) as the type of the function, the function assumes the length declared for it in the program unit that invokes it. This type of the resulting character function can have different lengths when it is invoked by different program units. An assumed-length character function cannot be directly recursive.

If the character length is specified as an integer constant, the value must agree with the length of the function specified in the program unit that invokes the function. If no length is specified, a length of 1 is assumed.

If the function is array-valued or a pointer, the declarations within the function must state these attributes for the function result name. The specification of the function result attributes, dummy argument attributes, and the information in the procedure heading collectively define the interface of the function.

The value of the result variable is returned by the function when it completes execution. Certain rules apply depending on whether the result is a pointer, as follows:

- If the result is a pointer, its allocation status must be determined before the function completes execution. The function must associate a target with the pointer, or cause the pointer to be explicitly disassociated from a target.

  If the pointer result points to a TARGET with the INTENT(IN) attribute, the function can give the result a value but the caller is not allowed to change the value pointed to.
- The shape of the value returned by the function is determined by the shape of the result variable when the function completes execution.
- If the result is not a pointer, its value must be defined before the function completes execution. If the result is an array, all the elements must be defined. If the result is a derived-type structure, all the components must be defined.

A function subprogram *cannot* contain a BLOCK DATA statement, a PROGRAM statement, a MODULE statement, or a SUBMODULE statement. A function can contain SUBROUTINE and FUNCTION statements to define internal procedures. ENTRY statements can be included to provide multiple entry points to the subprogram.

Example

The following example uses the Newton-Raphson iteration method `(F(X) = cosh(X) + cos(X) - A = 0)` to get the root of the function:

```
FUNCTION ROOT(A)
  X  = 1.0
  DO
    EX = EXP(X)
    EMINX = 1./EX
    ROOT  = X - ((EX+EMINX)*.5+COS(X)-A)/((EX-EMINX)*.5-SIN(X))
    IF (ABS((X-ROOT)/ROOT) .LT. 1E-6) RETURN
    X  = ROOT
  END DO
END
```

In the preceding example, the following formula is calculated repeatedly until the difference between X_i and X_{i+1} is less than 1.0E-6:

$$X_{i+1} = X_i - \frac{\cos(X_i) + \cos(X_i) - A}{\sinh(X_i) - \sin(X_i)}$$

The following example shows an assumed-length character function:

```
CHARACTER*(*) FUNCTION REDO(CARG)
  CHARACTER*1 CARG
  DO I=1,LEN(REDO)
    REDO(I:I) = CARG
  END DO
  RETURN
END FUNCTION
```

This function returns the value of its argument, repeated to fill the length of the function.

Within any given program unit, all references to an assumed-length character function must have the same length. In the following example, the REDO function has a length of 1000:

```
CHARACTER*1000 REDO, MANYAS, MANYZS
MANYAS = REDO('A')
MANYZS = REDO('Z')
```

Another program unit within the executable program can specify a different length. For example, the following REDO function has a length of 2:

```
CHARACTER HOLD*6, REDO*2
HOLD = REDO('A')//REDO('B')//REDO('C')
```

The following example shows a dynamic array-valued function:

```
FUNCTION SUB (N)
  REAL, DIMENSION(N) :: SUB
  ...
END FUNCTION
```

The following shows another example:

```
      INTEGER Divby2
10    PRINT *, 'Enter a number'
      READ *, i
```

```
      Print *, Divby2(i)
      GOTO 10
      END
C
C     This is the function definition
C
      INTEGER FUNCTION Divby2 (num)
      Divby2=num / 2
      END FUNCTION
```

The following example shows an allocatable function with allocatable arguments:

```
MODULE AP
CONTAINS

 FUNCTION ADD_VEC(P1,P2)
  ! Function to add two allocatable arrays of possibly differing lengths.
  ! The arrays may be thought of as polynomials (coefficients)
   REAL, ALLOCATABLE :: ADD_VEC(:), P1(:), P2(:)

  ! This function returns an allocatable array whose length is set to
  ! the length of the larger input array.
  ALLOCATE(ADD_VEC(MAX(SIZE(P1), SIZE(P2))))
  M = MIN(SIZE(P1), SIZE(P2))
  ! Add up to the shorter input array size
  ADD_VEC(:M) = P1(:M) + P2(:M)
  ! Use the larger input array elements afterwards (from P1 or P2)
  IF(SIZE(P1) > M) THEN
    ADD_VEC(M+1:) = P1(M+1:)
  ELSE IF(SIZE(P2) > M) THEN
    ADD_VEC(M+1:) = P2(M+1:)
  ENDIF
 END FUNCTION
END MODULE

PROGRAM TEST
 USE AP
 REAL, ALLOCATABLE :: P(:), Q(:), R(:), S(:)
 ALLOCATE(P(3))
 ALLOCATE(Q(2))
 ALLOCATE(R(3))
 ALLOCATE(S(3))

 ! Notice that P and Q differ in length
 P = (/4,2,1/)  ! P = X**2 + 2X + 4
 Q = (/-1,1/)   ! Q =         X - 1
 PRINT *,' Result should be:    3.000000       3.000000       1.000000'
 PRINT *,' Coefficients are: ', ADD_VEC(P, Q)  ! X**2 + 3X + 3

 P = (/1,1,1/)  ! P =  X**2 +  X + 1
 R = (/2,2,2/)  ! R = 2X**2 + 2X + 2
 S = (/3,3,3/)  ! S = 3X**2 + 3X + 3
 PRINT *,' Result should be:    6.000000       6.000000       6.000000'
 PRINT *,' Coefficients are: ', ADD_VEC(ADD_VEC(P,R), S)
END
```

Consider the following example:

```
module mymodule
type :: vec
  integer :: x(3)
contains
  procedure :: at
end type vec

contains
  function at( this, i ) result( p )
    implicit none
    class(vec), intent(in), target :: this
    integer, intent(in) :: i
    integer, pointer :: p

    p => this%x(i)
  end function at
end module mymodule

program test
use mymodule
implicit none
type(vec) :: myvec

myvec%x =  [1,2,3]

! pointer returned by function at gives the correct values: 1, 2, and 3
write(6,*) myvec%at(1), myvec%at(2), myvec%at(3)

! changing any array element is an error
myvec%at(1) = 4
myvec%at(2) = 5
myvec%at(3) = 6
…
end program test
```

See Also

ENTRY
SUBROUTINE
PURE
ELEMENTAL
RESULT keyword
Function References
Program Units and Procedures
General Rules for Function and Subroutine Subprograms

G

G

GAMMA

Elemental Intrinsic Function (Generic): *Returns the gamma value of its argument.*

Syntax

```
result = GAMMA (x)
```

x	(Input) Must be of type real. It must not be zero or a negative number.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the gamma function of x,

$$\Gamma(X) = \begin{cases} \int_0^\infty t^{X-1} \exp(-t)\, dt & X > 0 \\ \int_0^\infty t^{X-1} \left(\exp(-t) - \sum_{k=0}^{n} \frac{(-t)^k}{k!}\right) dt & -n-1 < X < -n,\ n \text{ an integer} \geq 0 \end{cases}$$

Example

GAMMA (1.0) has the approximate value 1.000.

GENERIC

Statement: *Declares a generic interface that is bound to a derived type-bound procedure, or specifies a generic identifier for one or more specific procedures.*

Syntax

The GENERIC statement takes the following form:

```
GENERIC [, access-spec]:: generic-spec => binding-name1 [, binding-name2]...
```

access-spec	(Optional) Is the PUBLIC or PRIVATE attribute. Only one can be specified. *access-spec* specifies the accessibility of *generic-spec*.
generic-spec	Is an explicit INTERFACE statement using one of the following: • generic-name *generic-spec* can be the name of a previously declared accessible generic name, in which case this statement extends that interface. For more information, see Defining Generic Names for Procedures. • OPERATOR (*op*) *op* is an intrinsic or a user-defined operator. For more information, see Defining Generic Operators.

- ASSIGNMENT (=)

  See Defining Generic Assignment.

The same *generic-spec* can be used in several generic bindings. In this case, every occurrence of the same *generic-spec* must have the same accessibility.

binding-spec1 [, binding-spec2,...]

Is the name of a specific procedure. It cannot be the name of a specific procedure that has been specified in an accessible generic interface with the same generic identifier. At most one of the specific names can be the same as the generic name.

Examples

Consider the following:

```
GENERIC :: GEN_FUNC => INT_FUNC, REAL_FUNC
GENERIC :: GEN_FUNC => CMPLX_FUNC
```

If INT_FUNC, REAL_FUNC, and CMPLX_FUNC are all functions with accessible interfaces, the first GENERIC statement declares GEN_FUNC to be a generic function name which can resolve to INT_FUNC or REAL_FUNC. The second GENERIC statement extends GEN_FUNC to be resolvable to a third specific function, CMPLX_FUNC.

Consider the following:

```
TYPE MY_TYPE2
...
CONTAINS
  PROCEDURE :: MYPROC => MYPROC1
  PROCEDURE,PASS(C) :: UPROC => UPROC1
  GENERIC :: OPERATOR(+) => MYPROC, UPROC
END TYPE MY_TYPE2
...
TYPE,EXTENDS(MY_TYPE2) :: MY_TYPE3
...
CONTAINS
  PROCEDURE :: MYPROC => MYPROC2
  PROCEDURE,PASS(C) :: UPROC => UPROC2
END TYPE MY_TYPE3
```

The type MY_TYPE3 inherits the generic operator '+'. Invoking the generic (+) invokes the specific type-bound procedure. For entities of type MY_TYPE3, that invokes the overriding actual procedure (MYPROC2 or UPROC2).

See Also

Type-Bound Procedures
Defining Generic Names for Procedures

GERROR

Run-Time Subroutine: *Returns a message for the last error detected by a Fortran run-time routine.*

Module

USE IFCORE

Syntax

```
CALL GERROR (string)
```

string	(Output) Character*(*). Message corresponding to the last detected error.

The last detected error does not necessarily correspond to the most recent function call. The compiler resets *string* only when another error occurs.

Example

```
USE IFCORE
character*40 errtext
character char1
integer*4 iflag, i4
. . .!Open unit 1 here
i4=fgetc(1,char1)
if (i4) then
  iflag = 1
  Call GERROR (errtext)
  print *, errtext
end if
```

See Also

PERROR
IERRNO

GETACTIVEQQ (W*S)

QuickWin Function: *Returns the unit number of the currently active child window.*

Module

USE IFQWIN

Syntax

```
result = GETACTIVEQQ( )
```

Results

The result type is INTEGER(4). The result is the unit number of the currently active window. If no child window is active, it returns the parameter QWIN$NOACTIVEWINDOW (defined in `IFQWIN.F90`).

See Also

SETACTIVEQQ
GETHWNDQQ

GETARCINFO (W*S)

Graphics Function: *Determines the endpoints (in viewport coordinates) of the most recently drawn arc or pie.*

Module

USE IFQWIN

Syntax

`result = GETARCINFO (lpstart, lpend, lppaint)`

lpstart	(Output) Derived type `xycoord`. Viewport coordinates of the starting point of the arc.
lpend	(Output) Derived type `xycoord`. Viewport coordinates of the end point of the arc.
lppaint	(Output) Derived type `xycoord`. Viewport coordinates of the point at which the fill begins.

Results

The result type is INTEGER(2). The result is nonzero if successful. The result is zero if neither the ARC nor the PIE function has been successfully called since the last time CLEARSCREEN or SETWINDOWCONFIG was successfully called, or since a new viewport was selected.

GETARCINFO updates the *lpstart* and *lpend*`xycoord` derived types to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to the ARC or PIE functions. The `xycoord` derived type, defined in `IFQWIN.F90`, is:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

The returned value in *lppaint* specifies a point from which a pie can be filled. You can use this to fill a pie in a color different from the border color. After a call to GETARCINFO, change colors using SETCOLORRGB. Use the new color, along with the coordinates in *lppaint*, as arguments for the FLOODFILLRGB function.

Example

```
USE IFQWIN
INTEGER(2) status, x1, y1, x2, y2, x3, y3, x4, y4
TYPE (xycoord) xystart, xyend, xyfillpt
x1 = 80; y1 = 50
x2 = 240; y2 = 150
x3 = 120; y3 = 80
x4 = 90; y4 = 180
status = ARC(x1, y1, x2, y2, x3, y3, x4, y4)
status = GETARCINFO(xystart, xyend, xyfillpt)
END
```

See Also

ARC
FLOODFILLRGB
GETCOLORRGB
GRSTATUS
PIE
SETCOLORRGB

GETARG

Intrinsic Subroutine: *Returns the specified command-line argument (where the command itself is argument number zero). Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL GETARG (n,buffer[,status])`

n	(Input) Must be a scalar of type integer. This value is the position of the command-line argument to retrieve. The command itself is argument number 0.
buffer	(Output) Must be a scalar of type default character. Its value is the returned command-line argument.
status	(Output; optional) Must be a scalar of type integer. If specified, its value is the returned completion status. If there were no errors, *status* returns the number of characters in the retrieved command-line argument before truncation or blank-padding. (That is, *status* is the original number of characters in the command-line argument.) Errors return a value of -1. Errors include specifying an argument position less than 0 or greater than the value returned by IARGC.

GETARG returns the *n*th command-line argument. If *n* is zero, the name of the executing program file is returned.

GETARG returns command-line arguments as they were entered. There is no case conversion.

If the command-line argument is shorter than *buffer*, GETARG pads *buffer* on the right with blanks. If the argument is longer than *buffer*, GETARG truncates the argument on the right. If there is an error, GETARG fills *buffer* with blanks.

Example

Assume a command-line invocation of `PROG1 -g -c -a`, and that *buffer* is at least five characters long. The following calls to GETARG return the corresponding arguments in *buffer* and *status*:

Statement	String returned in *buffer*	Length returned in *status*
CALL GETARG (0, buffer, status)	PROG1	5
CALL GETARG (1, buffer)	-g	undefined
CALL GETARG (2, buffer, status)	-c	2
CALL GETARG (3, buffer)	-a	undefined
CALL GETARG (4, buffer, status)	all blanks	-1

See Also

NARGS
IARGC
COMMAND_ARGUMENT_COUNT
GET_COMMAND

GET_COMMAND_ARGUMENT

GETBKCOLOR (W*S)

Graphics Function: *Returns the current background color index for both text and graphics output.*

Module

USE IFQWIN

Syntax

```
result = GETBKCOLOR( )
```

Results

The result type is INTEGER(4). The result is the current background color index.

GETBKCOLOR returns the current background color index for both text and graphics, as set with SETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR and returned with GETTEXTCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETBKCOLORRGB, SETCOLORRGB, and SETTEXTCOLORRGB.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are GETBKCOLOR and SETBKCOLOR. The default background index is 0, which is associated with black unless the user remaps the palette with REMAPPALETTERGB.

NOTE
The GETBKCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the GetBkColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$GetBkColor.

Example

```
USE IFQWIN
INTEGER(4) bcindex
bcindex = GETBKCOLOR()
```

See Also

GETBKCOLORRGB
SETBKCOLOR
GETCOLOR
GETTEXTCOLOR
REMAPALLPALETTERGB, REMAPPALETTERGB

GETBKCOLORRGB (W*S)

Graphics Function: *Returns the current background Red-Green-Blue (RGB) color value for both text and graphics.*

Module

USE IFQWIN

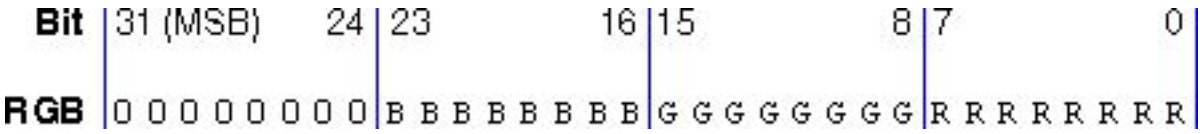
Syntax

```
result = GETBKCOLORRGB( )
```

Results

The result type is INTEGER(4). The result is the RGB value of the current background color for both text and graphics.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETBKCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set with SETBKCOLORRGB. The RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT) is set with SETTEXTCOLORRGB and returned with GETTEXTCOLORRGB. The RGB color value of graphics over the background color (used by graphics functions such as ARC, OUTGTEXT, and FLOODFILLRGB) is set with SETCOLORRGB and returned with GETCOLORRGB.

SETBKCOLORRGB (and the other RGB color selection functions SETCOLORRGB and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETBKCOLOR, SETCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(4) back, fore, oldcolor
INTEGER(2) status, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
oldcolor = SETCOLORRGB(Z'FF') ! red
! reverse the screen
back = GETBKCOLORRGB()
fore = GETCOLORRGB()
oldcolor = SETBKCOLORRGB(fore)
oldcolor = SETCOLORRGB(back)
CALL CLEARSCREEN ($GCLEARSCREEN)
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

See Also

GETCOLORRGB
GETTEXTCOLORRGB
SETBKCOLORRGB
GETBKCOLOR

GETC

Portability Function: *Reads the next available character from external unit 5, which is normally connected to the console.*

Module

USE IFPORT

Syntax

```
result = GETC (char)
```

char	(Output) Character*(*). First character typed at the keyboard after the call to GETC. If unit 5 is connected to a console device, then no characters are returned until the Enter key is pressed.

Results

The result type is INTEGER(4). The result is zero if successful, or -1 if an end-of-file was detected.

Example

```
   use IFPORT
   character ans,errtxt*40
   print *, 'Enter a character: '
   ISTAT = GETC (ans)
if (istat) then
  call gerror(errtxt)
end if
```

See Also

GETCHARQQ
GETSTRQQ

GETCHARQQ

Run-Time Function: *Returns the next keystroke.*

Module

USE IFCORE

Syntax

```
result = GETCHARQQ( )
```

Results

The result type is character with length 1. The result is the character representing the key that was pressed. The value can be any ASCII character.

If the key pressed is represented by a single ASCII character, GETCHARQQ returns the character. If the key pressed is a function or direction key, a hex Z'00' or Z'E0' is returned. If you need to know which function or direction was pressed, call GETCHARQQ a second time to get the extended code for the key.

If there is no keystroke waiting in the keyboard buffer, GETCHARQQ waits until there is one, and then returns it. Compare this to the function PEEKCHARQQ, which returns .TRUE. if there is a character waiting in the keyboard buffer, and .FALSE. if not. You can use PEEKCHARQQ to determine if GETCHARQQ should be called. This can prevent a program from hanging while GETCHARQQ waits for a keystroke that isn't there. Note that PEEKCHARQQ is only supported in console applications.

If your application is a QuickWin or Standard Graphics application, you may want to put a call to PASSDIRKEYSQQ in your program. This will enable the program to get characters that would otherwise be trapped. These extra characters are described in PASSDIRKEYSQQ.

Note that the GETCHARQQ routine used in a console application is a different routine than the one used in a QuickWin or Standard Graphics application. The GETCHARQQ used with a console application does not trap characters that are used in QuickWin for a special purpose, such as scrolling. Console applications do not need, and cannot use PASSDIRKEYSQQ.

Example

```
!  Program to demonstrate GETCHARQQ
USE IFCORE
CHARACTER(1) key / 'A' /
PARAMETER (ESC = 27)
PARAMETER (NOREP = 0)
WRITE (*,*) ' Type a key: (or q to quit)'
!  Read keys until ESC or q is pressed
DO WHILE (ICHAR (key) .NE. ESC)
   key = GETCHARQQ()
!  Some extended keys have no ASCII representation
   IF(ICHAR(key) .EQ. NOREP) THEN
     key = GETCHARQQ()
     WRITE (*, 900) 'Not ASCII. Char = NA'
     WRITE (*,*)
!  Otherwise, there is only one key
   ELSE
     WRITE (*,900) 'ASCII. Char = '
     WRITE (*,901) key
   END IF
   IF (key .EQ. 'q' ) THEN
     EXIT
   END IF
   END DO
900   FORMAT (1X, A, \)
901   FORMAT (A)
END
```

See Also

PEEKCHARQQ
GETSTRQQ
INCHARQQ
MBINCHARQQ
GETC
FGETC
PASSDIRKEYSQQ

GETCOLOR (W*S)

Graphics Function: *Returns the current graphics color index.*

Module

USE IFQWIN

Syntax

```
result = GETCOLOR( )
```

Results

The result type is INTEGER(2). The result is the current color index, if successful; otherwise, -1.

GETCOLOR returns the current color index used for graphics over the background color as set with SETCOLOR. The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR and returned with GETTEXTCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB.

Example

```
!  Program to demonstrate GETCOLOR
 PROGRAM COLORS
 USE IFQWIN
 INTEGER(2) loop, loop1, status, color
 LOGICAL(4) winstat
 REAL rnd1, rnd2, xnum, ynum
 type (windowconfig) wc
 status = SETCOLOR(INT2(0))
!  Color random pixels with 15 different colors
 DO loop1 = 1, 15
   color = INT2(MOD(GETCOLOR() +1, 16))
   status = SETCOLOR (color) ! Set to next color
   DO loop = 1, 75
!  Set color of random spot, normalized to be on screen
     CALL RANDOM(rnd1)
     CALL RANDOM(rnd2)
     winstat = GETWINDOWCONFIG(wc)
     xnum = wc%numxpixels
     ynum = wc%numypixels
     status = &
     SETPIXEL(INT2(rnd1*xnum+1),INT2(rnd2*ynum))
     status = &
     SETPIXEL(INT2(rnd1*xnum),INT2(rnd2*ynum+1))
     status = &
     SETPIXEL(INT2(rnd1*xnum-1),INT2( rnd2*ynum))
     status = &
     SETPIXEL(INT2(rnd1*xnum),INT2( rnd2*ynum-1))
   END DO
 END DO
 END
```

See Also

GETCOLORRGB
GETBKCOLOR
GETTEXTCOLOR
SETCOLOR

GETCOLORRGB (W*S)

Graphics Function: *Returns the current graphics color Red-Green-Blue (RGB) value (used by graphics functions such as ARC, ELLIPSE, and FLOODFILLRGB).*

Module

USE IFQWIN

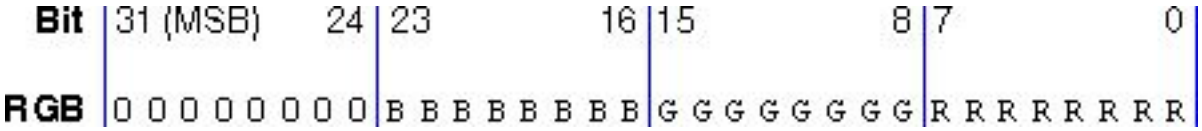
Syntax

```
result = GETCOLORRGB( )
```

Results

The result type is INTEGER(4). The result is the RGB value of the current graphics color.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETCOLORRGB returns the RGB color value of graphics over the background color (used by graphics functions such as ARC, ELLIPSE, and FLOODFILLRGB), set with SETCOLORRGB. GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set with SETBKCOLORRGB. GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT), set with SETTEXTCOLORRGB.

SETCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) numfonts
INTEGER(4) fore, oldcolor
numfonts = INITIALIZEFONTS ( )
oldcolor = SETCOLORRGB(Z'FF')   ! set graphics
                                !  color to red
fore = GETCOLORRGB()
oldcolor = SETBKCOLORRGB(fore)  ! set background
                                !  to graphics color
CALL CLEARSCREEN($GCLEARSCREEN)
oldcolor = SETCOLORRGB (Z'FF0000') ! set graphics
```

```
                                  !  color to blue
CALL OUTGTEXT("hello, world")
END
```

See Also

GETBKCOLORRGB
GETTEXTCOLORRGB
SETCOLORRGB
GETCOLOR

GET_COMMAND

Intrinsic Subroutine: *Returns the entire command that was used to invoke the program.*

Syntax

```
CALL GET_COMMAND ([command,length,status])
```

command	(Output; optional) Must be a scalar of type default character. If specified, its value is the entire command that was used to invoke the program. If the command cannot be determined, its value is all blanks.
length	(Output; optional) Must be a scalar of type integer. If specified, its value is the significant length of the command that was used to invoke the program. This length includes trailing blanks, but it does not include any truncation or padding used in the command. If the command length cannot be determined, its value is zero.
status	(Output; optional) Must be a scalar of type integer. If specified, its value is -1 if the command argument is present and has a length less than the significant length of the command. If the command cannot be retrieved, its value is positive; otherwise, it is assigned the value zero.

Example

See the example in COMMAND_ARGUMENT_COUNT.

See Also

GETARG
NARGS
IARGC
COMMAND_ARGUMENT_COUNT
GET_COMMAND_ARGUMENT

GET_COMMAND_ARGUMENT

Intrinsic Subroutine: *Returns a command line argument of the command that invoked the program. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL GET_COMMAND_ARGUMENT (number[,value,length,status])
```

number	(Input) Must be a scalar of type integer. It must be non-negative and less than or equal to the value returned by the COMMAND_ARGUMENT_COUNT function. Its value is the position of the command-line argument to retrieve. The command itself is argument number zero.
value	(Output; optional) Must be a scalar of type default character. If specified, its value is the returned command-line argument or all blanks if the value is unknown.
length	(Output; optional) Must be a scalar of type integer. If specified, its value is the length of the returned command-line argument or zero if the length of the argument is unknown. This length includes significant trailing blanks. It does not include any truncation or padding that occurs when the argument is assigned to the *value* argument.
status	(Output; optional) Must be a scalar of type integer. If specified, its value is the returned completion status. It is assigned the value -1 if the value argument is present and has a length less than the significant length of the command argument specified by *number*. It is assigned a processor-dependent positive value if the argument retrieval fails. Otherwise, it is assigned the value zero.

GET_COMMAND_ARGUMENT returns command-line arguments as they were entered. There is no case conversion.

Example

See the example in COMMAND_ARGUMENT_COUNT.

See Also

GETARG
NARGS
IARGC
COMMAND_ARGUMENT_COUNT
GET_COMMAND

GETCONTROLFPQQ

Portability Subroutine: *Returns the floating-point processor control word.*

Module

USE IFPORT

Syntax

```
CALL GETCONTROLFPQQ (controlword)
```

controlword	(Output) INTEGER(2). Floating-point processor control word. The floating-point control word is a bit flag that controls various modes of the floating-point processor. The control word can be any of the following constants (defined in IFPORT.F90):

Parameter name	Hex value	Description
FPCW$MCW_IC	Z'1000'	**Infinity control mask**
FPCW$AFFINE	Z'1000'	Affine infinity
FPCW$PROJECTIVE	Z'0000'	Projective infinity
FPCW$MCW_PC	Z'0300'	**Precision control mask**
FPCW$64	Z'0300'	64-bit precision
FPCW$53	Z'0200'	53-bit precision
FPCW$24	Z'0000'	24-bit precision
FPCW$MCW_RC	Z'0C00'	**Rounding control mask**
FPCW$CHOP	Z'0C00'	Truncate
FPCW$UP	Z'0800'	Round up
FPCW$DOWN	Z'0400'	Round down
FPCW$NEAR	Z'0000'	Round to nearest
FPCW$MCW_EM	Z'003F'	**Exception mask**
FPCW$INVALID	Z'0001'	Allow invalid numbers
FPCW$DENORMAL	Z'0002'	Allow subnormals (very small numbers)
FPCW$SUBNORMAL	Z'0002'	Allow subnormals (very small numbers)
FPCW$ZERODIVIDE	Z'0004'	Allow divide by zero
FPCW$OVERFLOW	Z'0008'	Allow overflow
FPCW$UNDERFLOW	Z'0010'	Allow underflow
FPCW$INEXACT	Z'0020'	Allow inexact precision

An exception is disabled if its control bit is set to 1. An exception is enabled if its control bit is cleared to 0. Exceptions can be disabled by setting the control bits to 1 with SETCONTROLFPQQ.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues.

You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ. If errors on floating-point exceptions are enabled (by clearing the control bits to 0 with SETCONTROLFPQQ), the operating system generates an interrupt when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

You can use GETCONTROLFPQQ to retrieve the current control word and SETCONTROLFPQQ to change the control word. Most users do not need to change the default settings.

Example

```
USE IFPORT
INTEGER(2) control
CALL GETCONTROLFPQQ (control)
     ! if not rounding down
IF (IAND(control, FPCW$DOWN) .NE. FPCW$DOWN) THEN
  control = IAND(control, NOT(FPCW$MCW_RC)) ! clear all
                                            !  rounding
  control = IOR(control, FPCW$DOWN)         !  set to
                                            !  round down
  CALL SETCONTROLFPQQ(control)
END IF
END
```

See Also

SETCONTROLFPQQ
GETSTATUSFPQQ
SIGNALQQ
CLEARSTATUSFPQQ

GETCURRENTPOSITION, GETCURRENTPOSITION_W (W*S)

Graphics Subroutines: *Return the coordinates of the current graphics position.*

Module

USE IFQWIN

Syntax

`CALL GETCURRENTPOSITION (s)`

`CALL GETCURRENTPOSITION_W (s)`

s

(Output) Derived type `xycoord`. Viewport coordinates of current graphics position. The derived type `xycoord` is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

LINETO, MOVETO, and OUTGTEXT all change the current graphics position. It is in the center of the screen when a window is created.

Graphics output starts at the current graphics position returned by GETCURRENTPOSITION or GETCURRENTPOSITION_W. This position is not related to normal text output (from OUTTEXT or WRITE, for example), which begins at the current text position (see SETTEXTPOSITION). It does, however, affect graphics text output from OUTGTEXT.

Example

```
! Program to demonstrate GETCURRENTPOSITION
USE IFQWIN
TYPE (xycoord) position
```

```
INTEGER(2)     result
result = LINETO(INT2(300), INT2(200))
CALL GETCURRENTPOSITION( position )
IF (position%xcoord .GT. 50) THEN
  CALL MOVETO(INT2(50), position%ycoord, position)
  WRITE(*,*) "Text unaffected by graphics position"
END IF
result = LINETO(INT2(300), INT2(200))
END
```

See Also

LINETO
MOVETO
OUTGTEXT
SETTEXTPOSITION
GETTEXTPOSITION

GETCWD

Portability Function: *Returns the path of the current working directory.*

Module

USE IFPORT

Syntax

`result = GETCWD (dirname)`

dirname	(Output) Character *(*). Name of the current working directory path, including drive letter.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code.

Example

```
    USE IFPORT
    character*30 dirname
!   variable dirname must be long enough to hold entire string
    integer(4) istat
    ISTAT = GETCWD (dirname)
    IF (ISTAT == 0) write *, 'Current directory is ',dirname
```

See Also

GETDRIVEDIRQQ

GETDAT

Portability Subroutine: *Returns the date.*

Module

USE IFPORT

Syntax

`CALL GETDAT (iyr, imon, iday)`

iyr	(Output) INTEGER(4) or INTEGER(2). Year (*xxxx*AD).
imon	(Output) INTEGER(4) or INTEGER(2). Month (1-12).
iday	(Output) INTEGER(4) or INTEGER(2). Day of the month (1-31).
	This subroutine is thread-safe.

All arguments must be of the same integer kind, that is, all must be INTEGER(2) or all must be INTEGER(4). If INTEGER(2) arguments are passed, you must specify USE IFPORT.

Example

```
! Program to demonstrate GETDAT and GETTIM
USE IFPORT
INTEGER(4) tmpday, tmpmonth, tmpyear
INTEGER(4) tmphour, tmpminute, tmpsecond, tmphund
CHARACTER(1) mer
CALL GETDAT(tmpyear, tmpmonth, tmpday)
CALL GETTIM(tmphour, tmpminute, tmpsecond, tmphund)
IF (tmphour .GT. 12) THEN
  mer = 'p'
  tmphour = tmphour - 12
ELSE
  mer = 'a'
END IF
WRITE (*, 900) tmpmonth, tmpday, tmpyear
900  FORMAT(I2, '/', I2.2, '/', I4.4)
WRITE (*, 901) tmphour, tmpminute, tmpsecond, tmphund, mer
901  FORMAT(I2, ':', I2.2, ':', I2.2, ':', I2.2, ' ',&
            A, 'm')
END
```

See Also

GETTIM

SETDAT

SETTIM

DATE portability routine

FDATE

IDATE portability routine

JDATE

GETDRIVEDIRQQ

Portability Function: *Returns the path of the current working directory on a specified drive.*

Module

USE IFPORT

Syntax

`result = GETDRIVEDIRQQ (drivedir)`

drivedir	(Input; output) Character*(*). On input, the drive whose current working directory path is to be returned. On output, the string containing the current directory on that drive in the form d:\dir.

Results

The result type is INTEGER(4). The result is the length (in bytes) of the full path of the directory on the specified drive. Zero is returned if the path is longer than the size of the character buffer *drivedir*.

You specify the drive from which to return the current working directory by putting the drive letter into *drivedir* before calling GETDRIVEDIRQQ. To make sure you get information about the current drive, put the symbolic constant FILE$CURDRIVE (defined in `IFPORT.F90`) into *drivedir*.

Because drives are identified by a single alphabetic character, GETDRIVEDIRQQ examines only the first letter of *drivedir*. For instance, if *drivedir* contains the path `c:\fps90\bin`, GETDRIVEDIRQQ (*drivedir*) returns the current working directory on drive C and disregards the rest of the path. The drive letter can be uppercase or lowercase.

The length of the path returned depends on how deeply the directories are nested on the drive specified in *drivedir*. If the full path is longer than the length of *drivedir*, GETDRIVEDIRQQ returns only the portion of the path that fits into *drivedir*. If you are likely to encounter a long path, allocate a buffer of size $MAXPATH ($MAXPATH = 260).

On Linux* and macOS* systems, the function gets a path only when symbolic constant FILE$CURDRIVE has been applied to *drivedir*.

Example

```
!  Program to demonstrate GETDRIVEDIRQQ
USE IFPORT
CHARACTER($MAXPATH) dir
INTEGER(4) length
!  Get current directory
dir = FILE$CURDRIVE
length = GETDRIVEDIRQQ(dir)
IF (length .GT. 0) THEN
  WRITE (*,*) 'Current directory is: '
  WRITE (*,*) dir
ELSE
  WRITE (*,*) 'Failed to get current directory'
END IF
END
```

See Also

CHANGEDRIVEQQ
CHANGEDIRQQ
GETDRIVESIZEQQ
GETDRIVESQQ
GETLASTERRORQQ
SPLITPATHQQ

GETDRIVESIZEQQ

Portability Function: *Returns the total size of the specified drive and space available on it.*

Module

USE IFPORT

Syntax

```
result = GETDRIVESIZEQQ (drive,total,avail)
```

drive	(Input) Character*(*). String containing the letter of the drive to get information about.
total	(Output) INTEGER(4) or INTEGER(4), DIMENSION(2) or INTEGER(8). Total number of bytes on the drive.
avail	(Output) INTEGER(4) or INTEGER(4), DIMENSION(2) or INTEGER(8). Number of bytes of available space on the drive.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The data types and dimension (if any) specified for the *total* and *avail* arguments must be the same. Specifying an array of two INTEGER(4) elements, or an INTEGER(8) argument, allows drive sizes larger than 2147483647 to be returned.

If an array of two INTEGER(4) elements is specified, the least-significant 32 bits are returned in the first element, the most-significant 32 bits in the second element. If an INTEGER(4) scalar is specified, the least-significant 32 bits are returned.

Because drives are identified by a single alphabetic character, GETDRIVESIZEQQ examines only the first letter of *drive*. The drive letter can be uppercase or lowercase. You can use the constant FILE$CURDRIVE (defined in `IFPORT.F90`) to get the size of the current drive.

If GETDRIVESIZEQQ fails, use GETLASTERRORQQ to determine the reason.

Example

```
! Program to demonstrate GETDRIVESQQ and GETDRIVESIZEQQ
USE IFPORT
CHARACTER(26) drives
CHARACTER(1) adrive
LOGICAL(4) status
INTEGER(4) total, avail
INTEGER(2) i
! Get the list of drives
drives = GETDRIVESQQ()
WRITE (*,'(A, A)') ' Drives available: ', drives
!
!Cycle through them for free space and write to console
DO i = 1, 26
  adrive = drives(i:i)
  status = .FALSE.
  WRITE (*,'(A, A, A, \)') ' Drive ', CHAR(i + 64), ':'
  IF (adrive .NE. ' ') THEN
     status = GETDRIVESIZEQQ(adrive, total, avail)
  END IF
```

```
  IF (status) THEN
     WRITE (*,*) avail, ' of ', total, ' bytes free.'
  ELSE
     WRITE (*,*) 'Not available'
  END IF
END DO
END
```

See Also

GETLASTERRORQQ
GETDRIVESQQ
GETDRIVEDIRQQ
CHANGEDRIVEQQ
CHANGEDIRQQ

GETDRIVESQQ

Portability Function: *Reports which drives are available to the system.*

Module

USE IFPORT

Syntax

```
result = GETDRIVESQQ( )
```

Results

The result type is character with length 26. It is the positional character string containing the letters of the drives available in the system.

The returned string contains letters for drives that are available, and blanks for drives that are not available. For example, on a system with A, C, and D drives, the string 'A CD' is returned.

On Linux* and macOS* systems, the function returns a string filled with spaces.

Example

See the example for GETDRIVESIZEQQ.

See Also

GETDRIVEDIRQQ
GETDRIVESIZEQQ
CHANGEDRIVEQQ

GETENV

Portability Subroutine: *Returns the value of an environment variable.*

Module

USE IFPORT

Syntax

```
CALL GETENV (ename,evalue)
```

ename	(Input) Character*(*). Environment variable to search for.
evalue	(Output) Character*(*). Value found for *ename*. Blank if *ename* is not found.

Example

```
use IFPORT
character*40 libname
CALL GETENV ("LIB",libname)
TYPE *, "The LIB variable points to ",libname
```

See Also

GETENVQQ

GET_ENVIRONMENT_VARIABLE

Intrinsic Subroutine: *Gets the value of an environment variable.*

Syntax

CALL GET_ENVIRONMENT_VARIABLE (*name* [,*value*,*length*,*status*,*trim_name*])

name	(Input) Must be a scalar of type default character. It is the name of the environment variable.
value	(Output; optional) Must be a scalar of type default character. If specified, it is assigned the value of the environment variable specified by *name*. If the environment variable does not exist, *value* is assigned all blanks.
length	(Output; optional) Must be a scalar of type integer. If specified, its value is the length of the environment variable, if it exists; otherwise, *length* is set to 0.
status	(Output; optional) Must be a scalar of type integer. If specified, it is assigned a value of 0 if the environment variable exists and either has no value or its value is successfully assigned to *value*. It is assigned a value of -1 if the *value* argument is present and has a length less than the significant length of the environment variable value. It is assigned a value of 1 if the environment variable does not exist. For other error conditions, it is assigned a processor-dependent value greater than 2.
trim_name	(Input; optional) Must be a scalar of type logical. If the value is FALSE, then trailing blanks in name are considered significant. Otherwise, they are not considered part of the environment variable's name.

Example

The following program asks for the name of an environment variable. If the environment variable exists in the program's environment, it prints out its value:

```
program print_env_var
character name*20, val*40
integer len, status
```

```
write (*,*) 'enter the name of the environment variable'
read (*,*) name
call get_environment_variable (name, val, len, status, .true.)
if (status .ge. 2) then
   write (*,*) 'get_environment_variable failed: status = ', status
   stop
end if
if (status .eq. 1) then
   write (*,*) 'env var does not exist'
   stop
end if
if (status .eq. -1) then
   write (*,*) 'env var length = ', len, ' truncated to 40'
   len = 40
end if
if (len .eq. 0) then
   write (*,*) 'env var exists  but has no value'
   stop
end if
write (*,*) 'env var value = ', val (1:len)
end
```

When the above program is invoked, the following line is displayed:

```
enter the name of the environment variable
```

The following shows an example of what could be displayed if you enter "HOME".

- On a Linux* or macOS* system:

```
env var value = /home/our_space/usr4
```

- On a Windows* system:

```
env var value = C:/
```

The following shows an example of what could be displayed if you enter "PATH".

- On a Linux or macOS* system:

```
env var length =          307  truncated to 40
env var value = /site/our_space/usr4/progs/build_area
```

- On a Windows system:

```
env var length =          829  truncated to 40
env var value = C:\OUR_SPACE\BUILD_AREA\build_objects\
```

GETENVQQ

Portability Function: *Returns the value of an environment variable.*

Module

USE IFPORT

Syntax

`result = GETENVQQ (varname,value)`

varname (Input) Character*(*). Name of environment variable.

value	(Output) Character*(*). Value of the specified environment variable, in uppercase.

Results

The result type is INTEGER(4). The result is the length of the string returned in *value*. Zero is returned if the given variable is not defined.

GETENVQQ searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment in which a process executes. For example, the LIB environment variable defines the default search path for libraries to be linked with a program.

Note that some environment variables may exist only on a per-process basis and may not be present at the command-line level.

GETENVQQ uses the C runtime routine getenv and SETENVQQ uses the C runtime routine _putenv. From the C documentation:

getenv and _putenv use the copy of the environment pointed to by the global variable _environ to access the environment. getenv operates only on the data structures accessible to the run-time library and not on the environment segment created for the process by the operating system.

In a program that uses the main function, _environ is initialized at program startup to settings taken from the operating system's environment.

Changes made outside the program by the console SET command, for example, SET MY_VAR=ABCDE, will be reflected by GETENVQQ.

GETENVQQ and SETENVQQ will not work properly with the Windows* APIs GetEnvironmentVariable and SetEnvironmentVariable.

Example

```
!  Program to demonstrate GETENVQQ and SETENVQQ
USE IFPORT
USE IFCORE
INTEGER(4) lenv, lval
CHARACTER(80) env, val, enval
WRITE (*,900) ' Enter environment variable name to create, &
               modify, or delete: '
lenv = GETSTRQQ(env)
IF (lenv .EQ. 0) STOP
WRITE (*,900) ' Value of variable (ENTER to delete): '
lval = GETSTRQQ(val)
IF (lval .EQ. 0) val = ' '
enval = env(1:lenv) // '=' // val(1:lval)
IF (SETENVQQ(enval)) THEN
  lval = GETENVQQ(env(1:lenv), val)
  IF (lval .EQ. 0) THEN
    WRITE (*,*) 'Can''t get environment variable'
  ELSE IF (lval .GT. LEN(val)) THEN
    WRITE (*,*) 'Buffer too small'
  ELSE
    WRITE (*,*) env(:lenv), ': ', val(:lval)
    WRITE (*,*) 'Length: ', lval
  END IF
ELSE
  WRITE (*,*) 'Can''t set environment variable'
```

```
END IF
900  FORMAT (A, \)
END
```

See Also

SETENVQQ
GETLASTERRORQQ

GETEXCEPTIONPTRSQQ (W*S)

Run-Time Function: *Returns a pointer to C run-time exception information pointers appropriate for use in signal handlers established with SIGNALQQ or direct calls to the C rtl signal( ) routine.*

Module

USE IFCORE

Syntax

```
result = GETEXCEPTIONPTRSQQ( )
```

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The return value is the address of a data structure whose members are pointers to exception information captured by the C runtime at the time of an exception. This result value can then be used as the *eptr* argument to routine TRACEBACKQQ to generate a stack trace from a user-defined handler or to inspect the exception context record directly.

Calling GETEXCEPTIONPTRSQQ is only valid within a user-defined handler that was established with SIGNALQQ or a direct call to the C rtl signal( ) function.

For a full description of exceptions and error handling, see Compiler Reference: Error Handling.

Example

```
PROGRAM SIGTEST
USE IFCORE
...
R3 = 0.0E0
STS = SIGNALQQ(MY_HANDLER)
! Cause a divide by zero exception
R1 = 3.0E0/R3
...
END
INTEGER(4) FUNCTION MY_HANDLER(SIGNUM,EXCNUM)
USE IFCORE
...
EPTRS = GETEXCEPTIONPTRSQQ()
...
CALL TRACEBACKQQ("Application SIGFPE error!",USER_EXIT_CODE=-1,EPTR=EPTRS)
...
MY_HANDLER = 1
END
```

See Also

TRACEBACKQQ
GETSTATUSFPQQ

CLEARSTATUSFPQQ
SETCONTROLFPQQ
GETCONTROLFPQQ
SIGNALQQ

GETEXITQQ (W*S)

QuickWin Function: *Returns the setting for a QuickWin application's exit behavior.*

Module

USE IFQWIN

Syntax

```
result = GETEXITQQ( )
```

Results

The result type is INTEGER(4). The result is exit mode with one of the following constants (defined in `IFQWIN.F90`):

- QWIN$EXITPROMPT - Displays a message box that reads "Program exited with exit status *n*. Exit Window?", where *n* is the exit status from the program.

  If you choose Yes, the application closes the window and terminates. If you choose No, the dialog box disappears and you can manipulate the window as usual. You must then close the window manually.
- QWIN$EXITNOPERSIST - Terminates the application without displaying a message box.
- QWIN$EXITPERSIST - Leaves the application open without displaying a message box.

The default for both QuickWin and Console Graphics applications is QWIN$EXITPROMPT.

Example

```
!  Program to demonstrate GETEXITQQ
     USE IFQWIN
     INTEGER i
     i = GETEXITQQ()
     SELECT CASE (i)
       CASE (QWIN$EXITPROMPT)
         WRITE(*, *) "Prompt on exit."
       CASE (QWIN$EXITNOPERSIST)
         WRITE(*,*) "Exit and close."
       CASE (QWIN$EXITPERSIST)
         WRITE(*,*) "Exit and leave open."
     END SELECT
     END
```

See Also

SETEXITQQ

GETFILEINFOQQ

Portability Function: *Returns information about the specified file. File names can contain wildcards (* and ?).*

Module

USE IFPORT

Syntax

```
result = GETFILEINFOQQ (files,buffer,handle)
```

files

(Input) Character*(*). Name or pattern of files you are searching for. Can include a full path and wildcards (* and ?).

buffer

(Output) Derived type FILE$INFO or derived type FILE$INFOI8. Information about a file that matches the search criteria in *files*.

The derived type FILE$INFO is defined in IFPORT.F90 as follows:

```
TYPE FILE$INFO
  INTEGER(4) CREATION             ! CREATION TIME (-1 on FAT)
  INTEGER(4) LASTWRITE            ! LAST WRITE TO FILE
  INTEGER(4) LASTACCESS           ! LAST ACCESS (-1 on FAT)
  INTEGER(4) LENGTH               ! LENGTH OF FILE
  INTEGER(4) PERMIT               ! FILE ACCESS MODE
  CHARACTER(255) NAME             ! FILE NAME
END TYPE FILE$INFO
```

The derived type FILE$INFOI8 is defined in IFPORT.F90 as follows:

```
TYPE FILE$INFOI8
  INTEGER(4) CREATION             ! CREATION TIME (-1 on FAT)
  INTEGER(4) LASTWRITE            ! LAST WRITE TO FILE
  INTEGER(4) LASTACCESS           ! LAST ACCESS (-1 on FAT)
  INTEGER(8) LENGTH               ! LENGTH OF FILE
  INTEGER(4) PERMIT               ! FILE ACCESS MODE
  CHARACTER(255) NAME             ! FILE NAME
END TYPE FILE$INFOI8
```

handle

(Input; output) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Control mechanism. One of the following constants, defined in IFPORT.F90:

- FILE$FIRST - First matching file found.
- FILE$LAST - Previous file was the last valid file.
- FILE$ERROR - No matching file found.

Results

The result type is INTEGER(4). The result is the nonblank length of the file name if a match was found, or 0 if no matching files were found.

To get information about one or more files, set the handle to FILE$FIRST and call GETFILEINFOQQ. This will return information about the first file which matches the name and return a handle. If the program wants more files, it should call GETFILEINFOQQ with the handle. GETFILEINFOQQ must be called with the handle until GETFILEINFOQQ sets handle to FILE$LAST, or system resources may be lost.

The derived-type element variables FILE$INFO%CREATION, FILE$INFO%LASTWRITE, and FILE$INFO %LASTACCESS contain packed date and time information that indicates when the file was created, last written to, and last accessed, respectively.

To break the time and date into component parts, call UNPACKTIMEQQ. FILE$INFO%LENGTH contains the length of the file in bytes. FILE$INFO%PERMIT contains a set of bit flags describing access information about the file as follows:

Bit flag	Access information for the file
FILE$ARCHIVE	Marked as having been copied to a backup device.

Bit flag	Access information for the file
FILE$DIR	A subdirectory of the current directory. Each MS-DOS* directory contains two special files, "." and "..". These are directory aliases created by MS-DOS for use in relative directory notation. The first refers to the current directory, and the second refers to the current directory's parent directory.
FILE$HIDDEN	Hidden. It does not appear in the directory list you request from the command line, the Microsoft* visual development environment browser, or File Manager.
FILE$READONLY	Write-protected. You can read the file, but you cannot make changes to it.
FILE$SYSTEM	Used by the operating system.
FILE$VOLUME	A logical volume, or partition, on a physical disk drive. This type of file appears only in the root directory of a physical device.

You can use the constant FILE$NORMAL to check that all bit flags are set to 0. If the derived-type element variable FILE$INFO%PERMIT is equal to FILE$NORMAL, the file has no special attributes. The variable FILE $INFO%NAME contains the short name of the file, not the full path of the file.

If an error occurs, call GETLASTERRORQQ to retrieve the error message, such as:

- ERR$NOENT: The file or path specified was not found.
- ERR$NOMEM: Not enough memory is available to execute the command, the available memory has been corrupted, or an invalid block exists, indicating that the process making the call was not allocated properly.

Example

```
USE IFPORT
USE IFCORE
  CALL SHOWPERMISSION( )
END

! SUBROUTINE to demonstrate GETFILEINFOQQ
SUBROUTINE SHOWPERMISSION( )
USE IFPORT
  CHARACTER(80) files
  INTEGER(KIND=INT_PTR_KIND( )) handle
  INTEGER(4) length
  CHARACTER(5) permit
  TYPE (FILE$INFO) info
    WRITE (*, 900) ' Enter wildcard of files to view: '
    900 FORMAT (A, \)

    length = GETSTRQQ(files)
    handle = FILE$FIRST

    DO WHILE (.TRUE.)
      length = GETFILEINFOQQ(files, info, handle)
      IF ((handle .EQ. FILE$LAST) .OR. &
          (handle .EQ. FILE$ERROR)) THEN
```

```
        SELECT CASE (GETLASTERRORQQ( ))
          CASE (ERR$NOMEM)
            WRITE (*,*) 'Out of memory'
          CASE (ERR$NOENT)
            EXIT
          CASE DEFAULT
            WRITE (*,*) 'Invalid file or path name'
        END SELECT
      END IF

      permit = ' '
      IF ((info%permit .AND. FILE$HIDDEN) .NE. 0) &
         permit(1:1) = 'H'
      IF ((info%permit .AND. FILE$SYSTEM) .NE. 0) &
         permit(2:2) = 'S'
      IF ((info%permit .AND. FILE$READONLY) .NE. 0) &
         permit(3:3) = 'R'
      IF ((info%permit .AND. FILE$ARCHIVE) .NE. 0) &
         permit(4:4) = 'A'
      IF ((info%permit .AND. FILE$DIR) .NE. 0) &
         permit(5:5) = 'D'
      WRITE (*, 9000) info%name, info%length, permit
      9000 FORMAT (1X, A5, I9, ' ',A6)
    END DO
END SUBROUTINE
```

See Also

SETFILEACCESSQQ
SETFILETIMEQQ
UNPACKTIMEQQ

GETFILLMASK (W*S)

Graphics Subroutine: *Returns the current pattern used to fill shapes.*

Module

USE IFQWIN

Syntax

`CALL GETFILLMASK (mask)`

mask (Output) INTEGER(1). One-dimensional array of length 8.

There are 8 bytes in *mask*, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of *mask* becomes the top 8 bits of the pattern, and the eighth element (byte) of *mask* becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of 0 are unchanged. The current graphics color is set with SETCOLORRGB or SETCOLOR. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with SETFILLMASK), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (FLOODFILLRGB, PIE, ELLIPSE, POLYGON, and RECTANGLE).

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(1) style(8). array(8)
INTEGER(2) i
style = 0
style(1) = Z'F'
style(3) = Z'F'
style(5) = Z'F'
style(7) = Z'F'
CALL SETFILLMASK (style)
...
CALL GETFILLMASK (array)
WRITE (*, *) 'Fill mask in bits: '
DO i = 1, 8
  WRITE (*, '(B8)') array(i)
END DO
END
```

See Also

ELLIPSE
FLOODFILLRGB
PIE
POLYGON
RECTANGLE
SETFILLMASK

GETFONTINFO (W*S)

Graphics Function: *Returns the current font characteristics.*

Module

USE IFQWIN

Syntax

`result = GETFONTINFO (font)`

font

(Output) Derived type FONTINFO. Set of characteristics of the current font. The FONTINFO derived type is defined in IFQWIN.F90 as follows:

```
TYPE FONTINFO
  INTEGER(4) type        ! 1 = truetype, 0 = bit map
  INTEGER(4) ascent      ! Pixel distance from top to
                         !   baseline
  INTEGER(4) pixwidth    ! Character width in pixels,
                         !   0=proportional
  INTEGER(4) pixheight   ! Character height in pixels
  INTEGER(4) avgwidth    ! Average character width in
                         !   pixels
  CHARACTER(81) filename ! File name including path
  CHARACTER(32) facename ! Font name
  LOGICAL(1) italic      ! .TRUE. if current font
                         !   formatted italic
  LOGICAL(1) emphasized  ! .TRUE. if current font
```

```
                          !   formatted bold
  LOGICAL(1) underline    ! .TRUE. if current font
                          !   formatted underlined
END TYPE FONTINFO
```

Results

The result type is INTEGER(2). The result is zero if successful; otherwise, -1.

You must initialize fonts with INITIALIZEFONTS before calling any font-related function, including GETFONTINFO.

Example

```
! Build as QuickWin or Standard Graphics
USE IFQWIN
TYPE (FONTINFO) info
INTEGER(2)      numfonts, return, line_spacing
numfonts = INITIALIZEFONTS ( )
return = GETFONTINFO(info)
line_spacing = info%pixheight + 2
END
```

See Also

GETGTEXTEXTENT
GETGTEXTROTATION
GRSTATUS
OUTGTEXT
INITIALIZEFONTS
SETFONT

GETGID

Portability Function: *Returns the group ID of the user of a process.*

Module

USE IFPORT

Syntax

```
result = GETGID( )
```

Results

The result type is INTEGER(4). The result corresponds to the primary group of the user under whose identity the program is running. The result is returned as follows:

- On Windows* systems, this function returns the last subauthority of the security identifier for the process. This is unique on a local machine and unique within a domain for domain accounts.

  Note that on Windows systems, domain accounts and local accounts can overlap.
- On Linux* and macOS* systems, this function returns the group identity for the current process.

Example

```
USE IFPORT
ISTAT = GETGID( )
```

GETGTEXTEXTENT (W*S)

Graphics Function: *Returns the width in pixels that would be required to print a given string of text (including any trailing blanks) with OUTGTEXT using the current font.*

Module

USE IFQWIN

Syntax

```
result = GETGTEXTEXTENT (text)
```

text (Input) Character*(*). Text to be analyzed.

Results

The result type is INTEGER(2). The result is the width of *text* in pixels if successful; otherwise, -1 (for example, if fonts have not been initialized with INITIALIZEFONTS).

This function is useful for determining the size of text that uses proportionally spaced fonts. You must initialize fonts with INITIALIZEFONTS before calling any font-related function, including GETGTEXTEXTENT.

Example

```
! Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(2) status, pwidth
CHARACTER(80) text
status= INITIALIZEFONTS( )
status= SETFONT('t''Arial''h22w10')
pwidth= GETGTEXTEXTENT('How many pixels wide is this?')
WRITE(*,*) pwidth
END
```

See Also

GETFONTINFO
OUTGTEXT
SETFONT
INITIALIZEFONTS
GETGTEXTROTATION

GETGTEXTROTATION (W*S)

Graphics Function: *Returns the current orientation of the font text output by OUTGTEXT.*

Module

USE IFQWIN

Syntax

```
result = GETGTEXTROTATION( )
```

Results

The result type is INTEGER(4). It is the current orientation of the font text output in tenths of degrees. Horizontal is 0°, and angles increase counterclockwise so that 900 tenths of degrees (90°) is straight up, 1800 tenths of degrees (180°) is upside-down and left, 2700 tenths of degrees (270°) is straight down, and so forth.

The orientation for text output with OUTGTEXT is set with SETGTEXTROTATION.

Example

```
! Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER ang
REAL rang
ang = GETGTEXTROTATION( )
rang = FLOAT(ang)/10.0
WRITE(*,*) "Text tilt in degrees is: ", rang
END
```

See Also

OUTGTEXT
SETFONT
SETGTEXTROTATION

GETHWNDQQ (W*S)

QuickWin Function: *Converts a window unit number into a Windows* handle.*

Module

USE IFQWIN

Syntax

`result = GETHWNDQQ (unit)`

unit	(Input) INTEGER(4). The window unit number. If *unit* is set to QWIN $FRAMEWINDOW (defined in IFQWIN.F90), the handle of the frame window is returned.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The result is a true Windows handle to the window. If *unit* is not open, it returns -1 .

See Also

GETACTIVEQQ
GETUNITQQ
SETACTIVEQQ

GETIMAGE, GETIMAGE_W (W*S)

Graphics Subroutines: *Store the screen image defined by a specified bounding rectangle.*

Module

USE IFQWIN

Syntax

CALL GETIMAGE (*x1*,*y1*,*x2*,*y2*,*image*)

CALL GETIMAGE_W (*wx1*,*wy1*,*wx2*,*wy2*,*image*)

x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.
image	(Output) INTEGER(1). Array of single-byte integers. Stored image buffer.

GETIMAGE defines the bounding rectangle in viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). GETIMAGE_W defines the bounding rectangle in window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

The buffer used to store the image must be large enough to hold it. You can determine the image size by calling IMAGESIZE at run time, or by using the formula described under IMAGESIZE. After you have determined the image size, you can dimension the buffer accordingly.

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(1), ALLOCATABLE:: buffer (:)
INTEGER(2) status, x, y, error
INTEGER(4) imsize
x = 50
y = 30
status = ELLIPSE ($GFILLINTERIOR, INT2(x-15), &
                  INT2(y-15), INT2( x+15), INT2(y+15))
imsize = IMAGESIZE (INT2(x-16), INT2(y-16), &
                  INT2( x+16), INT2(y+16))
ALLOCATE(buffer (imsize), STAT = error)
IF (error .NE. 0) THEN
  STOP 'ERROR: Insufficient memory'
END IF
CALL GETIMAGE (INT2(x-16), INT2(y-16), &
                  INT2( x+16), INT2(y+16), buffer)
END
```

See Also

IMAGESIZE

PUTIMAGE

GETLASTERROR

Portability Function: *Returns the last error set.*

Module

USE IFPORT

Syntax

```
result = GETLASTERROR( )
```

Results

The result type is INTEGER(4). The result is the integer corresponding to the last run-time error value that was set.

For example, if you use an ERR= specifier in an I/O statement, your program will not abort if an error occurs. GETLASTERROR provides a way to determine what the error condition was, with a better degree of certainty than just examining `errno`. Your application can then take appropriate action based upon the error number.

GETLASTERRORQQ

Portability Function: *Returns the last error set by a run-time procedure.*

Module

USE IFPORT

Syntax

```
result = GETLASTERRORQQ( )
```

Results

The result type is INTEGER(4). The result is the most recent error code generated by a run-time procedure.

Library functions that return a logical or integer value sometimes also provide an error code that identifies the cause of errors. GETLASTERRORQQ retrieves the most recent error message. The error constants are defined in `IFPORT.F90`. The following table shows some library routines and the errors each routine produces:

Library routine	Errors produced
BEEPQQ	no error
BSEARCHQQ	ERR$INVAL
CHANGEDIRQQ	ERR$NOMEM, ERR$NOENT
CHANGEDRIVEQQ	ERR$INVAL, ERR$NOENT
COMMITQQ	ERR$BADF
DELDIRQQ	ERR$NOMEM, ERR$ACCES, ERR$NOENT
DELFILESQQ	ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR $INVAL
FINDFILEQQ	ERR$NOMEM, ERR$NOENT
FULLPATHQQ	ERR$NOMEM, ERR$INVAL
GETCHARQQ	no error
GETDRIVEDIRQQ	ERR$NOMEM, ERR$RANGE
GETDRIVESIZEQQ	ERR$INVAL, ERR$NOENT

Library routine	Errors produced
GETDRIVESQQ	no error
GETENVQQ	ERR$NOMEM, ERR$NOENT
GETFILEINFOQQ	ERR$NOMEM, ERR$NOENT, ERR$INVAL
GETLASTERRORQQ	no error
GETSTRQQ	no error
MAKEDIRQQ	ERR$NOMEM, ERR$ACCES, ERR$EXIST, ERR $NOENT
PACKTIMEQQ	no error
PEEKCHARQQ	no error
RENAMEFILEQQ	ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$XDEV
RUNQQ	ERR$NOMEM, ERR$2BIG, ERR$INVAL, ERR$NOENT, ERR$NOEXEC
SETERRORMODEQQ	no error
SETENVQQ	ERR$NOMEM, ERR$INVAL
SETFILEACCESSQQ	ERR$NOMEM, ERR$INVAL, ERR$ACCES
SETFILETIMEQQ	ERR$NOMEM, ERR$ACCES, ERR$INVAL, ERR $MFILE, ERR$NOENT
SLEEPQQ	no error
SORTQQ	ERR$INVAL
SPLITPATHQQ	ERR$NOMEM, ERR$INVAL
SYSTEMQQ	ERR$NOMEM, ERR$2BIG, ERR$NOENT, ERR $NOEXEC
UNPACKTIMEQQ	no error

GETLINESTYLE (W*S)

Graphics Function: *Returns the current graphics line style.*

Module

USE IFQWIN

Syntax

```
result = GETLINESTYLE( )
```

Results

The result type is INTEGER(2). The result is the current line style.

GETLINESTYLE retrieves the mask (line style) used for line drawing. The mask is a 16-bit number, where each bit represents a pixel in the line being drawn.

If a bit is 1, the corresponding pixel is colored according to the current graphics color and logical write mode; if a bit is 0, the corresponding pixel is left unchanged. The mask is repeated for the entire length of the line. The default mask is Z'FFFF' (a solid line). A dashed line can be represented by Z'FF00' (long dashes) or Z'F0F0' (short dashes).

The line style is set with SETLINESTYLE. The current graphics color is set with SETCOLORRGB or SETCOLOR. SETWRITEMODE affects how the line is displayed.

The line style retrieved by GETLINESTYLE affects the drawing of straight lines as in LINETO, POLYGON and RECTANGLE, but not the drawing of curved lines as in ARC, ELLIPSE or PIE.

Example

```
!  Build as Graphics
     USE IFQWIN
     INTEGER(2) lstyle
     lstyle = GETLINESTYLE()
     WRITE (*, 100) lstyle, lstyle
100  FORMAT (1X, 'Line mask in Hex ', Z4, ' and binary ', B16)
     END
```

See Also

LINETO
POLYGON
RECTANGLE
SETCOLORRGB
SETFILLMASK
SETLINESTYLE
SETWRITEMODE

GETLINEWIDTHQQ (W*S)

Graphics Function: *Gets the current line width or the line width set by the last call to SETLINEWIDTHQQ.*

Module

USE IFQWIN

Syntax

```
result = GETLINEWIDTHQQ( )
```

Results

The result is of type INTEGER(4). It contains the current width of the line in pixels.

If there is no call to SETLINEWIDTHQQ in the program (or on that particular graphics window), it returns 1. (The default width of a line drawn by any graphics routine is 1 pixel.)

See Also

SETLINEWIDTHQQ (W*S)

GETLOG

Portability Subroutine: *Returns the user's login name.*

Module

USE IFPORT

Syntax

`CALL GETLOG (name)`

name	(Output) Character*(*). User's login name.

The login name must be less than or equal to 64 characters. If the login name is longer than 64 characters, it is truncated. The actual parameter corresponding to *name* should be long enough to hold the login name. If the supplied actual parameter is too short to hold the login name, the login name is truncated.

If the login name is shorter than the actual parameter corresponding to *name*, the login name is padded with blanks at the end, until it reaches the length of the actual parameter.

If the login name cannot be determined, all blanks are returned.

Example

```
use IFPORT
character*20 username
CALL GETLOG (username)
print *, "You logged in as ",username
```

GETPHYSCOORD (W*S)

Graphics Subroutine: *Translates viewport coordinates to physical coordinates.*

Module

USE IFQWIN

Syntax

`CALL GETPHYSCOORD (x,y,s)`

x,*y*	(Input) INTEGER(2). Viewport coordinates to be translated to physical coordinates.
s	(Output) Derived type `xycoord`. Physical coordinates of the input viewport position. The `xycoord` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with SETVIEWPORT. Both take integer coordinate values. Window coordinates refer to a window sized with SETWINDOW or SETWSIZEQQ. Window coordinates are floating-point values and allow easy scaling of data to the window area.

Example

```
!  Program to demonstrate GETPHYSCOORD, GETVIEWCOORD,
!  and GETWINDOWCOORD. Build as QuickWin or Standard
!  Graphics
USE IFQWIN
TYPE (xycoord)  viewxy, physxy
TYPE (wxycoord) windxy
CALL SETVIEWPORT(INT2(80), INT2(50), &
                 INT2(240), INT2(150))

!  Get viewport equivalent of point (100, 90)
CALL GETVIEWCOORD (INT2(100), INT2(90), viewxy)

!  Get physical equivalent of viewport coordinates
CALL GETPHYSCOORD (viewxy%xcoord, viewxy%ycoord, &
                   physxy)

!  Get physical equivalent of viewport coordinates
CALL GETWINDOWCOORD (viewxy%xcoord, viewxy%ycoord, &
                     windxy)

!  Write viewport coordinates
WRITE (*,*) viewxy%xcoord, viewxy%ycoord
!  Write physical coordinates
WRITE (*,*) physxy%xcoord, physxy%ycoord
!  Write window coordinates
WRITE (*,*) windxy%wx, windxy%wy
END
```

See Also

GETVIEWCOORD
GETWINDOWCOORD
SETCLIPRGN
SETVIEWPORT

GETPID

Portability Function: *Returns the process ID of the current process.*

Module

USE IFPORT

Syntax

```
result = GETPID( )
```

Results

The result type is INTEGER(4). The result is the process ID number of the current process.

Example

```
USE IFPORT
INTEGER(4) istat
istat = GETPID()
```

GETPIXEL, GETPIXEL_W (W*S)

Graphics Functions: *Return the color index of the pixel at a specified location.*

Module

USE IFQWIN

Syntax

```
result = GETPIXEL (x, y)
result = GETPIXEL_W (wx, wy)
```

x, y	(Input) INTEGER(2). Viewport coordinates for pixel position.
wx, wy	(Input) REAL(8). Window coordinates for pixel position.

Results

The result type is INTEGER(2). The result is the pixel color index if successful; otherwise, -1 (if the pixel lies outside the clipping region, for example).

Color routines without the RGB suffix, such as GETPIXEL, use color indexes, not true color values, and limit you to colors in the palette, at most 256. To access all system colors, use SETPIXELRGB to specify an explicit Red-Green-Blue value and retrieve the value with GETPIXELRGB.

NOTE
The GETPIXEL routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the GetPixel routine by including the IFWIN module, you need to specify the routine name as MSFWIN$GetPixel.

See Also

GETPIXELRGB
GRSTATUS
REMAPALLPALETTERGB, REMAPPALETTERGB
SETCOLOR
GETPIXELS
SETPIXEL

GETPIXELRGB, GETPIXELRGB_W (W*S)

Graphics Functions: *Return the Red-Green-Blue (RGB) color value of the pixel at a specified location.*

Module

USE IFQWIN

Syntax

```
result = GETPIXELRGB (x,y)
result = GETPIXELRGB_W (wx,wy)
```

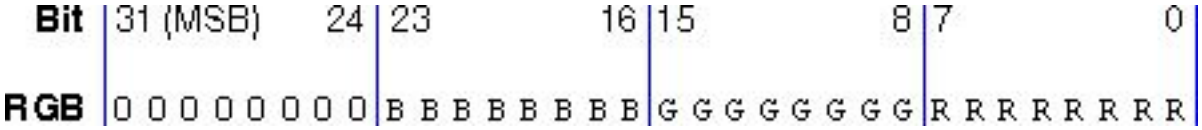
x, y	(Input) INTEGER(2). Viewport coordinates for pixel position.

wx, wy (Input) REAL(8). Window coordinates for pixel position.

Results

The result type is INTEGER(4). The result is the pixel's current RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETPIXELRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETPIXELRGB returns the true color value of the pixel, set with SETPIXELRGB, SETCOLORRGB, SETBKCOLORRGB, or SETTEXTCOLORRGB, depending on the pixel's position and the current configuration of the screen.

SETPIXELRGB (and the other RGB color selection functions SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB) sets colors to a color value chosen from the entire available range. The non-RGB color functions (SETPIXELS, SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(4) pixcolor, rseed
INTEGER(2) status
REAL rnd1, rnd2
LOGICAL(4) winstat
TYPE (windowconfig) wc
CALL GETTIM (status, status, status, INT2(rseed))
CALL SEED (rseed)
CALL RANDOM (rnd1)
CALL RANDOM (rnd2)
!  Get the color index of a random pixel, normalized to
!    be in the window. Then set current color to that
!    pixel color.
winstat = GETWINDOWCONFIG(wc)
xnum = wc%numxpixels
ynum = wc%numypixels
pixcolor = GETPIXELRGB( INT2( rnd1*xnum ), INT2( rnd2*ynum ))
status = SETCOLORRGB (pixcolor)
END
```

See Also

SETPIXELRGB
GETPIXELSRGB
SETPIXELSRGB
GETPIXEL, GETPIXEL_W

GETPIXELS (W*S)

Graphics Subroutine: *Returns the color indexes of multiple pixels.*

Module

USE IFQWIN

Syntax

CALL GETPIXELS (*n*,*x*,*y*,*color*)

n	(Input) INTEGER(4). Number of pixels to get. Sets the number of elements in the other arguments.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels to get.
color	(Output) INTEGER(2). Array to be filled with the color indexes of the pixels at *x* and *y*.

GETPIXELS fills in the array *color* with color indexes of the pixels specified by the two input arrays *x* and *y*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If the pixel is outside the clipping region, the value placed in the *color* array is undefined. Calls to GETPIXELS with *n* less than 1 are ignored. GETPIXELS is a much faster way to acquire multiple pixel color indexes than individual calls to GETPIXEL.

The range of possible pixel color index values is determined by the current video mode and palette, at most 256 colors. To access all system colors you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function such as SETPIXELSRGB and retrieve the value with GETPIXELSRGB, rather than a palette index with a non-RGB color function.

See Also

GETPIXELSRGB
SETPIXELSRGB
GETPIXEL
SETPIXELS

GETPIXELSRGB (W*S)

Graphics Subroutine: *Returns the Red-Green-Blue (RGB) color values of multiple pixels.*

Module

USE IFQWIN

Syntax

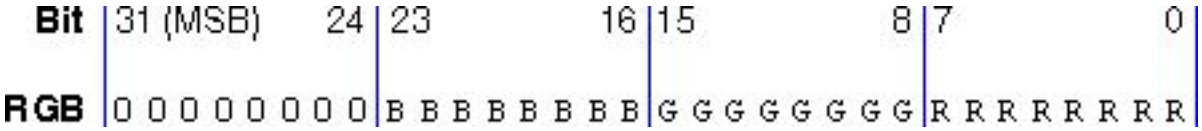
CALL GETPIXELSRGB (*n*,*x*,*y*,*color*)

n	(Input) INTEGER(4). Number of pixels to get. Sets the number of elements in the other argument arrays.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels.

color (Output) INTEGER(4). Array to be filled with RGB color values of the pixels at *x* and *y*.

GETPIXELS fills in the array *color* with the RGB color values of the pixels specified by the two input arrays *x* and *y*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the values you retrieve with GETPIXELSRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 11111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETPIXELSRGB is a much faster way to acquire multiple pixel RGB colors than individual calls to GETPIXELRGB. GETPIXELSRGB returns an array of true color values of multiple pixels, set with SETPIXELSRGB, SETCOLORRGB, SETBKCOLORRGB, or SETTEXTCOLORRGB, depending on the pixels' positions and the current configuration of the screen.

SETPIXELSRGB (and the other RGB color selection functions SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB) sets colors to a color value chosen from the entire available range. The non-RGB color functions (SETPIXELS, SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(4) color(50), result
INTEGER(2) x(50), y(50), status
TYPE (xycoord) pos
result = SETCOLORRGB(Z'FF')
CALL MOVETO(INT2(0), INT2(0), pos)
status = LINETO(INT2(100), INT2(200))
!  Get 50 pixels at line 30 in viewport
DO i = 1, 50
  x(i) = i-1
  y(i) = 30
END DO
CALL GETPIXELSRGB(300, x, y, color)
!  Move down 30 pixels and redisplay pixels
DO i = 1, 50
  y(i) = y(i) + 30
END DO
CALL SETPIXELSRGB (50, x, y, color)
END
```

See Also

SETPIXELSRGB
GETPIXELRGB, GETPIXELRGB_W

GETPIXELS
SETPIXELS

GETPOS, GETPOSI8

Portability Functions: *Return the current position of a file.*

Module

USE IFPORT

Syntax

`result = GETPOS (lunit)`

`result = GETPOSI8 (lunit)`

lunit	(Input) INTEGER(4). External unit number of a file. The value must be in the range 0 to 100 and the file must be connected.

Results

The result type is INTEGER(4) for GETPOS; INTEGER(8) for GETPOSI8. The result is the offset, in bytes, from the beginning of the file. If an error occurs, the result value is -1 and the following error code is returned in `errno`:

EINVAL: *lunit* is not a valid unit number, or is not open.

These functions are equivalent to FTELL, FTELLI8.

GETSTATUSFPQQ (W*S)

Portability Subroutine: *Returns the floating-point processor status word.*

Module

USE IFPORT

Syntax

`CALL GETSTATUSFPQQ (status)`

status	(Output) INTEGER(2). Floating-point processor status word.

The floating-point status word shows whether various floating-point exception conditions have occurred. The compiler initially clears (sets to 0) all status flags, but after an exception occurs it does not reset the flags before performing additional floating-point operations. A status flag with a value of one thus shows there has been at least one occurrence of the corresponding exception. The following table lists the status flags and their values:

Parameter name	Hex value	Description
FPSW$MSW_EM	Z'003F'	Status Mask (set all flags to 1)
FPSW$INVALID	Z'0001'	An invalid result occurred
FPSW$DENORMAL	Z'0002'	A subnormal (very small number) occurred

Parameter name	Hex value	Description
FPSW$SUBNORMAL	Z'0002'	A subnormal (very small number) occurred
FPSW$ZERODIVIDE	Z'0004'	A divide by zero occurred
FPSW$OVERFLOW	Z'0008'	An overflow occurred
FPSW$UNDERFLOW	Z'0010'	An underflow occurred
FPSW$INEXACT	Z'0020'	Inexact precision occurred

You can use a logical comparison on the status word returned by GETSTATUSFPQQ to determine which of the six floating-point exceptions listed in the table has occurred.

An exception is disabled if its control bit is set to 1. An exception is enabled if its control bit is cleared to 0. By default, all exception traps are disabled. Exceptions can be enabled and disabled by clearing and setting the flags with SETCONTROLFPQQ. You can use GETCONTROLFPQQ to determine which exceptions are currently enabled and disabled.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues. You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ.

If errors on floating-point exceptions are enabled (by clearing the flags to 0 with SETCONTROLFPQQ), the operating system generates an interrupt when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

Example

```
!  Program to demonstrate GETSTATUSFPQQ
 USE IFPORT
 INTEGER(2) status
 CALL GETSTATUSFPQQ(status)
!  check for divide by zero
 IF (IAND(status, FPSW$ZERODIVIDE) .NE. 0) THEN
   WRITE (*,*) 'Divide by zero occurred. Look    &
   for NaN or signed infinity in resultant data.'
 END IF
END
```

See Also

SETCONTROLFPQQ
GETCONTROLFPQQ
SIGNALQQ
CLEARSTATUSFPQQ

GETSTRQQ

Run-Time Function: *Reads a character string from the keyboard using buffered input.*

Module

USE IFCORE

Syntax

```
result = GETSTRQQ (buffer)
```

buffer	(Output) Character*(*). Character string returned from keyboard, padded on the right with blanks.

Results

The result type is INTEGER(4). The result is the number of characters placed in *buffer*.

The function does not complete until you press Return or Enter.

Example

```
!  Program to demonstrate GETSTRQQ
 USE IFCORE
 USE IFPORT
 INTEGER(4) length, result
 CHARACTER(80) prog, args
 WRITE (*, '(A, \)') ' Enter program to run: '
 length = GETSTRQQ (prog)
 WRITE (*, '(A, \)') ' Enter arguments: '
 length = GETSTRQQ (args)
 result = RUNQQ (prog, args)
 IF (result .EQ. -1) THEN
   WRITE (*,*) 'Couldn''t run program'
 ELSE
   WRITE (*, '(A, Z4, A)') 'Return code : ', result, 'h'
 END IF
 END
```

See Also

READ
GETCHARQQ
PEEKCHARQQ

GETTEXTCOLOR (W*S)

Graphics Function: *Returns the current text color index.*

Module

USE IFQWIN

Syntax

```
result = GETTEXTCOLOR( )
```

Results

The result type is INTEGER(2). It is the current text color index.

GETTEXTCOLOR returns the text color index set by SETTEXTCOLOR. SETTEXTCOLOR affects text output with OUTTEXT, WRITE, and PRINT. The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use SETTEXTCOLORRGB, SETBKCOLORRGB, and SETCOLORRGB.

The default text color index is 15, which is associated with white unless the user remaps the palette.

NOTE
The GETTEXTCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the GetTextColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$GetTextColor.

See Also
OUTTEXT
REMAPPALETTERGB
SETCOLOR
SETTEXTCOLOR

GETTEXTCOLORRGB (W*S)

Graphics Function: *Returns the Red-Green-Blue (RGB) value of the current text color (used with OUTTEXT, WRITE and PRINT).*

Module

USE IFQWIN

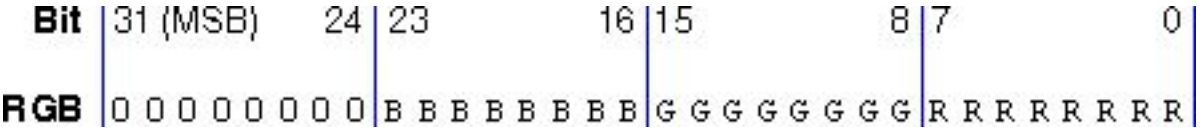
Syntax

```
result = GETTEXTCOLORRGB( )
```

Results

The result type is INTEGER(4). It is the RGB value of the current text color.

In each RGB color calue, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT), set with SETTEXTCOLORRGB. The RGB color value used for graphics is set and returned with SETCOLORRGB and GETCOLORRGB. SETCOLORRGB controls the color used by the graphics function OUTGTEXT, while SETTEXTCOLORRGB controls the color used by all other text output functions. The RGB background color value for both text and graphics is set and returned with SETBKCOLORRGB and GETBKCOLORRGB.

SETTEXTCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB, and SETCOLORRGB) sets the color to a color value chosen from the entire available range. The non-RGB color functions (SETTEXTCOLOR, SETBKCOLOR, and SETCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(4) oldtextc, oldbackc, temp
TYPE (rccoord) curpos
!  Save color settings
oldtextc = GETTEXTCOLORRGB()
oldbackc = GETBKCOLORRGB()
CALL CLEARSCREEN( $GCLEARSCREEN )
!  Reset colors
temp = SETTEXTCOLORRGB(Z'00FFFF')  ! full red + full green
                                   !  = full yellow text
temp = SETBKCOLORRGB(Z'FF0000')    ! blue background
CALL SETTEXTPOSITION( INT2(4), INT2(15), curpos)
CALL OUTTEXT( 'Hello, world')
!  Restore colors
temp = SETTEXTCOLORRGB(oldtextc)
temp = SETBKCOLORRGB(oldbackc)
END
```

See Also

SETTEXTCOLORRGB
GETBKCOLORRGB
GETCOLORRGB
GETTEXTCOLOR

GETTEXTPOSITION (W*S)

Graphics Subroutine: *Returns the current text position.*

Module

USE IFQWIN

Syntax

`CALL GETTEXTPOSITION (s)`

s

(Output) Derived type `rccoord`. Current text position. The derived type `rccoord` is defined in `IFQWIN.F90` as follows:

```
TYPE rccoord
  INTEGER(2) row   ! Row coordinate
  INTEGER(2) col   ! Column coordinate
END TYPE rccoord
```

The text position given by coordinates (1, 1) is defined as the upper-left corner of the text window. Text output from the OUTTEXT function (and WRITE and PRINT statements) begins at the current text position. Font text is not affected by the current text position. Graphics output, including OUTGTEXT output, begins at the current graphics output position, which is a separate position returned by GETCURRENTPOSITION.

Example

```
!  Build as QuickWin or Standard Graphics
     USE IFQWIN
     TYPE (rccoord) textpos
     CALL GETTEXTPOSITION (textpos)
     END
```

See Also
SETTEXTPOSITION
GETCURRENTPOSITION
OUTTEXT
WRITE
SETTEXTWINDOW

GETTEXTWINDOW (W*S)

Graphics Subroutine: *Finds the boundaries of the current text window.*

Module

USE IFQWIN

Syntax

CALL GETTEXTWINDOW (*r1*,*c1*,*r2*,*c2*)

r1, c1	(Output) INTEGER(2). Row and column coordinates for upper-left corner of the text window.
r2, c2	(Output) INTEGER(2). Row and column coordinates for lower-right corner of the text window.

Output from OUTTEXT and WRITE is limited to the text window. By default, this is the entire window, unless the text window is redefined by SETTEXTWINDOW.

The window defined by SETTEXTWINDOW has no effect on output from OUTGTEXT.

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(2) top, left, bottom, right
DO i = 1, 10
  WRITE(*,*) "Hello, world"
END DO
!  Save text window position
 CALL GETTEXTWINDOW (top, left, bottom, right)
!  Scroll text window down seven lines
 CALL SCROLLTEXTWINDOW (INT2(-7))
!  Restore text window
 CALL SETTEXTWINDOW (top, left, bottom, right)
 WRITE(*,*) "At beginning again"
 END
```

See Also
GETTEXTPOSITION
OUTTEXT
WRITE
SCROLLTEXTWINDOW
SETTEXTPOSITION
SETTEXTWINDOW
WRAPON

GETTIM

Portability Subroutine: *Returns the time.*

Module

USE IFPORT

Syntax

`CALL GETTIM (ihr, imin, isec, i100th)`

ihr	(Output) INTEGER(4) or INTEGER(2). Hour (0-23).
imin	(Output) INTEGER(4) or INTEGER(2). Minute (0-59).
isec	(Output) INTEGER(4) or INTEGER(2). Second (0-59).
i100th	(Output) INTEGER(4) or INTEGER(2). Hundredths of a second (0-99).

All arguments must be of the same integer kind, that is, all must be INTEGER(2) or all must be INTEGER(4).

If INTEGER(2) arguments are passed, you must specify USE IFPORT.

Example

See the example in GETDAT.

See Also

GETDAT

SETDAT

SETTIM

CLOCK

CTIME

DTIME

ETIME

GMTIME

ITIME

LTIME

RTC

SECNDS portability routine

TIME portability routine

TIMEF

GETTIMEOFDAY

Portability Subroutine: *Returns seconds and microseconds since 00:00 Jan 1, 1970.*

Module

USE IFPORT

Syntax

```
CALL GETTIMEOFDAY (ret, err)
```

ret (Output) INTEGER(4). One-dimensional array with 2 elements used to contain numeric time data. The elements of *ret* are returned as follows:

Element	Value
ret(1)	Seconds
ret(2)	Microseconds

err (Output) INTEGER(4).

If an error occurs, *err* contains a value equal to -1 and array *ret* contains zeros.

On Windows* systems, this subroutine has millisecond precision, and the last three digits of the returned value are not significant.

GETUID

Portability Function: *Returns the user ID of the calling process.*

Module

USE IFPORT

Syntax

```
result = GETUID( )
```

Results

The result type is INTEGER(4). The result corresponds to the user identity under which the program is running. The result is returned as follows:

- On Windows* systems, this function returns the last subauthority of the security identifier for the process. This is unique on a local machine and unique within a domain for domain accounts.

  Note that on Windows systems, domain accounts and local accounts can overlap.
- On Linux* and macOS* systems, this function returns the user identity for the current process.

Example

```
USE IFPORT
integer(4) istat
istat = GETUID( )
```

GETUNITQQ (W*S)

QuickWin Function: *Returns the unit number corresponding to the specified Windows* handle.*

Module

USE IFQWIN

Syntax

`result = GETUNITQQ (whandle)`

whandle	(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The Windows handle to the window; this is a unique ID.

Results

The result type is INTEGER(4). The result is the unit number corresponding to the specified Windows handle. If *whandle* does not exist, it returns -1 .

This routine is the inverse of GETHWNDQQ.

See Also

GETHWNDQQ

GETVIEWCOORD, GETVIEWCOORD_W (W*S)

Graphics Subroutines: *Translate physical coordinates or window coordinates to viewport coordinates.*

Module

USE IFQWIN

Syntax

`CALL GETVIEWCOORD (x, y, s)`

`CALL GETVIEWCOORD_W (wx, wy, s)`

x, y	(Input) INTEGER(2). Physical coordinates to be converted to viewport coordinates.
wx, wy	(Input) REAL(8). Window coordinates to be converted to viewport coordinates.
s	(Output) Derived type `xycoord`. Viewport coordinates. The `xycoord` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

Viewport coordinates refer to an area of the screen defined as the viewport with SETVIEWPORT. Physical coordinates refer to the whole screen. Both take integer coordinate values. Window coordinates refer to a window sized with SETWINDOW or SETWSIZEQQ. Window coordinates are floating-point values and allow easy scaling of data to the window area.

Example

See the example program in GETPHYSCOORD.

See Also

GETPHYSCOORD
GETWINDOWCOORD

GETWINDOWCONFIG (W*S)

QuickWin Function: *Returns the properties of the current window.*

Module

USE IFQWIN

Syntax

`result = GETWINDOWCONFIG (wc)`

wc

(Output) Derived type `windowconfig`. Contains window properties. The `windowconfig` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE windowconfig
  INTEGER(2) numxpixels             ! Number of pixels on x-
axis
  INTEGER(2) numypixels             ! Number of pixels on y-
axis
  INTEGER(2) numtextcols            ! Number of text columns
available
  INTEGER(2) numtextrows            ! Number of text rows
available
  INTEGER(2) numcolors              ! Number of color indexes
  INTEGER(4) fontsize           ! Size of default font. Set to
                                !   QWIN$EXTENDFONT when
specifying
                                !   extended attributes, in
which
                                !   case extendfontsize sets
the
                                !   font size
  CHARACTER(80) title               ! The window title
  INTEGER(2) bitsperpixel           ! The number of bits per
pixel
  INTEGER(2) numvideopages          ! Unused
  INTEGER(2) mode                   ! Controls scrolling mode
  INTEGER(2) adapter                ! Unused
  INTEGER(2) monitor                ! Unused
  INTEGER(2) memory                 ! Unused
  INTEGER(2) environment            ! Unused
!
! The next three parameters provide extended font attributes.
!
  CHARACTER(32) extendfontname      ! The name of the desired
font
  INTEGER(4) extendfontsize         ! Takes the same values
as fontsize,
                                    !   when fontsize is set
```

```
to
                                        !   QWIN$EXTENDFONT
  INTEGER(4) extendfontattributes       ! Font attributes such as
bold
                                        !  and italic
END TYPE windowconfig
```

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE. (for example, if there is no active child window).

GETWINDOWCONFIG returns information about the active child window. If you have not set the window properties with SETWINDOWCONFIG, GETWINDOWCONFIG returns default window values.

A typical set of values would be 1024 *x* pixels, 768 *y* pixels, 128 text columns, 48 text rows, and a font size of 8x16 pixels. The resolution of the display and the assumed font size of 8x16 pixels generates the number of text rows and text columns. The resolution (in this case, 1024 *x* pixels by 768 *y* pixels) is the size of the *virtual* window. To get the size of the *physical* window visible on the screen, use GETWSIZEQQ. In this case, GETWSIZEQQ returned the following values: (0,0) for the x and y position of the physical window, 25 for the height or number of rows, and 71 for the width or number of columns.

The number of colors returned depends on the video drive. The window title defaults to "Graphic1" for the default window. All of these values can be changed with SETWINDOWCONFIG.

Note that the bitsperpixel field in the `windowconfig` derived type is an output field only, while the other fields return output values to GETWINDOWCONFIG and accept input values from SETWINDOWCONFIG.

Example

```
!Build as QuickWin or Standard Graphics App.
USE IFQWIN
LOGICAL(4) status
TYPE (windowconfig) wc
status = GETWINDOWCONFIG(wc)
IF(wc%numtextrows .LT. 10) THEN
  wc%numtextrows = 10
  status = SETWINDOWCONFIG(wc)
  IF(.NOT. status ) THEN  ! if setwindowconfig error
    status = SETWINDOWCONFIG(wc)   ! reset
             ! setwindowconfig with corrected values
    status = GETWINDOWCONFIG(wc)
    IF(wc%numtextrows .NE. 10) THEN
       WRITE(*,*) 'Error: Cannot increase text rows to 10'
    END IF
  END IF
END IF
END
```

See Also

GETWSIZEQQ
SETWINDOWCONFIG
SETACTIVEQQ

GETWINDOWCOORD (W*S)

Graphics Subroutine: *Converts viewport coordinates to window coordinates.*

Module

USE IFQWIN

Syntax

```
CALL GETWINDOWCOORD (x,y,s)
```

x,*y* — (Input) INTEGER(2). Viewport coordinates to be converted to window coordinates.

s — (Output) Derived type `wxycoord`. Window coordinates. The `wxycoord` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  REAL(8) wx   ! x-coordinate
  REAL(8) wy   ! y-coordinate
END TYPE wxycoord
```

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with SETVIEWPORT. Both take integer coordinate values. Window coordinates refer to a window sized with SETWINDOW or SETWSIZEQQ. Window coordinates are floating-point values and allow easy scaling of data to the window area.

Example

See the example program in GETPHYSCOORD.

See Also

GETCURRENTPOSITION
GETPHYSCOORD
GETVIEWCOORD
MOVETO
SETVIEWPORT
SETWINDOW

GETWRITEMODE (W*S)

Graphics Function: *Returns the current logical write mode, which is used when drawing lines with the LINETO, POLYGON, and RECTANGLE functions.*

Module

USE IFQWIN

Syntax

```
result = GETWRITEMODE( )
```

Results

The result type is INTEGER(2). The result is the current write mode. Possible return values are:

- $GPSET - Causes lines to be drawn in the current graphics color. (default)
- $GAND - Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.
- $GOR - Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.
- $GPRESET - Causes lines to be drawn in the color that is the logical NOT of the current graphics color.

- $GXOR - Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.

The default value is $GPSET. These constants are defined in IFQWIN.F90.

The write mode is set with SETWRITEMODE.

Example

```
! Build as QuickWin or Standard Graphics App.
   USE IFQWIN
   INTEGER(2) mode
   mode = GETWRITEMODE()
   END
```

See Also

SETWRITEMODE
SETLINESTYLE
LINETO
POLYGON
PUTIMAGE
RECTANGLE
SETCOLORRGB
SETFILLMASK
GRSTATUS

GETWSIZEQQ (W*S)

QuickWin Function: *Returns the size and position of a window.*

Module

USE IFQWIN

Syntax

`result = GETWSIZEQQ (unit, ireq, winfo)`

unit
(Input) INTEGER(4). Specifies the window unit. Unit numbers 0, 5 and 6 refer to the default startup window only if you have not explicitly opened them with the OPEN statement. To access information about the frame window (as opposed to a child window), set *unit* to the symbolic constant QWIN$FRAMEWINDOW, defined in IFQWIN.F90.

ireq
(Input) INTEGER(4). Specifies what information is obtained. The following symbolic constants, defined in IFQWIN.F90, are available:

- QWIN$SIZEMAX - Gets information about the maximum window size.
- QWIN$SIZECURR - Gets information about the current window size.

winfo

(Output) Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame). The derived type qwinfois defined in IFQWIN.F90 as follows:

```
TYPE QWINFO
  INTEGER(2) TYPE  ! request type (controls
                   !   SETWSIZEQQ)
  INTEGER(2) X     ! x coordinate for upper left
  INTEGER(2) Y     ! y coordinate for upper left
  INTEGER(2) H     ! window height
  INTEGER(2) W     ! window width
END TYPE QWINFO
```

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width returned for a frame window reflects the size in pixels of the client area *excluding* any borders, menus, and status bar at the bottom of the frame window. You should adjust the values used in SETWSIZEQQ to take this into account.

The client area is the area actually available to place child windows.

See Also

GETWINDOWCONFIG
SETWSIZEQQ

GMTIME

Portability Subroutine: *Returns the Greenwich mean time in an array of time elements.*

Module

USE IFPORT

Syntax

CALL GMTIME (*stime*,*tarray*)

stime

(Input) INTEGER(4). Numeric time data to be formatted. Number of seconds since 00:00:00 Greenwich mean time, January 1, 1970.

tarray

(Output) INTEGER(4). One-dimensional array with 9 elements used to contain numeric time data. The elements of *tarray* are returned as follows:

Element	Value
tarray(1)	Seconds (0-61, where 60-61 can be returned for leap seconds)
tarray(2)	Minutes (0-59)
tarray(3)	Hours (0-23)

Element	Value
tarray(4)	Day of month (1-31)
tarray(5)	Month (0-11)
tarray(6)	Number of years since 1900
tarray(7)	Day of week (0-6, where 0 is Sunday)
tarray(8)	Day of year (0-365)
tarray(9)	Daylight saving flag (0 if standard time, 1 if daylight saving time)

Caution
This subroutine may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

```
use IFPORT
integer(4) stime, timearray(9)
! initialize stime to number of seconds since
!   00:00:00 GMT January 1, 1970
stime = time()
CALL GMTIME (stime, timearray)
print *, timearray
end
```

See Also
DATE_AND_TIME

GOTO - Assigned

Statement: *Transfers control to the statement whose label was most recently assigned to a variable. This feature has been deleted in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.*

Syntax

GOTO *var*[[,] (*label-list*)]

var — Is a scalar integer variable.

label-list — Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the assigned GO TO statement. The same label can appear more than once in this list.

The variable must have a statement label value assigned to it by an ASSIGN statement (not an arithmetic assignment statement) before the GO TO statement is executed.

If a list of labels appears, the statement label assigned to the variable is not checked against the labels in the list.

Both the assigned GO TO statement and its associated ASSIGN statement must be in the same scoping unit.

Example

The following example is equivalent to `GO TO 200`:

```
ASSIGN 200 TO IGO
GO TO IGO
```

The following example is equivalent to `GO TO 450`:

```
ASSIGN 450 TO IBEG
GO TO IBEG, (300,450,1000,25)
```

The following example shows an invalid use of an assigned variable:

```
ASSIGN 10 TO I
J = I
GO TO J
```

In this case, variable J is not the variable assigned to, so it cannot be used in the assigned GO TO statement.

The following shows another example:

```
    ASSIGN 10 TO N
    GOTO N
10  CONTINUE
```

The following example uses an assigned GOTO statement with a *label-list* but it is not checked against the value of VIEW:

```
ASSIGN 300 TO VIEW
    GOTO VIEW (100, 200, 400)   ! this will go to label 300
```

See Also

Obsolescent Language Features in the Fortran Standard
GOTO - Computed GOTO
GOTO - Unconditional GOTO
Execution Control

GOTO - Computed

Statement: *Transfers control to one of a set of labeled branch target statements based on the value of an expression. It is an obsolescent feature in Fortran 95.*

Syntax

`GOTO (`*`label-list`*`) [ ,] `*`expr`*

`label-list`	Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the computed GO TO statement. (Also called the *transfer list*.) The same label can appear more than once in this list.
`expr`	Is a scalar numeric expression in the range 1 to n, where *n* is the number of statement labels in *`label-list`*. If necessary, it is converted to integer data type.

When the computed GO TO statement is executed, the expression is evaluated first. The value of the expression represents the ordinal position of a label in the associated list of labels. Control is transferred to the statement identified by the label. For example, if the list contains (30,20,30,40) and the value of the expression is 2, control is transferred to the statement identified with label 20.

If the value of the expression is less than 1 or greater than the number of labels in the list, control is transferred to the next executable statement or construct following the computed GO TO statement.

Example

The following example shows valid computed GO TO statements:

```
GO TO (12,24,36), INDEX
GO TO (320,330,340,350,360), SITU(J,K) + 1
```

The following shows another example:

```
   next = 1
C
C  The following statement transfers control to statement 10:
C
   GOTO (10, 20) next
   ...
10 CONTINUE
   ...
20 CONTINUE
```

See Also

Obsolescent Language Features in the Fortran Standard
GOTO - Unconditional GOTO
Execution Control

GOTO - Unconditional

Statement: *Transfers control to the same branch target statement every time it executes.*

Syntax

`GO TO` *label*

label	Is the label of a valid branch target statement in the same scoping unit as the GO TO statement.

The unconditional GO TO statement transfers control to the branch target statement identified by the specified label.

Example

The following are examples of GO TO statements:

```
  GO TO 7734
  GO TO 99999
```

The following shows another example:

```
    integer(2) in
10  print *, 'enter a number from one to ten: '
    read *, in
    select case (in)
    case (1:10)
      exit
```

```
    case default
      print *, 'wrong entry, try again'
      goto 10
    end select
```

See Also

GOTO - Computed GOTO
Execution Control

GRSTATUS (W*S)

Graphics Function: *Returns the status of the most recently used graphics routine.*

Module

USE IFQWIN

Syntax

```
result = GRSTATUS( )
```

Results

The result type is INTEGER(2). The result is the status of the most recently used graphics function.

Use GRSTATUS immediately following a call to a graphics routine to determine if errors or warnings were generated. Return values less than 0 are errors, and values greater than 0 are warnings.

The following symbolic constants are defined in the `IFQWIN.F90` module file for use with GRSTATUS:

Constant	Meaning
$GRFILEWRITEERROR	Error writing bitmap file
$GRFILEOPENERROR	Error opening bitmap file
$GRIMAGEREADERROR	Error reading image
$GRBITMAPDISPLAYERROR	Error displaying bitmap
$GRBITMAPTOOLARGE	Bitmap too large
$GRIMPROPERBITMAPFORMAT	Improper format for bitmap file
$GRFILEREADERROR	Error reading file
$GRNOBITMAPFILE	No bitmap file
$GRINVALIDIMAGEBUFFER	Image buffer data inconsistent
$GRINSUFFICIENTMEMORY	Not enough memory to allocate buffer or to complete a fill operation
$GRINVALIDPARAMETER	One or more parameters invalid
$GRMODENOTSUPPORTED	Requested video mode not supported
$GRERROR	Graphics error
$GROK	Success

Constant	Meaning
$GRNOOUTPUT	No action taken
$GRCLIPPED	Output was clipped to viewport
$GRPARAMETERALTERED	One or more input parameters was altered to be within range, or pairs of parameters were interchanged to be in the proper order

After a graphics call, compare the return value of GRSTATUS to $GROK. to determine if an error has occurred. For example:

```
IF ( GRSTATUS .LT. $GROK ) THEN
! Code to handle graphics error goes here
ENDIF
```

The following routines cannot give errors, and they all set GRSTATUS to $GROK:

DISPLAYCURSOR	GETCOLORRGB	GETTEXTWINDOW
GETBKCOLOR	GETTEXTCOLOR	OUTTEXT
GETBKCOLORRGB	GETTEXTCOLORRGB	WRAPON
GETCOLOR	GETTEXTPOSITION	

The following table lists some other routines with the error or warning messages they produce for GRSTATUS:

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
ARC, ARC_W	$GRINVALIDPARAMETER	$GRNOOUTPUT
CLEARSCREEN	$GRINVALIDPARAMETER	
ELLIPSE, ELLIPSE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
FLOODFILLRGB	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
GETARCINFO	$GRERROR	
GETFILLMASK	$GRERROR, $GRINVALIDPARAMETER	
GETFONTINFO	$GRERROR	
GETGTEXTEXTENT	$GRERROR	
GETIMAGE	$GRINSUFFICIENTMEMORY	$GRPARAMETERALTERED
GETPIXEL	$GRBITMAPTOOLARGE	
GETPIXELRGB	$GRBITMAPTOOLARGE	
LINETO, LINETO_W		$GRNOOUTPUT, $GRCLIPPED

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
LOADIMAGE	$GRFILEOPENERROR, $GRNOBITMAPFILE, $GRALEREADERROR, $GRIMPROPERBITMAPFORMAT, $GRBITMAPTOOLARGE, $GRIMAGEREADERROR	
OUTGTEXT		$GRNOOUTPUT, $GRCLIPPED
PIE, PIE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
POLYGON, POLYGON_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT, $GRCLIPPED
PUTIMAGE, PUTIMAGE_W	$GRERROR, $GRINVALIDPARAMETER, $GRINVALIDIMAGEBUFFER $GRBITMAPDISPLAYERROR	$GRPARAMETERALTERED, $GRNOOUTPUT
RECTANGLE, RECTANGLE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT, $GRCLIPPED
REMAPPALETTERGB	$GRERROR, $GRINVALIDPARAMETER	
REMAPALLPALETTERGB	$GRERROR, $GRINVALIDPARAMETER	
SAVEIMAGE	$GRFILEOPENERROR	
SCROLLTEXTWINDOW		$GRNOOUTPUT
SETBKCOLOR	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETBKCOLORRGB	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETCLIPRGN		$GRPARAMETERALTERED
SETCOLOR		$GRPARAMETERALTERED
SETFONT	$GRERROR, $GRINSUFFICIENTMEMORY	$GRPARAMETERALTERED
SETPIXEL, SETPIXEL_W		$GRNOOUTPUT
SETPIXELRGB, SETPIXELRGB_W		$GRNOOUTPUT
SETTEXTCOLOR		$GRPARAMETERALTERED
SETTEXTCOLORRGB		$GRPARAMETERALTERED
SETTEXTPOSITION		$GRPARAMETERALTERED
SETTEXTWINDOW		$GRPARAMETERALTERED
SETVIEWPORT		$GRPARAMETERALTERED

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
SETWINDOW	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETWRITEMODE	$GRINVALIDPARAMETER	

See Also

ARC
ELLIPSE
FLOODFILLRGB
LINETO
PIE
POLYGON
REMAPALLPALETTERGB
SETBKCOLORRGB
SETCOLORRGB
SETPIXELRGB
SETTEXTCOLORRGB
SETWINDOW
SETWRITEMODE

H to I

H to I

HOSTNAM

Portability Function: *Returns the current host computer name. This function can also be specified as HOSTNM.*

Module

USE IFPORT

Syntax

```
result = HOSTNAM (name)
```

name	(Output) Character*(*). Name of the current host. Should be at least as long as MAX_HOSTNAM_LENGTH + 1. MAX_HOSTNAM_LENGTH is defined in the IFPORT module.

Results

The result type is INTEGER(4). The result is zero if successful. If *name* is not long enough to contain all of the host name, the function truncates the host name and returns -1.

Example

```
use IFPORT
character(MAX_HOSTNAM_LENGTH + 1) hostname
integer(4) istat
ISTAT = HOSTNAM (hostname)
```

HUGE

Inquiry Intrinsic Function (Generic): *Returns the largest number in the model representing the same type and kind parameters as the argument.*

Syntax

`result = HUGE (x)`

x	(Input) Must be of type integer or real; it can be scalar or array valued.

Results

The result is a scalar of the same type and kind parameters as *x*. If *x* is of type integer, the result has the value $r^q - 1$. If *x* is of type real, the result has the value $(1 - b^{-p})b^{e_{max}}$.

Integer parameters *r* and *q* are defined in Model for Integer Data; real parameters *b*, *p*, and e_{max} are defined in Model for Real Data.

Example

If X is of type REAL(4), HUGE (X) has the value $(1 - 2^{-24}) \times 2^{128}$.

See Also

TINY

Data Representation Models

HYPOT

Elemental Intrinsic Function (Generic): *Returns the value of the Euclidean distance of the arguments.*

Syntax

`result = HYPOT (x,y)`

x	(Input) Must be of type real.
y	(Input) Must be of type real. It must be the same type and kind as *x*.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the Euclidean distance, $\sqrt{X^2 + Y^2}$, without undue overflow or underflow.

Example

HYPOT (3.0, 4.0) has the approximate value 5.0.

IACHAR

Elemental Intrinsic Function (Generic): *Returns the position of a character in the ASCII character set, even if the processor's default character set is different. In Intel® Fortran, IACHAR is equivalent to the ICHAR function.*

Syntax

```
result = IACHAR (c [, kind])
```

c	(Input) Must be of type character of length 1.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by kind; otherwise, the kind parameter of the result is that of default integer. If *c* is in the ASCII collating sequence, the result is the position of *c* in that sequence and satisfies the inequality (0 .le. IACHAR(c) .le. 127).

The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE(C, D) is true, IACHAR(C) .LE. IACHAR(D) is also true.

Example

IACHAR ('Y') has the value 89.

IACHAR ('%') has the value 37.

See Also

ASCII and Key Code Charts
ACHAR
CHAR
ICHAR
LGE
LGT
LLE
LLT

IALL

Transformational Intrinsic Function (Generic): *Reduces an array with a bitwise AND operation.*

Syntax

```
result = IALL (array, dim [, mask])
```

```
result = IALL (array [, mask])
```

array	(Input) Must be an array of type integer.
dim	(Input) Must be a scalar integer with a value in the range 1 <= *dim* <= *n*, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result has the same type and kind parameters as *array*. It is scalar if *dim* does not appear or if *array* has rank one; otherwise, the result is an array of rank *n* - 1 and shape [d_1, d_2, ..., d_{dim-1}, d_{dim+1} , ..., d_n] where [d_1, d_2,..., d_n] is the shape of *array*.

If *array* has size zero the result value is equal to NOT (INT (0, KIND (*array*))). Otherwise, the result of IALL (*array*) has a value equal to the bitwise AND of all the elements of *array*.

The result of IALL (*array*, MASK=*mask*) has a value equal to IALL (PACK (*array*, *mask*)).

The result of IALL (*array*, DIM=*dim* [, MASK=*mask*]) has a value equal to that of IALL (*array* [, MASK=*mask*]) if *array* has rank one. Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1} , ..., s_n) of the result is equal to IALL (*array* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n) [, MASK = *mask* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n)]).

Example

IALL ([14, 13, 11]) has the value 8. IALL ([14, 13, 11], MASK=[.true., .false., .true]) has the value 10.

See Also

IANY
IPARITY

IAND

Elemental Intrinsic Function (Generic): *Performs a logical AND on corresponding bits. This function can also be specified as AND.*

Syntax

```
result = IAND (i,j)
```

i — (Input) Must be of type integer or logical (which is treated as an integer), or a binary, octal, or hexadecimal literal constant.

j — (Input) Must be of type integer or logical, or a binary, octal, or hexadecimal literal constant.

If both *i* and *j* are of type integer or logical, they must have the same kind type parameter. If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with its sign bit on the left and the larger kind is used for the operation and the result. *i* and *j* must not both be binary, octal, or hexadecimal literal constants.

Results

If both *i* and *j* are of type integer or logical, the result type and kind are the same as *i*. If either *i* or *j* is a binary, octal, or hexadecimal literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other.

The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

i	j	IAND (i, j)
1	1	1
1	0	0
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BIAND	INTEGER(1)	INTEGER(1)
IIAND [1]	INTEGER(2)	INTEGER(2)
JIAND	INTEGER(4)	INTEGER(4)
KIAND	INTEGER(8)	INTEGER(8)

[1]Or HIAND.

Example

IAND (2, 3) has the value 2.

IAND (4, 6) has the value 4.

The following shows another example:

```
INTEGER(1) i, m
INTEGER result
INTEGER(2) result2
i = 1
m = 3
result = IAND(i,m) ! returns an integer of default type
        ! (INTEGER(4) unless reset by user) whose
        ! value = 1
result2 = IAND(i,m) ! returns an INTEGER(2) with value = 1
```

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
IEOR
IOR
NOT
IALL
IANY
IPARITY

IANY

Transformational Intrinsic Function (Generic):
Reduces an array with a bitwise OR operation.

Syntax

`result = IANY (array, dim [, mask])`

`result = IANY (array [, mask])`

array	(Input) Must be an array of type integer.
dim	(Input) Must be a scalar integer with a value in the range 1 <= *dim* <= *n*, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result has the same type and kind parameters as *array*. It is scalar if *dim* does not appear; otherwise, the result is an array of rank *n* - 1 and shape [d_1, d_2, ..., d_{dim-1}, d_{dim+1} , ..., d_n] where [d_1, d_2,..., d_n] is the shape of *array*.

The result of IANY (*array*) is the bitwise OR of all the elements of *array*. If *array* has size zero, the result value is equal to zero.

The result of IANY (*array*, MASK=*mask*) has a value equal to IANY (PACK (*array*, *mask*)).

The result of IANY (*array*, DIM=*dim* [, MASK=*mask*]) has a value equal to that of IANY (*array* [, MASK=*mask*]) if *array* has rank one. Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1} , ..., s_n) of the result is equal to IANY (*array* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n) [, MASK = *mask* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n)]).

Example

IANY ([14, 13, 8]) has the value 15. IANY ([14, 13, 8], MASK=[.true., .false., .true]) has the value 14.

See Also

IALL
IPARITY

IARGC

Inquiry Intrinsic Function (Specific): *Returns the index of the last command-line argument. It cannot be passed as an actual argument. This function can also be specified as IARG or NUMARG.*

Syntax

```
result = IARGC( )
```

Results

The result type is INTEGER(4). The result is the index of the last command-line argument, which is also the number of arguments on the command line. The command is not included in the count. For example, IARGC returns 3 for the command-line invocation of `PROG1 -g -c -a`.

IARGC returns a value that is 1 less than that returned by NARGS.

Example

```
integer(4) no_of_arguments
no_of_arguments = IARGC ()
print *, 'total command line arguments are ', no_of_arguments
```

For a command-line invocation of `PROG1 -g -c -a`, the program above prints:

```
total command line arguments are 3
```

See Also

GETARG
NARGS
COMMAND_ARGUMENT_COUNT
GET_COMMAND
GET_COMMAND_ARGUMENT

IBCHNG

Elemental Intrinsic Function (Generic): *Reverses the value of a specified bit in an integer.*

Syntax

```
result = IBCHNG (i,pos)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer). This argument contains the bit to be reversed.
pos	(Input) Must be of type integer. This argument is the position of the bit to be changed. The rightmost (least significant) bit of *i* is in position 0.

Results

The result type and kind are the same as *i*. The result is equal to *i* with the bit in position *pos* reversed.

For more information, see Bit Functions.

Example

```
INTEGER J, K
J = IBCHNG(10, 2)     ! returns 14 = 1110
K = IBCHNG(10, 1)     ! returns 8 = 1000
```

See Also

BTEST
IAND
IBCLR
IBSET
IEOR
IOR
ISHA
ISHC
ISHL
ISHFT
NOT

IBCLR

Elemental Intrinsic Function (Generic): *Clears one bit to zero.*

Syntax

```
result = IBCLR (i,pos)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
pos	(Input) Must be of type integer. It must not be negative and it must be less than BIT_SIZE(*i*).

The rightmost (least significant) bit of *i* is in position 0.

Results

The result type and kind are the same as *i*. The result has the value of the sequence of bits of *i*, except that bit *pos* of *i* is set to zero.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BBCLR	INTEGER(1)	INTEGER(1)
IIBCLR[1]	INTEGER(2)	INTEGER(2)
JIBCLR	INTEGER(4)	INTEGER(4)
KIBCLR	INTEGER(8)	INTEGER(8)

[1]Or HBCLR.

Example

IBCLR (18, 1) has the value 16.

If V has the value (1, 2, 3, 4), the value of IBCLR (POS = V, I = 15) is (13, 11, 7, 15).

The following shows another example:

```
INTEGER J, K
J = IBCLR(7, 1) ! returns 5 = 0101
K = IBCLR(5, 1) ! returns 5 = 0101
```

See Also

BTEST
IAND
IBCHNG
IBSET
IEOR
IOR
ISHA
ISHC
ISHL
ISHFT
NOT

IBITS

Elemental Intrinsic Function (Generic): *Extracts a sequence of bits (a bit field).*

Syntax

```
result = IBITS (i,pos,len)
```

i (Input) Must be of type integer.

pos	(Input) Must be of type integer. It must not be negative and *pos*+ *len* must be less than or equal to BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.
len	(Input) Must be of type integer. It must not be negative.

Results

The result type and kind are the same as *i*. The result has the value of the sequence of *len* bits in *i*, beginning at *pos*, right-adjusted and with all other bits zero.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BBITS	INTEGER(1)	INTEGER(1)
IIBITS[1]	INTEGER(2)	INTEGER(2)
JIBITS	INTEGER(4)	INTEGER(4)
KIBITS	INTEGER(8)	INTEGER(8)

[1] Or HBITS

Example

IBITS (12, 1, 4) has the value 6.

IBITS (10, 1, 7) has the value 5.

See Also

BTEST
BIT_SIZE
IBCLR
IBSET
ISHFT
ISHFTC
MVBITS

IBSET

Elemental Intrinsic Function (Generic): *Sets one bit to 1.*

Syntax

```
result = IBSET (i,pos)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
pos	(Input) Must be of type integer. It must not be negative and it must be less than BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.

Results

The result type and kind are the same as *i*. The result has the value of the sequence of bits of *i*, except that bit *pos* of *i* is set to 1.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BBSET	INTEGER(1)	INTEGER(1)
IIBSET[1]	INTEGER(2)	INTEGER(2)
JIBSET	INTEGER(4)	INTEGER(4)
KIBSET	INTEGER(8)	INTEGER(8)

[1]Or HBSET.

Example

IBSET (8, 1) has the value 10.

If V has the value (1, 2, 3, 4), the value of IBSET (POS = V, I = 2) is (2, 6, 10, 18).

The following shows another example:

```
INTEGER I
I = IBSET(8, 2) ! returns 12 = 1100
```

See Also

BTEST
IAND
IBCHNG
IBCLR
IEOR
IOR
ISHA
ISHC
ISHL
ISHFT
NOT

ICHAR

Elemental Intrinsic Function (Generic): *Returns the position of a character in the processor's character set.*

Syntax

```
result = ICHAR (c [, kind])
```

c	(Input) Must be of type character of length 1.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer.

The result value is the position of *c* in the processor's character set. *c* is in the range zero to n - 1, where *n* is the number of characters in the character set.

For any characters C and D (capable of representation in the processor), C .LE. D is true only if ICHAR(C) .LE. ICHAR(D) is true, and C .EQ. D is true only if ICHAR(C) .EQ. ICHAR(D) is true.

Specific Name	Argument Type	Result Type
	CHARACTER	INTEGER(2)
ICHAR [1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

[1]This specific function cannot be passed as an actual argument.

Example

ICHAR ('W') has the value 87.

ICHAR ('#') has the value 35.

See Also

IACHAR
CHAR
ASCII and Key Code Charts

IDATE Intrinsic Procedure

Intrinsic Subroutine (Generic): *Returns three integer values representing the current month, day, and year. IDATE can be used as an intrinsic subroutine or as a portability routine. It is an intrinsic procedure unless you specify USE IFPORT. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL IDATE (i, j, k)
```

i	(Output) Must be of type integer. It is the current month.
j	(Output) Must be of type integer with the same kind type parameter as *i*. It is the current day.
k	(Output) Must be of type integer with the same kind type parameter as *i*. It is the current year.

The current month is returned in *i*; the current day in *j*. The last two digits of the current year are returned in *k*.

Caution
The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

If the current date is September 16, 1999, the values of the integer variables upon return are: I = 9, J = 16, and K = 99.

See Also

DATE intrinsic procedure

DATE_AND_TIME

GETDAT

IDATE portability routine

IDATE Portability Routine

Portability Subroutine: *Returns the month, day, and year of the current system. IDATE can be used as a portability subroutine or as an intrinsic procedure. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

```
CALL IDATE (i, j, k)
```

-or-

```
CALL IDATE (iarray)
```

i	(Output) INTEGER(4). Is the current system month.
j	(Output) INTEGER(4). Is the current system day.
k	(Output) INTEGER(4). Is the current system year as an offset from 1900.
iarray	(Output) INTEGER(4). Is a three-element array that holds day as element 1, month as element 2, and year as element 3. The month is between 1 and 12. The year is greater than or equal to 1969 and is returned as 2 digits.

Caution
The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

Consider the following:

```
    use IFPORT
    integer(4) imonth, iday, iyear, datarray(3)
! If the date is July 11, 1999:
    CALL IDATE(IMONTH, IDAY, IYEAR)
```

```
! sets IMONTH to 7, IDAY to 11 and IYEAR to 99.
    CALL IDATE (DATARRAY)
! datarray is (/11,7,99/)
```

See Also

DATE portability routine

DATE_AND_TIME

GETDAT

IDATE intrinsic procedure

IDATE4

Portability Subroutine: *Returns the month, day, and year of the current system.*

Module

USE IFPORT

Syntax

CALL IDATE4 (*i*,*j*,*k*)

-or-

CALL IDATE4 (*iarray*)

i	(Output) INTEGER(4). The current system month.
j	(Output) INTEGER(4). The current system day.
k	(Output) INTEGER(4). The current system year as an offset from 1900.
iarray	(Output) INTEGER(4). A three-element array that holds day as element 1, month as element 2, and year as element 3. The month is between 1 and 12. The year is returned as an offset from 1900, if the year is less than 2000. For years greater than or equal to 2000, this element simply returns the integer year, such as 2003.

IDENT

General Compiler Directive: *Specifies a string that identifies an object module. The compiler places the string in the identification field of an object module when it generates the module for each source program unit.*

Syntax

!DIR$ IDENT *string*

string	Is a character constant containing printable characters. The number of characters is limited by the length of the source line.

Only the first IDENT directive is effective; the compiler ignores any additional IDENT directives in a program unit or module.

See Also

General Compiler Directives

Syntax Rules for Compiler Directives

IDFLOAT

Portability Function: *Converts an INTEGER(4) argument to double-precision real type.*

Module

USE IFPORT

Syntax

```
result = IDFLOAT (i)
```

i	(Input) Must be of type INTEGER(4).

Results

The result type is double-precision real (REAL(8) or REAL*8).

See Also

DFLOAT

IEEE_CLASS

Elemental Module Intrinsic Function (Generic): *Returns the IEEE class.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_CLASS (x)
```

x	(Input) Must be of type REAL.

Results

The result is of type TYPE(IEEE_CLASS_TYPE). The result value is one of the following:

IEEE_SIGNALING_NAN	IEEE_NEGATIVE_NORMAL
IEEE_QUIET_NAN	IEEE_POSITIVE_DENORMAL
IEEE_POSITIVE_INF	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_INF	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_ZERO

IEEE_CLASS does not return IEEE_OTHER_VALUE in Intel® Fortran.

Example

IEEE_CLASS(1.0) has the value IEEE_POSITIVE_NORMAL.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_COPY_SIGN

Elemental Module Intrinsic Function (Generic): *Returns an argument with a copied sign. This is equivalent to the IEEE copysign function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_COPY_SIGN (x,y)
```

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type and kind are the same as *x*. The result has the value *x* with the sign of *y*. This is true even for IEEE special values, such as NaN or infinity.

The flags information is returned as a set of 1-bit flags.

Example

The value of IEEE_COPY_SIGN (X,3.0) is ABS (X), even when X is NaN.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_FLAGS

Portability Function: *Gets, sets or clears IEEE flags for rounding direction and precision as well as queries or controls exception status. This function provides easy access to the modes and status required to use the features of ISO/IEC/IEEE 60559:2011 arithmetic in a Fortran program.*

Module

USE IFPORT

Syntax

```
result = IEEE_FLAGS (action,mode,in,out)
```

action	(Input) Character*(*). One of the following literal values: 'GET', 'SET', 'CLEAR', or 'CLEARALL'.

mode (Input) Character*(*). One of the following literal values: 'direction', 'precision', or 'exception'.

in (Input) Character*(*). One of the following literal values: 'inexact', 'division', 'underflow','overflow', 'invalid', 'all', 'common', 'nearest', 'tozero', 'negative', 'positive','extended', 'double', 'single', or ' ', which represents an unused (null) value.

out (Output) Must be at least CHARACTER*9. One of the literal values listed for *in*.

The descriptions for the values allowed for *in* and *out* can be summarized as follows:

Value	Description
'nearest'	Rounding direction flags
'tozero'	
'negative'	
'positive'	
'single'	Rounding precision flags
'double'	
'extended'	
'inexact'	Math exception flags
'underflow'	
'overflow'	
'division'	
'invalid'	
'all'	All math exception flags above
'common'	The math exception flags: 'invalid', 'division', 'overflow', and 'underflow'

The values for *in* and *out* depend on the *action* and *mode* they are used with. The interaction of the parameters can be summarized as follows:

Value of *action*	Value of *mode*	Value of *in*	Value of *out*	Functionality and return value
GET	'direction'	Null (' ')	One of 'nearest', 'tozero', 'negative', or 'positive'	Tests rounding direction settings.

Value of *action*	Value of *mode*	Value of *in*	Value of *out*	Functionality and return value
				Returns the current setting, or 'not available'.
	'exception'	Null (' ')	One of 'inexact', 'division', 'underflow', 'overflow', 'invalid', 'all', or 'common'	Tests math exception settings. Returns the current setting, or 0.
	'precision'	Null (' ')	One of 'single ', 'double ', or 'extended'	Tests rounding precision settings. Returns the current setting, or 'not available'.
SET	'direction'	One of 'nearest', 'tozero', 'negative', or 'positive'	Null (' ')	Sets a rounding direction.
	'exception'	One of 'inexact', 'division', 'underflow', 'overflow', 'invalid', 'all', or 'common'	Null (' ')	Sets a floating-point math exception.
	'precision'	One of 'single ', 'double ', or 'extended'	Null (' ')	Sets a rounding precision.

Value of *action*	Value of *mode*	Value of *in*	Value of *out*	Functionality and return value
CLEAR	'direction'	Null (' ')	Null (' ')	Clears the *mode*. Sets rounding to 'nearest'. Returns 0 if successful.
	'exception'	One of 'inexact', 'division', 'underflow', 'overflow', 'invalid', 'all', or 'common'	Null (' ')	Clears the *mode*. Returns 0 if successful.
	'precision'	Null (' ')	Null (' ')	Clears the *mode*. Sets precision to 'double' (W*S) or 'extended' (L*X, M*X). Returns 0 if successful.
CLEARALL	Null (' ')	Null (' ')	Null (' ')	Clears all flags. Sets rounding to 'nearest', sets precision to 'double' (W*S) or 'extended' (L*X, M*X), and sets all exception flags to 0. Returns 0 if successful.

Results

IEEE_FLAGS is an elemental, integer-valued function that sets IEEE flags for GET, SET, CLEAR, or CLEARALL procedures. It lets you control rounding direction and rounding precision, query exception status, and control exception enabling or disabling by using the SET or CLEAR procedures, respectively.

The flags information is returned as a set of 1-bit flags.

Example

The following example gets the highest priority exception that has a flag raised. It passes the input argument *in* as a null string:

```
USE IFPORT
INTEGER*4 iflag
CHARACTER*9 out
iflag = ieee_flags('get', 'exception', '', out)
PRINT *, out, ' flag raised'
```

The following example sets the rounding direction to round toward zero, unless the hardware does not support directed rounding modes:

```
USE IFPORT
INTEGER*4 iflag
CHARACTER*10 mode, out, in
iflag = ieee_flags('set', 'direction', 'tozero', out)
```

The following example sets the rounding direction to the default ('nearest'):

```
USE IFPORT
INTEGER*4 iflag
CHARACTER*10 out, in
iflag = ieee_flags('clear','direction', '', '' )
```

The following example clears all exceptions:

```
USE IFPORT
INTEGER*4 iflag
CHARACTER*10 out
iflag = ieee_flags('clear','exception', 'all', '' )
```

The following example restores default direction and precision settings, and sets all exception flags to 0:

```
USE IFPORT
INTEGER*4 iflag
CHARACTER*10 mode, out, in
iflag = ieee_flags('clearall', '', '', '')
```

The following example detects an underflow exception:

```
USE IFPORT
CHARACTER*20 out, in
excep_detect = ieee_flags('get', 'exception', 'underflow', out)
if (out .eq.'underflow') stop 'underflow'
```

IEEE_FMA

Elemental Module Intrinsic Function (Generic):
Returns the result of a fused multiply-add operation. This is equivalent to the IEEE fusedMultiplyAdd operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_FMA (a,b,c)`

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.
c	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is REAL with the same kind type parameter as *a*. When the result is in range, the value of the result is the mathematical value of (*a* * *b*) + *c* rounded according to the rounding mode of the representation method of *a*. Only the final step in the calculation may cause IEEE_INEXACT, IEEE_OVERFLOW, or IEEE_UNDERFLOW to signal; intermediate calculations do not.

This is the fusedMultiplyAdd operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_FMA (TINY(0.0), TINY(0.0) 1.0) is equal to 1.0 when the rounding mode is set to NEAREST. The exception IEEE_INEXACT is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_GET_FLAG

Elemental Module Intrinsic Subroutine (Generic): *Returns whether an exception flag is signaling.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

`CALL IEEE_GET_FLAG (flag, flag_value)`

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
flag_value	(Output) Must be of type logical. If the exception in *flag* is signaling, the result is true; otherwise, false.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL ON
...
CALL IEEE_GET_FLAG(IEEE_INVALID, ON)
```

If flag IEEE_INVALID is signaling, the value of ON is true; if it is quiet, the value of ON is false.

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_GET_HALTING_MODE

Elemental Module Intrinsic Subroutine (Generic): *Stores the halting mode for an exception.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_GET_HALTING_MODE (*flag*, *halting*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
halting	(Output) Must be of type logical. If the exception in *flag* causes halting, the result is true; otherwise, false.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL HALT
...
CALL IEEE_GET_HALTING_MODE(IEEE_INVALID, HALT)    ! Stores the halting mode
CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, .FALSE.) ! Stops halting
...
CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, HALT) ! Restores halting
```

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_GET_MODES

Intrinsic Module Subroutine (Generic): *Stores the current IEEE floating-point modes. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_GET_MODES (*modes*)

modes	(Output) Must be scalar and of type TYPE (IEEE_MODES_TYPE). It is assigned the value of the floating-point modes.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS ! Can also use IEEE_ARITHMETIC
TYPE (IEEE_MODE_TYPE) MODES
...
CALL IEEE_GET_MODES (MODES) ! Stores the floating-point modes
CALL IEEE_SET_UNDERFLOW_MODE (.TRUE.) ! Sets the underflow mode to gradual
...
CALL IEEE_SET_MODES (MODES) ! Restores the floating-point modes
```

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_GET_ROUNDING_MODE

Intrinsic Module Subroutine (Generic): *Stores the current IEEE rounding mode. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

CALL IEEE_GET_ROUNDING_MODE (*round_value* [, *radix*])

round_value	(Output) Must be scalar and of type TYPE (IEEE_ROUND_TYPE). It returns one of the following IEEE floating-point rounding values: IEEE_DOWN, IEEE_NEAREST, IEEE_TO_ZERO, or IEEE_UP; otherwise, IEEE_OTHER. The result can only be used if IEEE_SET_ROUNDING_MODE is invoked.
radix	(Input; optional) Must be an integer scalar with a value of two or ten. The rounding mode queried is the binary rounding mode, unless *radix* is present with the value ten, in which case it is the decimal rounding mode queried.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND
...
CALL IEEE_GET_ROUNDING_MODE(ROUND)   ! Stores the rounding mode
CALL IEEE_SET_ROUNDING_MODE(IEEE_UP) ! Resets the rounding mode
...
CALL IEEE_SET_ROUNDING_MODE(VALUE)   ! Restores the previous rounding mode
```

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_GET_STATUS

Intrinsic Module Subroutine (Generic): *Stores the current state of the floating-point environment. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_GET_STATUS (*status_value*)

status_value	(Input) Must be scalar and of type TYPE (IEEE_STATUS_TYPE). It stores the floating-point status. The result can only be used if IEEE_SET_STATUS is invoked.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS    ! Can also use IEEE_ARITHMETIC
TYPE(IEEE_ STATUS_TYPE) STATUS
...
CALL IEEE_GET_STATUS(STATUS)   ! Stores the floating-point status
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Sets all flags to be quiet
...
CALL IEEE_SET_STATUS(STATUS)   ! Restores the floating-point status
```

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_GET_UNDERFLOW_MODE

Intrinsic Module Subroutine (Generic): *Stores the current underflow mode. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

CALL IEEE_GET_UNDERFLOW_MODE (*gradual*)

gradual	(Output) Must be logical scalar. The result is true if the current underflow mode is gradual (IEEE subnormals are allowed) and false if the current underflow mode is abrupt (underflowed results are set to zero).

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL GRAD
...
```

```
CALL IEEE_GET_UNDERFLOW_MODE(GRAD)
IF (GRAD) THEN ! underflows are gradual
...
ELSE ! underflows are abrupt
...
END IF
```

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_HANDLER

Portability Function: *Establishes a handler for IEEE exceptions.*

Module

USE IFPORT

Syntax

`result = IEEE_HANDLER (action, exception, handler)`

action	(Input) Character*(*). One of the following literal IEEE actions: 'GET', 'SET', or 'CLEAR'. For more details on these actions, see IEEE_FLAGS.
exception	(Input) Character*(*). One of the following literal IEEE exception flags: 'inexact', 'underflow', 'overflow', 'division', 'invalid', 'all' (which equals the previous five flags), or 'common' (which equals 'invalid', 'overflow', 'underflow', and 'division'). The flags 'all' or 'common' should only be used for actions SET or CLEAR.
handler	(Input) The address of an external signal-handling routine.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The result is 0 if successful; otherwise, 1.

IEEE_HANDLER calls a signal-handling routine to establish a handler for IEEE exceptions. It also enables an FPU trap corresponding to the required exception.

The state of the FPU is not defined in the handler routine. When the FPU trap occurs, the program invokes the handler routine. After the handler routine is executed, the program terminates.

The handler routine gets the exception code in the SIGINFO argument. SIGNO is the number of the system signal. The meaning of the SIGINFO constants appear in the following table (defined in the IFPORT module):

FPE$INVALID	Invalid operation
FPE$ZERODIVIDE	Divide-by-zero
FPE$OVERFLOW	Numeric overflow
FPE$UNDERFLOW	Numeric underflow
FPE$INEXACT	Inexact result (precision)

'GET' actions return the location of the current handler routine for exception cast to an INTEGER.

Example

The following example creates a handler routine and sets it to trap divide-by-zero:

```
PROGRAM TEST_IEEE
 REAL :: X, Y, Z
 CALL FPE_SETUP
 X = 0.
 Y = 1.
 Z = Y / X
END PROGRAM
SUBROUTINE FPE_SETUP
USE IFPORT
IMPLICIT NONE
 INTERFACE
   SUBROUTINE FPE_HANDLER(SIGNO, SIGINFO)
     INTEGER(4), INTENT(IN) :: SIGNO, SIGINFO
   END SUBROUTINE
 END INTERFACE
 INTEGER IR
 IR = IEEE_HANDLER('set','division',FPE_HANDLER)
END SUBROUTINE FPE_SETUP
SUBROUTINE FPE_HANDLER(SIG, CODE)
 USE IFPORT
 IMPLICIT NONE
 INTEGER SIG, CODE
 IF(CODE.EQ.FPE$ZERODIVIDE) PRINT *,'Occurred divide by zero.'
 CALL ABORT
END SUBROUTINE FPE_HANDLER
```

See Also

IEEE_FLAGS

IEEE_INT

Elemental Module Intrinsic Function (Generic):
Enables conversion to INTEGER type.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_INT (a, round [, kind])`

a	(Input) Must be of type REAL or INTEGER.
round	(Input) Must be of type IEEE_ROUND_TYPE.
kind	(Input; optional) Must be a scalar INTEGER constant.

Results

The result type is INTEGER. If *kind* is present, the kind type parameter is that specified by *kind*; otherwise, the kind type parameter is default integer.

The result is the value of *a* converted to integer according to the rounding mode specified by *round* if the values can be represented in the representation method of the result type kind; otherwise, the result is processor dependent and IEEE_INVALID is signaled.

The result must be consistent with the ISO/IEC/IEEE 60559:2011 operation convertToInteger{round} or convertToIntegerExact{round}. The processor consistently chooses which operation is performed.

Example

The result value of IEEE_INT (63.5, IEEE_DOWN) is 63. If converToIntegerExact{round} is used, IEEE_INEXACT will signal.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_IS_FINITE

Elemental Module Intrinsic Function (Generic):
Returns whether an IEEE value is finite.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_FINITE (x)
```

x	(Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of *x* is finite; otherwise, false.

An IEEE value is finite if IEEE_CLASS(x) has one of the following values:

IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_NORMAL	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_DENORMAL	IEEE_NEGATIVE_ZERO

Example

IEEE_IS_FINITE (-2.0) has the value true.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_IS_NAN

Elemental Module Intrinsic Function (Generic):
Returns whether an IEEE value is Not-a-Number (NaN).

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_NAN (x)
```

x (Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of *x* is NaN; otherwise, false.

Example

IEEE_IS_NAN (SQRT(-2.0)) has the value true if IEEE_SUPPORT_SQRT (2.0) has the value true.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_IS_NEGATIVE

Elemental Module Intrinsic Function (Generic):
Returns whether an IEEE value is negative.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_NEGATIVE (x)
```

x (Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of x is negative; otherwise, false.

An IEEE value is negative if IEEE_CLASS(x) has one of the following values::

IEEE_NEGATIVE_NORMAL	IEEE_NEGATIVE_ZERO
IEEE_NEGATIVE_DENORMAL	IEEE_NEGATIVE_INF

Example

IEEE_IS_NEGATIVE (2.0) has the value false.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_IS_NORMAL

Elemental Module Intrinsic Function (Generic):
Returns whether an IEEE value is normal.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_NORMAL (x)
```

x (Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of *x* is normal; otherwise, false.

An IEEE value is normal if IEEE_CLASS(x) has one of the following values:

IEEE_POSITIVE_NORMAL	IEEE_POSITIVE_ZERO
IEEE_NEGATIVE_NORMAL	IEEE_NEGATIVE_ZERO

Example

IEEE_IS_NORMAL (SQRT(-2.0)) has the value false if IEEE_SUPPORT_SQRT (-2.0) has the value true.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_LOGB

Elemental Module Intrinsic Function (Generic):
Returns a floating-point value equal to the unbiased exponent of the argument. This is equivalent to the IEEE logb function.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_LOGB (x)
```

x (Input) Must be of type REAL.

Results

The result type and kind are the same as *x*. The result has the value of the unbiased exponent of *x* if the value of *x* is not zero, infinity, or NaN. The value of the result is equal to EXPONENT(x) - 1.

If *x* is equal to 0, the result is -infinity if IEEE_SUPPORT_INF(x) is true; otherwise, -HUGE(x). In either case, the IEEE_DIVIDE_BY_ZERO exception is signaled.

Example

IEEE_LOGB (3.4) has the value 1.0; IEEE_LOGB (4.0) has the value 2.0.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_MAX_NUM

Inquiry Module Intrinsic Function (Generic):
Returns the maximum of two values. This is equivalent to the IEEE maxNum operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_MAX_NUM (x,y)
```

x (Input) Must be of type REAL.

y (Input) Must be of type REAL with the same kind type parameter as *x*.

Results

The result type is REAL with the same kind type parameter as *x*. The result is *x* if *y* < *x*. The result is *y* if *x* < *y*.

If one of the arguments is a quiet NaN, the result is the value of the argument which is not a quiet NaN. If either or both of the arguments is a signaling NaN, the result is a NaN and IEEE_INVALID signals. Otherwise, the result value is that of either *x* or *y* (processor dependent). No exceptions are signaled unless *x* or *y* is a signaling NaN.

This is the maxNum operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_MAX_NUM (3.7, IEEE_VALUE (0.0, IEEE_SIGNALING_NAN)) is a NaN. The exception IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module

IEEE Intrinsic Modules Quick Reference Tables

IEEE_MAX_NUM_MAG

Inquiry Module Intrinsic Function (Generic):
Returns the maximum magnitude of two values. This is equivalent to the IEEE maxNumMag operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_MAX_NUM_MAG (x,y)
```

x (Input) Must be of type REAL.

y (Input) Must be of type REAL with the same kind type parameter as *x*.

Results

The result type is REAL with the same kind type parameter as *x*. The result is *x* if ABS(*y*) < ABS(*x*). The result is the value of *y* if ABS(*x*) < ABS(*y*).

If one of the arguments is a quiet NaN, the result is the value of the argument which is not a quiet Nan. If either or both of the arguments is a signaling NaN, the result is a NaN and IEEE_INVALID signals. Otherwise, the result value is that of either *x* or *y* (processor dependent). No exceptions are signaled unless *x* or *y* is a signaling NaN.

This is the maxNumMag operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_MAX_NUM_MAG (3.7, -7.5) is -7.5.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_MIN_NUM

Inquiry Module Intrinsic Function (Generic):
Returns the minimum of two values. This is equivalent to the IEEE minNum operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_MIN_NUM (x,y)
```

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL with the same kind type parameter as *x*.

Results

The result type is REAL with the same kind type parameter as *x*. The result is *x* if *x* < *y*. The result is *y* if *y* < *x*.

If one of the arguments is a quiet NaN, the result is the value of the argument which is not a quiet NaN. If either or both of the arguments is a signaling NaN, the result is a NaN and IEEE_INVALID signals. Otherwise, the result value is that of either *x* or *y* (processor dependent). No exceptions are signaled unless *x* or *y* is a signaling NaN.

This is the minNum operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_MIN_NUM (3.7, IEEE_VALUE (0.0, IEEE_QUIET _NAN)) is 3.7. No exceptions are signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_MIN_NUM_MAG

Inquiry Module Intrinsic Function (Generic):
Returns the minimum magnitude of two values. This is equivalent to the IEEE minNumMag operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_MIN_NUM_MAG (x,y)
```

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL with the same kind type parameter as *x*.

Results

The result type is REAL with the same kind type parameter as *x*. The result is y if ABS(*y*) < ABS(*x*). The result is the value of *x* if ABS(*x*) < ABS(*y*).

If one of the arguments is a quiet NaN, the result is the value of the argument which is not a quiet NaN. If either or both of the arguments is a signaling NaN, the result is a NaN and IEEE_INVALID signals. Otherwise, the result value is that of either *x* or *y* (processor dependent). No exceptions are signaled unless *x* or *y* is a signaling NaN.

This is the minNumMag operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_MIN_NUM_MAG (3.7, -7.5) is 3.7.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_NEXT_AFTER

Elemental Module Intrinsic Function (Generic):
Returns the next representable value after X toward Y.
This is equivalent to the IEEE nextafter function.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_NEXT_AFTER (x,y)
```

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type and kind are the same as x. If x is equal to y, the result is x; no exception is signaled. If x is not equal to y, the result has the value of the next representable neighbor of x toward y. The neighbors of zero (of either sign) are both nonzero.

The following exceptions are signaled under certain cases:

Exception	Signaled
IEEE_OVERFLOW	When X is finite but IEEE_NEXT_AFTER(X,Y) is infinite
IEEE_UNDERFLOW	When IEEE_NEXT_AFTER(X,Y) is subnormal
IEEE_INEXACT	In both the above cases

Example

The value of IEEE_NEXT_AFTER (2.0,3.0) is 2.0 + EPSILON (X).

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_NEXT_DOWN

Elemental Module Intrinsic Function (Generic):
Returns the next lower adjacent machine number.
This is equivalent to the IEEE nextDown operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_NEXT_DOWN (x)
```

x	(Input) Must be of type REAL.

Results

The result type and kind are the same as *x*. The value of the result is the greatest value that compares less than *x* - except when *x* has the value NaN, the result is NaN, and when *x* has the value - ∞ , the result is - ∞ . If *x* is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

This is the nextDown operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The value of IEEE_NEXT_DOWN(+0.0) is the negative subnormal number with the least magnitude if the value if IEEE_SUPPORT_SUBNORMAL (0.0) is .TRUE..

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_NEXT_UP

Elemental Module Intrinsic Function (Generic):
Returns the next higher adjacent machine number.
This is equivalent to the IEEE nextUp operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_NEXT_UP (x)
```

x	(Input) Must be of type REAL.

Results

The result type and kind are the same as *x*. The value of the result is the least value that compares greater than *x* - except when *x* has the value NaN, the result is NaN, and when *x* has the value + ∞, the result is + ∞. If *x* is a signaling NaN, IEEE_INVALID signals; otherwise, no exception is signaled.

This is the nextUp operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The value of IEEE_NEXT_UP (HUGE (x)) is + ∞ if the value of IEEE_SUPPORT_INF (x) is .true..

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_QUIET_EQ

Elemental Module Intrinsic Function (Generic):
Performs a non-signaling comparison for equality. This is equivalent to the IEEE compareQuietEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_QUIET_EQ (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares equal to *b*. If *a* or *b* is a NaN, the result is false. IEEE_INVALID signals if either *a* or *b* is a signaling NaN, otherwise no exception is signaled.

This is the compareQuietEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_QUIET_EQ (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and no exception is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_QUIET_GE

Elemental Module Intrinsic Function (Generic):
Performs a non-signaling comparison for greater than or equal. This is equivalent to the IEEE compareQuietGreaterEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_QUIET_GE (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares greater than or equal to *b*. If *a* or *b* is a NaN, the result is false. IEEE_INVALID signals if either *a* or *b* is a signaling NaN, otherwise no exception is signaled.

This is the compareQuietGreaterEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_QUIET_GE (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and no exception is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_QUIET_GT

Elemental Module Intrinsic Function (Generic):
Performs a non-signaling comparison for greater than. This is equivalent to the IEEE compareQuietGreater operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_QUIET_GT (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares greater than *b*. If *a* or *b* is a NaN, the result is false. IEEE_INVALID signals if either *a* or *b* is a signaling NaN, otherwise no exception is signaled.

This is the compareQuietGreater operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_QUIET_GT (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and no exception is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_QUIET_LE

Elemental Module Intrinsic Function (Generic):
Performs a non-signaling comparison for less than or equal. This is equivalent to the IEEE compareQuietLessEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_QUIET_LE (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares less than or equal to *b*. If *a* or *b* is a NaN, the result is false. IEEE_INVALID signals if either *a* or *b* is a signaling NaN, otherwise no exception is signaled.

This is the compareQuietLessEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_QUIET_LE (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and no exception is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_QUIET_LT

Elemental Module Intrinsic Function (Generic):
Performs a non-signaling comparison for less than. This is equivalent to the IEEE compareQuietLess operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_QUIET_LT (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares less than *b*. If *a* or *b* is a NaN, the result is false. IEEE_INVALID signals if either *a* or *b* is a signaling NaN, otherwise no exception is signaled.

This is the compareQuietLess operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_QUIET_LT (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and no exception is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_QUIET_NE

Elemental Module Intrinsic Function (Generic):
Performs a non-signaling comparison for inequality. This is equivalent to the IEEE compareQuietNotEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_QUIET_NE (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares not equal to *b*. If *a* or *b* is a NaN, the result is true. IEEE_INVALID signals if either *a* or *b* is a signaling NaN, otherwise no exception is signaled.

This is the compareQuietNotEqual operation specified as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_QUIET_NE (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is true and no exception is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_REAL

Elemental Module Intrinsic Function (Generic):
Enables conversion to REAL type.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_REAL (a [, kind])
```

a	(Input) Must be of type REAL or INTEGER.
kind	(Input; optional) Must be a scalar INTEGER constant.

Results

The result type is REAL. If *kind* is present, the kind type parameter is that specified by *kind*; otherwise, the kind type parameter is default real.

The result has the same value as *a* if that value is representable in the representation method of the result type kind; otherwise, it is rounded according to the current rounding mode.

The result must be consistent with the ISO/IEC/IEEE 60559:2011 operation convertFromInt if *a* is an integer, and with operation convertFormat if *a* is real.

Example

The result value of IEEE_REAL (987) is 987.0.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_REM

Elemental Module Intrinsic Function (Generic):
Returns the result of an exact remainder operation. This is equivalent to the IEEE remainder function.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_REM (x,y)
```

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL with a radix that is the same as that of *x*.

Results

The result type is real with the kind type parameter of whichever argument has greater precision.

Regardless of the rounding mode, the result value is x - y*N, where N is the integer nearest to the value x / y. If |N – x / y | = 1/2, N is even. If the result value is zero, the sign is the same as x.

Example

The value of IEEE_REM (5.0,4.0) is 1.0; the value of IEEE_REM (2.0,1.0) is 0.0; the value of IEEE_REM (3.0,2.0) is -1.0.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_RINT

Elemental Module Intrinsic Function (Generic):
Returns an integer value rounded according to the current rounding mode.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_RINT (x [, round])
```

x	(Input) Must be of type REAL.
round	(Input; optional) Must be of type TYPE (IEEE_ROUND_TYPE).

Results

The result type and kind are the same as *x*.

The value of the result is *x* rounded to an integer according to the current rounding mode if *round* is not present. If *round* is present, the value of the result is *x* rounded to an integer according to the mode specified by *round*; this is the operation roundToInteger (rounding) as specified by ISO/IEC/IEEE 60559:2011.

If the result value is zero, the sign is the same as *x*.

The rounding mode specified by *round*, if present, is used to perform the conversion. The rounding mode before and after the call remains unchanged.

Example

If the current rounding mode is IEEE_UP, the value of IEEE_RINT (2.2) is 3.0.

If the current rounding mode is IEEE_NEAREST, the value of IEEE_RINT (2.2) is 2.0.

The value of IEEE_RINT (7.4, IEEE_DOWN) is 7.0.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SCALB

Elemental Module Intrinsic Function (Generic):
Returns the exponent of a radix-independent floating-point number. This is equivalent to the IEEE scalb function.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SCALB (x,i)
```

x	(Input) Must be of type REAL.
i	(Input) Must be of type INTEGER.

Results

The result type and kind are the same as *x*. The result is *x* multiplied by 2**i, if the value can be represented as a normal number.

If x (2**i) is too small and there is a loss of accuracy, the exception IEEE_UNDERFLOW is signaled. The result value is the nearest number that can be represented with the same sign as *x*.

If *x* is finite and x (2**i) is too large, an IEEE_OVERFLOW exception occurs. If IEEE_SUPPORT_INF (x) is true, the result value is infinity with the same sign as *x*; otherwise, the result value is SIGN (HUGE(x), x).

If *x* is infinite, the result is the same as *x*; no exception is signaled.

Example

The value of IEEE_SCALB (2.0,3) is 16.0.

See Also

IEEE_ARITHMETIC Intrinsic Module

IEEE Intrinsic Modules Quick Reference Tables

IEEE_SELECTED_REAL_KIND

Transformational Module Intrinsic Function (Generic): *Returns the value of the kind parameter of an IEEE REAL data type.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SELECTED_REAL_KIND ([p][,r][,radix])`

`p`	(Input; optional) Must be scalar and of type INTEGER.
`r`	(Input; optional) Must be scalar and of type INTEGER.
`radix`	(Input; optional) Must be scalar and of type INTEGER.

At least argument `p` or `r` must be present.

Results

If `p` or `r` is absent, the result is as if the argument was present with the value zero. If `radix` is absent, there is no requirement on the radix of the selected kind.

The result is a scalar of type default integer. The result has a value equal to a value of the kind parameter of an IEEE real data type with decimal precision, as returned by the function PRECISION, of at least *p* digits, a decimal exponent range, as returned by the function RANGE, of at least *r*, and a *radix*, as returned by the function RADIX, of *radix*.

If no such kind type parameter is available on the processor, the result is as follows:

- -1 if the precision is not available
- -2 if the exponent range is not available
- -3 if neither the precision nor the exponent range is available
- -4 if one but not both of the precision and the exponent range is available
- -5 if the radix is not available

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision.

Example

IEEE_SELECTED_REAL_KIND (6, 70, 2) = 8.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables
SELECTED_REAL_KIND
Model for Real Data
for information on the real model.

IEEE_SET_FLAG

Elemental Module Intrinsic Function (Generic): *Assigns a value to an exception flag. This is a pure subroutine.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_FLAG (*flag*,*flag_value*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
flag_value	(Output) Must be of type logical. If it has the value true, the exception in *flag* is set to signal; otherwise, the exception is set to be quiet.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS   ! Can also use module IEEE_ARITHMETIC
...
CALL IEEE_SET_FLAG (IEEE_INVALID, .TRUE.) ! Sets the IEEE_INVALID flag to signal
```

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SET_HALTING_MODE

Elemental Module Intrinsic Function (Generic):
Controls halting or continuation after an exception.
This is a pure subroutine.

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_HALTING_MODE (*flag*, *halting*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW
halting	(Input) Must be scalar and of type logical. If the value is true, the exception specified in *flag* will cause halting; otherwise, execution will continue after this exception.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL HALT
...
CALL IEEE_GET_HALTING_MODE(IEEE_INVALID, HALT)  ! Stores the halting mode
```

```
CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, .FALSE.) ! Stops halting
...
CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, HALT)  ! Restores halting
```

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SET_MODES

Intrinsic Module Subroutine (Generic): *Restores the current IEEE floating-point modes. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_MODES (*modes*)

modes	(Input) Must be scalar and of type TYPE (IEEE_MODES_TYPE). Its value must be a value that was assigned to the modes argument of a previous call to IEEE_GET_MODES. A call to IEEE_SET_MODES restores the value of the floating-point modes to the state at the time IEEE_GET_MODES was called.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS ! Can also use IEEE_ARITHMETIC
TYPE (IEEE_MODE_TYPE) MODES
...
CALL IEEE_GET_MODES (MODES) ! Stores the floating-point modes
CALL IEEE_SET_UNDERFLOW_MODE (.TRUE.) ! Sets the underflow mode to gradual
...
CALL IEEE_SET_MODES (MODES) ! Restores the floating-point modes
```

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SET_ROUNDING_MODE

Intrinsic Module Subroutine (Generic): *Sets the IEEE rounding mode. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

CALL IEEE_SET_ROUNDING_MODE (*round_value* [, *radix*])

round_value	(Input) Must be scalar and of type TYPE (IEEE_ROUND_TYPE). It specifies one of the following IEEE floating-point rounding values: IEEE_DOWN, IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, or IEEE_OTHER.
radix	(Input; optional) Must be an integer scalar with a value of ten or two. The rounding mode set is the binary rounding mode unless *radix* is present with the value of ten, in which case it is the decimal rounding mode set.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE (IEEE_ROUND_TYPE) ROUND
...
CALL IEEE_GET_ROUNDING_MODE (ROUND)   ! Stores the rounding mode
CALL IEEE_SET_ROUNDING_MODE (IEEE_UP) ! Resets the rounding mode
...
CALL IEEE_SET_ROUNDING_MODE (VALUE)   ! Restores the previous rounding mode
```

IEEE_SET_STATUS

Intrinsic Module Subroutine (Generic): *Restores the state of the floating-point environment. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

`CALL IEEE_SET_STATUS (`*status_value*`)`

status_value	(Input) Must be scalar and of type TYPE (IEEE_STATUS_TYPE). Its value must be set in a previous invocation of IEEE_GET_STATUS.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS ! Can also use IEEE_ARITHMETIC
TYPE (IEEE_STATUS_TYPE) STATUS
...
CALL IEEE_GET_STATUS (STATUS) ! Stores the floating-point status
CALL IEEE_SET_FLAG (IEEE_ALL,.FALSE.) ! Sets all flags to be quiet
...
CALL IEEE_SET_STATUS (STATUS) ! Restores the floating-point status
```

IEEE_SET_UNDERFLOW_MODE

Intrinsic Module Subroutine (Generic): *Sets the current underflow mode. This is an impure subroutine.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
CALL IEEE_SET_UNDERFLOW_MODE (gradual)
```

gradual	(Input) Must be scalar and of type logical. If it is true, the current underflow mode is set to gradual underflow (subnormals may be produced on underflow). If it is false, the current underflow mode is set to abrupt (underflowed results are set to zero).

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL :: SG
...
CALL IEEE_GET_UNDERFLOW_MODE (SG) ! Stores underflow mode
CALL IEEE_SET_UNDERFLOW_MODE (.FALSE.) ! Resets underflow mode
...                                    ! Abrupt underflows happens here
CALL IEEE_SET_UNDERFLOW_MODE (SG) ! Restores previous underflow mode
```

IEEE_SIGNALING_EQ

Elemental Module Intrinsic Function (Generic):
Performs a signaling comparison for equality. This is equivalent to the IEEE compareSignalingEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SIGNALING_EQ (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares equal to *b*. If *a* or *b* is a NaN, the result is false and IEEE_INVALID signals, otherwise no exception is signaled.

This is the compareSignalingEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_SIGNALING_EQ (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SIGNALING_GE

Elemental Module Intrinsic Function (Generic):
Performs a signaling comparison for greater than or equal. This is equivalent to the IEEE compareSignalingGreaterEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SIGNALING_GE (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares greater than or equal to *b*. If *a* or *b* is a NaN, the result is false and IEEE_INVALID signals, otherwise no exception is signaled.

This is the compareSignalingGreaterEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_SIGNALING_GE (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SIGNALING_GT

Elemental Module Intrinsic Function (Generic):
Performs a signaling comparison for greater than. This is equivalent to the IEEE compareSignalingGreater operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SIGNALING_GT (a,b)
```

a	(Input) Must be of type REAL.
b	(Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares greater than *b*. If *a* or *b* is a NaN, the result is false and IEEE_INVALID signals, otherwise no exception is signaled.

This is the compareSignalingGreater operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_SIGNALING_GT (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SIGNALING_LE

Elemental Module Intrinsic Function (Generic):
Performs a signaling comparison for less than or equal. This is equivalent to the IEEE compareSignalingLessEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SIGNALING_LE (a,b)`

a (Input) Must be of type REAL.

b (Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares less than or equal to *b*. If *a* or *b* is a NaN, the result is false and IEEE_INVALID signals, otherwise no exception is signaled.

This is the compareSignalingLessEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_SIGNALING_LE (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SIGNALING_LT

Elemental Module Intrinsic Function (Generic):
Performs a signaling comparison for less than. This is equivalent to the IEEE compareSignalingLess operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SIGNALING_LT (a,b)`

a (Input) Must be of type REAL.

b (Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares less than *b*. If *a* or *b* is a NaN, the result is false and IEEE_INVALID signals, otherwise no exception is signaled.

This is the compareSignalingLess operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_SIGNALING_LT (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is false and IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SIGNALING_NE

Elemental Module Intrinsic Function (Generic):
Performs a signaling comparison for inequality. This is equivalent to the IEEE compareSignalingNotEqual operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SIGNALING_NE (a,b)`

a (Input) Must be of type REAL.

b (Input) Must be of type REAL with the same kind type parameter as *a*.

Results

The result type is default LOGICAL. It has the value true if *a* compares not equal to *b*. If *a* or *b* is a NaN, the result is true and IEEE_INVALID signals, otherwise no exception is signaled.

This is the compareSignalingNotEqual operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result value of IEEE_SIGNALING_NE (3.7, IEEE_VALUE (0.0, IEEE_QUIET_NAN)) is true and IEEE_INVALID is signaled.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SIGNBIT

Elemental Module Intrinsic Function (Generic):
Tests to determine if the argument's sign bit is set. This is equivalent to the IEEE SignMinus operation.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SIGNBIT (x)
```

x (Input) Must be of type REAL.

Results

The result is a scalar of type default logical. The result has the value .true. if the sign bit of *x* is set (nonzero); otherwise, it has the value .false.. No exception is signaled even if *x* has the value of a signaling NaN.

This is the SignMinus operation as specified in the ISO/IEC/IEEE 60559:2011 standard.

Example

The result of IEEE_SIGNBIT (-3.14) is .true..

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_DATATYPE

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE arithmetic.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_DATATYPE ([x])
```

x (Input; optional) Must be scalar and of type REAL.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports IEEE arithmetic for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports IEEE arithmetic for real variables of the same kind type parameter as *x*; otherwise, false.

If real values are implemented according to the IEEE standard except that underflowed values flush to zero (abrupt) instead of being subnormal.

Example

IEEE_SUPPORT_DATATYPE (3.0) has the value true.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_DENORMAL

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE subnormal numbers.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_DENORMAL ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports arithmetic operations and assignments with subnormal numbers for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports arithmetic operations and assignments with subnormal numbers for real variables of the same kind type parameter as *x*; otherwise, false.

IEEE_SUPPORT_DENORMAL() and IEEE_SUPPORT_DENORMAL(0.0_16) return .TRUE. even though Intel® Fortran's implementation does not signal when an underflow results in a REAL(16) denormal value. Intel® Fortran's implementation does signal when an underflow results in a REAL(16) zero.

Example

IEEE_SUPPORT_DENORMAL () has the value true if IEEE subnormal numbers are supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_DIVIDE

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE divide.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_DIVIDE ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports divide with the accuracy specified by the IEEE standard for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports divide with the accuracy specified by the IEEE standard for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_DIVIDE ( ) has the value true if IEEE divide is supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_FLAG

Transformational Module Intrinsic Function (Generic): *Returns whether the processor supports IEEE exceptions.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

```
result = IEEE_SUPPORT_FLAG (flag [, x])
```

flag	(Input) Must be a scalar of type TYPE (IEEE_FLAG_TYPE). Its value is one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports detection of the exception specified by *flag* for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports detection of the exception specified by *flag* for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_FLAG (IEEE_UNDERFLOW) has the value true if the IEEE_UNDERFLOW exception is supported for all real types.

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_HALTING

Transformational Module Intrinsic Function (Generic): *Returns whether the processor supports IEEE halting.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

```
result = IEEE_SUPPORT_HALTING(flag)
```

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.

Results

The result is a scalar of type default logical. The result has the value true if the processor supports the ability to control halting after the exception specified by *flag*; otherwise, false.

Example

IEEE_SUPPORT_HALTING (IEEE_UNDERFLOW) has the value true if halting is supported after an IEEE_UNDERFLOW exception

See Also

IEEE_EXCEPTIONS Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_INF

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE infinities.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_INF ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports IEEE infinities (positive and negative) for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports IEEE infinities for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_INF( ) has the value true if IEEE infinities are supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_IO

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE base conversion rounding during formatted I/O.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_IO ([x])
```

x (Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports base conversion rounding during formatted input and output for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports base conversion rounding during formatted input and output for real variables of the same kind type parameter as *x*; otherwise, false.

The base conversion rounding applies to modes IEEE_UP, IEEE_DOWN, IEEE_TO_ZERO, and IEEE_NEAREST.

Example

IEEE_SUPPORT_IO ( ) has the value true if base conversion rounding is supported for all real types during formatted I/O.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_NAN

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE Not-a-Number feature.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_NAN ([x])
```

x (Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports NaNs for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports NaNs for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_NAN () has the value true if IEEE NaNs are supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_ROUNDING

Transformational Module Intrinsic Function (Generic): *Returns whether the processor supports IEEE rounding mode.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SUPPORT_ROUNDING (round_value [, x])`

round_value	(Input) Must be of type TYPE(IEEE_ROUND_TYPE). It specifies one of the following rounding modes: IEEE_AWAY, IEEE_DOWN, IEEE_NEAREST, IEEE_TO_ZERO, and IEEE_UP.
x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports the rounding mode specified by round_value for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports the rounding mode specified by round_value for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_ROUNDING (IEEE_DOWN) has the value true if rounding mode IEEE_DOWN is supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_SQRT

Inquiry Module Intrinsic Function (Generic): *Returns whether the processor supports IEEE SQRT (square root).*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SUPPORT_SQRT ([x])`

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor implements SQRT in accord with the IEEE standard for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor implements SQRT in accord with the IEEE standard for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_SQRT ( ) has the value true if IEEE SQRT is supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_STANDARD

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE features defined in the standard.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_STANDARD ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. The result has the value true if the results of all the following functions are true (x can be omitted):

IEEE_SUPPORT_DATATYPE([x])
IEEE_SUPPORT_DENORMAL([x])
IEEE_SUPPORT_DIVIDE([x])
IEEE_SUPPORT_FLAG(flag [, x])[1]
IEEE_SUPPORT_HALTING(flag)[1]
IEEE_SUPPORT_INF([x])
IEEE_SUPPORT_NAN([x])
IEEE_SUPPORT_ROUNDING(round_value [, x])[2]
IEEE_SUPPORT_SQRT([x])
[1] "flag" must be a valid value
[2] "round_value" must be a valid value

Otherwise, the result has the value, false.

Example

IEEE_SUPPORT_STANDARD () has the value false if both IEEE and non-IEEE real kinds are supported.

See Also

IEEE_ARITHMETIC Intrinsic Module

IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_SUBNORMAL

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE subnormal numbers.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_SUBNORMAL ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports arithmetic operations and assignments with subnormal numbers for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports arithmetic operations and assignments with subnormal numbers for real variables of the same kind type parameter as *x*; otherwise, false.

IEEE_SUPPORT_SUBNORMAL () and IEEE_SUPPORT_SUBNORMAL (0.0_16) return .TRUE. even though Intel® Fortran's implementation does not signal when an underflow results in a REAL (16) subnormal value. Intel® Fortran's implementation does signal when an underflow results in a REAL (16) zero.

Example

IEEE_SUPPORT_SUBNORMAL () has the value true if IEEE subnormal numbers are supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_SUPPORT_UNDERFLOW_CONTROL

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports the ability to control the underflow mode.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_UNDERFLOW_CONTROL ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If *x* is omitted, the result has the value true if the processor supports controlling the underflow mode for all real values; otherwise, false.

If *x* is specified, the result has the value true if the processor supports controlling the underflow mode for real variables of the same kind type parameter as *x*; otherwise, false.

Example

IEEE_SUPPORT_UNDERFLOW _CONTROL () has the value true if controlling the underflow mode is supported for all real types.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_UNORDERED

Elemental Module Intrinsic Function (Generic):
Returns whether one or more of the arguments is Not-a-Number (NaN). This is equivalent to the IEEE unordered function.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_UNORDERED (x,y)`

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if *x* or *y* is a NaN, or both are NaNs; otherwise, false.

Example

IEEE_UNORDERED (0.0, SQRT(-2.0)) has the value true if IEEE_SUPPORT_SQRT (2.0) has the value true.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEEE_VALUE

Elemental Module Intrinsic Function (Generic):
Creates an IEEE value.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_VALUE (x, class)`

x	(Input) Must be of type REAL.
class	(Input) Must be of type TYPE (IEEE_CLASS_TYPE). Its value is one of the following:

IEEE_SIGNALING_NAN	IEEE_NEGATIVE_NORMAL
IEEE_QUIET_NAN	IEEE_POSITIVE_DENORMAL
IEEE_POSITIVE_INF	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_INF	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_ZERO

Results

The result type and kind are the same as x. The result value is an IEEE value as specified by "class".

When IEEE_VALUE returns a signaling NaN, it is processor dependent whether or not invalid is signaled and processor dependent whether or not the signaling NaN is converted to a quiet NaN.

Example

IEEE_VALUE (1.0,IEEE_POSITIVE_INF) has the value +infinity.

See Also

IEEE_ARITHMETIC Intrinsic Module
IEEE Intrinsic Modules Quick Reference Tables

IEOR

Elemental Intrinsic Function (Generic): *Performs an exclusive OR on corresponding bits. This function can also be specified as XOR or IXOR.*

Syntax

```
result = IEOR (i,j)
```

i — (Input) Must be of type integer, logical (which is treated as an integer), or a binary, octal, or hexadecimal literal constant.

j — (Input) Must be of type integer or logical, or a binary, octal, or hexadecimal literal constant.

If both *i* and *j* are of type integer or logical, they must have the same kind type parameter. If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with its sign bit on the left and the larger kind is used for the operation and the result. *i* and *j* must not both be binary, octal, or hexadecimal literal constants.

Results

The result is the same as *i* if *i* is of type integer or logical; otherwise, the result is the same as *j*. If either *i* or *j* is a binary, octal, or hexadecimal literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other.

The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

i	j	IEOR (i, j)
1	1	0
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BIEOR[1]	INTEGER(1)	INTEGER(1)
IIEOR[2]	INTEGER(2)	INTEGER(2)
JIEOR[3]	INTEGER(4)	INTEGER(4)
KIEOR	INTEGER(8)	INTEGER(8)

[1]Or BIXOR

[2]Or HIEOR, HIXOR, or IIXOR

[3]Or JIXOR

Example

IEOR (12, 7) has the value 11; binary 1100 exclusive OR with binary 0111 is binary 1011.

The following shows another example:

```
INTEGER I
I = IEOR(240, 90)  ! returns 170
                   ! IEOR (B'11110000', B'1011010') == B'10101010'
```

The following shows an example using alternate option XOR:

```
 INTEGER i, j, k
 i = 3             ! B'011'
 j = 5             ! B'101'
 k = XOR(i, j)     ! returns 6 = B'110'
```

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
IAND
IOR
NOT
IALL
IANY
IPARITY

IERRNO

Portability Function: *Returns the number of the last detected error from any routines in the IFPORT module that return error codes.*

Module

USE IFPORT

Syntax

```
result = IERRNO( )
```

Results

The result type is INTEGER(4). The result value is the last error code from any portability routines that return error codes. These error codes are analogous to `errno` on a Linux* or macOS* system. The module `IFPORT.F90` provides parameter definitions for the following `errno` names (typically found in `errno.h` on Linux systems):

Symbolic name	Number	Description
EPERM	1	Insufficient permission for operation
ENOENT	2	No such file or directory
ESRCH	3	No such process
EIO	5	I/O error
E2BIG	7	Argument list too long
ENOEXEC	8	File is not executable
ENOMEM	12	Not enough resources
EACCES	13	Permission denied
EXDEV	18	Cross-device link
ENOTDIR	20	Not a directory
EINVAL	22	Invalid argument

The value returned by IERRNO is updated only when an error occurs. For example, if an error occurs on a GETLOG call and then two CHMOD calls succeed, a subsequent call to IERRNO returns the error for the GETLOG call.

Examine IERRNO immediately after returning from a portability routine. IERRNO is set on a per thread basis.

Example

```
USE IFPORT
CHARACTER*20 username
INTEGER(4) ierrval
ierrval=0 !initialize return value
CALL GETLOG(username)
IF (IERRNO( ) == ierrval) then
  print *, 'User name is ',username
  exit
ELSE
  ierrval = ierrno()
  print *, 'Error is ',ierrval
END IF
```

IF - Arithmetic

Statement: *Conditionally transfers control to one of three statements, based on the value of an arithmetic expression. The arithmetic IF statement is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.*

Syntax

```
IF (expr) label1,label2,label3
```

expr	Is a scalar numeric expression of type integer or real (enclosed in parentheses).
label1, label2, label3	Are the labels of valid branch target statements that are in the same scoping unit as the arithmetic IF statement.

Description

All three labels are required, but they do not need to refer to three different statements. The same label can appear more than once in the same arithmetic IF statement.

During execution, the expression is evaluated first. Depending on the value of the expression, control is then transferred as follows:

If the Value of *expr* is:	Control Transfers To:
Less than 0	Statement *label1*
Equal to 0	Statement *label2*
Greater than 0	Statement *label3*

Example

The following example transfers control to statement 50 if the real variable `THETA` is less than or equal to the real variable `CHI`. Control passes to statement 100 only if `THETA` is greater than `CHI`.

```
IF (THETA-CHI) 50,50,100
```

The following example transfers control to statement 40 if the value of the integer variable `NUMBER`is even. It transfers control to statement 20 if the value is odd.

```
IF (NUMBER / 2*2 - NUMBER) 20,40,20
```

The following statement transfers control to statement 10 for n < 10, to statement 20 for n = 10, and to statement 30 for n > 10:

```
IF (n-10) 10, 20, 30
```

The following statement transfers control to statement 10 if n <= 10, and to statement 30 for n > 10:

```
IF (n-10) 10, 10, 30
```

See Also

SELECT CASE...END SELECT
Execution Control
Deleted Language Features in the Fortran Standard

IF - Logical

Statement: *Conditionally executes one statement based on the value of a logical expression. (This statement was called a logical IF statement in FORTRAN 77.)*

Syntax

```
IF (expr) stmt
```

expr	Is a scalar logical expression enclosed in parentheses.
stmt	Is any complete, unlabeled, executable Fortran statement, except for the following: • A CASE, DO, IF, FORALL, or WHERE construct • Another IF statement • The END statement for a program, function, or subroutine

When an IF statement is executed, the logical expression is evaluated first. If the value is true, the statement is executed. If the value is false, the statement is not executed and control transfers to the next statement in the program.

Example

The following examples show valid IF statements:

```
IF (J.GT.4 .OR. J.LT.1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.5D0)

IF (ENDRUN) CALL EXIT
```

The following shows another example:

```
   USE IFPORT
   INTEGER(4) istat, errget
   character(inchar)
   real x
   istat = getc(inchar)
   IF (istat) errget = -1
   ...
!   IF (x .GT. 2.3) call new_subr(x)
   ...
```

See Also

IF Construct
Execution Control

IF Clause

Parallel Directive Clause: *Specifies a conditional expression. If the expression evaluates to .FALSE., the construct is not executed.*

Syntax

IF ([*directive-name-modifier*:] *scalar-logical-expression*)

directive-name-modifier	Names the associated construct that the IF clause applies to. Currently, you can specify one of the following associated constructs (directives): PARALLEL, SIMD, TARGET DATA, TARGET, TARGET UPDATE, TARGET ENTER DATA, TARGET EXIT DATA, TASK, or TASKLOOP. These directives are only available on Linux* systems: TARGET DATA, TARGET, TARGET UPDATE, TARGET ENTER DATA, TARGET EXIT DATA.
scalar-logical-expression	Must be a scalar logical expression that evaluates to .TRUE. or .FALSE..

At most one IF clause can appear in a non-combined directive. In combined directives, IF clauses with different *directive-name-modifiers* can occur, at most one for each constituent directive making up the combined directive where IF is allowed.

Description

The effect of the IF clause depends on the construct to which it is applied:

- For combined or composite constructs, the IF clause only applies to the semantics of the construct named in the *directive-name-modifier* if one is specified.
- If no *directive-name-modifier* is specified for a combined or composite construct then the IF clause applies to all constructs to which an IF clause can apply.

The following are additional rules that apply to specific OpenMP Fortran directives:

- For the CANCEL OpenMP* Fortran directive, if *scalar-logical-expression* evaluates to false, the construct does not request cancellation. Note that *directive-name-modifier* cannot specify CANCEL.
- For the PARALLEL OpenMP Fortran directive:
  - The enclosed code section is executed in parallel only if *scalar-logical-expression* evaluates to .TRUE.. Otherwise, the parallel region is serialized. If this clause is not used, the region is executed as if an IF(.TRUE.) clause were specified.
  - This clause is evaluated in the context outside of this construct.
- For the SIMD OpenMP Fortran directive, if *scalar-logical-expression* evaluates to .FALSE., the number of iterations to be executed concurrently is one.
- For the TARGET OpenMP Fortran directive, if *scalar-logical-expression* evaluates to .FALSE., the target region is not executed by the device. It is executed by the encountering task.
- For the TARGET DATA OpenMP Fortran directive, if *scalar-logical-expression* evaluates to .FALSE., the new device data environment is not created.
- For the TARGET UPDATE OpenMP Fortran directive, if *scalar-logical-expression* evaluates to .FALSE., the TARGET UPDATE directive is ignored.
- For the TASK OpenMP Fortran directive:
  - If *scalar-logical-expression* evaluates to .FALSE., the encountering thread must suspend the current task region and begin execution of the generated task immediately. The suspended task region will not be resumed until the generated task is completed.
  - This clause is evaluated in the context outside of this construct.

IF Construct

Statement: *Conditionally executes one block of constructs or statements depending on the evaluation of a logical expression. (This construct was called a block IF statement in FORTRAN 77.)*

Syntax

```
[name:] IF (expr) THEN
    block
[ELSE IF (expr) THEN [name]
   block]
[ELSE [name]
   block]
END IF [name]
```

name (Optional) Is the name of the IF construct.

expr	Is a scalar logical expression enclosed in parentheses.
block	Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified at the beginning of an IF THEN statement, the same name must appear in the corresponding END IF statement. If a construct name is specified on an ELSE IF or ELSE statement, the same name must appear in the corresponding IF THEN and END IF statements.

The same construct name must not be used for different named constructs in the same scoping unit.

Depending on the evaluation of the logical expression, one block or no block is executed. The logical expressions are evaluated in the order in which they appear, until a true value is found or an ELSE or END IF statement is encountered.

Once a true value is found or an ELSE statement is encountered, the block immediately following it is executed and the construct execution terminates.

If none of the logical expressions evaluate to true and no ELSE statement appears in the construct, no block in the construct is executed and the construct execution terminates.

NOTE
No additional statement can be placed after the IF THEN statement in a block IF construct. For example, the following statement is invalid in the block IF construct:

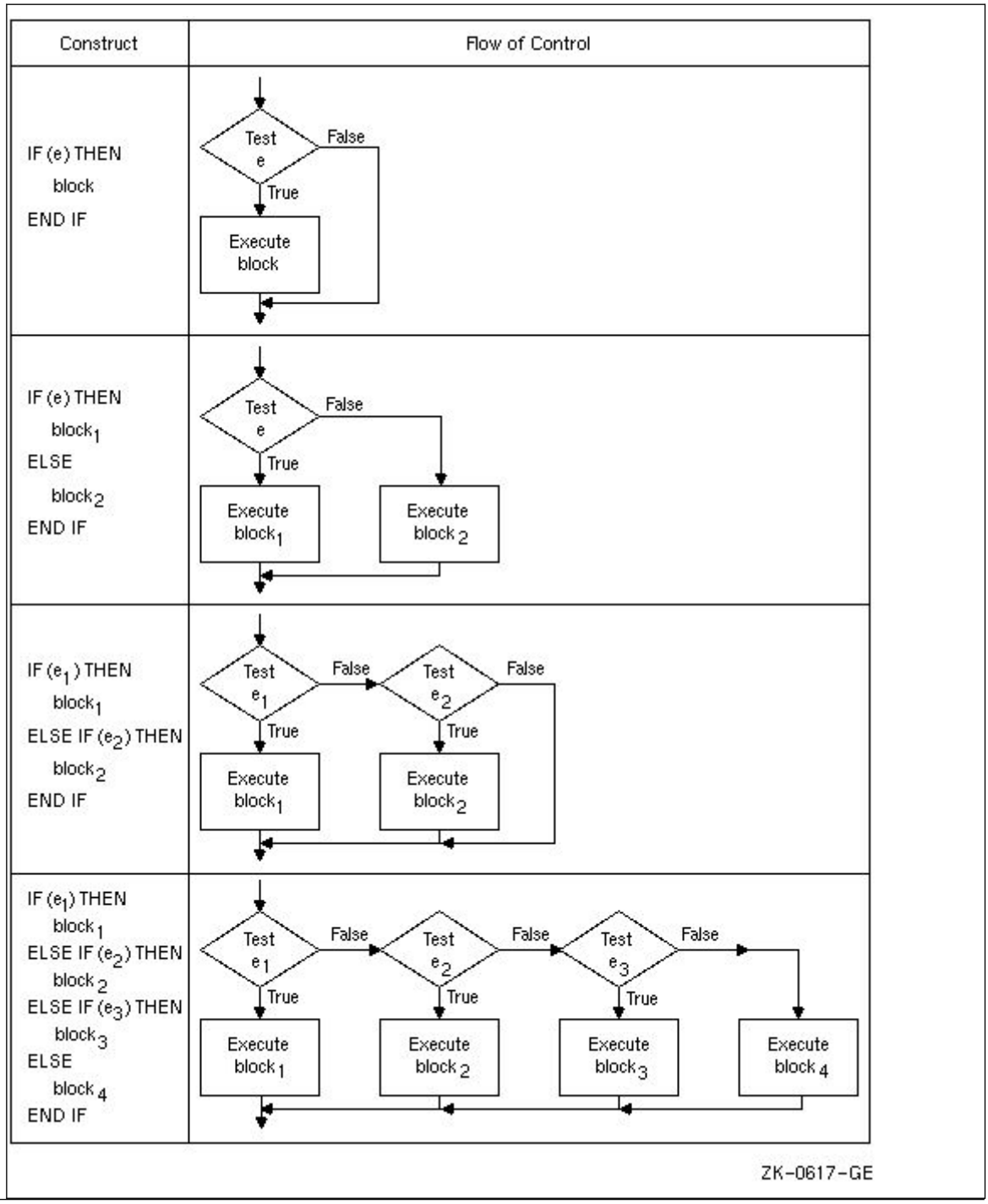
```
IF (e) THEN I = J
```

This statement is translated as the following logical IF statement:

```
IF (e) THENI = J
```

You cannot use branching statements to transfer control to an ELSE IF statement or ELSE statement. However, you can branch to an END IF statement from within the IF construct.

The following figure shows the flow of control in IF constructs:



You can include an IF construct in the statement block of another IF construct, if the nested IF construct is completely contained within a statement block. It cannot overlap statement blocks.

Example

The following example shows the simplest form of an IF construct:

```
Form                  Example
  IF (expr) THEN        IF (ABS(ADJU) .GE. 1.0E-6) THEN
    block                 TOTERR = TOTERR + ABS(ADJU)
                          QUEST = ADJU/FNDVAL
  END IF                END IF
```

This construct conditionally executes the block of statements between the IF THEN and the END IF statements.

The following shows another example:

```
!  Simple block IF:
   IF (i .LT. 10) THEN
      ! the next two statements are only executed if i < 10
      j = i
      slice = TAN (angle)
   END IF
```

The following example shows a named IF construct:

```
  BLOCK_A: IF (D > 0.0) THEN        ! Initial statement for named construct

    RADIANS = ACOS(D)               ! These two statements
    DEGREES = ACOSD(D)              !       form a block

  END IF BLOCK_A                    ! Terminal statement for named construct
```

The following example shows an IF construct containing an ELSE statement:

```
Form                  Example
  IF (expr) THEN        IF (NAME .LT. 'N') THEN
    block1                IFRONT = IFRONT + 1
                          FRLET(IFRONT) = NAME(1:2)
  ELSE                  ELSE
    block2                IBACK = IBACK + 1
  END IF                END IF
```

Block1 consists of all the statements between the IF THEN and ELSE statements. Block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than `'N '`, block1 is executed. If the value of NAME is greater than or equal to `'N '`, block2 is executed.

The following example shows an IF construct containing an ELSE IF THEN statement:

```
Form                    Example
    IF (expr) THEN          IF (A .GT. B) THEN
    block1                  D = B
                            F = A - B
  ELSE IF (expr) THEN     ELSE IF (A .GT. B/2.) THEN
    block2                  D = B/2.
                            F = A - B/2.
  END IF                  END IF
```

If A is greater than B, block1 is executed. If A is not greater than B, but A is greater than B/2, block2 is executed. If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed. Control transfers directly to the next executable statement after the END IF statement.

The following shows another example:

```
!  Block IF with ELSE IF statements:

   IF (j .GT. 1000) THEN
      ! Statements here are executed only if J > 1000
   ELSE IF (j .GT. 100) THEN
      ! Statements here are executed only if J > 100 and j <= 1000
   ELSE IF (j .GT. 10) THEN
      ! Statements here are executed only if J > 10 and j <= 100
   ELSE
      ! Statements here are executed only if j <= 10
   END IF
```

The following example shows an IF construct containing several ELSE IF THEN statements and an ELSE statement:

```
Form                      Example
  IF (expr) THEN            IF (A .GT. B) THEN
    block1                    D = B
                              F = A - B
  ELSE IF (expr) THEN       ELSE IF (A .GT. C) THEN
    block2                    D = C
                              F = A - C
  ELSE IF (expr) THEN       ELSE IF (A .GT. Z) THEN
    block3                    D = Z
                              F = A - Z
  ELSE                      ELSE
    block4                    D = 0.0
                              F = A
  END IF                    END IF
```

If A is greater than B, block1 is executed. If A is not greater than B but is greater than C, block2 is executed. If A is not greater than B or C but is greater than Z, block3 is executed. If A is not greater than B, C, or Z, block4 is executed.

The following example shows a nested IF construct:

```
Form                      Example
  IF (expr) THEN            IF (A .LT. 100) THEN
    block1                    INRAN = INRAN + 1
    IF (expr2) THEN           IF (ABS(A-AVG) .LE. 5.) THEN
      block1a                   INAVG = INAVG + 1
    ELSE                      ELSE
      block1b                   OUTAVG = OUTAVG + 1
    END IF                    END IF
  ELSE                      ELSE
    block2                    OUTRAN = OUTRAN + 1
  END IF                    END IF
```

If A is less than 100, the code immediately following the IF is executed. This code contains a nested IF construct. If the absolute value of A minus AVG is less than or equal to 5, block1a is executed. If the absolute value of A minus AVG is greater than 5, block1b is executed.

If A is greater than or equal to 100, block2 is executed, and the nested IF construct (in block1) is not executed.

The following shows another example:

```
!  Nesting of constructs and use of an ELSE statement following
!  a block IF without intervening ELSE IF statements:
   IF (i .LT. 100) THEN
```

```
    ! Statements here executed only if i < 100
    IF (j .LT. 10) THEN
    ! Statements here executed only if i < 100 and j < 10
    END IF
    ! Statements here executed only if i < 100
  ELSE
    ! Statements here executed only if i >= 100
    IF (j .LT. 10) THEN
    ! Statements here executed only if i >= 100 and j < 10
    END IF
    ! Statements here executed only if i >= 100
  END IF
```

See Also

Execution Control
IF - Logical
IF - Arithmetic

IF Directive Construct

General Compiler Directive: *A conditional compilation construct that begins with an IF or IF DEFINED directive. IF tests whether a logical expression is .TRUE. or .FALSE.. IF DEFINED tests whether a symbol has been defined.*

Syntax

```
!DIR$ IF (expr) -or- !DIR$ IF DEFINED (name)
   block
[!DIR$ ELSEIF (expr)
   block] ...
[!DIR$ ELSE
   block]
!DIR$ ENDIF
```

expr	Is a logical expression that evaluates to .TRUE. or .FALSE..
name	Is the name of a symbol to be tested for definition.
block	Are executable statements that are compiled (or not) depending on the value of logical expressions in the IF directive construct.

The IF and IF DEFINED directive constructs end with an ENDIF directive and can contain one or more ELSEIF directives and at most one ELSE directive. If the logical condition within a directive evaluates to .TRUE. at compilation, and all preceding conditions in the IF construct evaluate to .FALSE., then the statements contained in the directive block are compiled.

A *name* can be defined with a DEFINE directive, and can optionally be assigned an integer value. If the symbol has been defined, with or without being assigned a value, IF DEFINED (name) evaluates to .TRUE.; otherwise, it evaluates to .FALSE..

If the logical condition in the IF or IF DEFINED directive is .TRUE., statements within the IF or IF DEFINED block are compiled. If the condition is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If the logical expression in an ELSEIF directive is .TRUE., statements within the ELSEIF block are compiled. If the expression is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If control reaches an ELSE directive because all previous logical conditions in the IF construct evaluated to .FALSE., the statements in an ELSE block are compiled unconditionally.

You can use any Fortran logical or relational operator or symbol in the logical expression of the directive, including: .LT., <, .GT., >, .EQ., ==, .LE., <=, .GE., >=, .NE., /=, .EQV., .NEQV., .NOT., .AND., .OR., and .XOR.. The logical expression can be as complex as you like, but the whole directive must fit on one line.

Example

```
!  When the following code is compiled and run,
!  the output is:
!  Or this compiled if all preceding conditions .FALSE.
!
!DIR$ DEFINE flag=3
!DIR$ IF (flag .LT. 2)
   WRITE (*,*) "This is compiled if flag less than 2."
!DIR$ ELSEIF (flag >= 8)
   WRITE (*,*) "Or this compiled if flag greater than &
                or equal to 8."
!DIR$ ELSE
   WRITE (*,*) "Or this compiled if all preceding &
                conditions .FALSE."
!DIR$ ENDIF
END
```

See Also

DEFINE and UNDEFINE
IF Construct
General Compiler Directives
Syntax Rules for Compiler Directives

IF DEFINED Directive

Statement: *Marks the start of an IF Directive Construct.*

See Also

See IF Directive Construct.

IFIX

Elemental Intrinsic Function (Generic): *Converts a single-precision real argument to an integer by truncating.*

See Also

See INT.

IFLOATI, IFLOATJ

Portability Functions: *Convert an integer to single-precision real type.*

Module

USE IFPORT

Syntax

```
result = IFLOATI (i)
result = IFLOATJ (j)
```

i	(Input) Must be of type INTEGER(2).
j	(Input) Must be of type INTEGER(4).

Results

The result type is single-precision real (REAL(4) or REAL*4).

See Also

DFLOAT

ILEN

Inquiry Intrinsic Function (Generic): *Returns the length (in bits) of the two's complement representation of an integer.*

Syntax

```
result = ILEN (i)
```

i	Must be of type integer.

Results

The result type and kind are the same as *i*. The result value is (LOG_2(*i*+ 1)) if *i* is not negative; otherwise, the result value is (LOG_2(- *i*)).

Example

ILEN (4) has the value 3.

ILEN (-4) has the value 2.

IMAGE_INDEX

Transformational Intrinsic Function (Generic): *Converts cosubscripts to an image index.*

Syntax

```
result = IMAGE_INDEX (coarray,sub)
```

coarray	(Input) Must be a coarray; it can be of any type.
sub	(Input) Must be a rank-one integer array of size equal to the corank of *coarray*.

Results

The result is default integer scalar. The result is the index of the corresponding image if the value of *sub* is a valid sequence of cosubscripts for *coarray*. Otherwise, the result is zero.

Example

If coarray D is declared as D [0:*] and coarray C is declared as C(5,10) [10, 0:9, 0:*], IMAGE_INDEX (D, [0]) has the value 1 and IMAGE_INDEX (C, [3, 1, 2]) has the value 213 (on any image).

IMAGESIZE, IMAGESIZE_W (W*S)

Graphics Functions: *Return the number of bytes needed to store the image inside the specified bounding rectangle. IMAGESIZE is useful for determining how much memory is needed for a call to GETIMAGE.*

Module

USE IFQWIN

Syntax

```
result = IMAGESIZE (x1,y1,x2,y2)
result = IMAGESIZE_W (wx1,wy1,wx2,wy2)
```

x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of image.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of image.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of image.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of image.

Results

The result type is INTEGER(4). The result is the storage size of an image in bytes.

IMAGESIZE defines the bounding rectangle in viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). IMAGESIZE_W defines the bounding rectangle in window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

IMAGESIZE_W defines the bounding rectangle in terms of window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

Example

See the example in GETIMAGE.

See Also

GETIMAGE
GRSTATUS
PUTIMAGE

IMAGE_STATUS

Elemental Intrinsic Function (Generic): *Returns the execution status value of the specified image.*

Syntax

```
result = IMAGE_STATUS (image)
```

image	(Input) Must be a positive integer with a value equal to or less than the number of executing images on the current (initial) team.

Results

The result type is default integer. If *image* has initiated normal termination, the result is the value STAT_STOPPED_IMAGE defined in the intrinsic module ISO_FORTRAN_ENV. If *image* has failed, the result is the value STAT_FAILED_IMAGE defined in the intrinsic module ISO_FORTRAN_ENV. Otherwise, the result value is zero.

NOTE
IMAGE_STATUS argument *team* has not yet been implemented. It will be added in a future release.

Example

If image 5 on the current team has initiated normal termination, and image 12 on the current team is known to have failed, then IMAGE_STATUS (5) returns the value STAT_STOPPED_IMAGE and IMAGE_STATUS (12) returns the value STAT_FAILED_IMAGE. If image 3 on the current team has neither initiated normal termination nor failed, IMAGE_STATUS (3) returns the value zero.

See Also

FAILED_IMAGES
STOPPED_IMAGES
ISO_FORTRAN_ENV Module

IMPLICIT

Statement: *Overrides the default implicit typing rules for names. (The default data type is default INTEGER kind for names beginning with the letters I through N, and default REAL kind for names beginning with any other letter.) An IMPLICIT NONE statement overrides all implicit typing in a scoping unit, or it indicates that all dummy procedures and externals must explicitly be given the external attribute.*

Syntax

The IMPLICIT statement takes one of the following forms:

```
IMPLICIT type(a[,a]...)[,type(a[,a]...)]...
IMPLICIT NONE [(spec-list])
```

type	Is a data type specifier (CHARACTER*(*) is not allowed).
a	Is a single letter, a dollar sign ($),or a range of letters in alphabetical order. The form for a range of letters is a_1-a_2, where the second letter follows the first alphabetically (for example, A-C).
spec	Is TYPE or EXTERNAL. The keyword must appear in parentheses. If more than one keyword is present, they must be separated by commas.

The dollar sign can be used at the end of a range of letters, since IMPLICIT interprets the dollar sign to alphabetically follow the letter Z. For example, a range of X-$ would apply to identifiers beginning with the letters X, Y, Z, or $.

In Intel® Fortran, the parentheses around the list of letters are optional.

Description

The IMPLICIT statement assigns the specified data type (and kind parameter) to all names that have no explicit data type and begin with the specified letter or range of letters. It has no effect on the default types of intrinsic procedures.

When the data type is CHARACTER*len, *len* is the length for character type. The *len* is an unsigned integer constant or an integer specification expression enclosed in parentheses. The range for *len* is 1 to 2**31-1 on IA-32 architecture; 1 to 2**63-1 on Intel® 64 architecture.

Names beginning with a dollar sign ($) are implicitly INTEGER.

The IMPLICIT NONE statement disables all implicit typing defaults. When IMPLICIT NONE is used, all names in a program unit must be explicitly declared. An IMPLICIT NONE statement must precede any PARAMETER statements, and there must be no other IMPLICIT statements in the scoping unit. An IMPLICIT NONE (TYPE) statement is the same as an IMPLICIT NONE statement. If an IMPLICIT NONE (EXTERNAL) statement appears in a scoping unit, all dummy procedures and external procedures in that scope or a contained scope of BLOCK must have an accessible explicit interface or be declared EXTERNAL.

NOTE
To receive diagnostic messages when variables are used but not declared, you can specify compiler option warn declarations instead of using IMPLICIT NONE or IMPLICIT NONE (TYPE). To receive diagnostic messages when external and dummy procedures have not explicitly been given the EXTERNAL attribute, you can specify compiler option `warn externals` instead of using IMPLICIT NONE (EXTERNAL).

The following IMPLICIT statement represents the default typing as specified by the Fortran Standard for names when they are not explicitly typed:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

Example

The following are examples of the IMPLICIT statement:

```
IMPLICIT DOUBLE PRECISION (D) IMPLICIT COMPLEX (S,Y), LOGICAL(1) (L,A-C)
IMPLICIT CHARACTER*32 (T-V)
IMPLICIT CHARACTER*2 (W)
IMPLICIT TYPE(COLORS) (E-F), INTEGER (G-H)
IMPLICIT NONE (EXTERNALS, TYPE)    ! Must be the only IMPLICIT statement in a scoping unit
```

The following shows another example:

```
SUBROUTINE FF (J)
IMPLICIT INTEGER (a-b), CHARACTER*(J+1) (n), TYPE(fried) (c-d)
TYPE fried
INTEGER e, f
REAL g, h
END TYPE
age = 10 ! integer
name = 'Paul' ! character
c%e = 1 ! type fried, integer component
```

See Also

Data Types, Constants, and Variables
warn declarations compiler option
warn externals compiler option

IMPORT

Statement: *Controls accessibility of host entities in a submodule, module procedure, a contained procedure, a block construct, or in the interface body of an interface block.*

Syntax

The IMPORT statement takes the following form:

```
IMPORT [[::] import-name-list]

IMPORT,ONLY: import-name-list

IMPORT,NONE

IMPORT,ALL
```

import-name-list	(Input) Is the name of one or more entities accessible in the host scoping unit.

An IMPORT statement can appear in a submodule, module procedure, a contained procedure, the specification part of a BLOCK construct, or in an interface body. It can not appear in the specification part of a main program, external procedure, module, or block data except in an interface body.

An IMPORT statement must appear after any USE statements, and before any other specification statements. Each of the named entities must be an accessible entity in the host scoping unit. Within an interface body each named entity must be explicitly declared before the interface body, or accessible by use or host association in the host containing the IMPORT statement.

If IMPORT, ONLY appears within a scoping unit, all other IMPORT statements in that scoping unit must be IMPORT, ONLY statements. An entity is host associated in a scoping unit which contains an IMPORT, ONLY statement if it appears in an *import-name-list* in that scoping unit.

If IMPORT, NONE is specified, no entities in the host are accessible by host association in that scoping unit. This is the default behavior for interface bodies for a dummy or external procedure. IMPORT, NONE must not appear in the specification part of a submodule.

If an IMPORT, ALL or an IMPORT, NONE statement appears in a scoping unit, it must be the only IMPORT statement in that scoping unit.

If *import-name-list* is not specified, and if ALL, ONLY, or NONE are not specified, all of the accessible named entities in the host scoping unit are imported unless they are made inaccessible by an entity of the same name in the local scope. This is the default behavior for a nested scoping unit, other than an interface body, for a dummy or external procedure.

If IMPORT, ALL is specified, all entities in the host are accessible by host association. If an entity is made accessible by an IMPORT, ALL statement or by its name appearing in an *import-name-list*, it cannot be made inaccessible by declaring another entity with the same name in the local scope.

If an IMPORT statement with an *import-name-list* appears, only the named entities are available by host association.

Examples

The following examples show how the IMPORT statement can be applied.

```
module mymod
type mytype
  integer comp
end type mytype
interface
```

```
  subroutine sub (arg)
    import
    type(mytype) :: arg
  end subroutine sub
end interface
end module mymod

module host
  integer  :: i, j, k
  contains
  subroutine sub1 ()
    import :: i, j
    k = i + j      ! only i and j are host associated, k is local to sub1
  end subroutine
  subroutine sub2 ()
    import, none
    k = i + j      ! i, j, and k are local to sub2
  end subroutine
  subroutine sub3 ()
    import all
    k = i + j      ! i, j, and k are all host associated
  end subroutine
  subroutine sub4 ()
    import, only : i
    import, only : j
    k = i + j      ! i and j are host associated, k is local
  end subroutine
end module
```

IMPURE

Keyword: *Asserts that a user-defined procedure has side effects.*

Description

This kind of procedure is specified by using the prefix IMPURE in a FUNCTION or SUBROUTINE statement. By default all user-defined procedures are impure, that is, they are allowed to have side effects, except for elemental procedures.

An IMPURE elemental procedure has the restrictions that apply to elemental procedures, but it does not have any of the restrictions of PURE elemental procedures.

An impure elemental procedure can have side effects and it can contain the following:

- Any external I/O statement (including a READ or WRITE statement whose I/O unit is an external file unit number or *)
- A PAUSE statement
- A STOP statement or an ERROR STOP statement
- An image control statement

An impure elemental procedure cannot be referenced in a context that requires a procedure to be pure; for example:

- It cannot be called directly in a FORALL statement or be used in the mask expression of a FORALL statement.
- It cannot be called from a pure procedure. Pure procedures can only call other pure procedures, including one referenced by means of a defined operator, defined assignment, or finalization.
- It cannot be passed as an actual argument to a pure procedure.

Example

```
module my_rand_mod
integer, save :: my_rand_seed (8)

  contains
    impure elemental subroutine my_rand (r)

    ! my_rand updates module variable my_rand_seed (an array)
    ! and returns the next value of a "pseudo" random sequence in
    ! output dummy argument r

      real, intent(out) :: r

      ! code goes here

    end subroutine my_rand
end module my_rand_mod
```

See Also

FUNCTION
SUBROUTINE

IN_REDUCTION

Specifies that a task participates in a reduction. The IN_REDUCTION clause is a reduction participating clause.

Syntax

IN_REDUCTION (*reduction-identifier* : *list*)

Arguments *reduction-identifier* and *list* are defined in REDUCTION clause. All the common restrictions for the REDUCTION clause apply to this clause.

A *list* item that appears in an IN_REDUCTION clause of a TASK construct must appear in a TASK_REDUCTION clause of a construct associated with a TASKGROUP region that includes the participating task in its taskgroup set. The construct associated with the innermost region that meets this condition must specify the same operator or intrinsic as the IN_REDUCTION clause.

In the following example, the IN_REDUCTION clause at (3) must name the same operator (+) and variable (a) as in the TASK_REDUCTION clause in (2). The IN_REDUCTION clause at (4) must name the same operator (*) and variable (a) as in the TASK_REDUCTION clause in (1).

```
!$omp taskgroup task_reduction(*:a)       ! (1) *:a
    ...
!$omp taskgroup task_reduction(+:a)       ! (2) +:a
!$omp task in_reduction(+:a)              ! (3) +:a matches (2)
    a = a + x
!$omp end task                            ! ends (3) +:a
    ...
!$omp end taskgroup                          ! ends (2) +:a
    ...
!$omp task in_reduction(*:a)              ! (4) *:a matches (1)
    a = a * y
!$omp end task                            ! ends (4) *:a
!$omp end taskgroup                          ! ends (1) *:a
```

See Also

REDUCTION
DECLARE REDUCTION
TASK_REDUCTION
TASK
TASKLOOP

INCHARQQ (W*S)

QuickWin Function: *Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering.*

Module

USE IFQWIN

Syntax

```
result = INCHARQQ( )
```

Results

The result type is INTEGER(2). The result is the ASCII key code.

The keystroke is read from the child window that currently has the focus. You must call INCHARQQ before the keystroke is made (INCHARQQ does not read the keyboard buffer). This function does not echo its input. For function keys, INCHARQQ returns 0xE0 as the upper 8 bits, and the ASCII code as the lower 8 bits.

For direction keys, INCHARQQ returns 0xF0 as the upper 8 bits, and the ASCII code as the lower 8 bits. To allow direction keys to be read, you must use the PASSDIRKEYSQQ function. The escape characters (the upper 8 bits) are different from those of GETCHARQQ. Note that console applications do not need, and cannot use PASSDIRKEYSQQ.

Example

```
use IFQWIN
integer*4 res
integer*2 exchar
character*1 ch, ch1

Print *,"Type X to exit, S to scroll, D to pass Direction keys"

123 continue
exchar = incharqq()
! check for escapes
! 0xE0 0x?? is a function key
! 0xF0 0x?? is a direction key

ch = char(rshift(exchar,8) .and. Z'00FF')
ch1= char(exchar .and. Z'00FF')

if (ichar(ch) .eq. 224) then
  print *,"function key  = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
  goto 123
endif

if (ichar(ch) .eq. 240) then
  print *,"direction key = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
```

```
  goto 123
endif

print *,"other key  = ",ichar(ch),"  ",ichar(ch1),"  ",ch1

if(ch1 .eq. 'S') then
  res = passdirkeysqq(.false.)
  print *, "Entering Scroll mode"
endif

if(ch1 .eq. 'D')  then
  res = passdirkeysqq(.true.)
  print *, "Entering Direction keys mode"
endif

if(ch1 .ne. 'X')  go to 123
end
```

See Also

GETCHARQQ
READ
MBINCHARQQ
GETC
PASSDIRKEYSQQ

INCLUDE

Statement: *Directs the compiler to stop reading statements from the current file and read statements in an included file or text module.*

Syntax

The INCLUDE line takes the following form:

```
INCLUDE 'filename[/[NO]LIST]'
```

filename	Is a character string specifying the name of the file to be included; it must not be a named constant. The form of the file name must be acceptable to the operating system, as described in your system documentation.
[NO]LIST	Specifies whether the incorporated code is to appear in the compilation source listing. In the listing, a number precedes each incorporated statement. The number indicates the "include" nesting depth of the code. The default is /NOLIST. /LIST and /NOLIST must be spelled completely. You can only use /[NO]LIST if you specify compiler option vms (which sets OpenVMS defaults).

Description

An INCLUDE line can appear anywhere within a scoping unit. The line can span more than one source line, but no other statement can appear on the same line. The source line cannot be labeled.

An included file or text module cannot begin with a continuation line, and each Fortran statement must be completely contained within a single file.

An included file or text module can contain any source text, but it cannot begin or end with an incomplete Fortran statement.

The included statements, when combined with the other statements in the compilation, must satisfy the statement-ordering restrictions shown in Statements.

Included files or text modules can contain additional INCLUDE lines, but they must not be recursive. INCLUDE lines can be nested until system resources are exhausted.

When the included file or text module completes execution, compilation resumes with the statement following the INCLUDE line.

You can use modules instead of include files to achieve encapsulation of related data types and procedures. For example, one module can contain derived type definitions as well as special operators and procedures that apply to those types. For information on how to use modules, see Program Units and Procedures.

Example

In the following example, a file named COMMON.FOR (in the current working directory) is included and read as input.

Including Text from a File

Main Program File

```
PROGRAM
  INCLUDE 'COMMON.FOR'
  REAL, DIMENSION(M) :: Z
  CALL CUBE
  DO I = 1, M
    Z(I) = X(I) + SQRT(Y(I))
    ...
  END DO
END

SUBROUTINE CUBE
  INCLUDE 'COMMON.FOR'
  DO I=1,M
    X(I) = Y(I)**3
  END DO
  RETURN
END
```

COMMON.FOR File

```
INTEGER, PARAMETER :: M=100
REAL, DIMENSION(M) :: X, Y
COMMON X, Y
```

The file COMMON.FOR defines a named constant M, and defines arrays X and Y as part of blank common.

The following example program declares its common data in an include file. The contents of the file INCLUDE.INC are inserted in the source code in place of every INCLUDE 'INCLUDE.INC' line. This guarantees that all references to common storage variables are consistent.

```
INTEGER i
REAL x
INCLUDE 'INCLUDE.INC'

DO i = 1, 5
  READ (*, '(F10.5)') x
  CALL Push (x)
END DO
```

See Also

MODULE

USE

INDEX

Elemental Intrinsic Function (Generic): *Returns the starting position of a substring within a string.*

Syntax

```
result = INDEX (string, substring [,back] [, kind])
```

string	(Input) Must be of type character.
substring	(Input) Must be of type character.
back	(Input; optional) Must be of type logical.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* does not appear (or appears with the value false), the value returned is the minimum value of I such that *string*(I : I + LEN (*substring*) - 1) = *substring*(or zero if there is no such value). If LEN (*string*) < LEN (*substring*), zero is returned. If LEN (*substring*) = zero, 1 is returned.

If *back* appears with the value true, the value returned is the maximum value of I such that *string*(I : I + LEN (*substring*) - 1) = *substring* (or zero if there is no such value). If LEN(*string*) < LEN (*substring*), zero is returned. If LEN (*substring*) = zero, LEN (*string*) + 1 is returned.

Specific Name	Argument Type	Result Type
	CHARACTER	INTEGER(1)
	CHARACTER	INTEGER(2)
INDEX [1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

[1]The setting of compiler options specifying integer size can affect this function.

Example

INDEX ('FORTRAN', 'O', BACK = .TRUE.) has the value 2.

INDEX ('XXXX', " ", BACK = .TRUE.) has the value 5.

The following shows another example:

```
I = INDEX('banana','an', BACK = .TRUE.) ! returns 4
I = INDEX('banana', 'an') ! returns 2
```

See Also

SCAN

INITIALIZEFONTS (W*S)

Graphics Function: *Initializes Windows* fonts.*

Module

USE IFQWIN

Syntax

```
result = INITIALIZEFONTS( )
```

Results

The result type is INTEGER(2). The result is the number of fonts initialized.

All fonts on Windows systems become available after a call to INITIALIZEFONTS. Fonts must be initialized with INITIALIZEFONTS before any other font-related library function (such as GETFONTINFO, GETGTEXTEXTENT, SETFONT, OUTGTEXT) can be used.

The font functions affect the output of OUTGTEXT only. They do not affect other Fortran I/O functions (such as WRITE) or graphics output functions (such as OUTTEXT).

For each window you open, you must call INITIALIZEFONTS before calling SETFONT. INITIALIZEFONTS needs to be executed after each new child window is opened in order for a subsequent SETFONT call to be successful.

Example

```
! build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) numfonts
numfonts = INITIALIZEFONTS()
WRITE (*,*) numfonts
END
```

See Also

SETFONT
OUTGTEXT

INITIALSETTINGS (W*S)

QuickWin Function: *Initializes QuickWin.*

Module

USE IFQWIN

Syntax

```
result = INITIALSETTINGS( )
```

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can change the initial appearance of an application's default frame window and menus by defining an INITIALSETTINGS function. Do not use INITIALSETTINGS to open or change the properties of child windows.

If no user-defined INITIALSETTINGS function is supplied, QuickWin calls a predefined INITIALSETTINGS routine to control the default frame window and menu appearance. You do not need to call INITIALSETTINGS if you define it, since it will be called automatically during initialization.

See Also

APPENDMENUQQ
INSERTMENUQQ

DELETEMENUQQ
SETWSIZEQQ

INLINE, FORCEINLINE, and NOINLINE

General Compiler Directives: *Tell the compiler to perform the specified inlining on routines within statements or DO loops.*

Syntax

```
!DIR$ INLINE [RECURSIVE]

!DIR$ FORCEINLINE [RECURSIVE]

!DIR$ NOINLINE
```

The INLINE directive specifies that the routines can be inlined.

The FORCEINLINE directive specifies that a routine should be inlined whenever the compiler can do so. This condition can also be specified by using compiler option `[Q]inline-forceinline`.

The NOINLINE directive specifies that a routine should not be inlined.

These directives apply only to the immediately following statement or DO construct. All statements within the immediately following DO construct are affected.

If keyword RECURSIVE is specified, the specified inlining is performed when the routine calls itself directly or indirectly.

The inlining can be ignored by the compiler if inline heuristics determine it may have a negative impact on performance or will cause too much of an increase in code size.

Caution
When you use directive FORCEINLINE, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Example

Consider the following:

```
!DIR$ INLINE
A = F(B) + G(C)  ! inline the call to function F and inline the call to function G

!DIR$ INLINE
   DO I = 1, N
      CALL F1 ( G1(A), G2(A)) ! inline the call to F1 and the function executions of G1 and G2
      !DIR$ NOINLINE
         DO J = 1, M
            M(J) = F (M(J))   ! do not inline this call to F {M is a data array}
         END DO
      M(I) = F2 (X)           ! F2 gets inlined from the directive before DO I

      !DIR$ FORCEINLINE RECURSIVE
      CALL F3 ()       ! F3 must be inlined and it calls itself recursively so inline those
calls too
   END DO
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements
`inline-forceinline, Qinline-forceinline` compiler option

INMAX

Portability Function: *Returns the maximum positive value for an integer.*

Module

USE IFPORT

Syntax

```
result = INMAX (i)
```

i	(Input) INTEGER(4).

Results

The result type is INTEGER(4). The result is the maximum 4-byte signed integer value for the argument.

INQFOCUSQQ (W*S)

QuickWin Function: *Determines which window has the focus.*

Module

USE IFQWIN

Syntax

```
result = INQFOCUSQQ (unit)
```

unit	(Output) INTEGER(4). Unit number of the window that has the I/O focus.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero. The function fails if the window with the focus is associated with a closed unit.

Unit numbers 0, 5, and 6 refer to the default window only if the program has not specifically opened them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

The window with focus is always in the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling SETACTIVEQQ.

A window has focus when it is given the focus by FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when an I/O statement is executed on that unit. For example:

```
  OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

By default IOFOCUS=.TRUE., except for child windows opened with as unit *. If IOFOCUS=.TRUE., the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

See Also
FOCUSQQ

INQUIRE
Statement: *Returns information on the status of specified properties of a file, logical unit, or directory. It takes one of the following forms:*

Syntax
Inquiring by File:

```
INQUIRE (FILE=name[, ERR=label] [, ID=id-var] [, IOMSG=msg-var] [, SIZE=sz] [, IOSTAT=i-var] [, DEFAULTFILE=def] slist)
```

Inquiring by Unit:

```
INQUIRE ([UNIT=]io-unit [, ERR=label] [, ID=id-var] [, IOMSG=msg-var] [, SIZE=sz] [, IOSTAT=i-var] slist)
```

Inquiring by Directory:

```
INQUIRE (DIRECTORY=dir, EXIST=ex [, DIRSPEC=dirspec] [, ERR=label] [, ID=id-var] [, IOMSG=msg-var] [, SIZE=sz] [, IOSTAT=i-var])
```

Inquiring by Output List:

```
INQUIRE (IOLENGTH=len) out-item-list
```

name	Is a scalar default character expression specifying the name of the file for inquiry. For more information, see FILE Specifier and STATUS Specifier.
label	Is the label of the branch target statement that receives control if an error occurs. For more information, see Branch Specifiers.
id-var	Is a scalar integer expression identifying a data transfer operation that was returned using the ID= specifier in a previous asynchronous READ or WRITE statement. For more information, see ID Specifier.
msg-var	Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs. For more information, see I/O Message Specifier.
sz	Is a scalar integer variable that is assigned the size of the file in file storage units. For more information, see Size Specifier.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs. For more information, see I/O Status Specifier.
def	Is a scalar default character expression specifying a default file pathname string. (For more information, see the DEFAULTFILE specifier.)
slist	Is one or more of the following inquiry specifiers (each specifier can appear only once):

ACCESS	DELIM	NEXTREC	RECL
ACTION	DIRECT	NUMBER	RECORDTYPE
ASYNCHRONOUS	ENCODING	OPENED	ROUND
BINARY	EXIST	ORGANIZATION	SEQUENTIAL
BLANK	FORM	PAD	SHARE
BLOCKSIZE	FORMATTED	PENDING	SIGN
BUFFERED	IOFOCUS	POS	SIZE
CARRIAGECONTROL	MODE	POSITION	UNFORMATTED
CONVERT	NAME	READ	WRITE
DECIMAL	NAMED	READWRITE	

io-unit
Is an external unit specifier.

The unit does not have to exist, nor does it need to be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

dir
Is a scalar default character expression specifying the name of the directory for inquiry. If you are inquiring by directory, it must be present.

ex
Is a scalar default logical variable that is assigned the value .TRUE. if *dir* names a directory that exists; otherwise, *ex* is assigned the value .FALSE.. If you are inquiring by directory, it must be present. For more information, see the EXIST Specifier.

dirspec
Is a scalar default character variable that is assigned the value of the full directory specification of *dir* if *ex* is assigned the value .TRUE.. This specifier can only be used when inquiring by directory.

len
(Output) Is a scalar integer variable that is assigned a value corresponding to the length of an unformatted, direct-access record resulting from the use of the *out-item-list* in a WRITE statement.

The value is suitable to use as a RECL specifier value in an OPEN statement that connects a file for unformatted, direct access.

The unit of the value is 4-byte longwords, by default. However, if you specify compiler option assume byterecl, the unit is bytes.

out-item-list
(Output) Is a list of one or more output items (see I/O Lists).

Description

The control specifiers ([UNIT=] *io-unit*, ERR= *label*, and IOSTAT= *i-var*) and inquiry specifiers can appear anywhere within the parentheses following INQUIRE. However, if the UNIT keyword is omitted, the *io-unit* must appear first in the list.

An INQUIRE statement can be executed before, during, or after a file is connected to a unit. The specifier values returned are those that are current when the INQUIRE statement executes.

To get file characteristics, specify the INQUIRE statement after opening the file.

Example

The following are examples of INQUIRE statements:

```
INQUIRE (FILE='FILE_B', EXIST=EXT)
INQUIRE (4, FORM=FM, IOSTAT=IOS, ERR=20)
INQUIRE (IOLENGTH=LEN) A, B
```

In the last statement, you can use the length returned in LEN as the value for the RECL specifier in an OPEN statement that connects a file for unformatted direct access. If you have already specified a value for RECL, you can check LEN to verify that A and B are less than or equal to the record length you specified.

The following shows another example:

```
!    This program prompts for the name of a data file.
!    The INQUIRE statement then determines whether
!    the file exists. If it does not, the program
!    prompts for another file name.

      CHARACTER*12 fname
      LOGICAL exists

!     Get the name of a file:
100   WRITE (*, '(1X, A\)') 'Enter the file name: '
      READ (*, '(A)') fname

!     INQUIRE about file's existence:
      INQUIRE (FILE = fname, EXIST = exists)

      IF (.NOT. exists) THEN
        WRITE (*,'(2A/)') ' >> Cannot find file ', fname
        GOTO 100
      END IF
      END
```

See Also

OPEN statement
UNIT control specifier
ERR control specifier
ID control specifier
IOMSG control specifier
IOSTAT control specifier
RECL specifier in OPEN statements
FILE specifier in OPEN statements
DEFAULTFILE specifier in OPEN statements
assume:minus0 compiler option

INSERTMENUQQ (W*S)

QuickWin Function: *Inserts a menu item into a QuickWin menu and registers its callback routine.*

Module

USE IFQWIN

Syntax

```
result = INSERTMENUQQ (menuID,itemID,flag,text,routine)
```

menuID (Input) INTEGER(4). Identifies the menu in which the item is inserted, starting with 1 as the leftmost menu.

itemID (Input) INTEGER(4). Identifies the position in the menu where the item is inserted, starting with 0 as the top menu item.

flag (Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see Results section below). The following constants are available:

- $MENUGRAYED - Disables and grays out the menu item.
- $MENUDISABLED - Disables but does not gray out the menu item.
- $MENUENABLED - Enables the menu item.
- $MENUSEPARATOR - Draws a separator bar.
- $MENUCHECKED - Puts a check by the menu item.
- $MENUUNCHECKED - Removes the check by the menu item.

text (Input) Character*(*). Menu item name. Must be a null-terminated C string, for example, words of text'C.

routine (Input) EXERNAL. Callback subroutine that is called if the menu item is selected. You can assign the following predefined routines to menus:

- WINPRINT - Prints the program.
- WINSAVE - Saves the program.
- WINEXIT - Terminates the program.
- WINSELECTTEXT - Selects text from the current window.
- WINSELECTGRAPHICS - Selects graphics from the current window.
- WINSELECTALL - Selects the entire contents of the current window.
- WININPUT - Brings to the top the child window requesting input and makes it the current window.
- WINCOPY - Copies the selected text and/or graphics from current window to the Clipboard.
- WINPASTE - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.
- WINCLEARPASTE - Clears the paste buffer.
- WINSIZETOFIT - Sizes output to fit window.
- WINFULLSCREEN - Displays output in full screen.
- WINSTATE - Toggles between pause and resume states of text output.
- WINCASCADE - Cascades active windows.
- WINTILE - Tiles active windows.
- WINARRANGE - Arranges icons.
- WINSTATUS - Enables a status bar.
- WININDEX - Displays the index for QuickWin help.
- WINUSING - Displays information on how to use Help.
- WINABOUT - Displays information about the current QuickWin application.
- NUL - No callback routine.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE.

Menus and menu items must be defined in order from left to right and top to bottom. For example, INSERTMENUQQ fails if you try to insert menu item 7 when 5 and 6 are not defined yet. For a top-level menu item, the callback routine is ignored if there are subitems under it.

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to INSERTMENUQQ as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY(ALT+Rin the example) as an alternative to selecting the item with the mouse.

Example

```
USE IFQWIN
LOGICAL(4)     :: status
CHARACTER (80) :: text_input

! insert new item into Menu 5 (Window)
status= INSERTMENUQQ(5, 5, $MENUCHECKED, 'New Item:Status 5_5'C, WINSTATUS)
! insert new menu in position 2
status= INSERTMENUQQ(2, 0, $MENUENABLED, 'New Menu:2_0'C, NUL)
! insert new menu in position 3, and mark it disabled & grayed
status= INSERTMENUQQ(3, 0, IOR($MENUGRAYED,$MENUDISABLED), 'New Disabled Menu:3_0'C, NUL)
! insert new item under the new menu 2 in position 1
status= INSERTMENUQQ(2, 1, $MENUENABLED, 'New Item:Print'C, WINPRINT)
! insert separator bar under the new menu 2 in position 2
status= INSERTMENUQQ(2, 2, $MENUSEPARATOR, 'New Separator'C, NUL)
! insert new item under the new menu 2 in position 3
status= INSERTMENUQQ(2, 3, IOR($MENUCHECKED,$MENUENABLED), 'New Item:Select Text'C, &
                     WINSELECTTEXT)
! insert new item under the new menu 2 in position 5
status= INSERTMENUQQ(2, 4, $MENUENABLED, 'New Item:Copy Selected Text'C, WINCOPY)
! insert new item under the new menu 2 in position 6
status= INSERTMENUQQ(2, 5, $MENUENABLED, 'New Item:Paste Selected Text'C, WINPASTE)

write(*,'("Enter (or cut and paste) a text value: ")',advance='no')
read (*,"(A)") text_input
write(*,'("You just entered: ", A1, A, A1)') "'",trim(text_input),"'"

END
```

See Also

APPENDMENUQQ
DELETEMENUQQ
MODIFYMENUFLAGSQQ
MODIFYMENUROUTINEQQ
MODIFYMENUSTRINGQQ

INT

Elemental Intrinsic Function (Generic): *Converts a value to integer type.*

Syntax

```
result = INT (a[,kind])
```

a	(Input) Must be of type integer, real, or complex, or a binary, octal, or hexadecimal literal constant.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

The result value depends on the type and absolute value of *a* as follows:

- If *a* is of type integer, INT(*a*) = *a*.
- If *a* is of type real and | *a* | < 1, INT(*a*) has the value zero.

  If *a* is of type real and | *a* | >=1, INT(*a*) is the integer whose magnitude is the largest integer that does not exceed the magnitude of *a* and whose sign is the same as the sign of *a*.
- If *a* is of type complex, INT(*a*) is the value obtained by applying the preceding rules (for a real argument) to the real part of *a*.
- If *a* is a binary, octal, or hexadecimal literal constant, the value of the result is the value whose bit sequence according to the model in Bit Model is the same as that of *a* as modified by padding or truncation according to the following:
  - If the length of the sequence of bits specified by *a* is less than the size in bits of a scalar variable of the same type and kind type parameter as the result, the binary, octal, or hexadecimal literal constant is treated as if it were extended to a length equal to the size in bits of the result by padding on the left with zero bits.
  - If the length of the sequence of bits specified by *a* is greater than the size in bits of a scalar variable of the same type and kind type parameter as the result, the binary, octal, or hexadecimal literal constant is treated as if it were truncated from the left to a length equal to the size in bits of the result.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1), INTEGER(2), INTEGER(4)	INTEGER(4)
	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8)	INTEGER(8)
IJINT	INTEGER(4)	INTEGER(2)
IIFIX[2]	REAL(4)	INTEGER(2)
IINT	REAL(4)	INTEGER(2)

Specific Name [1]	Argument Type	Result Type
IFIX [3, 4]	REAL(4)	INTEGER(4)
JFIX	INTEGER(1), INTEGER(2), INTEGER(4),	INTEGER(4)
	INTEGER(8), REAL(4), REAL(8), REAL(16),	
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	
INT [5, 6, 7]	REAL(4)	INTEGER(4)
KIFIX	REAL(4)	INTEGER(8)
KINT	REAL(4)	INTEGER(8)
IIDINT	REAL(8)	INTEGER(2)
IDINT [6, 8]	REAL(8)	INTEGER(4)
KIDINT	REAL(8)	INTEGER(8)
IIQINT	REAL(16)	INTEGER(2)
IQINT[6, 9]	REAL(16)	INTEGER(4)
KIQINT	REAL(16)	INTEGER(8)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(2)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(8)
INT1[10]	INTEGER(1), INTEGER(2), INTEGER(4),	INTEGER(1)
	INTEGER(8), REAL(4), REAL(8), REAL(16),	
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	
INT2[10]	INTEGER(1), INTEGER(2), INTEGER(4),	INTEGER(2)
	INTEGER(8), REAL(4), REAL(8), REAL(16),	
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	
INT4[10]	INTEGER(1), INTEGER(2), INTEGER(4),	INTEGER(4)
	INTEGER(8), REAL(4), REAL(8), REAL(16),	

Specific Name [1]	Argument Type	Result Type
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	
INT8[10]	INTEGER(1), INTEGER(2), INTEGER(4),	INTEGER(8)
	INTEGER(8), REAL(4), REAL(8), REAL(16),	
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	

[1]These specific functions cannot be passed as actual arguments.

[2]This function can also be specified as HFIX.

[3]The setting of compiler options specifying integer size or real size can affect IFIX.

[4]For compatibility with older versions of Fortran, IFIX is treated as a generic function.

[5]Or JINT.

[6]The setting of compiler options specifying integer size can affect INT, IDINT, and IQINT.

[7]Or JIFIX.

[8]Or JIDINT. For compatibility with older versions of Fortran, IDINT can also be specified as a generic function.

[9]Or JIQINT. For compatibility with older versions of Fortran, IQINT can also be specified as a generic function.

[10]For compatibility, these functions can also be specified as generic functions.

If the argument is a binary, octal, or hexadecimal constant, the result is affected by the `assume old-boz` option. The default option setting, `noold-boz`, treats the argument as a bit string that represents a value of the data type of the intrinsic, that is, the bits are not converted. If setting `old-boz` is specified, the argument is treated as a signed integer and the bits are converted.

Example

INT (-4.2) has the value -4.

INT (7.8) has the value 7.

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
NINT
AINT
ANINT
REAL
DBLE
SNGL

INTC

Portability Function: *Converts an INTEGER(4) argument to INTEGER(2) type.*

Module

USE IFPORT

Syntax

```
result = INTC (i)
```

i (Input) INTEGER(4). A value or expression.

Results

The result type is INTEGER(2). The result is the value of *i* with type INTEGER(2). Overflow is ignored.

INT_PTR_KIND

Inquiry Intrinsic Function (Specific): *Returns the INTEGER KIND that will hold an address. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.*

Syntax

```
result = INT_PTR_KIND( )
```

Results

The result type is default integer. The result is a scalar with the value equal to the value of the kind parameter of the integer data type that can represent an address on the targeted platform.

The result value is 4 on IA-32 target architecture; 8 on Intel® 64 architecture.

Example

```
REAL A(100)
POINTER (P, A)
INTEGER (KIND=INT_PTR_KIND()) SAVE_P
P = MALLOC (400)
SAVE_P = P
```

INTEGER Statement

Statement: *Specifies the INTEGER data type.*

Syntax

```
INTEGER
INTEGER([KIND=] n)
INTEGER*n
```

n Is kind 1, 2, 4, or 8.

If a kind parameter is specified, the integer has the kind specified. If a kind parameter is not specified, integer constants are interpreted as follows:

- If the integer constant is within the default integer kind range, the kind is default integer.
- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind which holds the constant.

The default kind can also be changed by using the INTEGER directive or compiler options specifying integer size.

Example

```
! Entity-oriented declarations:
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER (2) :: k=4
INTEGER (2), PARAMETER :: limit=12

! Attribute-oriented declarations:
INTEGER days, hours
INTEGER (2):: k=4, limit
DIMENSION days(:), hours(:)
POINTER days, hours
PARAMETER (limit=12)
```

See Also

INTEGER Directive
Integer Data Types
Integer Constants

INTEGER Directive

General Compiler Directive: *Specifies the default integer kind.*

Syntax

```
!DIR$ INTEGER:{ 2 | 4 | 8 }
```

The INTEGER directive specifies a size of 2 (KIND=2), 4 (KIND=4), or 8 (KIND=8) bytes for default integer numbers.

When the INTEGER directive is in effect, all default integer variables are of the kind specified. Only numbers specified or implied as INTEGER without KIND are affected.

The INTEGER directive can only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. INTEGER cannot appear at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

The default logical kind is the same as the default integer kind. So, when you change the default integer kind you also change the default logical kind.

Example

```
INTEGER i            ! a 4-byte integer
WRITE(*,*) KIND(i)
CALL INTEGER2( )
WRITE(*,*) KIND(i)   ! still a 4-byte integer
                     !   not affected by setting in subroutine
END

SUBROUTINE INTEGER2( )
  !DIR$ INTEGER:2
  INTEGER j          ! a 2-byte integer
  WRITE(*,*) KIND(j)
END SUBROUTINE
```

See Also
INTEGER
REAL Directive
General Compiler Directives
Syntax Rules for Compiler Directives
Integer Data Types
Integer Constants

INTEGERTORGB (W*S)

QuickWin Subroutine: *Converts an RGB color value into its red, green, and blue components.*

Module

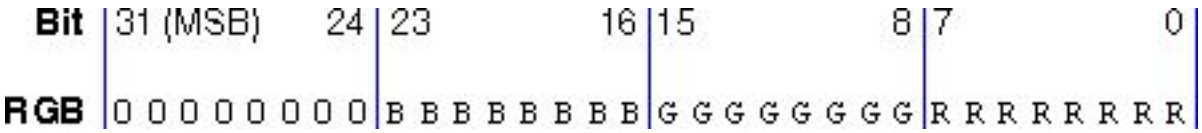
USE IFQWIN

Syntax

```
CALL INTEGERTORGB (rgb,red,green,blue)
```

rgb	(Input) INTEGER(4). RGB color value whose red, green, and blue components are to be returned.
red	(Output) INTEGER(4). Intensity of the red component of the RGB color value.
green	(Output) INTEGER(4). Intensity of the green component of the RGB color value.
blue	(Output) INTEGER(4). Intensity of the blue component of the RGB color value.

INTEGERTORGB separates the four-byte RGB color value into the three components as follows:



Example

```
! build as a QuickWin App.
USE IFQWIN
INTEGER(4) r, g, b
CALL INTEGERTORGB(2456, r, g, b)
write(*,*) r, g, b
END
```

See Also
RGBTOINTEGER
GETCOLORRGB
GETBKCOLORRGB
GETPIXELRRGB
GETPIXELSRGB
GETTEXTCOLORRGB

INTENT

Statement and Attribute: *Specifies the intended use of one or more dummy arguments.*

Syntax

The INTENT attribute can be specified in a type declaration statement or an INTENT statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] INTENT (*intent-spec*) [, *att-ls*] :: *d-arg*[, *d-arg*]...

Statement:

INTENT (*intent-spec*) [::] *d-arg*[, *d-arg*] ...

type
Is a data type specifier.

att-ls
Is an optional list of attribute specifiers.

intent-spec
Is one of the following specifiers:

IN
Specifies that the dummy argument will be used only to provide data to the procedure. The dummy argument must not be redefined (or become undefined) during execution of the procedure.

OUT
Specifies that the dummy argument will be used to pass data from the procedure back to the calling program. The dummy argument is undefined on entry, although it may have subcomponents that are initialized by default. An undefined argument must be defined before it is referenced in the procedure.

Any associated actual argument must be definable.

INOUT
Specifies that the dummy argument can both provide data to the procedure and return data to the calling program.

Any associated actual argument must be definable.

d-arg
Is the name of a dummy argument or dummy pointer. It cannot be a dummy procedure.

Description

The INTENT statement can only appear in the specification part of a subprogram or interface body.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument.

If a function specifies a defined operator, the dummy arguments must have intent IN.

If a subroutine specifies defined assignment, the first argument must have intent OUT or INOUT, and the second argument must have intent IN or the VALUE attribute, or both IN and the VALUE attribute.

An entity with the INTENT (OUT) attribute must not be an allocatable coarray or have a subobject that is an allocatable coarray. It must not be of, or have a subcomponent of, type EVENT_TYPE or type LOCK_TYPE from the ISO_FORTRAN_ENV module.

A non-pointer dummy argument with intent IN (or a subobject of such a dummy argument) must *not* appear as any of the following:

- A DO variable
- The variable of an assignment statement
- The *pointer-object* of a pointer assignment statement
- An *object* or STAT variable in an ALLOCATE or DEALLOCATE statement
- An input item in a READ statement
- A variable name in a NAMELIST statement if the namelist group name appears in a NML specifier in a READ statement
- An internal file unit in a WRITE statement
- A definable variable in an INQUIRE statement
- An IOSTAT or SIZE specifier in an I/O statement
- An actual argument in a reference to a procedure with an explicit interface if the associated dummy argument has intent OUT or INOUT

INTENT on a pointer dummy argument refers to the pointer association status of the pointer and has no effect on the value of the target of the pointer.

A pointer dummy argument with intent IN (or a subobject of such a pointer argument) must *not* appear as any of the following:

- A *pointer-object* in a NULLIFY statement
- A *pointer-object* in a pointer assignment statement
- An *object* in an ALLOCATE or DEALLOCATE statement
- An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

A pointer dummy argument with INTENT(IN) can be argument associated with a non-pointer actual argument with the TARGET attribute. During the execution of the procedure, it is pointer associated with the actual argument.

If an actual argument is an array section with a vector subscript, it cannot be associated with a dummy array that is defined or redefined (has intent OUT or INOUT).

On entry to a routine, given an INTENT(OUT) dummy argument:

- If it is a pointer, it should be deallocated.
- If it is an allocatable, all of its allocatable subcomponents should be deallocated, and then it should also be deallocated.
- If it is a non-pointer, non-allocatable, all its allocatable subcomponents should be deallocated, and then default initialization should be applied, as specified by the program.

Example

The following example shows type declaration statements specifying the INTENT attribute:

```
SUBROUTINE TEST(I, J)
  INTEGER, INTENT(IN) :: I
  INTEGER, INTENT(OUT), DIMENSION(I) :: J
```

The following are examples of the INTENT statement:

```
SUBROUTINE TEST(A, B, X)
   INTENT(INOUT) :: A, B
   ...
SUBROUTINE CHANGE(FROM, TO)
   USE EMPLOYEE_MODULE
```

```
  TYPE(EMPLOYEE) FROM, TO
  INTENT(IN) FROM
  INTENT(OUT) TO
  ...
```

The following shows another example:

```
SUBROUTINE AVERAGE(value,data1, cube_ave)
  TYPE DATA
          INTEGER count
          REAL avg
        END TYPE
        TYPE(DATA) data1
        REAL tmp
  ! value cannot be changed, while cube_ave must be defined
  ! before it can be used. Data1 is defined when the procedure is
  ! invoked, and becomes redefined in the subroutine.
  INTENT(IN)::value; INTENT(OUT)::cube_ave
  INTENT(INOUT)::data1
  ! count number of times AVERAGE has been called on the data set
  ! being passed.
  tmp = data1%count*data1%avg + value
  data1%count = data1%count + 1
  data1%avg = tmp/data1%count
  cube_ave = data1%avg**3
END SUBROUTINE
```

See Also

Argument Association
Type Declarations
ISO_FORTRAN_ENV
Compatible attributes

INTERFACE

Statement: *Defines an explicit interface for an external or dummy procedure. It can also be used to define a generic name for procedures, a new operator for functions, and a new form of assignment for subroutines.*

Syntax

```
INTERFACE [generic-spec]
   [interface-body]...
   [[MODULE]PROCEDURE [::]name-list]...
END INTERFACE [generic-spec]
```

generic-spec

(Optional) Is one of the following:

- A generic name

  For information on generic names, see Defining Generic Names for Procedures.
- OPERATOR (*op*)

Defines a generic operator (*op*). It can be a defined unary, defined binary, or extended intrinsic operator. For information on defined operators, see Defining Generic Operators.

- ASSIGNMENT (=)

  Defines generic assignment. For information on defined assignment, see Defining Generic Assignment.

interface-body

Is one or more function or subroutine subprograms or a procedure pointer. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

The subprogram must *not* contain a statement function or a DATA, ENTRY, or FORMAT statement; an entry name can be used as a procedure name.

The subprogram can contain a USE statement. It can also contain the prefix MODULE before FUNCTION or SUBROUTINE to indicate a separate module procedure.

name-list

Is the name of one or more nonintrinsic procedures that are accessible in the host. The MODULE keyword is only allowed if the interface block specifies a *generic-spec* and has a host that is a module, or accesses a module by use association.

The characteristics of module procedures or internal procedures are not given in interface blocks, but are assumed from the module subprogram definitions or the USE associated interfaces.

Description

Interface blocks can appear in the specification part of the program unit that invokes the external or dummy procedure.

A *generic-spec* can only appear in the END INTERFACE statement if one appears in the INTERFACE statement; they must be identical.

The characteristics specified for the external or dummy procedure must be consistent with those specified in the procedure's definition.

An interface block must not appear in a block data program unit.

An interface block comprises its own scoping unit, and does not inherit anything from its host through host association.

Internal, module, and intrinsic procedures are all considered to have explicit interfaces. External procedures have implicit interfaces by default; when you specify an interface block for them, their interface becomes explicit. A procedure must not have more than one explicit interface in a given scoping unit. This means that you cannot include internal, module, or intrinsic procedures in an interface block, unless you want to define a generic name for them.

The function or subroutine named in the interface-body cannot have the same name as a keyword that specifies an intrinsic type.

A interface block containing *generic-spec* specifies a generic interface for the following procedures:

- The procedures within the interface block

  Any generic name, defined operator, or equals symbol that appears is a generic identifier for all the procedures in the interface block. For the rules on how any two procedures with the same generic identifier must differ, see Unambiguous Generic Procedure References.
- The module procedures listed in the MODULE PROCEDURE statement

The module procedures must be accessible by a USE statement.

To make an interface block available to multiple program units (through a USE statement), place the interface block in a module.

The following rules apply to interface blocks containing pure procedures:

- The interface specification of a pure procedure must declare the INTENT of all dummy arguments except pointer and procedure arguments.
- A procedure that is declared pure in its definition can also be declared pure in an interface block. However, if it is not declared pure in its definition, it must not be declared pure in an interface block.

Example

The following example shows a simple procedure interface block with no generic specification:

```
SUBROUTINE SUB_B (B, FB)
  REAL B
  ...
  INTERFACE
    FUNCTION FB (GN)
      REAL FB, GN
    END FUNCTION
  END INTERFACE
```

The following shows another example:

```
!An interface to an external subroutine SUB1 with header:
!SUBROUTINE SUB1(I1,I2,R1,R2)
!INTEGER I1,I2
!REAL R1,R2
INTERFACE
  SUBROUTINE SUB1(int1,int2,real1,real2)
    INTEGER int1,int2
    REAL real1,real2
  END SUBROUTINE SUB1
END INTERFACE
INTEGER int
...
```

See Also

ABSTRACT INTERFACE
CALL
PROCEDURE
FUNCTION
MODULE
SUBMODULE
MODULE PROCEDURE
SUBROUTINE
PURE
Procedure Interfaces
Procedures that Require Explicit Interfaces
Use and Host Association

INTERFACE TO

Statement: *Identifies a subprogram and its actual arguments before it is referenced or called.*

Syntax

```
INTERFACE TO subprogram-stmt
   [formal-declarations]
END
```

subprogram-stmt	Is a function or subroutine declaration statement.
formal-declarations	(Optional) Are type declaration statements (including optional attributes) for the arguments.

The INTERFACE TO block defines an explicit interface, but it contains specifications for only the procedure declared in the INTERFACE TO statement. The explicit interface is defined only in the program unit that contains the INTERFACE TO statement.

The recommended method for defining explicit interfaces is to use an INTERFACE block.

Example

Consider that a C function that has the following prototype:

```
extern void Foo (int i);
```

The following INTERFACE TO block declares the Fortran call to this function:

```
INTERFACE TO SUBROUTINE Foo [C.ALIAS: '_Foo'] (I)
  INTEGER*4 I
END
```

See Also

INTERFACE

INTRINSIC

Statement and Attribute: *Allows the specific name of an intrinsic procedure to be used as an actual argument.*

Syntax

The INTRINSIC attribute can be specified in a type declaration statement or an INTRINSIC statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] INTRINSIC [, *att-ls*] :: *in-pro*[, *in-pro*]...

Statement:

INTRINSIC [::] *in-pro*[, *in-pro*] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
in-pro	Is the name of an intrinsic procedure.

Description

In a type declaration statement, only *functions* can be declared INTRINSIC. However, you can use the INTRINSIC *statement* to declare subroutines, as well as functions, to be intrinsic.

The name declared INTRINSIC is assumed to be the name of an intrinsic procedure. If a generic intrinsic function name is given the INTRINSIC attribute, the name retains its generic properties.

Some specific intrinsic function names cannot be used as actual arguments. For more information, see table Specific Functions Not Allowed as Actual Arguments in Intrinsic Procedures.

Example

The following example shows a type declaration statement specifying the INTRINSIC attribute:

```
PROGRAM EXAMPLE
...
REAL(8), INTRINSIC :: DACOS
...
CALL TEST(X, DACOS)     ! Intrinsic function DACOS is an actual argument
```

The following example shows an INTRINSIC statement:

Main Program	Subprogram
EXTERNAL CTN	SUBROUTINE TRIG(X,F,Y)
INTRINSIC SIN, COS	Y = F(X)
. . .	RETURN
	END
CALL TRIG(ANGLE,SIN,SINE)	
. . .	FUNCTION CTN(X)
	CTN = COS(X)/SIN(X)
CALL TRIG(ANGLE,COS,COSINE)	RETURN
. . .	END
CALL TRIG(ANGLE,CTN,COTANGENT)	

Note that when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the Standard Fortran library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

The following shows another example:

```
    INTRINSIC SIN, COS
    REAL X, Y, R
!   SIN and COS are arguments to Calc2:
    R = Calc2 (SIN, COS)
```

See Also

References to Generic Procedures
Type Declarations
Compatible attributes

INUM

Elemental Intrinsic Function (Specific): *Converts a character string to an INTEGER(2) value. This function cannot be passed as an actual argument.*

Syntax

```
result = INUM (i)
```

i (Input) Must be of type character.

Results

The result type is INTEGER(2). The result value is the INTEGER(2) value represented by the character string *i*.

If the argument contains characters that are illegal in an integer value, an error is signaled and execution stops.

Example

INUM ("451") has the value 451 of type INTEGER(2).

IOR

Elemental Intrinsic Function (Generic): *Performs an inclusive OR on corresponding bits. This function can also be specified as OR.*

Syntax

```
result = IOR (i,j)
```

i (Input) Must be of type integer, logical (which is treated as an integer), or a binary, octal, or hexadecimal literal constant.

j (Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.

If both *i* and *j* are of type integer, they must have the same kind type parameter. If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with its sign bit on the left and the larger kind is used for the operation and the result. *i* and *j* must not both be binary, octal, or hexadecimal literal constants.

Results

The result is the same as *i* if *i* is of type integer; otherwise, the result is the same as *j*. If either *i* or *j* is a binary, octal, or hexadecimal literal constant, it is first converted as if by the intrinsic function INT to type integer with the kind type parameter of the other.

The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

```
  i   j   IOR (i, j)
  1   1       1
  1   0       1
  0   1       1
  0   0       0
```

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BIOR	INTEGER(1)	INTEGER(1)
IIOR[1]	INTEGER(2)	INTEGER(2)
JIOR	INTEGER(4)	INTEGER(4)
KIOR	INTEGER(8)	INTEGER(8)

Specific Name	Argument Type	Result Type
[1]Or HIOR.		

Example

IOR (1, 4) has the value 5.

IOR (1, 2) has the value 3.

The following shows another example:

```
INTEGER result
result = IOR(240, 90) ! returns 250
```

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
IAND
IEOR
NOT
IALL
IANY
IPARITY

IPARITY

Transformational Intrinsic Function (Generic):
Returns the result of a bitwise exclusive OR operation.

Syntax

`result = IPARITY (array, dim [, mask])`

`result = IPARITY (array [, mask])`

array	(Input) Must be an array of type integer.
dim	(Input) Must be a scalar integer with a value in the range 1 <= *dim* <= *n*, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result has the same type and kind parameters as *array*. It is scalar if *dim* does not appear; otherwise, the result has rank *n* - 1 and shape [d_1, d_2, ..., d_{dim-1}, d_{dim+1} , ..., d_n] where [d_1, d_2,..., d_n] is the shape of *array*.

The result of IPARITY (*array*) has a value equal to the bitwise exclusive OR of all the elements of *array*. If *array* has size zero, the result value is equal to zero.

The result of IPARITY (*array*, MASK=*mask*) has a value equal to IPARITY (PACK (*array*, *mask*)).

The result of IPARITY (*array*, DIM=*dim* [, MASK=*mask*]) has a value equal to that of IPARITY (*array* [, MASK=*mask*]) if *array* has rank one. Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1} , ..., s_n) of the result is equal to IPARITY (*array* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n) [, MASK = *mask* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n)]).

Example

IPARITY ([14, 13, 8]) has the value 11. IPARITY ([14, 13, 8], MASK=[.true., .false., .true]) has the value 6.

See Also

IANY
IALL

IPXFARGC

POSIX Function: *Returns the index of the last command-line argument.*

Module

USE IFPOSIX

Syntax

```
result = IPXFARGC( )
```

Results

The result type is INTEGER(4). The result value is the number of command-line arguments, excluding the command name, in the command used to invoke the executing program. A return value of zero indicates there are no command-line arguments other than the command name itself.

See Also

PXFGETARG

IPXFCONST

POSIX Function: *Returns the value associated with a constant defined in the C POSIX standard.*

Module

USE IFPOSIX

Syntax

```
result = IPXFCONST (constname)
```

constname (Input) Character. The name of a C POSIX standard constant.

Results

The result type is INTEGER(4). If *constname* corresponds to a defined constant in the C POSIX standard, the result value is the integer that is associated with the constant. Otherwise, the result value is -1.

See Also

PXFGETARG
PXFCONST

IPXFLENTRIM

POSIX Function: *Returns the index of the last non-blank character in an input string.*

Module

USE IFPOSIX

Syntax

`result = IPXFLENTRIM (string)`

string	(Input) Character. A character string.

Results

The result type is INTEGER(4). The result value is the index of the last non-blank character in the input argument *string*, or zero if all characters in *string* are blank characters.

IPXFWEXITSTATUS (L*X, M*X)

POSIX Function: *Returns the exit code of a child process.*

Module

USE IFPOSIX

Syntax

`result = IPXFWEXITSTATUS (istat)`

istat	(Input) INTEGER(4). The value of output argument *istat* from PXFWAIT or PXFWAITPID.

Results

The result type is INTEGER(4). The result is the low-order eight bits of the output argument of PXFWAIT or PXFWAITPID.

The IPXFWEXITSTATUS function should only be used if PXFWIFEXITED returns TRUE.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
```

```
      print *, " The child exited normally"
      istat_ret = IPXFWEXITSTATUS(ISTAT)
      print 10," The low byte of the child exit code is", istat_ret
    end if
  end if
10 FORMAT (A,Z)
end program
```

See Also

PXFWAIT
PXFWAITPID
PXFWIFEXITED

IPXFWSTOPSIG (L*X, M*X)

POSIX Function: *Returns the number of the signal that caused a child process to stop.*

Module

USE IFPOSIX

Syntax

`result = IPXFWSTOPSIG (istat)`

istat	(Input) INTEGER(4). The value of output argument *istat* from PXFWAIT or PXFWAITPID.

Results

The result type is INTEGER(4). The result is the number of the signal that caused the child process to stop.

The IPXFWSTOPSIG function should only be used if PXFWIFSTOPPED returns TRUE.

See Also

PXFWAIT
PXFWAITPID
PXFWIFSTOPPED

IPXFWTERMSIG (L*X, M*X)

POSIX Function: *Returns the number of the signal that caused a child process to terminate.*

Module

USE IFPOSIX

Syntax

`result = IPXFWTERMSIG (istat)`

istat	(Input) INTEGER(4). The value of output argument *istat* from PXFWAIT or PXFWAITPID.

Results

The result type is INTEGER(4). The result is the number of the signal that caused the child process to terminate.

The IPXFWTERMSIG function should only be used if PXFWIFSIGNALED returns TRUE.

See Also

PXFWAIT
PXFWAITPID
PXFWIFSIGNALED

IRAND, IRANDM

Portability Functions: *Return random numbers in the range 0 through (2**31)-1, or 0 through (2**15)-1 if called without an argument.*

Module

USE IFPORT

Syntax

```
result = IRAND ([iflag])
result = IRANDM ([iflag])
```

iflag	(Input) INTEGER(4). Optional for IRAND. Controls the way the returned random number is chosen. If *iflag* is omitted, it is assumed to be 0, and the return range is 0 through (2**15)-1 (inclusive).

Results

The result type is INTEGER(4). If *iflag* is 1, the generator is restarted and the first random value is returned. If *iflag* is 0, the next random number in the sequence is returned. If *iflag* is neither zero nor 1, it is used as a new seed for the random number generator, and the functions return the first new random value.

IRAND and IRANDM are equivalent and return the same random numbers. Both functions are included to ensure portability of existing code that references one or both of them.

You can use SRAND to restart the pseudorandom number generator used by these functions.

Example

```
USE IFPORT
INTEGER(4) istat, flag_value, r_nums(20)
flag_value=1
r_nums(1) = IRAND (flag_value)
flag_value=0
do istat=2,20
   r_nums(istat) = irand(flag_value)
end do
```

See Also

RANDOM_NUMBER
RANDOM_SEED
SRAND

IRANGET

Portability Subroutine: *Returns the current seed.*

Module

USE IFPORT

Syntax

```
CALL IRANGET (seed)
```

seed	(Output) INTEGER(4). The current seed value.

See Also

IRANSET

IRANSET

Portability Subroutine: *Sets the seed for the random number generator.*

Module

USE IFPORT

Syntax

```
CALL IRANSET (seed)
```

seed	(Input) INTEGER(4). The reset value for the seed.

See Also

IRANGET

IS_CONTIGUOUS

Inquiry Intrinsic Function (Generic): *Tests the contiguity of an array.*

Syntax

```
result = IS_CONTIGUOUS (array)
```

array	(Input) Is an array; it can be of any data type. If it is a pointer, it must be associated.

Results

The result is default logical scalar. The result has the value .TRUE. if array is contiguous; otherwise, .FALSE..

Example

After the pointer assignment MY_P => TARGET (2:20:4), IS_CONTIGUOUS (MY_P) has the value .FALSE..

See Also

CONTIGUOUS

IS_IOSTAT_END

Elemental Intrinsic Function (Generic): *Tests for an end-of-file condition.*

Syntax

```
result=IS_IOSTAT_END(i)
```

i	(Input) Must be of type integer.

Results

The result type is default logical. The value of the result is true only if *i* is a value that could be assigned to the scalar integer variable in an IOSTAT= specifier to indicate an end-of-file condition.

Example

```
INTEGER IO_STATUS
…
READ (20, IOSTAT=IO_STATUS) A, B, C
IF (IS_IOSTAT_END (IO_STATUS)) THEN
…                                ! process end of file
ENDIF
…                                ! process data read
```

IS_IOSTAT_EOR

Elemental Intrinsic Function (Generic): *Tests for an end-of-record condition.*

Syntax

`result=IS_IOSTAT_EOR(i)`

i	(Input) Must be of type integer.

Results

The result type is default logical. The value of the result is true only if *i* is a value that could be assigned to the scalar integer variable in an IOSTAT= specifier to indicate an end-of-record condition.

Example

```
INTEGER IO_STATUS
…
READ (30, ADVANCE='YES', IOSTAT=IO_STATUS) A, B, C
IF (IS_IOSTAT_EOR (IO_STATUS)) THEN
…                                ! process end of record
ENDIF
…                                ! process data read
```

ISATTY

Portability Function: *Checks whether a logical unit number is a terminal.*

Module

USE IFPORT

Syntax

`result = ISATTY (lunit)`

iunit	(Input) INTEGER(4). An integer expression corresponding to a Fortran logical unit number. Must be in the range 0 to 100 and must be connected.

Results

The result type is LOGICAL(4). The result is .TRUE. if the specified logical unit is connected to a terminal device; otherwise, .FALSE..

If *lunit* is out of range or is not connected, zero is returned.

ISHA

Elemental Intrinsic Function (Generic):
Arithmetically shifts an integer left or right by a specified number of bits.

Syntax

```
result = ISHA (i,shift)
```

i	(Input) Must be of type integer. This argument is the value to be shifted.
shift	(Input) Must be of type integer. This argument is the direction and distance of shift. Positive shifts are left (toward the most significant bit); negative shifts are right (toward the least significant bit).

Results

The result type and kind are the same as *i*. The result is equal to *i* shifted arithmetically by *shift* bits.

If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. If the shift is to the left, zeros are shifted in on the right. If the shift is to the right, copies of the sign bit (0 for non-negative *i*; 1 for negative *i*) are shifted in on the left.

The kind of integer is important in arithmetic shifting because sign varies among integer representations (see the following example). If you want to shift a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Example

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = -128             ! equal to 10000000
j = -32768           ! equal to 10000000 00000000
res1 = ISHA (i, -4)  ! returns 11111000 = -8
res2 = ISHA (j, -4)  ! returns 11111000 10100000 = -2048
```

See Also

ISHC
ISHL
ISHFT
ISHFTC

ISHC

Elemental Intrinsic Function (Generic): *Rotates an integer left or right by specified number of bits. Bits shifted out one end are shifted in the other end. No bits are lost.*

Syntax

```
result = ISHC (i,shift)
```

i	(Input) Must be of type integer. This argument is the value to be rotated.
shift	(Input) Must be of type integer. This argument is the direction and distance of rotation. Positive rotations are left (toward the most significant bit); negative rotations are right (toward the least significant bit).

Results

The result type and kind are the same as *i*. The result is equal to *i* circularly rotated by *shift* bits.

If *shift* is positive, *i* is rotated left *shift* bits. If *shift* is negative, *i* is rotated right *shift* bits. Bits shifted out one end are shifted in the other. No bits are lost.

The kind of integer is important in circular shifting. With an INTEGER(4) argument, all 32 bits are shifted. If you want to rotate a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Example

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10  ! equal to 00001010
j = 10  ! equal to 00000000 00001010
res1 = ISHC (i, -3)   ! returns 01000001 = 65
res2 = ISHC (j, -3)   ! returns 01000000 00000001 =
                      ! 16385
```

See Also

ISHA
ISHL
ISHFT
ISHFTC

ISHFT

Elemental Intrinsic Function (Generic): *Performs a logical shift.*

Syntax

```
result = ISHFT (i,shift)
```

i	(Input) Must be of type integer.
shift	(Input) Must be of type integer. The absolute value for *shift* must be less than or equal to BIT_SIZE(*i*).

Results

The result type and kind are the same as *i*. The result has the value obtained by shifting the bits of *i* by *shift* positions. If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

ISHFT with a positive *shift* can also be specified as LSHIFT (or LSHFT). ISHFT with a negative *shift* can also be specified as RSHIFT (or RSHFT) with | *shift* |.

For more information on bit functions, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BSHFT	INTEGER(1)	INTEGER(1)
IISHFT[1]	INTEGER(2)	INTEGER(2)
JISHFT	INTEGER(4)	INTEGER(4)
KISHFT	INTEGER(8)	INTEGER(8)

[1]Or HSHFT.

Example

ISHFT (2, 1) has the value 4.

ISHFT (2, -1) has the value 1.

The following shows another example:

```
INTEGER(1) i, res1
INTEGER(2) j, k(3), res2
i = 10 ! equal to 00001010
j = 10 ! equal to 00000000 00001010
res1 = ISHFT (i, 5)  ! returns 01000000 = 64
res2 = ISHFT (j, 5)  ! returns 00000001 01000000 =
                     ! 320
k = ISHFT((/3, 5, 1/), (/1, -1, 0/))  ! returns array
                                      ! /6, 2, 1/
```

See Also

BIT_SIZE
ISHFTC
ISHA
ISHC

ISHFTC

Elemental Intrinsic Function (Generic): *Performs a circular shift of the rightmost bits.*

Syntax

```
result = ISHFTC (i,shift[,size])
```

i (Input) Must be of type integer.

shift	(Input) Must be of type integer. The absolute value of *shift* must be less than or equal to *size*.
size	(Input;optional) Must be of type integer. The value of *size*must be positive and must not exceed BIT_SIZE(*i*). If *size*is omitted, it is assumed to have the value of BIT_SIZE(*i*).

Results

The result type and kind are the same as *i*. The result value is obtained by circular shifting the *size* rightmost bits of *i* by *shift* positions. If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

No bits are lost. Bits in *i* beyond the value specified by *size* are unaffected.

For more information on bit functions, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BSHFTC	INTEGER(1)	INTEGER(1)
IISHFTC[1]	INTEGER(2)	INTEGER(2)
JISHFTC	INTEGER(4)	INTEGER(4)
KISHFTC	INTEGER(8)	INTEGER(8)

[1]Or HSHFTC.

Example

ISHFTC (4, 2, 4) has the value 1.

ISHFTC (3, 1, 3) has the value 6.

The following shows another example:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10  ! equal to 00001010
j = 10  ! equal to 00000000 00001010
res1 = ISHFTC (i, 2, 3)   ! rotates the 3 rightmost
                          ! bits by 2 (left) and
                          ! returns 00001001 = 9
res1 = ISHFTC (i, -2, 3)  ! rotates the 3 rightmost
                          ! bits by -2 (right) and
                          ! returns 00001100 = 12
res2 = ISHFTC (j, 2, 3)   ! rotates the 3 rightmost
                          ! bits by 2 and returns
                          ! 00000000 00001001 = 9
```

See Also

BIT_SIZE
ISHFT
MVBITS

ISHL

Elemental Intrinsic Function (Generic): *Logically shifts an integer left or right by the specified bits. Zeros are shifted in from the opposite end.*

Syntax

```
result = ISHL (i,shift)
```

i	(Input) Must be of type integer. This argument is the value to be shifted.
shift	(Input) Must be of type integer. This argument is the direction and distance of shift. If positive, *i* is shifted left (toward the most significant bit). If negative, *i* is shifted right (toward the least significant bit).

Results

The result type and kind are the same as *i*. The result is equal to *i* logically shifted by *shift* bits. Zeros are shifted in from the opposite end.

Unlike circular or arithmetic shifts, which can shift ones into the number being shifted, logical shifts shift in zeros only, regardless of the direction or size of the shift. The integer kind, however, still determines the end that bits are shifted out of, which can make a difference in the result (see the following example).

Example

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10   ! equal to 00001010
j = 10   ! equal to 00000000 00001010
res1 = ISHL (i, 5)   ! returns 01000000 = 64
res2 = ISHL (j, 5)   ! returns 00000001 01000000 = 320
```

See Also

ISHA
ISHC
ISHFT
ISHFTC

ISNAN

Elemental Intrinsic Function (Generic): *Tests whether IEEE* real (binary32, binary64, and binary128) numbers are Not-a-Number (NaN) values.*

Syntax

```
result = ISNAN (x)
```

x	(Input) Must be of type real.

Results

The result type is default logical. The result is .TRUE. if *x* is an IEEE NaN; otherwise, the result is .FALSE..

Example

```
LOGICAL A
DOUBLE PRECISION B
...
A = ISNAN(B)
```

A is assigned the value .TRUE. if B is an IEEE NaN; otherwise, the value assigned is .FALSE..

ITIME

Portability Subroutine: *Returns the time in numeric form.*

Module

USE IFPORT

Syntax

CALL ITIME (*array*)

array	(Output) INTEGER(4). A rank one array with three elements used to store numeric time data: • *array*(1) - the hour • *array*(2) - the minute • *array*(3) - the second

Example

```
USE IFPORT
 INTEGER(4) time_array(3)
 CALL ITIME (time_array)
 write(*,10) time_array
 10 format (1X,I2,':',I2,':',I2)
 END
```

See Also

DATE_AND_TIME

IVDEP

General Compiler Directive: *Assists the compiler's dependence analysis of iterative DO loops.*

Syntax

!DIR$ IVDEP [: *option*]

option	Is LOOP or BACK. This argument is only available on processors using IA-32 architecture.

The IVDEP directive is an assertion to the compiler's optimizer about the order of memory references inside a DO loop.

IVDEP:LOOP implies no loop-carried dependencies. IVDEP:BACK implies no backward dependencies.

When no *option* is specified, the following occurs:

- The compiler begins dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.

!DIR$ IVDEP with no option can also be spelled !DIR$ INIT_DEP_FWD (INITialize DEPendences ForWarD).

The IVDEP directive is applied to a DO loop in which the user knows that dependences are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is "earlier" than the left-hand side.

The IVDEP directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependences, to choose other execution orders.

Example

In the following example, the IVDEP directive provides more information about the dependences within the loop, which may enable loop transformations to occur:

```
!DIR$ IVDEP
      DO I=1, N
         A(INDARR(I)) = A(INDARR(I)) + B(I)
      END DO
```

In this case, the scalar execution order follows:

1. Retrieve INDARR(I).
2. Use the result from step 1 to retrieve A(INDARR(I)).
3. Retrieve B(I).
4. Add the results from steps 2 and 3.
5. Store the results from step 4 into the location indicated by A(INDARR(I)) from step 1.

IVDEP directs the compiler to initially assume that when steps 1 and 5 access a common memory location, step 1 always accesses the location first because step 1 occurs earlier in the execution sequence. This approach lets the compiler reorder instructions, as long as it chooses an instruction schedule that maintains the relative order of the array references.

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements

J to L

J to L

JABS

Portability Function: *Returns an absolute value.*

Module

USE IFPORT

Syntax

```
result = JABS (i)
```

i (Input) INTEGER(4). A value.

Results

The result type is INTEGER(4). The value of the result is | *i* |.

JDATE

Portability Function: *Returns an 8-character string with the Julian date in the form "yyddd". Three spaces terminate this string.*

Module

USE IFPORT

Syntax

```
result = JDATE( )
```

Results

The result type is character with length 8. The result is the Julian date, in the form YYDDD, followed by three spaces.

The Julian date is a five-digit number whose first two digits are the last two digits of the year, and whose final three digits represent the day of the year (1 for January 1, 366 for December 31 of a leap year, and so on). For example, the Julian date for February 1, 1999 is 99032.

Caution
The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

```
! Sets julian to today's julian date
    USE IFPORT
    CHARACTER*8 julian
    julian = JDATE( )
```

See Also

DATE_AND_TIME

JDATE4

Portability Function: *Returns a 10-character string with the Julian date in the form "yyyyddd". Three spaces terminate this string.*

Module

USE IFPORT

Syntax

```
result = JDATE4( )
```

Results

The result type is character with length 10. The result is the Julian date, in the form YYYYDDD, followed by three spaces.

The Julian date is a seven-digit number whose first four digits are the year, and whose final three represent the day of the year (1 for January 1, 366 for December 31 of a leap year, and so on). For example, the Julian date for February 1, 1999 is 1999032.

See Also

DATE_AND_TIME

JNUM

Elemental Intrinsic Function (Specific): *Converts a character string to an INTEGER(4) value. This function cannot be passed as an actual argument.*

Syntax

```
result = JNUM (i)
```

i	(Input) Must be of type character.

Results

The result type is INTEGER(4). The result value is the integer value represented by the character string *i*.

If the argument contains characters that are illegal in an integer value, an error is signaled and execution stops.

Example

JNUM ("46616") has the value 46616 of type INTEGER(4).

KILL

Portability Function: *Sends a signal to the process given by ID.*

Module

USE IFPORT

Syntax

```
result = KILL (pid,signum)
```

pid	(Input) INTEGER(4). ID of a process to be signaled.
signum	(Input) INTEGER(4). A signal value. For the definition of signal values, see the SIGNALfunction.

Results

The result type is INTEGER(4). The result is zero if the call was successful; otherwise, an error code. Possible error codes are:

- EINVAL: The *signum* is not a valid signal number, or PID is not the same as getpid( ) and *signum* does not equal SIGKILL.
- ESRCH: The given PID could not be found.
- EPERM: The current process does not have permission to send a signal to the process given by PID.

On Windows* systems, arbitrary signals can be sent only to the calling process (where *pid*= getpid( )). Other processes can send only the SIGKILL signal (*signum*= 9), and only if the calling process has permission.

Example

```
USE IFPORT
integer(4) id_number, sig_val, istat
id_number=getpid( )
ISTAT = KILL (id_number, sig_val)
```

See Also

SIGNAL
RAISEQQ
SIGNALQQ

KIND

Inquiry Intrinsic Function (Generic): *Returns the kind parameter of the argument.*

Syntax

`result = KIND (x)`

x	(Input) Can be of any intrinsic type.

Results

The result is a scalar of type default integer. The result has a value equal to the kind type parameter value of *x*.

Example

KIND (0.0) has the kind value of default real type.

KIND (12) has the kind value of default integer type.

The following shows another example:

```
INTEGER i ! a 4-byte integer
WRITE(*,*) KIND(i)
CALL INTEGER2( )
WRITE(*,*) KIND(i)  ! still a 4-byte integer
                    ! not affected by setting in subroutine
END
SUBROUTINE INTEGER2( )
  !DIR$INTEGER:2
  INTEGER j ! a 2-byte integer
  WRITE(*,*) KIND(j)
END SUBROUTINE
```

See Also

SELECTED_INT_KIND
SELECTED_REAL_KIND
CMPLX
INT
REAL
LOGICAL
CHAR
Intrinsic Data Types
Argument Keywords in Intrinsic Procedures

KNUM

Elemental Intrinsic Function (Specific): *Converts a character string to an INTEGER(8) value.*

Syntax

```
result = KNUM (i)
```

i	(Input) Must be of type character.

Results

The result type is INTEGER(8). The result value is the integer value represented by the character string *i*.

If the argument contains characters that are illegal in an integer value, an error is signaled and execution stops.

Example

KNUM ("46616") has the value 46616 of type INTEGER(8).

LASTPRIVATE

Parallel Directive Clause: *Provides a superset of the functionality provided by the PRIVATE clause. It declares one or more variables to be private to an implicit task, and causes the corresponding original variable to be updated after the end of the region.*

Syntax

```
LASTPRIVATE ([CONDITIONAL:] list)
```

CONDITIONAL	Is an optional modifier specifying that the last value assigned to a *list* item can be from the sequentially last iteration of the associated loops or the lexically last section construct.
list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /). If CONDITIONAL is present, *list* items must be scalar of intrinsic type (INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER) and they must have neither the POINTER nor the ALLOCATABLE attribute.

Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause semantics. In addition, once the parallel region is exited, each variable has the value provided by the sequentially last section or loop iteration.

Multiple LASTPRIVATE clauses are allowed on all of the directives that accept them. A *list* item may not appear in more than one LASTPRIVATE clause. LASTPRIVATE (CONDITIONAL:A,B) is equivalent to LASTPRIVATE(CONDITIONAL:A) LASTPRIVATE(CONDITIONAL:B).

LASTPRIVATE (CONDITIONAL:list) can appear in the following directives:

- OMP DISTRIBUTE
- OMP DO
- OMP SECTIONS
- OMP SIMD
- OMP TASKLOOP
- Any and all of the combination directives of the above directives

The OMP PARALLEL directive does not allow the LASTPRIVATE clause.

If the original variable has the POINTER attribute, its update occurs as if by pointer assignment.

If the original variable does not have the POINTER attribute, its update occurs as if by intrinsic assignment.

When a LASTPRIVATE clause without the CONDITIONAL modifier appears in a worksharing or a SIMD directive, the value of each new *list* item from the sequentially last iteration of the associated loops, or the lexically last section, is assigned to the original *list* item.

When the CONDITIONAL modifier is specified, the final value written by the sequentially last iteration or lexically last section that writes to a *list* item, if any, is assigned to the original *list* item.

When the CONDITIONAL modifier is *not* specified, *list* items that are not assigned a value by the sequentially last iteration of the loops, or by the lexically last section, have unspecified values after the construct.

Therefore, variables specified in the *list* of a LASTPRIVATE clause that contains a CONDITIONAL modifier should be variables that are conditionally assigned to in the loop/region/sections. The variables in the *list* of a LASTPRIVATE clause that does *not* contain a CONDITIONAL modifier should be variables that are unconditionally assigned to in the loop or sections.

Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

The original variable becomes defined at the end of the construct if there is an implicit barrier at that point. To avoid race conditions, concurrent reads or updates of the original variable must be synchronized with the update of the original variable that occurs as a result of the LASTPRIVATE clause.

If the LASTPRIVATE clause is used in a construct for which NOWAIT is specified, accesses to the original variable may create a data race. To avoid this, synchronization must be inserted to ensure that the sequentially last iteration or lexically last section construct has stored and flushed that variable.

The following are restrictions for the LASTPRIVATE clause:

- A variable that is part of another variable (as an array or structure element) must not appear in a PRIVATE clause.
- A variable that is private within a parallel region, or that appears in the REDUCTION clause of a PARALLEL construct, must not appear in a LASTPRIVATE clause on a worksharing construct if any of the corresponding worksharing regions ever binds to any of the corresponding parallel regions.
- A variable that appears in a LASTPRIVATE clause must be definable.
- An original variable with the ALLOCATABLE attribute must be in the allocated state at entry to the construct containing the LASTPRIVATE clause. The variable in the sequentially last iteration or lexically last section must be in the allocated state upon exit from that iteration or section with the same bounds as the corresponding original variable.
- Assumed-size arrays must not appear in a PRIVATE clause.

- Variables that appear in NAMELIST statements, in variable format expressions, and in expressions for statement function definitions, must not appear in a PRIVATE clause.
- If a list item appears in both the FIRSTPRIVATE and LASTPRIVATE clauses, the update required for LASTPRIVATE occurs after all of the initializations for FIRSTPRIVATE.

NOTE
If a variable appears in both FIRSTPRIVATE and LASTPRIVATE clauses, the update required for LASTPRIVATE occurs after all initializations for FIRSTPRIVATE.

NOTE
If a variable appears in a LASTPRIVATE clause on a combined construct for which the first construct is TARGET, it is treated as if it had appeared in a MAP clause with a *map-type* of FROM.

Example

Consider the following:

```
!$OMP DO PRIVATE(I) LASTPRIVATE(B)
      DO I = 1, 1000
         B = I
      ENDDO
!$OMP END DO
```

In this case, after the construct is exited, variable B has the value 1000.

Consider the following:

```
!$OMP SECTIONS LASTPRIVATE(B)
!$OMP SECTION
      B = 2
!$OMP SECTION
      B = 4
!$OMP SECTION
      D = 6
!$OMP END SECTIONS
```

In this case the thread that executes the lexically last SECTION updates the original variable B to have a value of 4. However, variable D was not specified in the LASTPRIVATE clause, so it has an undefined value after the construct is exited.

Consider the following example:

```
      P = 0
!$OMP DO PRIVATE(I), FIRSTPRIVATE(P), LASTPRIVATE(CONDITIONAL:P)
      DO I = 1, 1000
         IF (A(I) .EQ. B) THEN
           P = I
           EXIT
         END IF
      ENDDO
!$OMP END DO
```

After the loop, P will be the index of the first element of A to match B; if no match is found, P will be zero.

See Also

PRIVATE Clause
FIRSTPRIVATE clause

LBOUND

Inquiry Intrinsic Function (Generic): *Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.*

Syntax

`result = LBOUND (array [, dim] [, kind])`

array	(Input) Must be an array; it can be assumed-rank. It may be of any data type. It must not be an allocatable array that is not allocated, or a disassociated pointer.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank *array*.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *dim* is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of *array*. Each element in the result corresponds to a dimension of *array*.

If *array* is an array section or an array expression that is not a whole array or array structure component, each element of the result has the value 1.

If *array* is a whole array or array structure component, LBOUND (*array*, *dim*) has a value equal to the lower bound for subscript *dim* of *array*(if *dim* is nonzero or *array* is an assumed-size array of rank *dim*). Otherwise, the corresponding element of the result has the value 1.

If LBOUND is invoked for an assumed-rank object that is associated with a scalar and *dim* is absent, the result is a zero-sized array. LBOUND cannot be invoked for an assumed-rank object that is associated with a scalar if *dim* is present because the rank of a scalar is zero and *dim* must be >= 1.

The setting of compiler options specifying integer size can affect this function.

Example

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
```

LBOUND(ARRAY_A) is (1, 5). LBOUND(ARRAY_A, DIM=2) is 5.

LBOUND(ARRAY_B) is (2, -3). LBOUND(ARRAY_B (5:8, :)) is (1,1) because the arguments are array sections.

The following shows another example:

```
REAL ar1(2:3, 4:5, -1:14), vec1(35)
INTEGER res1(3), res2, res3(1)
res1 = LBOUND (ar1)          ! returns [2, 4, -1]
res2 = LBOUND (ar1, DIM= 3)  ! returns -1
res3 = LBOUND (vec1)         ! returns [1]
END
```

See Also

UBOUND

LCOBOUND

Inquiry Intrinsic Function (Generic): *Returns the lower cobounds of a coarray.*

Syntax

```
result = LCOBOUND (coarray [,dim [, kind])
```

coarray	(Input) Must be a coarray; it can be of any type. It can be a scalar or an array. If it is allocatable, it must be allocated.
dim	(Input; optional) Must be an integer scalar with a value in the range 1 <= *dim* <= *n*, where *n* is the corank of *coarray*. The corresponding actual argument must not be an optional dummy argument.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default integer type. The result is scalar if *dim* is present; otherwise, the result is an array of rank one and size *n*, where*n* is the corank of *coarray*.

The result depends on whether *dim* is specified:

- If *dim* is specified, LCOBOUND (COARRAY, DIM) has a value equal to the lower cobound for cosubscript *dim* of *coarray*.
- If *dim* is not specified, LCOBOUND (COARRAY) has a value whose ith element is equal to LCOBOUND (COARRAY, i), for i = 1, 2,. . . , *n*, where *n* is the corank of *coarray*.

Example

If B is allocated by the statement ALLOCATE (B [2:3, 8:*]), then LCOBOUND (B) is [2, 8] and LCOBOUND (B, DIM=2) is 8.

LCWRQQ

Portability Subroutine: *Sets the value of the floating-point processor control word.*

Module

USE IFPORT

Syntax

```
CALL LCWRQQ (controlword)
```

controlword	(Input) INTEGER(2). Floating-point processor control word.

LCWRQQ performs the same function as the run-time subroutine SETCONTROLFPQQ and is provided for compatibility.

Example

```
USE IFPORT
INTEGER(2) control
CALL SCWRQQ(control) ! get control word
!  Set control word to make processor round up
control = control .AND. (.NOT. FPCW$MCW_RC)  ! Clear
```

```
                                          !  control word with inverse
                                          !  of rounding control mask
control = control .OR. FPCW$UP  ! Set control word
                                !  to round up
CALL LCWRQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

See Also

SETCONTROLFPQQ

LEADZ

Elemental Intrinsic Function (Specific): *Returns the number of leading zero bits in an integer.*

Syntax

`result = LEADZ (i)`

i	(Input) Must be of type integer or logical.

Results

The result type is default integer. The result value is the number of leading zeros in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

Consider the following:

```
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
  TYPE *, LEADZ(TWO**J)  ! Prints 64 down to 23 (leading zeros)
ENDDO
END
```

LEN

Inquiry Intrinsic Function (Generic): *Returns the length of a character expression.*

Syntax

`result = LEN (string [, kind])`

string	(Input) Must be of type character; it can be scalar or array valued. (It can be an array of strings.)
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters in *string*(if it is scalar) or in an element of *string*(if it is array valued).

Specific Name	Argument Type	Result Type
LEN [1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

[1]The setting of compiler options specifying integer size can affect this function.

Example

Consider the following example:

```
  CHARACTER (15) C (50)
  CHARACTER (25) D
```

LEN (C) has the value 15, and LEN (D) has the value 25.

The following shows another example:

```
CHARACTER(11) STR(100)
INTEGER I
I = LEN (STR)  ! returns 11
I = LEN('A phrase with 5 trailing blanks.^^^^^')
               ! returns 37
```

See Also

LEN_TRIM
Declaration Statements for Character Types
Character Data Type

LEN_TRIM

Elemental Intrinsic Function (Generic): *Returns the length of the character argument without counting trailing blank characters.*

Syntax

`result = LEN_TRIM (`*`string`* `[, `*`kind`*`])`

string	(Input) Must be of type character.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters remaining after any trailing blanks in *string* are removed. If the argument contains only blank characters, the result is zero.

The setting of compiler options specifying integer size can affect this function.

Example

LEN_TRIM (' C D ') has the value 4.

LEN_TRIM (' ') has the value 0.

The following shows another example:

```
INTEGER LEN1
LEN1 = LEN_TRIM (' GOOD DAY ') ! returns 9
LEN1 = LEN_TRIM (' ')          ! returns 0
```

See Also

LEN
LNBLNK

LGE

Elemental Intrinsic Function (Generic):
Determines if a string is lexically greater than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel® Fortran, LGE is equivalent to the (.GE.) operator.

Syntax

`result = LGE (string_a,string_b)`

string_a	(Input) Must be of type character.
string_b	(Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if the strings are equal, both strings are of zero length, or if *string_a* follows *string_b* in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LGE [1,2]	CHARACTER	LOGICAL(4)

[1] This specific function cannot be passed as an actual argument.

[2]The setting of compiler options specifying integer size can affect this function.

Example

LGE ('ONE', 'SIX') has the value false.

LGE ('TWO', 'THREE') has the value true.

The following shows another example:

```
LOGICAL L
L = LGE('ABC','ABD')       ! returns .FALSE.
L = LGE ('AB', 'AAAAAAAB') ! returns .TRUE.
```

See Also

LGT
LLE
LLT
ASCII and Key Code Charts

LGT

Elemental Intrinsic Function (Generic): *Determines whether a string is lexically greater than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel® Fortran, LGT is equivalent to the > (.GT.) operator.*

Syntax

`result = LGT (string_a,string_b)`

string_a	(Input) Must be of type character.
string_b	(Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if *string_a* follows *string_b* in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LGT [1,2]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

[2]The setting of compiler options specifying integer size can affect this function.

Example

LGT ('TWO', 'THREE') has the value true.

LGT ('ONE', 'FOUR') has the value true.

The following shows another example:

```
LOGICAL L
L = LGT('ABC', 'ABC')  ! returns .FALSE.
L = LGT('ABC', 'AABC') ! returns .TRUE.
```

See Also

LGE
LLE
LLT
ASCII and Key Code Charts

LINEAR Clause

Parallel Directive Clause: *Specifies that all variables in a list are private to a SIMD lane and that they have a linear relationship within the iteration space of a loop.*

Syntax

It takes one of the following forms:

In Construct Directive !$OMP SIMD:

`LINEAR (var-list[: linear-step])`

In Declarative Directive !$OMP DECLARE SIMD:

`LINEAR (linear-list[: linear-step])`

var-list
Is a comma-separated list of one or more integer variables. Each list item must comply with PRIVATE clause semantics.

For each iteration of a scalar loop, the value of each *var* is incremented by *linear-step*. For each iteration of a vector loop, the value of each *var* is incremented by *linear-step* times the vector length for the loop.

linear-step
Is a compile-time positive, integer, scalar constant expression. If *linear-step* is not specified, it is assumed to be 1.

linear-list
Takes one of the following forms:

- *list*
- *modifier* (*list*)

list
Is a comma-separated list of one or more dummy arguments of the procedure that will be invoked concurrently on each SIMD lane. Each list item must comply with PRIVATE clause semantics.

If *modifier* is not specified or it is VAL or UVAL, each *list* item must be a scalar of type integer. If *modifier* is REF, each *list* item must be a scalar of intrinsic type.

modifier
Is one of the following:

- REF

  Specifies that the storage location of each *list* item on each lane corresponds to an array at the storage location upon entry to the function indexed by the logical number of the lane times *linear-step*.

  The referenced values passed into the routine (which forms a vector for calculations) are located sequentially, like in an array, with the distance between elements equal to step.

  This modifier can be used only for dummy arguments passed by reference.
- VAL

  Specifies that the value of each *list* item on each lane corresponds to the value of the *list* item upon entry to the function plus the logical number of the lane times *linear-step*.

  On each iteration of the loop from which the routine is called, the value of the parameter can be calculated as (value_on_previous_iteration + step).
- UVAL

  Similar to VAL but each invocation uses the same storage location for each SIMD lane. This storage location is updated with the final value of the logically last lane. It differs from VAL as follows:

- For VAL, a vector of addresses (references) is passed to the vector variant of the routine.
- For UVAL, only one address (reference) is passed, which may improve performance.

This modifier can be used only for dummy arguments passed by reference.

If *modifier* is not specified or it is VAL or UVAL, the value of each *list* item on each lane corresponds to the value of the *list* item upon entry to the function plus the logical number of the lane times *linear-step*.

You can only use REF or VAL if the *list* item is a dummy argument without the Fortran Standard VALUE attribute.

If more than one *linear-step* is specified for a *var*, a compile-time error occurs. Multiple LINEAR clauses can be used to specify different values for *linear-step*. The *linear-step* expression must be invariant (must not be changed) during the execution of the associated construct. Multiple LINEAR clauses are merged as a union.

The value corresponding to the sequentially last iteration of the associated loops is assigned to the original *list* item.

A *list* item in a LINEAR clause cannot appear in any other data scope attribute clause, in another LINEAR clause, or more than once in *list*. For a list of data scope attribute clauses, see the first table in Clauses Used in Multiple OpenMP* Fortran Directives.

In the sequentially last iteration, any *list* item with the ALLOCATABLE attribute must have an allocation status of allocated upon exit from that iteration.

If a *list* item is a dummy argument without the Fortran Standard VALUE attribute and the REF modifier is not specified, then a read of the *list* item must be executed before any writes to the *list* item.

If a LINEAR clause appears in a directive that also contains a REDUCTION clause with the INSCAN modifier, only a loop iteration variable of a loop associated with the construct can appear as a list item in the LINEAR clause.

The following cannot appear in a LINEAR clause:

- Variables that have the POINTER attribute
- Cray* pointers

The LINEAR (REF) clause is very important because Fortran passes dummy arguments by reference. By default, the compiler places consecutive addresses in a vector register, which leads to an inefficient gather of the addresses.

In the following example, LINEAR (REF (X)) tells the compiler that the 4 addresses for the dummy argument X are consecutive, so the code only needs to dereference X once and then copy consecutive values to a vector register.

```
subroutine test_linear(x, y)
!$omp declare simd (test_linear) linear(ref(x, y))    ! arguments by reference
    real(8),intent(in)  :: x
    real(8),intent(out) :: y
    y = 1. + sin(x)**3
end subroutine test_linear
…                                ! procedure that calls test_linear
interface                        ! test_linear needs an explicit interface
…
do j = 1,n
     call test_linear(a(j), b(j))    ! loop vectorized via qopenmp-simd
enddo
…
```

Example

Consider the following:

```
! universal but slowest definition; it matches the use of func in loops 1, 2, and 3:
!$OMP DECLARE SIMD
!
! matches the use of func in loop 1:
!$OMP DECLARE SIMD LINEAR(in1) LINEAR(REF(in2)) UNIFORM(mul)
!
! matches the use of func in loops 2 and 3:
!$OMP DECLARE SIMD LINEAR(REF(in2))
!
! matches the use of func in loop 2:
!$OMP DECLARE SIMD LINEAR(REF(in2)) LINEAR(mul)
!
! matches the use of func in loop 3:
!$OMP DECLARE SIMD LINEAR(VAL(in2:2))

function func(in1, in2, mul)
integral   in1
integral   in2
integral   mul
...
integral a, k
integral b(100)
integral c(100)
integral ndx(100)
...
```

The following are the loop examples referenced above.

```
!loop 1
!$OMP SIMD
   DO i=1,100
      c(i) = func(a + i, b(i), mul) ! the value of the 1st parameter is changed linearly,
                                    !   the 2nd parameter reference is changed linearly,
                                    !   the 3rd parameter is not changed
   END DO

!loop 2
!$OMP SIMD
   DO i=1,100
      c(i) = func(b(ndx(i)), b(i), i + 1 ! the value of the 1st parameter is unpredictable,
                                         !   the 2nd reference is changed linearly,
                                         !   the 3rd parameter is changed linearly
   END DO

!loop 3
!$OMP SIMD
   DO i=1,100
      k = i * 2                       ! during vectorization, private variables are
      c(i) = func(b(ndx(i)), k, b(i)) !     transformed into arrays: k -> k_vec(simdlen)
                                      ! the value of the 1st parameter is unpredictable,
                                      ! for the 2nd parameter both value and reference
                                      !     to its location can be considered linear,
                                      ! the value for the 3rd parameter is unpredictable
                                      !
                                      ! the !$OMP DECLARE SIMD LINEAR(VAL(in2:2))) will
   END DO                             !     be chosen from the two matching variants)
```

See Also

VALUE statement and attribute
SIMD Directive (OpenMP* API)
DECLARE SIMD

LINETO, LINETO_W (W*S)

Graphics Function: *Draws a line from the current graphics position up to and including the end point.*

Module

USE IFQWIN

Syntax

```
result = LINETO (x,y)
```

```
result = LINETO_W (wx, wy)
```

x, y	(Input) INTEGER(2). Viewport coordinates of end point.
wx, wy	(Input) REAL(8). Window coordinates of end point.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0.

The line is drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

If no error occurs, LINETO sets the current graphics position to the viewport point (*x*, *y*), and LINETO_W sets the current graphics position to the window point (*wx*, *wy*).

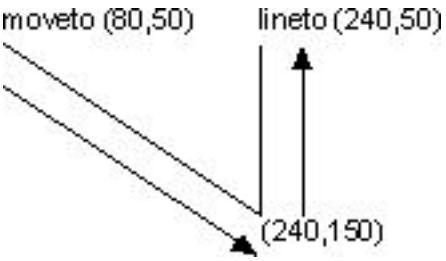
If you use FLOODFILLRGB to fill in a closed figure drawn with LINETO, the figure must be drawn with a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE
The LINETO routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the LineTo routine by including the IFWIN module, you need to specify the routine name as MSFWIN$LineTo.

Example

This program draws the figure shown below.

```
! Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(2) status
TYPE (xycoord) xy

CALL MOVETO(INT2(80), INT2(50), xy)
status = LINETO(INT2(240), INT2(150))
status = LINETO(INT2(240), INT2(50))
END
```



See Also

GETCURRENTPOSITION
GETLINESTYLE
GRSTATUS
MOVETO
POLYGON
POLYLINEQQ
SETLINESTYLE
SETWRITEMODE

LINETOAR (W*S)

Graphics Function: *Draws a line between each x,y point in the from-array to each corresponding x,y point in the to-array.*

Module

USE IFQWIN

Syntax

`result = LINETOAR (loc(fx), loc(fy), loc(tx) loc(ty), cnt)`

fx	(Input) INTEGER(2). From x viewport coordinate array.
fy	(Input) INTEGER(2). From y viewport coordinate array.
tx	(Input) INTEGER(2). To x viewport coordinate array.
ty	(Input) INTEGER(2). To y viewport coordinate array.
cnt	(Input) INTEGER(4). Length of each coordinate array; all should be the same size.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, zero.

The lines are drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

Example

```
! Build for QuickWin or Standard Graphics
USE IFQWIN
integer(2) fx(3),fy(3),tx(3),ty(3),result
integer(4) cnt, i
! load the points
do i = 1,3
```

```
   !from here
   fx(i) =20*i
   fy(i) =10
   !to there
   tx(i) =20*i
   ty(i) =60
end do
! draw the lines all at once
! 3 white vertical lines in upper left corner
result = LINETOAR(loc(fx),loc(fy),loc(tx),loc(ty), 3)
end
```

See Also

LINETO
LINETOAREX
LOC
SETCOLORRGB
SETLINESTYLE
SETWRITEMODE

LINETOAREX (W*S)

Graphics Function: *Draws a line between each x,y point in the from-array to each corresponding x,y point in the to-array. Each line is drawn with the specified graphics color and line style.*

Module

USE IFQWIN

Syntax

`result = LINETOAREX (loc(fx), loc(fy), loc(tx) loc(ty), loc(C), loc(S), cnt)`

fx	(Input) INTEGER(2). From x viewport coordinate array.
fy	(Input) INTEGER(2). From y viewport coordinate array.
tx	(Input) INTEGER(2). To x viewport coordinate array.
ty	(Input) INTEGER(2). To y viewport coordinate array.
C	(Input) INTEGER(4). Color array.
S	(Input) INTEGER(4). Style array.
cnt	(Input) INTEGER(4). Length of each coordinate array; also the length of the color array and style array. All of the arrays should be the same size.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, zero.

The lines are drawn using the specified graphics colors and line styles, and with the current write mode. The current write mode is set with SETWRITEMODE.

If the color has the Z'80000000' bit set, the color is an RGB color; otherwise, the color is a palette color.

The styles are as follows from `wingdi.h`:

```
SOLID            0
DASH             1       /* -------  */
DOT              2       /* .......  */
DASHDOT          3       /* _._._._  */
DASHDOTDOT       4       /* _.._.._  */
NULL             5
```

Example

```
! Build for QuickWin or Standard Graphics
USE IFQWIN
integer(2) fx(3),fy(3),tx(3),ty(3),result
integer(4) C(3),S(3),cnt,i,color

color = #000000FF

! load the points
do i = 1,3
   S(i) = 0  ! all lines solid
   C(i) = IOR(Z'80000000',color)
   color = color*256 ! pick another of RGB
   if(IAND(color,Z'00FFFFFF').eq.0) color = Z'000000FF'
   !from here
   fx(i) =20*i
   fy(i) =10
   !to there
   tx(i) =20*i
   ty(i) =60
 end do
 ! draw the lines all at once
 ! 3 vertical lines in upper left corner, Red, Green, and Blue
 result = LINETOAREX(loc(fx),loc(fy),loc(tx),loc(ty),loc(C),loc(S),3)
 end
```

See Also

LINETO
LINETOAR
LOC
POLYLINEQQ
SETWRITEMODE

LLE

Elemental Intrinsic Function (Generic):
Determines whether a string is lexically less than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel® Fortran, LLE is equivalent to the (.LE.) operator.

Syntax

`result = LLE (string_a,string_b)`

string_a (Input) Must be of type character.

string_b (Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if the strings are equal, both strings are of zero length, or if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LLE [1,2]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

[2]The setting of compiler options specifying integer size can affect this function.

Example

LLE ('TWO', 'THREE') has the value false.

LLE ('ONE', 'FOUR') has the value false.

The following shows another example:

```
LOGICAL L
L = LLE('ABC', 'ABC')   ! returns .TRUE.
L = LLE('ABC', 'AABCD') ! returns .FALSE.
```

See Also

LGE
LGT
LLT
ASCII and Key Code Charts

LLT

Elemental Intrinsic Function (Generic):
Determines whether a string is lexically less than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel® Fortran, LLT is equivalent to the < (.LT.) operator.

Syntax

`result = LLT (string_a,string_b)`

string_a (Input) Must be of type character.

string_b (Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LLT [1,2]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

[2]The setting of compiler options specifying integer size can affect this function.

Example

LLT ('ONE', 'SIX') has the value true.

LLT ('ONE', 'FOUR') has the value false.

The following shows another example:

```
LOGICAL L
L = LLT ('ABC', 'ABC')     ! returns .FALSE.
L = LLT ('AAXYZ', 'ABCDE') ! returns .TRUE.
```

See Also

LGE
LGT
LLE
ASCII and Key Code Charts

LNBLNK

Portability Function: *Locates the position of the last nonblank character in a string.*

Module

USE IFPORT

Syntax

`result = LNBLNK (string)`

string (Input) Character*(*). String to be searched. Cannot be an array.

Results

The result type is INTEGER(4). The result is the index of the last nonblank character in *string*.

LNBLNK is very similar to the intrinsic function LEN_TRIM, except that *string* cannot be an array.

Example

```
USE IFPORT
integer(4) p
p = LNBLNK(' GOOD DAY ') ! returns 9
p = LNBLNK(' ')          ! returns 0
```

See Also

LEN_TRIM

LOADIMAGE, LOADIMAGE_W (W*S)

Graphics Functions: *Read an image from a Windows bitmap file and display it at a specified location.*

Module

USE IFQWIN

Syntax

`result = LOADIMAGE (filename,xcoord,ycoord)`

`result = LOADIMAGE_W (filename,wxcoord,wycoord)`

filename	(Input) Character*(*). Path of the bitmap file.
xcoord, ycoord	(Input) INTEGER(4). Viewport coordinates for upper-left corner of image display.
wxcoord, wycoord	(Input) REAL(8). Window coordinates for upper-left corner of image display.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The image is displayed with the colors in the bitmap file. If the color palette in the bitmap file is different from the current system palette, the current palette is discarded and the bitmap's palette is loaded.

LOADIMAGE specifies the screen placement of the image in viewport coordinates. LOADIMAGE_W specifies the screen placement of the image in window coordinates.

See Also

SAVEIMAGE, SAVEIMAGE_W

LOC

Inquiry Intrinsic Function (Generic): *Returns the internal address of a storage item. This function cannot be passed as an actual argument.*

Syntax

`result = LOC (x)`

x	(Input) Is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of a statement function. If it is a pointer, it must be defined and associated with a target.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

This function performs the same function as the %LOC built-in function.

Example

The Fortran standard provides the C_LOC intrinsic module function as an alternative to the non-standard LOC. For more information, see the descriptions of C_LOC, C_FUNLOC, C_F_POINTER and C_F_PROCPOINTER.

```
! Example of using the LOC intrinsic
integer :: array(2) = [10,20]
integer :: t
pointer (p,t) ! Integer pointer extension
                ! p is pointer, t is pointee (target)
                ! This declares p as an address-sized integer

p = loc(array(1)) ! Address of array(1)
print *, t           ! Prints 10
p = loc(array(2)) ! Address of array(2)
print *, t           ! Prints 20
```

%LOC

Built-in Function: *Computes the internal address of a storage item.*

Syntax

```
result = %LOC (a)
```

a	(Input) Is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of a statement function. If it is a pointer, it must be defined and associated with a target.

Description

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

This function performs the same function as the LOC intrinsic function.

LOCK and UNLOCK

Statements: *A LOCK statement causes a lock variable to become locked by an image. An UNLOCK statement causes the lock variable to become unlocked. They take the following forms:*

Syntax

```
LOCK (lock-var [, ACQUIRED_LOCK=log-var] [, STAT=stat-var] [, ERRMSG=err-var])

UNLOCK (lock-var [, STAT=stat-var] [, ERRMSG=err-var])
```

lock-var	Is a scalar variable of type LOCK_TYPE. For more information, see intrinsic module ISO_FORTRAN_ENV.
log-var	Is a scalar logical variable.
stat-var	Is a scalar integer variable in which the status of the synchronization is stored.

err-var	Is a scalar default character variable in which explanatory text is stored if an error occurs.

ACQUIRED_LOCK=, STAT=, and ERRMSG= can appear in any order, but only once in a LOCK statement.

STAT= and ERRMSG= can appear in either order, but only once in an UNLOCK statement.

Description

A lock variable is unlocked if its value is equal to that of the structure constructor LOCK_TYPE ( ). If it has any other value, it is locked.

A lock variable is locked by an image if it was locked by execution of a LOCK statement on that image and has not been subsequently unlocked by execution of an UNLOCK statement on the same image.

When a LOCK statement is specified without an ACQUIRED_LOCK= specifier, it causes the lock variable to become locked by that image. If the lock variable is already locked by another image, that LOCK statement causes the lock variable to become defined after the other image causes the lock variable to become unlocked.

If the lock variable is unlocked, successful execution of a LOCK statement with an ACQUIRED LOCK= specifier causes the lock variable to become locked by that image and the *log-var* to become defined with the value TRUE. If the lock variable is already locked by a different image, successful execution of a LOCK statement with an ACQUIRED LOCK= specifier leaves the lock variable unchanged and causes the *log-var* to become defined with the value FALSE.

During the execution of the program, the value of a lock variable changes through a sequence of locked and unlocked states when LOCK and UNLOCK statements are executed. If a lock variable becomes unlocked by execution of an UNLOCK statement on image M and next becomes locked by execution of a LOCK statement on image T, the segments preceding the UNLOCK statement on image M precede the segments following the LOCK statement on image T. Execution of a LOCK statement that does not cause the lock variable to become locked does not affect segment ordering.

An error condition occurs in the following cases:

- If the lock variable in a LOCK statement is already locked by the executing image
- If the lock variable in an UNLOCK statement is not already locked by the executing image

If an error condition occurs during execution of a LOCK or UNLOCK statement, the value of the lock variable is not changed and the value of the ACQUIRED_LOCK variable, if any, is not changed.

Example

The following example shows the use of LOCK and UNLOCK statements to manage a work queue:

```
USE, INTRINSIC :: ISO_FORTRAN_ENV

TYPE(Task) :: work_queue(50)[*]          ! List of tasks on queue to perform
INTEGER :: work_queue_size[*]
TYPE(LOCK_TYPE) :: work_queue_lock[*]    ! Lock to manage the work queue

TYPE(Task) :: current_task
INTEGER :: my_image

my_image = THIS_IMAGE()
DO
   ! Process the next task in the work queue
   LOCK (work_queue_lock)              ! Start of new segment A
   ! Segment A is ordered with respect to segment B
   ! executed by image my_image-1 below because of lock exclusion
   IF (work_queue_size>0) THEN
```

```
      ! Get the next job from the queue
      current_task = work_queue(work_queue_size)
      work_queue_size = work_queue_size-1
   END IF
   UNLOCK (work_queue_lock) ! Segment ends
...
   ! Process the task

   ! Add a new task on the neighboring queue:
   LOCK(work_queue_lock[my_image+1])      ! Starts segment B
   ! Segment B is ordered with respect to segment A
   ! executed by image my_image+1 above because of lock exclusion
   IF (work_queue_size[my_image+1]<SIZE(work_queue)) THEN
      work_queue_size[my_image+1] = work_queue_size[ti+1]+1
      work_queue(work_queue_size[my_image+1])[my_image+1] = current_task
   END IF
   UNLOCK (work_queue_lock[my_image+1]) ! Ends segment B
END DO
```

See Also

Image Control Statements
Coarrays
Using Coarrays

LOG

Elemental Intrinsic Function (Generic): *Returns the natural logarithm of the argument.*

Syntax

`result = LOG (x)`

x	(Input) Must be of type real or complex. If *x* is real, its value must be greater than zero. If *x* is complex, its value must not be zero.

Results

The result type and kind are the same as *x*. The result value is approximately equal to $\log_e x$.

If the arguments are complex, the result is the principal value with imaginary part omega in the range -pi <= omega <= pi.

If the real part of *x* < 0 and the imaginary part of *x* is a positive real zero, the imaginary part of the result is pi.

If the real part of *x* < 0 and the imaginary part of *x* is a negative real zero, the imaginary part of the result is -pi.

Specific Name	Argument Type	Result Type
ALOG [1,2]	REAL(4)	REAL(4)
DLOG	REAL(8)	REAL(8)
QLOG	REAL(16)	REAL(16)
CLOG [2]	COMPLEX(4)	COMPLEX(4)

Specific Name	Argument Type	Result Type
CDLOG[3,4]	COMPLEX(8)	COMPLEX(8)
CQLOG	COMPLEX(16)	COMPLEX(16)

[1]This function is treated like LOG.

[2]The setting of compiler options specifying real size can affect ALOG, LOG, and CLOG.

[3]This function can also be specified as ZLOG.

[4]The setting of compiler options specifying double size can affect CDLOG.

Example

LOG (8.0) has the value 2.079442.

LOG (25.0) has the value 3.218876.

The following shows another example:

```
REAL r
r = LOG(10.0)  ! returns 2.302585
```

See Also

EXP
LOG10

LOG_GAMMA

Elemental Intrinsic Function (Generic): *Returns the logarithm of the absolute value of the gamma function of the argument.*

Syntax

```
result = LOG_GAMMA (x)
```

x (Input) Must be of type real. It must not be zero or a negative integer.

Results

The result type and kind are the same as *x*.

The result has a value equal to a processor-dependent approximation to the natural logarithm of the absolute value of the gamma function of *x*.

Example

LOG_GAMMA (3.0) has the approximate value 0.693.

LOG10

Elemental Intrinsic Function (Generic): *Returns the common logarithm of the argument.*

Syntax

```
result = LOG10 (x)
```

x	(Input) Must be of type real or complex. If *x* is real, its value must be greater than zero. If *x* is complex, its value must not be zero.

Results

The result type and kind are the same as *x*. The result value is approximately equal to $\log_{10}x$.

Specific Name	Argument Type	Result Type
ALOG10 [1,2]	REAL(4)	REAL(4)
DLOG10 [3]	REAL(8)	REAL(8)
QLOG10	REAL(16)	REAL(16)
CLOG10[2]	COMPLEX(4)	COMPLEX(4)
CDLOG10[3]	COMPLEX(8)	COMPLEX(8)
CQLOG10	COMPLEX(16)	COMPLEX(16)

[1]This function is treated like LOG10.

[2]The setting of compiler options specifying real size can affect ALOG10, CLOG10, and LOG10.

[3]The setting of compiler options specifying double size can affect DLOG10 and CDLOG10.

Example

LOG10 (8.0) has the value 0.9030900.

LOG10 (15.0) has the value 1.176091.

The following shows another example:

```
REAL r
r = LOG10(10.0)  ! returns 1.0
```

See Also

LOG

LOGICAL Function

Elemental Intrinsic Function (Generic): *Converts the logical value of the argument to a logical value with different kind parameters.*

Syntax

```
result = LOGICAL (l[,kind])
```

l	(Input) Must be of type logical.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is of type logical. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default logical. The result value is that of *l*.

The setting of compiler options specifying integer size can affect this function.

Example

LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical regardless of the kind parameter of logical variable L.

LOGICAL (.FALSE., 2) has the value false, with the parameter of kind 2.

See Also

CMPLX
INT
REAL
Logical Data Types

LOGICAL Statement

Statement: *Specifies the LOGICAL data type.*

Syntax

```
LOGICAL
LOGICAL([KIND=] n)
LOGICAL*n
```

n	Is kind 1, 2, 4, or 8.

If a kind parameter is specified, the logical constant has the kind specified. If no kind parameter is specified, the kind of the constant is default logical.

Example

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (2), SAVE :: doit, dont=.FALSE.
LOGICAL switch

! An equivalent declaration is:
LOGICAL flag1, flag2
LOGICAL (2) doit, dont=.FALSE.
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

See Also

Logical Data Types
Logical Constants
Data Types, Constants, and Variables

LONG

Portability Function: *Converts an INTEGER(2) argument to INTEGER(4) type.*

Module

USE IFPORT

Syntax

```
result = LONG (int2)
```

int2	(Input) INTEGER(2). Value to be converted.

Results

The result type is INTEGER(4). The result is the value of *int2* with type INTEGER(4). The upper 16 bits of the result are zeros and the lower 16 are equal to *int2*.

See Also

INT
KIND

LOOP COUNT

General Compiler Directive: *Specifies the iterations (typical trip count) for a DO loop.*

Syntax

```
!DIR$ LOOP COUNT (n1[, n2]...)
!DIR$ LOOP COUNT= n1[, n2]...
!DIR$ LOOP COUNT qualifier(n)[, qualifier(n)]...
!DIR$ LOOP COUNT qualifier=n[, qualifier=n]...
```

n1, *n2*	Is a non-negative integer constant. It indicates that the next DO loop will iterate *n1*, *n2*, or some other number of times.
qualifier	Is one or more of the following: • MAX - specifies the maximum loop trip count. • MIN - specifies the minimum loop trip count. • AVG - specifies the average loop trip count.

The value of the loop count affects heuristics used in software pipelining, vectorization, and loop-transformations.

Example

Consider the following:

```
!DIR$ LOOP COUNT (10000)
do i =1,m
b(i) = a(i) +1 ! This is likely to enable the loop to get software-pipelined
enddo
```

Note that you can specify more than one LOOP COUNT directive for a DO loop. For example, the following directives are valid:

```
!DIR$ LOOP COUNT (10, 20, 30)
!DIR$ LOOP COUNT MAX=100, MIN=3, AVG=17
DO
...
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements

LSHIFT

Elemental Intrinsic Function (Generic): *Shifts the bits in an integer left by a specified number of positions. This is the same as specifying ISHFT with a positive shift.*

See Also

See ISHFT.

LSTAT

Portability Function: *Returns detailed information about a file.*

Module

USE IFPORT

Syntax

`result = LSTAT (name,statb)`

name	(Input) Character*(*). Name of the file to examine.
statb	(Output) INTEGER(4) or INTEGER(8). One-dimensional array of size 12; where the system information is stored. See STATfor the possible values returned in *statb*.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code (see IERRNO).

LSTAT returns detailed information about the file named in *name*.

On Linux* and macOS* systems, if the file denoted by *name* is a link, LSTAT provides information on the link, while STAT provides information on the file at the destination of the link.

On Windows* systems, LSTAT returns exactly the same information as STAT (because there are no symbolic links on these systems). STAT is the preferred function.

INQUIRE also provides information about file properties.

Example

```
USE IFPORT
INTEGER(4) info_array(12), istatus
character*20 file_name
print *, "Enter name of file to examime: "
read *, file_name
ISTATUS = LSTAT (file_name, info_array)
if (.NOT. ISTATUS) then
   print *, info_array
else
   print *, 'Error ',istatus
end if
```

See Also

INQUIRE
GETFILEINFOQQ
STAT

FSTAT

LTIME

Portability Subroutine: *Returns the components of the local time zone time in a nine-element array.*

Module

USE IFPORT

Syntax

CALL LTIME (*time*,*array*)

time (Input) INTEGER(4). An elapsed time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.

array (Output) INTEGER(4). One-dimensional array with 9 elements to contain local date and time data derived from *time*.

The elements of *array* are returned as follows:

Element	Value
array(1)	Seconds (0 - 59)
array(2)	Minutes (0 - 59)
array(3)	Hours (0 - 23)
array(4)	Day of month (1 - 31)
array(5)	Month (0 - 11)
array(6)	Years since 1900
array(7)	Day of week (0 - 6, where 0 is Sunday)
array(8)	Day of year (1 - 365)
array(9)	1 if daylight saving time is in effect; otherwise, 0.

Caution
This subroutine is not year-2000 compliant, use DATE_AND_TIME instead.

On Linux* and macOS*, *time* can be a negative number returning the time before 00:00:00 Greenwich mean time, January 1, 1970. On Windows*, *time* can not be negative, in which case all 9 elements of *array* are set to -1. On all operating systems, if there is a system error in getting the local time, all 9 elements of *array* are set to -1.

Example

```
USE IFPORT
INTEGER(4) input_time, time_array(9)
!   find number of seconds since 1/1/70
input_time=TIME()
```

```
!   convert number of seconds to time array
CALL LTIME (input_time, time_array)
PRINT *, time_array
```

See Also
DATE_AND_TIME

M to N

M to N

MAKEDIRQQ

Portability Function: *Creates a new directory with a specified name.*

Module

USE IFPORT

Syntax

`result = MAKEDIRQQ (dirname)`

dirname	(Input) Character*(*). Name of directory to be created.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

MAKEDIRQQ can create only one directory at a time. You cannot create a new directory and a subdirectory below it in a single command. MAKEDIRQQ does not translate path delimiters. You can use either slash (/) or backslash (\) as valid delimiters.

If an error occurs, call GETLASTERRORQQ to retrieve the error message. Possible errors include:

- ERR$ACCES - Permission denied. The file's (or directory's) permission setting does not allow the specified access.
- ERR$EXIST - The directory already exists.
- ERR$NOENT - The file or path specified was not found.

Example

```
USE IFPORT
LOGICAL(4) result
result = MAKEDIRQQ('mynewdir')
IF (result) THEN
   WRITE (*,*) 'New subdirectory successfully created'
ELSE
   WRITE (*,*) 'Failed to create subdirectory'
END IF
END
```

See Also
DELDIRQQ
CHANGEDIRQQ
GETLASTERRORQQ

MALLOC

Elemental Intrinsic Function (Generic): *Allocates a block of memory. This is a generic function that has no specific function associated with it. It must not be passed as an actual argument.*

Syntax

```
result = MALLOC (size)
```

size	(Input) Must be of type integer. This value is the size (in bytes) of memory to be allocated.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The result is the starting address of the allocated memory. The memory allocated can be freed by using the FREE intrinsic function.

Example

```
INTEGER(4) SIZE
REAL(4) STORAGE(*)
POINTER (ADDR, STORAGE)     ! ADDR will point to STORAGE
SIZE = 1024                 ! Size in bytes
ADDR = MALLOC(SIZE)         ! Allocate the memory
CALL FREE(ADDR)             ! Free it
```

MAP Clause

Clause for TARGET Directives: *Maps a variable from the data environment of the current task to the data environment of the device associated with the construct. This clause only applies to the TARGET directives.*

Syntax

```
MAP ([[map-type-modifier[,]] map-type :] list)
```

map-type-modifier	Is the following:	
	ALWAYS	Specifies that initialization should always occur for the *list* items.
map-type	Determines how a *list* item is initialized. Possible values are:	
	ALLOC	On entry to the outermost device region, each new corresponding list item has an undefined initial value.
	FROM	On exit from the device region, the value of the corresponding list item is assigned to each original list item. This is ignored for nested regions unless *map-type-modifier*ALWAYS is specified.

	TO	On entry to the device region, each new corresponding list item is initialized with the value of the original list item. This is ignored for nested regions unless *map-type-modifier*ALWAYS is specified.
	TOFROM	On entry to the device region, each new corresponding list item is initialized with the value of the original list item. On exit from the device region, the value of the corresponding list item is assigned to each original list item. This is ignored for nested regions unless *map-type-modifier*ALWAYS is specified.
	DELETE	On exit from the device region, if the corresponding list item is present on the device, it is then deleted from the device.
	RELEASE	On exit from the outermost device region, the corresponding list item is deleted from the device.
	If a *map-type* is not specified, the default is TOFROM. The map initialization and assignment are done as if by intrinsic assignment, that is, through bitwise copy.	
list	Is the name of one or more variables, array sections, or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a common block name must appear between slashes (/ /). If a *list* item is an array section, it must specify contiguous storage. A *list* item can appear in at most one of the TO, FROM, or TOFROM clauses. If a *list* item appears in an ALLOC clause, it cannot appear on a TO or TOFROM clause.	

On entry to an outermost target region where this clause is used, for each original list item, a new corresponding list item is created on the device. On exit from the outermost target region, if the corresponding list item is present on the device, it is then deleted from the device.

At least one MAP clause must appear in a directive that allows the clause.

THREADPRIVATE variables cannot appear in a MAP clause.

For the TARGET ENTER DATA directive, *map-type* must be either TO or ALLOC.

For the TARGET EXIT DATA directive, *map-type* must be FROM, RELEASE, or DELETE.

For the TARGET and TARGET DATA directives, *map-type* must be TO, FROM, TOFROM, or ALLOC.

For the TARGET UPDATE directive, *map-type* must be TO or FROM.

If original and corresponding list items share storage, data races can result when intervening synchronization between tasks does not occur. If variables that share storage are mapped, it causes unspecified behavior.

Any variables within a TARGET MAP region that are not specified in a MAP clause are treated as shared variables within the region.

A list item must not contain any components that have the ALLOCATABLE attribute.

If the allocation status of a list item with the ALLOCATABLE attribute is unallocated upon entry to a target region, the list item must be unallocated upon exit from the region.

If the allocation status of a list item with the ALLOCATABLE attribute is allocated upon entry to a target region, the allocation status of the corresponding list item must not be changed and must not be reshaped in the region.

If an array section of an allocatable array is mapped and the size of the section is smaller than that of the whole array, the target region must not have any reference to the whole array.

See Also

TARGET directive clause DEFAULTMAP

MAP and END MAP

Statement: *Specifies mapped field declarations that are part of a UNION declaration within a STRUCTURE declaration.*

Example

```
UNION
  MAP
     CHARACTER*20 string
  END MAP
  MAP
     INTEGER*2 number(10)
  END MAP
END UNION

UNION
  MAP
     RECORD /Cartesian/ xcoord, ycoord
  END MAP
  MAP
     RECORD /Polar/ length, angle
  END MAP
END UNION
```

See Also

See STRUCTURE.

MASKL

Elemental Intrinsic Function (Generic): *Returns a left-justified mask.*

Syntax

`result = MASKL (i[,kind])`

i	(Input) Must be of type integer or of type logical (which is treated as an integer). It must be nonnegative and less than or equal to the number of bits *s* of the model integer defined for bit manipulation contexts.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value has its leftmost I bits set to 1 and the remaining bits set to 0.

The model for the interpretation of an integer value as a sequence of bits is in Model for Bit Data.

Example

MASKL (3) has the value SHIFTL (7, BIT_SIZE (0) - 3).

See Also

MASKR

MASKR

Elemental Intrinsic Function (Generic): *Returns a right-justified mask.*

Syntax

`result = MASKR (i[,kind])`

i	(Input) Must be of type integer or of type logical (which is treated as an integer). It must be nonnegative and less than or equal to the number of bits *s* of the model integer defined for bit manipulation contexts.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value has its rightmost I bits set to 1 and the remaining bits set to 0.

The model for the interpretation of an integer value as a sequence of bits is in Model for Bit Data.

Example

MASKR (3) has the value 7.

See Also

MASKL

MASTER

OpenMP* Fortran Compiler Directive: *Specifies a block of code to be executed by the master thread of the team.*

Syntax

```
!$OMP MASTER
   block
!$OMP END MASTER
```

block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

The binding thread set for a MAS construct is the current team. A master region binds to the innermost enclosing parallel region.

When the MASTER directive is specified, the other threads in the team skip the enclosed block (section) of code and continue execution. There is no implied barrier, either on entry to or exit from the master section.

Example

The following example forces the master thread to execute the routines OUTPUT and INPUT:

```
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
!$OMP MASTER
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END MASTER
      CALL WORK(Y)
!$OMP END PARALLEL
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

MATMUL

Transformational Intrinsic Function (Generic):
Performs matrix multiplication of numeric or logical matrices.

Syntax

`result = MATMUL (matrix_a,matrix_b)`

matrix_a	(Input) Must be an array of rank one or two. It must be of numeric (integer, real, or complex) or logical type.
matrix_b	(Input) Must be an array of rank one or two. It must be of numeric type if *matrix_a* is of numeric type or logical type if *matrix_a* is logical type. At least one argument must be of rank two. The size of the first (or only) dimension of *matrix_b* must equal the size of the last (or only) dimension of *matrix_a*.

Results

The result is an array whose type depends on the data type of the arguments, according to the rules described in Data Type of Numeric Expressions. The rank and shape of the result depends on the rank and shapes of the arguments, as follows:

- If *matrix_a* has shape (n, m) and *matrix_b* has shape (m, k), the result is a rank-two array with shape (n, k).
- If *matrix_a* has shape (m) and *matrix_b* has shape (m, k), the result is a rank-one array with shape (k).
- If *matrix_a* has shape (n, m) and *matrix_b* has shape (m), the result is a rank-one array with shape (n).

If the arguments are of numeric type, element (i, j) of the result has the value SUM((row i of *matrix_a*) * (column j of *matrix_b*)). If the arguments are of logical type, element (i, j) of the result has the value ANY((row i of *matrix_a*) .AND. (column j of *matrix_b*)).

Example

A is matrix

```
 [ 2  3  4 ]
 [ 3  4  5 ],
```

B is matrix

```
 [ 2  3 ]
 [ 3  4 ]
 [ 4  5 ],
```

X is vector (1, 2), and Y is vector (1, 2, 3).

The result of MATMUL (A, B) is the matrix-matrix product AB with the value

```
 [ 29  38 ]
 [ 38  50 ].
```

The result of MATMUL (X, A) is the vector-matrix product XA with the value (8, 11, 14).

The result of MATMUL (A, Y) is the matrix-vector product AY with the value (20, 26).

The following shows another example:

```
INTEGER a(2,3), b(3,2), c(2), d(3), e(2,2), f(3), g(2)
a = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  a is   1 3 5
!         2 4 6
b = RESHAPE((/1, 2, 3, 4, 5, 6/), (/3, 2/))
!  b is   1 4
!         2 5
!         3 6
c = (/1, 2/)       ! c is [1 2]
d = (/1, 2, 3/)    ! d is [1 2 3]

e = MATMUL(a, b)     ! returns 22 49
                     !         28 64

f = MATMUL(c,a)    ! returns [5 11 17]
g = MATMUL(a,d)    ! returns [22 28]
WRITE(*,*) e, f, g
END
```

See Also

TRANSPOSE
PRODUCT

MAX

Elemental Intrinsic Function (Generic): *Returns the maximum value of the arguments.*

Syntax

`result = MAX (a1,a2[,a3]...)`

a1, a2, a3	(Input) All must have the same type (integer, real, or character) and kind parameters.

Results

For arguments of character type, the result type is character, and the length of the result is the length of the longest argument. For MAX0, AMAX1, DMAX1, QMAX1, IMAX0, JMAX0, and KMAX0, the result type is the same as the arguments. For MAX1, IMAX1, JMAX1, and KMAX1, the result type is integer. For AMAX0, AIMAX0, AJMAX0, and AKMAX0, the result type is real. The value of the result is that of the largest argument. For character arguments, the comparison is done using the ASCII collating sequence. If the selected argument is shorter than the longest argument, the result is extended to the length of the longest argument by inserting blank characters on the right.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IMAX0	INTEGER(2)	INTEGER(2)
AIMAX0	INTEGER(2)	REAL(4)
MAX0 [2]	INTEGER(4)	INTEGER(4)
AMAX0 [3, 4]	INTEGER(4)	REAL(4)
KMAX0	INTEGER(8)	INTEGER(8)
AKMAX0	INTEGER(8)	REAL(4)
IMAX1	REAL(4)	INTEGER(2)
MAX1 [4, 5, 6]	REAL(4)	INTEGER(4)
KMAX1	REAL(4)	INTEGER(8)
AMAX1 [7]	REAL(4)	REAL(4)
DMAX1	REAL(8)	REAL(8)
QMAX1	REAL(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

[2]Or JMAX0.

[3]Or AJMAX0.AMAX0 is the same as REAL (MAX).

[4]In Standard Fortran, AMAX0 and MAX1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.

[5]Or JMAX1. MAX1 is the same as INT(MAX).

[6]The setting of compiler options specifying integer size can affect MAX1.

[7]The setting of compiler options specifying real size can affect AMAX1.

Example

MAX (2.0, -8.0, 6.0) has the value 6.0.

MAX (14, 32, -50) has the value 32.

The following shows another example:

```
INTEGER m1, m2
REAL r1, r2
m1 = MAX(5, 6, 7)           ! returns 7
m2 = MAX1(5.7, 3.2, -8.3)   ! returns 5
r1 = AMAX0(5, 6, 7)         ! returns 7.0
r2 = AMAX1(6.4, -12.2, 4.9) ! returns 6.4
```

See Also

MAXLOC
MAXVAL
MIN

MAXEXPONENT

Inquiry Intrinsic Function (Generic): *Returns the maximum exponent in the model representing the same type and kind parameters as the argument.*

Syntax

`result = MAXEXPONENT (x)`

x	(Input) Must be of type real; it can be scalar or array valued.

Results

The result is a scalar of type default integer. The result has the value e_{max}, as defined in Model for Real Data.

Example

```
REAL(4) x
INTEGER i
i = MAXEXPONENT(x)    ! returns 128.
```

See Also

MINEXPONENT

MAXLOC

Transformational Intrinsic Function (Generic): *Returns the location of the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.*

Syntax

`result = MAXLOC (array, dim [, mask, kind, back])`

`result = MAXLOC (array [, mask, kind, back])`

array	(Input) Must be an array of type integer, real, or character.
dim	(Input) Must be a scalar integer with a value in the range 1 to *n*, where *n* is the rank of *array*.
mask	(Input; optional) Must be a logical array that is conformable with *array*.

kind	(Input; optional) Must be a scalar integer constant expression.
back	(Input; optional) Must be a scalar of type logical.

Results

The result is an array of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if *dim* is omitted:

- The array result has rank one and a size equal to the rank of *array*.
- If MAXLOC(*array*) is specified, the elements in the array result form the subscript of the location of the element with the maximum value in *array*.

  The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array*.
- If MAXLOC(*array*, MASK= *mask*) is specified, the elements in the array result form the subscript of the location of the element with the maximum value corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2,...d_{dim-1}, d_{dim+1},...d_n), where (d_1, d_2,...d_n) is the shape of *array*.
- If *array* has rank one, MAXLOC(*array*, *dim* [, *mask*]) is a scalar and has a value equal to the first element of MAXLOC(*array* [, MASK = *mask*]). Otherwise, the value of element (s_1, s_2,...s_{dim-1}, s_{dim+1},...s_n) of MAXLOC(*array*, *dim* [, *mask*]) is equal to MAXLOC(*array*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n) [, MASK = *mask*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n)]).

If only one element has the maximum value, that element's subscripts are returned. Otherwise, if more than one element has the maximum value and *back* is absent or present with the value .FALSE., the element whose subscripts are returned is the first such element, taken in array element order. If *back* is present with the value .TRUE., the element whose subscripts are returned is the last such element, taken in array element order.

If *array* has size zero, or every element of *mask* has the value .FALSE., the value of the result is controlled by compiler option `assume [no]old_maxminloc`, which can set the value of the result to either 1 or 0.

If *array* is of type character, the comparison is done using the ASCII collating sequence.

The setting of compiler options specifying integer size can affect this function.

Example

The value of MAXLOC((/3, 7, 4, 7/)) is (2), which is the subscript of the location of the first occurrence of the maximum value in the rank-one array.

A is the array

```
[  4    0    4    2 ]
[  3    1   -2    6 ]
[ -1   -4    5    5 ].
```

MAXLOC (A, MASK=A .LT. 5) has the value (1, 1) because these are the subscripts of the location of the first maximum value (4) that is less than 5.

MAXLOC (A, DIM=1) has the value (1, 2, 3, 2). 1 is the subscript of the location of the first maximum value (4) in column 1; 2 is the subscript of the location of the first maximum value (1) in column 2; and so forth.

MAXLOC (A, DIM=2) has the value (1, 4, 3). 1 is the subscript of the location of the first maximum value in row 1; 4 is the subscript of the location of the first maximum value in row 2; and so forth.

MAXLOC (A, DIM=2, BACK=.TRUE.) has the value (3, 4, 4). 3 is the subscript of the location of the last maximum value in row 1; 4 is the subscript of the location of the last maximum value in row 2; and so forth.

The following shows another example:

```
INTEGER i, maxl(1), max
INTEGER array(3, 3)
INTEGER, ALLOCATABLE :: AR1(:)
! put values in array
array = RESHAPE((/7, 9, -1, -2, 5, 0, 3, 6, 9/),     &
                (/3, 3/))
! array is  7 -2 3
!           9  5 6
!          -1  0 9
i = SIZE(SHAPE(array))   ! Get number of dimensions
                         ! in array
ALLOCATE ( AR1(i))       ! Allocate AR1 to number
                         ! of dimensions in array
AR1 = MAXLOC (array, MASK = array .LT. 7) ! Get
                         ! the location (subscripts) of
                         ! largest element less than 7
                         ! in array
!
! MASK = array .LT. 7 creates a mask array the same
! size and shape as array whose elements are .TRUE. if
! the corresponding element in array is less than 7,
! and .FALSE. if it is not. This mask causes MAXLOC to
! return the index of the element in array with the
! greatest value less than 7.
!
! array is  7 -2 3 and MASK=array .LT. 7 is  F T T
!           9  5 6                           F T T
!          -1  0 9                           T T F
! and AR1 = MAXLOC(array, MASK = array .LT. 7) returns
! (2, 3), the location of the element with value 6

maxl = MAXLOC((/1, 4, 3, 4/))    ! returns 2, the first
                                 ! occurrence of maximum
END
```

See Also

MAXVAL
MINLOC
MINVAL
FINDLOC

MAXVAL

Transformational Intrinsic Function (Generic):
Returns the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

`result = MAXVAL (`*`array`*`[,`*`dim`*`] [,`*`mask`*`])`

array	(Input) Must be an array of type integer, real, or character.
dim	(Input; optional) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be a logical array that is conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If MAXVAL(*array*) is specified, the result has a value equal to the maximum value of all the elements in *array*.
- If MAXVAL(*array*, MASK= *mask*) is specified, the result has a value equal to the maximum value of the elements in *array* corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- An array result has a rank that is one less than *array*, and shape (d_1, d_2,...,d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- If *array* has rank one, MAXVAL(*array*, *dim*[, *mask*]) has a value equal to that of MAXVAL(*array*[,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of MAXVAL(*array*, *dim*, [, *mask*]) is equal to MAXVAL(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK = *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

If *array* has size zero or if there are no true elements in *mask*, the result (if *dim* is omitted), or each element in the result array (if *dim* is specified), has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind parameters of *array*.

If *array* is of type character, the comparison is done using the ASCII collating sequence.

Example

The value of MAXVAL ((/2, 3, 4/)) is 4 because that is the maximum value in the rank-one array.

MAXVAL (B, MASK=B .LT. 0.0) finds the maximum value of the negative elements of B.

C is the array

```
 [ 2  3  4 ]
 [ 5  6  7 ].
```

MAXVAL (C, DIM=1) has the value (5, 6, 7). 5 is the maximum value in column 1; 6 is the maximum value in column 2; and so forth.

MAXVAL (C, DIM=2) has the value (4, 7). 4 is the maximum value in row 1 and 7 is the maximum value in row 2.

The following shows another example:

```
INTEGER array(2,3), i(2), max
INTEGER, ALLOCATABLE :: AR1(:), AR2(:)
array = RESHAPE((/1, 4, 5, 2, 3, 6/),(/2, 3/))
! array is   1 5 3
!            4 2 6
i = SHAPE(array)      ! i = [2 3]
ALLOCATE (AR1(i(2)))  ! dimension AR1 to the number of
                      ! elements in dimension 2
                      ! (a column) of array
```

```
ALLOCATE (AR2(i(1)))  ! dimension AR2 to the number of
                      ! elements in dimension 1
                      ! (a row) of array
max = MAXVAL(array, MASK = array .LT. 4) ! returns 3
AR1 = MAXVAL(array, DIM = 1)  ! returns [ 4 5 6 ]
AR2 = MAXVAL(array, DIM = 2)  ! returns [ 5 6 ]
END
```

See Also

MAXLOC
MINVAL
MINLOC

MBCharLen (W*S)

NLS Function: *Returns the length, in bytes, of the first character in a multibyte-character string.*

Module

USE IFNLS

Syntax

`result = MBCharLen (string)`

string	(Input) Character*(*). String containing the character whose length is to be determined. Can contain multibyte characters.

Results

The result type is INTEGER(4). The result is the number of bytes in the first character contained in *string*. The function returns 0 if *string* has no characters (is length 0).

MBCharLen does not test for multibyte character validity.

See Also

MBCurMax
MBLead
MBLen
MBLen_Trim

MBConvertMBToUnicode (W*S)

NLS Function: *Converts a multibyte-character string from the current codepage to a Unicode string.*

Module

USE IFNLS

Syntax

`result = MBConvertMBToUnicode (mbstr,unicodestr[,flags])`

mbstr	(Input) Character*(*). Multibyte codepage string to be converted.

unicodestr	(Output) INTEGER(2). Array of integers that is the translation of the input string into Unicode.
flags	(Input; optional) INTEGER(4). If specified, modifies the string conversion. If *flags* is omitted, the value NLS$Precomposed is used. Available values (defined in IFNLS.F90) are: • NLS$Precomposed - Use precomposed characters always. (default) • NLS$Composite - Use composite wide characters always. • NLS$UseGlyphChars - Use glyph characters instead of control characters. • NLS$ErrorOnInvalidChars - Returns -1 if an invalid input character is encountered. The flags NLS$Precomposed and NLS$Composite are mutually exclusive. You can combine NLS$UseGlyphChars with either NLS $Precomposed or NLS$Composite using an inclusive OR (IOR or OR).

Results

The result type is INTEGER(4). If no error occurs, the result is the number of bytes written to *unicodestr*(bytes are counted, not characters), or the number of bytes required to hold the output string if *unicodestr* has zero size. If the *unicodestr* array is bigger than needed to hold the translation, the extra elements are set to space characters. If *unicodestr* has zero size, the function returns the number of bytes required to hold the translation and nothing is written to *unicodestr*.

If an error occurs, one of the following negative values is returned:

- NLS$ErrorInsufficentBuffer - The *unicodestr* argument is too small, but not zero size so that the needed number of bytes would be returned.
- NLS$ErrorInvalidFlags - The *flags* argument has an illegal value.
- NLS$ErrorInvalidCharacter - A character with no Unicode translation was encountered in *mbstr*. This error can occur only if the NLS$InvalidCharsError flag was used in *flags*.

NOTE
By default, or if *flags* is set to NLS$Precomposed, the function MBConvertMBToUnicode attempts to translate the multibyte codepage string to a precomposed Unicode string. If a precomposed form does not exist, the function attempts to translate the codepage string to a composite form.

See Also

MBConvertUnicodeToMB

MBConvertUnicodeToMB (W*S)

NLS Function: *Converts a Unicode string to a multibyte-character string from the current codepage.*

Module

USE IFNLS

Syntax

```
result = MBConvertUnicodeToMB (unicodestr,mbstr[,flags])
```

unicodestr	(Input) INTEGER(2). Array of integers holding the Unicode string to be translated.
mbstr	(Output) Character*(*). Translation of Unicode string into multibyte character string from the current codepage.
flags	(Input; optional) INTEGER(4). If specified, argument to modify the string conversion. If *flags* is omitted, no extra checking of the conversion takes place. Available values (defined in IFNLS.F90) are: • NLS$CompositeCheck - Convert composite characters to precomposed. • NLS$SepChars - Generate separate characters. • NLS$DiscardDns - Discard nonspacing characters. • NLS$DefaultChars - Replace exceptions with default character. The last three flags (NLS$SepChars, NLS$DiscardDns, and NLS $DefaultChars) are mutually exclusive and can be used only if NLS $CompositeCheck is set, in which case one (and only one) of them is combined with NLS$CompositeCheck using an inclusive OR (IOR or OR). These flags determine what translation to make when there is no precomposed mapping for a base character/nonspace character combination in the Unicode wide character string. The default (IOR(NLS$CompositeCheck, NLS$SepChars)) is to generate separate characters.

Results

The result type is INTEGER(4). If no error occurs, returns the number of bytes written to *mbstr* (bytes are counted, not characters), or the number of bytes required to hold the output string if *mbstr* has zero length. If *mbstr* is longer than the translation, it is blank-padded. If *mbstr* is zero length, the function returns the number of bytes required to hold the translation and nothing is written to *mbstr*.

If an error occurs, one of the following negative values is returned:

- NLS$ErrorInsufficientBuffer - The *mbstr* argument is too small, but not zero length so that the needed number of bytes is returned.
- NLS$ErrorInvalidFlags - The *flags* argument has an illegal value.

See Also

MBConvertMBToUnicode

MBCurMax (W*S)

NLS Function: *Returns the longest possible multibyte character length, in bytes, for the current codepage.*

Module

USE IFNLS

Syntax

```
result = MBCurMax( )
```

Results

The result type is INTEGER(4). The result is the longest possible multibyte character, in bytes, for the current codepage.

The MBLenMax parameter, defined in the module `IFNLS.F90`, is the longest length, in bytes, of any character in any codepage installed on the system.

See Also

MBCharLen

MBINCHARQQ (W*S)

NLS Function: *Performs the same function as INCHARQQ except that it can read a single multibyte character at once, and it returns the number of bytes read as well as the character.*

Module

USE IFNLS

Syntax

`result = MBINCHARQQ (`*`string`*`)`

`string`	(Output) CHARACTER(MBLenMax). String containing the read characters, padded with blanks up to the length MBLenMax. The MBLenMax parameter, defined in the module `IFNLS.F90`, is the longest length, in bytes, of any character in any codepage installed on the system.

Results

The result type is INTEGER(4). The result is the number of characters read.

See Also

INCHARQQ
MBCurMax
MBCharLen
MBLead

MBINDEX (W*S)

NLS Function: *Performs the same function as INDEX except that the strings manipulated can contain multibyte characters.*

Module

USE IFNLS

Syntax

`result = MBINDEX (`*`string`*`,`*`substring`*`[,`*`back`*`])`

`string`	(Input) CHARACTER*(*). String to be searched for the presence of *substring*. Can contain multibyte characters.
`substring`	(Input) CHARACTER*(*). Substring whose position within *`string`* is to be determined. Can contain multibyte characters.

back (Input; optional) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results

The result type is INTEGER(4). If *back* is omitted or is .FALSE., it returns the leftmost position in *string* that contains the start of *substring*. If *back* is .TRUE., it returns the rightmost position in *string* that contains the start of *substring*. If *string* does not contain *substring*, it returns 0. If *substring* occurs more than once, it returns the starting position of the first occurrence ("first" is determined by the presence and value of *back*).

The position returned is a byte index, not a character index because of the confounding case of multibyte characters. As an example, if a substring which is a single multi-byte character matches the second multibyte character in a string consisting of two two-byte multi-byte characters the position is 3, not 2.

See Also

INDEX
MBSCAN
MBVERIFY

MBJISToJMS, MBJMSToJIS (W*S)

NLS Functions: *Converts Japan Industry Standard (JIS) characters to Microsoft Kanji (JMS) characters, or converts JMS characters to JIS characters.*

Module

USE IFNLS

Syntax

```
result = MBJISToJMS (char)
result = MBJMSToJIS (char)
```

char (Input) CHARACTER(2). JIS or JMS character to be converted.

A JIS character is converted only if the lead and trail bytes are in the hexadecimal range 21 through 7E.

A JMS character is converted only if the lead byte is in the hexadecimal range 81 through 9F or E0 through FC, and the trail byte is in the hexadecimal range 40 through 7E or 80 through FC.

Results

The result type is character with length 2. MBJISToJMS returns a Microsoft Kanji (Shift JIS or JMS) character. MBJMSToJIS returns a Japan Industry Standard (JIS) character.

Only computers with Japanese installed as one of the available languages can use the MBJISToJMS and MBJMSToJIS conversion functions.

See Also

NLSEnumLocales
NLSEnumCodepages
NLSGetLocale
NLSSetLocale

MBLead (W*S)

NLS Function: *Determines whether a given character is the lead (first) byte of a multibyte character sequence.*

Module

USE IFNLS

Syntax

`result = MBLead (char)`

char	(Input) CHARACTER(1). Character to be tested for lead status.

Results

The result type is LOGICAL(4). The result is .TRUE. if *char* is the first character of a multibyte character sequence; otherwise, .FALSE..

MBLead only works stepping forward through a whole multibyte character string. For example:

```
      DO i = 1, LEN(str)  ! LEN returns the number of bytes, not the
                          !  number of characters in str
         WRITE(*, 100) MBLead (str(i:i))
      END DO
100   FORMAT (L2, \)
```

MBLead is passed only one character at a time and must start on a lead byte and step through a string to establish context for the character. MBLead does not correctly identify a nonlead byte if it is passed only the second byte of a multibyte character because the status of lead byte or trail byte depends on context.

The function MBStrLead is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context. So, MBStrLead can be much slower than MBLead (up to *n* times slower, where *n* is the length of the string).

See Also

MBStrLead
MBCharLen

MBLen (W*S)

NLS Function: *Returns the number of characters in a multibyte-character string, including trailing blanks.*

Module

USE IFNLS

Syntax

`result = MBLen (string)`

string	(Input) CHARACTER*(*). String whose characters are to be counted. Can contain multibyte characters.

Results

The result type is INTEGER(4). The result is the number of characters in *string*.

MBLen recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.

See Also

MBLen_Trim
MBStrLead

MBLen_Trim (W*S)

NLS Function: *Returns the number of characters in a multibyte-character string, not including trailing blanks.*

Module

USE IFNLS

Syntax

```
result = MBLen_Trim (string)
```

string	(Input) Character*(*). String whose characters are to be counted. Can contain multibyte characters.

Results

The result type is INTEGER(4). The result is the number of characters in *string* minus any trailing blanks (blanks are bytes containing character 32 (hex 20) in the ASCII collating sequence).

MBLen_Trim recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.

See Also

MBLen
MBStrLead

MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE (W*S)

NLS Functions: *Perform the same functions as LGE, LGT, LLE, LLT and the logical operators .EQ. and .NE. except that the strings being compared can include multibyte characters, and optional flags can modify the comparison.*

Module

USE IFNLS

Syntax

```
result = MBLGE (string_a,string_b, [ flags])
result = MBLGT (string_a,string_b, [ flags])
result = MBLLE (string_a,string_b, [ flags])
result = MBLLT (string_a,string_b, [ flags])
result = MBLEQ (string_a,string_b, [ flags])
result = MBLNE (string_a,string_b, [ flags])
```

string_a, *string_b*	(Input) Character*(*). Strings to be compared. Can contain multibyte characters.
flags	(Input; optional) INTEGER(4). If specified, determines which character traits to use or ignore when comparing strings. You can combine several flags using an inclusive OR (IOR or OR). There are no illegal combinations of flags, and the functions may be used without flags, in which case all flag options are turned off. The available values (defined in `IFNLS.F90`) are: • NLS$MB_IgnoreCase - Ignore case. • NLS$MB_IgnoreNonspace - Ignore nonspacing characters (this flag removes Japanese accent characters if they exist). • NLS$MB_IgnoreSymbols - Ignore symbols. • NLS$MB_IgnoreKanaType - Do not differentiate between Japanese Hiragana and Katakana characters (corresponding Hiragana and Katakana characters will compare as equal). • NLS$MB_IgnoreWidth - Do not differentiate between a single-byte character and the same character as a double byte. • NLS$MB_StringSort - Sort all symbols at the beginning, including the apostrophe and hyphen (see the NOTE below).

Results

The result type is LOGICAL(4). Comparisons are made using the current locale, not the current codepage. The codepage used is the default for the language/country combination of the current locale.

The results of these functions are as follows:

- MBLGE returns .TRUE. if the strings are equal or *string_a* comes last in the collating sequence; otherwise, .FALSE..
- MBLGT returns .TRUE. if *string_a* comes last in the collating sequence; otherwise, .FALSE..
- MBLLE returns .TRUE. if the strings are equal or *string_a* comes first in the collating sequence; otherwise, .FALSE..
- MBLLT returns .TRUE. if *string_a* comes first in the collating sequence; otherwise, .FALSE..
- MBLEQ returns .TRUE. if the strings are equal in the collating sequence; otherwise, .FALSE..
- MBLNE returns .TRUE. if the strings are not equal in the collating sequence; otherwise, .FALSE..

If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, then the return value indicates that the longer string is greater.

If *flags* is invalid, the functions return .FALSE..

If the strings supplied contain Arabic Kashidas, the Kashidas are ignored during the comparison. Therefore, if the two strings are identical except for Kashidas within the strings, the functions return a value indicating they are "equal" in the collation sense, though not necessarily identical.

NOTE
When not using the NLS$MB_StringSort flag, the hyphen and apostrophe are special symbols and are treated differently than others. This is to ensure that words like coop and co-op stay together within a list. All symbols, except the hyphen and apostrophe, sort before any other alphanumeric character. If you specify the NLS$MB_StringSort flag, hyphen and apostrophe sort at the beginning also.

See Also

LGE
LGT

LLE
LLT

MBNext (W*S)

NLS Function: *Returns the position of the first lead byte or single-byte character immediately following the given position in a multibyte-character string.*

Module

USE IFNLS

Syntax

`result = MBNext (string,position)`

string	(Input) Character*(*). String to be searched for the first lead byte or single-byte character after the current position. Can contain multibyte characters.
position	(Input) INTEGER(4). Position in *string* to search from. Must be the position of a lead byte or a single-byte character. Cannot be the position of a trail (second) byte of a multibyte character.

Results

The result type is INTEGER(4). The result is the position of the first lead byte or single-byte character in *string* immediately following the position given in *position*, or 0 if no following first byte is found in *string*.

See Also

MBPrev

MBPrev (W*S)

NLS Function: *Returns the position of the first lead byte or single-byte character immediately preceding the given string position in a multibyte-character string.*

Module

USE IFNLS

Syntax

`result = MBPrev (string,position)`

string	(Input) Character*(*). String to be searched for the first lead byte or single-byte character before the current position. Can contain multibyte characters.
position	(Input) INTEGER(4). Position in *string* to search from. Must be the position of a lead byte or single-byte character. Cannot be the position of the trail (second) byte of a multibyte character.

Results

The result type is INTEGER(4). The result is the position of the first lead byte or single-byte character in *string* immediately preceding the position given in *position*, or 0 if no preceding first byte is found in *string*.

See Also
MBNext

MBSCAN (W*S)
NLS Function: *Performs the same function as SCAN except that the strings manipulated can contain multibyte characters.*

Module
USE IFNLS

Syntax
`result = MBSCAN (string,set[,back])`

string	(Input) Character*(*). String to be searched for the presence of any character in *set*.
set	(Input) Character*(*). Characters to search for.
back	(Input; optional) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results
The result type is INTEGER(4). If *back* is .FALSE. or is omitted, it returns the position of the leftmost character in *string* that is in *set*. If *back* is .TRUE., it returns the rightmost character in *string* that is in *set*. If no characters in *string* are in *set*, it returns 0.

See Also
SCAN
MBINDEX
MBVERIFY

MBStrLead (W*S)
NLS Function: *Performs a context-sensitive test to determine whether a given character byte in a string is a multibyte-character lead byte.*

Module
USE IFNLS

Syntax
`result = MBStrLead (string,position)`

string	(Input) Character*(*). String containing the character byte to be tested for lead status.
position	(Input) INTEGER(4). Position in *string* of the character byte in the string to be tested.

Results

The result type is LOGICAL(4). The result is .TRUE. if the character byte in *position* of *string* is a lead byte; otherwise, .FALSE..

MBStrLead is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context.

MBLead is passed only one character at a time and must start on a lead byte and step through a string one character at a time to establish context for the character. So, MBStrLead can be much slower than MBLead (up to *n* times slower, where *n* is the length of the string).

See Also

MBLead

MBVERIFY (W*S)

NLS Function: *Performs the same function as VERIFY except that the strings manipulated can contain multibyte characters.*

Module

USE IFNLS

Syntax

```
result = MBVERIFY (string,set[,back])
```

string	(Input) Character*(*). String to be searched for presence of any character not in *set*.
set	(Input) Character*(*). Set of characters tested to verify that it includes all the characters in *string*.
back	(Input; optional) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results

The result type is INTEGER(4). If *back* is .FALSE. or is omitted, it returns the position of the leftmost character in *string* that is not in *set*. If *back* is .TRUE., it returns the rightmost character in *string* that is not in *set*. If all the characters in *string* are in *set*, it returns 0.

See Also

VERIFY
MBINDEX
MBSCAN

MCLOCK

Inquiry Intrinsic Function (Specific): *Returns time accounting for a program.*

Syntax

```
result = MCLOCK( )
```

Results

The result type is INTEGER(4). The result is the sum (in units of milliseconds) of the current process's user time and the user and system time of all its child processes.

MERGE

Elemental Intrinsic Function (Generic): *Selects between two values or between corresponding elements in two arrays, according to the condition specified by a logical mask.*

Syntax

`result = MERGE (tsource,fsource,mask)`

tsource	(Input) May be of any data type.
fsource	(Input) Must be of the same type and type parameters as *tsource*.
mask	(Input) Must be of type logical.

Results

The result type and kind are the same as *tsource*. The value of *mask* determines whether the result value is taken from *tsource* (if *mask* is true) or *fsource* (if *mask* is false).

Example

For MERGE (1.0, 0.0, R < 0), R = -3 has the value 1.0, and R = 7 has the value 0.0.

TSOURCE is the array

```
 [ 1  3  5 ]
 [ 2  4  6 ],
```

FSOURCE is the array

```
 [ 8  9  0 ]
 [ 1  2  3 ],
```

and MASK is the array

```
 [ F  T  T]
 [ T  T  F].
```

MERGE (TSOURCE, FSOURCE, MASK) produces the result:

```
 [ 8  3  5 ]
 [ 2  4  3 ].
```

The following shows another example:

```
INTEGER tsource(2, 3), fsource(2, 3), AR1 (2, 3)
LOGICAL mask(2, 3)
tsource = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2, 3/))
fsource = RESHAPE((/7, 0, 8, -1, 9, -2/), (/2, 3/))
mask = RESHAPE((/.TRUE., .FALSE., .FALSE., .TRUE.,         &
                .TRUE., .FALSE./), (/2,3/))
! tsource is  1 2 3 , fsource is  7  8  9 , mask is  T F T
!             4 5 6               0 -1 -2            F T F
```

```
AR1 = MERGE(tsource, fsource, mask) ! returns  1 8  3
                                    !          0 5 -2
END
```

MERGE_BITS

Elemental Intrinsic Function (Generic): *Merges bits by using a mask.*

Syntax

`result = MERGE_BITS (i,j,mask)`

i	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.
j	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.
mask	(Input) Must be of type integer, or a binary, octal, or hexadecimal literal constant.

If both *i* and *j* are of type integer they must have the same kind type parameter. They cannot both be binary, octal, or hexadecimal literal constants.

If *mask* is of type integer, it must have the same kind type parameter as each other argument of type integer.

Results

The result type and kind are the same as *i* if *i* is of type integer; otherwise, the result type and kind are the same as *j*.

If any argument is a binary, octal, or hexadecimal literal constant, it is first converted as if by the intrinsic function INT to the type and kind parameter of the result. The result has the value of IOR (IAND (*i*, *mask*), IAND (*j*, NOT (*mask*))).

Example

MERGE_BITS (13, 18, 22) has the value 4.

MERGEABLE Clause

Parallel Directive Clause: *Specifies that the implementation may generate a merged task.*

Syntax

`MERGEABLE`

When the generated task is an undeferred task or an included task, it specifies that the implementation may instead generate a merged task.

When the MERGEABLE clause is present on a TASKLOOP construct, each generated task is a mergeable task.

MESSAGE

General Compiler Directive: *Specifies a character string to be sent to the standard output device during the first compiler pass; this aids debugging.*

Syntax

```
!DIR$ MESSAGE:string
```

string	Is a character constant specifying a message.

Example

```
!DIR$ MESSAGE:'Compiling Sound Speed Equations'
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives

MESSAGEBOXQQ (W*S)

QuickWin Function: *Displays a message box in a QuickWin window.*

Module

USE IFQWIN

Syntax

```
result = MESSAGEBOXQQ (msg,caption,mtype)
```

msg	(Input) Character*(*). Null-terminated C string. Message the box displays.
caption	(Input) Character*(*). Null-terminated C string. Caption that appears in the title bar.
mtype	(Input) INTEGER(4). Symbolic constant that determines the objects (buttons and icons) and properties of the message box. You can combine several constants (defined in `IFQWIN.F90`) using an inclusive OR (IOR or OR). The symbolic constants and their associated objects or properties are as follows: • MB$ABORTRETRYIGNORE - The Abort, Retry, and Ignore buttons. • MB$DEFBUTTON1 - The first button is the default. • MB$DEFBUTTON2 - The second button is the default. • MB$DEFBUTTON3 - The third button is the default. • MB$ICONASTERISK, MB$ICONINFORMATION - Lowercase *i* in blue circle icon. • MB$ICONEXCLAMATION - The exclamation-mark icon. • MB$ICONHAND, MB$ICONSTOP - The stop-sign icon. • MB$ICONQUESTION - The question-mark icon. • MB$OK - The OK button. • MB$OKCANCEL - The OK and Cancel buttons. • MB$RETRYCANCEL - The Retry and Cancel buttons. • MB$SYSTEMMODAL - Box is system-modal: all applications are suspended until the user responds. • MB$YESNO - The Yes and No buttons. • MB$YESNOCANCEL - The Yes, No, and Cancel buttons.

Results

The result type is INTEGER(4). The result is zero if memory is not sufficient for displaying the message box. Otherwise, the result is one of the following values, indicating the user's response to the message box:

- MB$IDABORT - The Abort button was pressed.
- MB$IDCANCEL - The Cancel button was pressed.
- MB$IDIGNORE - The Ignore button was pressed.
- MB$IDNO - The No button was pressed.
- MB$IDOK - The OK button was pressed.
- MB$IDRETRY - The Retry button was pressed.
- MB$IDYES - The Yes button was pressed.

Example

```
! Build as QuickWin app
USE IFQWIN
message = MESSAGEBOXQQ('Do you want to continue?'C,    &
          'Matrix'C,   &
          MB$ICONQUESTION.OR.MB$YESNO.OR.MB$DEFBUTTON1)
END
```

See Also

ABOUTBOXQQ
SETMESSAGEQQ

MIN

Elemental Intrinsic Function (Generic): *Returns the minimum value of the arguments.*

Syntax

```
result = MIN (a1,a2[,a3...])
```

a1, *a2*, *a3*	(Input) All must have the same type (integer, real, or character) and kind parameters.

Results

For arguments of character type, the result type is character, and the length of the result is the length of the longest argument. For MIN0, AMIN1, DMIN1, QMIN1, IMIN0, JMIN0, and KMIN0, the result type is the same as the arguments. For MIN1, IMIN1, JMIN1, and KMIN1, the result type is integer. For AMIN0, AIMIN0, AJMIN0, and AKMIN0, the result type is real. The value of the result is that of the smallest argument. For character arguments, the comparison is done using the ASCII collating sequence. If the selected argument is shorter than the longest argument, the result is extended to the length of the longest argument by inserting blank characters on the right.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IMIN0	INTEGER(2)	INTEGER(2)
AIMIN0	INTEGER(2)	REAL(4)
MIN0 [2]	INTEGER(4)	INTEGER(4)
AMIN0 [3, 4]	INTEGER(4)	REAL(4)

Specific Name [1]	Argument Type	Result Type
KMIN0	INTEGER(8)	INTEGER(8)
AKMIN0	INTEGER(8)	REAL(4)
IMIN1	REAL(4)	INTEGER(2)
MIN1 [4, 5, 6]	REAL(4)	INTEGER(4)
KMIN1	REAL(4)	INTEGER(8)
AMIN1 [7]	REAL(4)	REAL(4)
DMIN1	REAL(8)	REAL(8)
QMIN1	REAL(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

[2]Or JMIN0.

[3]Or AJMIN0.AMIN0 is the same as REAL (MIN).

[4]In Standard Fortran, AMIN0 and MIN1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.

[5]Or JMIN1.MIN1 is the same as INT (MIN).

[6]The setting of compiler options specifying integer size can affect MIN1.

[7]The setting of compiler options specifying real size can affect AMIN1.

Example

MIN (2.0, -8.0, 6.0) has the value -8.0.

MIN (14, 32, -50) has the value -50.

The following shows another example:

```
INTEGER m1, m2
REAL r1, r2
m1 = MIN (5, 6, 7)            ! returns 5
m2 = MIN1 (-5.7, 1.23, -3.8) ! returns -5
r1 = AMIN0 (-5, -6, -7)       ! returns -7.0
r2 = AMIN1(-5.7, 1.23, -3.8) ! returns -5.7
```

See Also

MINLOC
MINVAL
MAX

MINEXPONENT

Inquiry Intrinsic Function (Generic): *Returns the minimum exponent in the model representing the same type and kind parameters as the argument.*

Syntax

```
result = MINEXPONENT (x)
```

x	(Input) must be of type real; it can be scalar or array valued.

Results

The result is a scalar of type default integer. The result has the value e_{min}, as defined in Model for Real Data.

Example

If X is of type REAL(4), MINEXPONENT (X) has the value -125.

The following shows another example:

```
REAL(8) r1  ! DOUBLE PRECISION REAL
INTEGER i
i = MINEXPONENT (r1) ! returns - 1021.
```

See Also

MAXEXPONENT

MINLOC

Transformational Intrinsic Function (Generic):
Returns the location of the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

`result = MINLOC (array, dim [, mask, kind, back])`

`result = MINLOC (array [, mask, kind, back])`

array	(Input) Must be an array of type integer, real, or character.
dim	(Input) Must be a scalar integer with a value in the range 1 to *n*, where *n* is the rank of *array*.
mask	(Input; optional) Must be a logical array that is conformable with *array*.
kind	(Input; optional) Must be a scalar integer constant expression.
back	(Input; optional) Must be a scalar of type logical.

Results

The result is an array of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if *dim* is omitted:

- The array result has rank one and a size equal to the rank of *array*.
- If MINLOC(*array*) is specified, the elements in the array result form the subscript of the location of the element with the minimum value in *array*.

  The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array*.
- If MINLOC(*array*, MASK= *mask*) is specified, the elements in the array result form the subscript of the location of the element with the minimum value corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2,...d_{dim-1}, d_{dim+1},...d_n), where (d_1, d_2,...d_n) is the shape of *array*.
- If *array* has rank one, MINLOC(*array*, *dim* [, *mask*]) is a scalar and has a value equal to the first element of MINLOC(*array* [,MASK = *mask*]). Otherwise, the value of element (s_1, s_2,...s_{dim-1}, s_{dim+1},...s_n) of MINLOC(*array*, *dim* [, *mask*]) is equal to MINLOC(*array*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n) [, MASK = *mask*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n)]).

If only one element has the minimum value, that element's subscripts are returned. Otherwise, if more than one element has the minimum value and *back* is absent or present with the value .FALSE., the element whose subscripts are returned is the first such element, taken in array element order. If *back* is present with the value .TRUE., the element whose subscripts are returned is the last such element, taken in array element order.

If *array* has size zero, or every element of *mask* has the value .FALSE., the value of the result is controlled by compiler option `assume [no]old_maxminloc`, which can set the value of the result to either 1 or 0.

If *array* is of type character, the comparison is done using the ASCII collating sequence.

The setting of compiler options specifying integer size can affect this function.

Example

The value of MINLOC ((/3, 1, 4, 1/)) is (2), which is the subscript of the location of the first occurrence of the minimum value in the rank-one array.

A is the array

```
 [  4    0   0    2 ]
 [  3   -6   -2    6 ]
 [ -1   -4    5   -4 ].
```

MINLOC (A, MASK=A .GT. -5) has the value (3, 2) because these are the subscripts of the location of the first minimum value (-4) that is greater than -5.

MINLOC (A, DIM=1) has the value (3, 2, 2, 3). 3 is the subscript of the location of the first minimum value (-1) in column 1; 3 is the subscript of the location of the first minimum value (-6) in column 2; and so forth.

MINLOC (A, DIM=2) has the value (2, 2, 2). 2 is the subscript of the location of the first minimum value (0) in row 1; 2 is the subscript of the location of the first minimum value (-6) in row 2; and so forth.

MINLOC (A, DIM=2, BACK=.TRUE.) has the value (3, 2, 4). 3 is the subscript of the location of the last minimum value (0) in row 1; 2 is the subscript of the location of the last minimum value (-6) in row 2; and so forth.

The following shows another example:

```
INTEGER i, minl(1)
INTEGER array(2, 3)
INTEGER, ALLOCATABLE :: AR1(:)
! put values in array
array = RESHAPE((/-7, 1, -2, -9, 5, 0/),(/2, 3/))
!   array is   -7 -2 5
!               1 -9 0
i = SIZE(SHAPE(array))  ! Get the number of dimensions
                        !   in array
ALLOCATE (AR1 (i))      ! Allocate AR1 to number
                        !   of dimensions in array
AR1 = MINLOC (array, MASK = array .GT. -5)   ! Get the
                        !   location (subscripts) of
                        !   smallest element greater
                        !   than -5 in array
```

```
!
! MASK = array .GT. -5 creates a mask array the same
! size and shape as array whose elements are .TRUE. if
! the corresponding element in array is greater than
! -5, and .FALSE. if it is not. This mask causes MINLOC
! to return the index of the element in array with the
! smallest value greater than -5.
!
!array is  -7 -2 5 and MASK= array .GT. -5 is  F T T
!           1 -9 0                              T F T
! and AR1 = MINLOC(array, MASK = array .GT. -5) returns
! (1, 2), the location of the element with value -2

minl = MINLOC((/-7,2,-7,5/))   ! returns 1, first
                               ! occurrence of minimum
END
```

See Also

MAXLOC
MINVAL
MAXVAL
FINDLOC

MINVAL

Transformational Intrinsic Function (Generic):
Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

`result = MINVAL (array[,dim] [,mask])`

array	(Input) Must be an array of type integer, real, or character.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be a logical array that is conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If MINVAL(*array*) is specified, the result has a value equal to the minimum value of all the elements in *array*.
- If MINVAL(*array*, MASK= *mask*) is specified, the result has a value equal to the minimum value of the elements in *array* corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.

- If *array* has rank one, MINVAL(*array*, *dim*[, *mask*]) has a value equal to that of MINVAL(*array*[,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of MINVAL(*array*, *dim*, [, *mask*]) is equal to MINVAL(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK = *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

If *array* has size zero or if there are no true elements in *mask*, the result (if *dim* is omitted), or each element in the result array (if *dim* is specified), has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind parameters of *array*.

If *array* is of type character, the comparison is done using the ASCII collating sequence.

Example

The value of MINVAL ((/2, 3, 4/)) is 2 because that is the minimum value in the rank-one array.

The value of MINVAL (B, MASK=B .GT. 0.0) finds the minimum value of the positive elements of B.

C is the array

```
 [ 2  3  4 ]
 [ 5  6  7 ].
```

MINVAL (C, DIM=1) has the value (2, 3, 4). 2 is the minimum value in column 1; 3 is the minimum value in column 2; and so forth.

MINVAL (C, DIM=2) has the value (2, 5). 2 is the minimum value in row 1 and 5 is the minimum value in row 2.

The following shows another example:

```
INTEGER array(2, 3), i(2), minv
INTEGER, ALLOCATABLE :: AR1(:), AR2(:)
array = RESHAPE((/1, 4, 5, 2, 3, 6/), (/2, 3/))
!    array is    1 5 3
!                4 2 6
i = SHAPE(array)    ! i = [2 3]
ALLOCATE(AR1(i(2))) ! dimension AR1 to number of
                    ! elements in dimension 2
                    ! (a column) of array.
ALLOCATE(AR2(i(1))) ! dimension AR2 to number of
                    ! elements in dimension 1
                    ! (a row) of array
minv = MINVAL(array, MASK = array .GT. 4) ! returns 5
AR1 = MINVAL(array, DIM = 1)   ! returns [ 1 2 3 ]
AR2 = MINVAL(array, DIM = 2)   ! returns [ 1 2 ]
END
```

See Also

MAXVAL
MINLOC
MAXLOC

MM_PREFETCH

Intrinsic Subroutine (Generic): *Prefetches data from the specified address on one memory cache line. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL MM_PREFETCH (address[,hint] [,fault] [,exclusive])
```

address

(Input) Is the name of a scalar or array; it can be of any type or rank. It specifies the address of the data on the cache line to prefetch.

hint

(Input; optional) Is an optional default integer constant with one of the following values:

Value	Prefetch Constant	Description
0	FOR_K_PREFETCH_T0	Prefetches into the L1 cache (and the L2 and the L3 cache). Use this for integer data.
1	FOR_K_PREFETCH_T1	Prefetches into the L2 cache (and the L3 cache); floating-point data is used from the L2 cache, not the L1 cache. Use this for real data.
2	FOR_K_PREFETCH_T2	Prefetches into the L2 cache (and the L3 cache); this line will be marked for early displacement. Use this if you are not going to reuse the cache line frequently.
3	FOR_K_PREFETCH_NTA	Prefetches into the L2 cache (but *not* the L3 cache); this line will be marked for early displacement. Use this if you are not going to reuse the cache line.

The preceding constants are defined in file `fordef.for` on Windows* systems and file `fordef.f` on Linux* and macOS* systems.

If *hint* is omitted, 0 is assumed.

fault

(Input; optional) Is an optional default logical constant. If .TRUE. is specified, page faults are allowed to occur, if necessary; if .FALSE. is specified, page faults are not allowed to occur. If *fault* is omitted, .FALSE. is assumed. This argument is currently ignored.

exclusive	(Input; optional) Is an optional default logical constant. If.TRUE. is specified, you get exclusive ownership of the cache line because you intend to assign to it; if .FALSE. is specified, there is no exclusive ownership. If *exclusive* is omitted, .FALSE.is assumed. This argument is currently ignored.

Example

```
subroutine spread_lf (a, b)
PARAMETER (n = 1025)

real*8 a(n,n), b(n,n), c(n)
do j = 1,n
  do i = 1,100
    a(i, j) = b(i-1, j) + b(i+1, j)
    call mm_prefetch (a(i+20, j), 1)
    call mm_prefetch (b(i+21, j), 1)
  enddo
enddo

print *, a(2, 567)

stop
end
```

MOD

Elemental Intrinsic Function (Generic): *Returns the remainder when the first argument is divided by the second argument.*

Syntax

`result = MOD (a, p)`

a	(Input) Must be of type integer or real.
p	(Input)Must have the same type and kind parameters as *a*. It must not have a value of zero.

Results

The result type and kind are the same as *a*. If *p* is not equal to zero, the value of the result is *a* - INT(*a*/ *p*) * *p*. If *p* is equal to zero, the result is undefined.

Specific Name	Argument Type	Result Type
BMOD	INTEGER(1)	INTEGER(1)
IMOD[1]	INTEGER(2)	INTEGER(2)
MOD [2]	INTEGER(4)	INTEGER(4)
KMOD	INTEGER(8)	INTEGER(8)
AMOD [3]	REAL(4)	REAL(4)
DMOD [3,4]	REAL(8)	REAL(8)
QMOD	REAL(16)	REAL(16)

Specific Name	Argument Type	Result Type

[1] Or HMOD.

[2] Or JMOD.

[3] The setting of compiler options specifying real size can affect AMOD and DMOD.

[4] The setting of compiler options specifying double size can affect DMOD.

Example

MOD (7, 3) has the value 1.

MOD (9, -6) has the value 3.

MOD (-9, 6) has the value -3.

The following shows more examples:

```
INTEGER I
REAL R
R = MOD(9.0, 2.0)  ! returns 1.0
I = MOD(18, 5)     ! returns 3
I = MOD(-18, 5)    ! returns -3
I = MOD( 8, 5)     ! returns 3
I = MOD(-8, 5)     ! returns -3
I = MOD( 8,-5)     ! returns 3
I = MOD(-8,-5)     ! returns -3
```

See Also

MODULO

MODIFYMENUFLAGSQQ (W*S)

QuickWin Function: *Modifies a menu item's state.*

Module

USE IFQWIN

Syntax

`result = MODIFYMENUFLAGSQQ (menuID,itemID,flag)`

menuID — (Input) INTEGER(4). Identifies the menu containing the item whose state is to be modified, starting with 1 as the leftmost menu.

itemID — (Input) INTEGER(4). Identifies the menu item whose state is to be modified, starting with 0 as the top item.

flags — (Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see the Results section below). The following constants are available:

- $MENUGRAYED - Disables and grays out the menu item.
- $MENUDISABLED - Disables but does not gray out the menu item.
- $MENUENABLED - Enables the menu item.
- $MENUSEPARATOR - Draws a separator bar.
- $MENUCHECKED - Puts a check by the menu item.
- $MENUUNCHECKED - Removes the check by the menu item.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

Example

```
USE IFQWIN
LOGICAL(4)    result
CHARACTER(20) str
! Append item to the bottom of the first (FILE) menu
str = '&Add to File Menu'C
result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
! Gray out and disable the first two menu items in the
!  first (FILE) menu
result = MODIFYMENUFLAGSQQ (1, 1, $MENUGRAYED)
result = MODIFYMENUFLAGSQQ (1, 2, $MENUGRAYED)
END
```

See Also

APPENDMENUQQ
DELETEMENUQQ
INSERTMENUQQ
MODIFYMENUROUTINEQQ
MODIFYMENUSTRINGQQ

MODIFYMENUROUTINEQQ (W*S)

QuickWin Function: *Changes a menu item's callback routine.*

Module

USE IFQWIN

Syntax

`result = MODIFYMENUROUTINEQQ (menuIdD,itemID,routine)`

menuID — (Input) INTEGER(4). Identifies the menu that contains the item whose callback routine is be changed, starting with 1 as the leftmost menu.

itemID — (Input) INTEGER(4). Identifies the menu item whose callback routine is to be changed, starting with 0 as the top item.

routine — (Input) EXTERNAL. Callback subroutine called if the menu item is selected. All routines take a single LOGICAL parameter that indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- WINPRINT - Prints the program.
- WINSAVE - Saves the program.
- WINEXIT - Terminates the program.
- WINSELECTTEXT - Selects text from the current window.
- WINSELECTGRAPHICS - Selects graphics from the current window.

- WINSELECTALL - Selects the entire contents of the current window.
- WININPUT - Brings to the top the child window requesting input and makes it the current window.
- WINCOPY - Copies the selected text and/or graphics from the current window to the Clipboard.
- WINPASTE - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.
- WINCLEARPASTE - Clears the paste buffer.
- WINSIZETOFIT - Sizes output to fit window.
- WINFULLSCREEN - Displays output in full screen.
- WINSTATE - Toggles between pause and resume states of text output.
- WINCASCADE - Cascades active windows.
- WINTILE - Tiles active windows.
- WINARRANGE - Arranges icons.
- WINSTATUS - Enables a status bar.
- WININDEX - Displays the index for QuickWin help.
- WINUSING - Displays information on how to use Help.
- WINABOUT - Displays information about the current QuickWin application.
- NUL - No callback routine.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

See Also

APPENDMENUQQ
DELETEMENUQQ
INSERTMENUQQ
MODIFYMENUFLAGSQQ
MODIFYMENUSTRINGQQ

MODIFYMENUSTRINGQQ (W*S)

QuickWin Function: *Changes a menu item's text string.*

Module

USE IFQWIN

Syntax

```
result = MODIFYMENUSTRINGQQ (menuID,itemID,text)
```

menuID	(Input) INTEGER(4). Identifies the menu containing the item whose text string is to be changed, starting with 1 as the leftmost item.
itemID	(Input) INTEGER(4). Identifies the menu item whose text string is to be changed, starting with 0 as the top menu item.
text	(Input) Character*(*). Menu item name. Must be a null-terminated C string. For example, words of text'C.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can add access keys in your text strings by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, use "P&rint"C as *text*.

Example

```
USE IFQWIN
LOGICAL(4) result
CHARACTER(25) str
! Append item to the bottom of the first (FILE) menu
str = '&Add to File Menu'C
result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
! Change the name of the first item in the first menu
str ='&Browse'C
result = MODIFYMENUSTRINGQQ (1, 1, str)
END
```

See Also

APPENDMENUQQ
DELETEMENUQQ
INSERTMENUQQ
SETMESSAGEQQ
MODIFYMENUFLAGSQQ
MODIFYMENUROUTINEQQ

MODULE

Statement: *Marks the beginning of a module program unit, which contains specifications and definitions that can be used by one or more program units.*

Syntax

```
MODULE name
   [specification-part]
[CONTAINS
   [module-subprogram
   [module-subprogram]...] ]
END[ MODULE [name]]
```

name
Is the name of the module.

specification-part
Is one or more specification statements, except for the following:

- ENTRY
- FORMAT
- AUTOMATIC (or its equivalent attribute)
- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- Statement functions

An automatic object must not appear in a specification statement.

module-subprogram — Is a function or subroutine subprogram that defines the module procedure. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.

A module subprogram can contain internal procedures.

Description

If a name follows the END statement, it must be the same as the name specified in the MODULE statement.

The module name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A module is host to any module procedures it contains, and entities in the module are accessible to the module procedures through host association.

A module must not reference itself (either directly or indirectly).

You can use the PRIVATE attribute to restrict access to procedures or variables within a module.

Although ENTRY statements, FORMAT statements, and statement functions are not allowed in the specification part of a module, they are allowed in the specification part of a module subprogram.

The following rules also apply to modules:

- The specification part of a module must not contain IMPORT, ENTRY, FORMAT, executable, or statement function statements.
- A variable, common block, or procedure pointer declared in a submodule implicitly has the SAVE attribute, which may be confirmed by explicit specification.
- If a specification or constant expression in the *specification-part* of a module includes a reference to a generic entity, there must be no specific procedures of the generic entity defined in the submodule subsequent to the specification or constant expression.

Any executable statements in a module can only be specified in a module subprogram.

A module can contain one or more procedure interface blocks, which let you specify an explicit interface for an external subprogram or dummy subprogram.

A module can be extended by one or more program units called submodules. A submodule can in turn be extended by one or more submodules.

Example

The following example shows a simple module that can be used to provide global data:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5)
END MODULE MOD_A
...
SUBROUTINE SUB_Z
  USE MOD_A               ! Makes scalar variables B and C, and array
  ...                     !   E available to this subroutine
END SUBROUTINE SUB_Z
```

The following example shows a module procedure:

```
MODULE RESULTS
...
CONTAINS
  FUNCTION MOD_RESULTS(X,Y)  ! A module procedure
```

```
  ...
  END FUNCTION MOD_RESULTS
END MODULE RESULTS
```

The following example shows a module containing a derived type:

```
MODULE EMPLOYEE_DATA
  TYPE EMPLOYEE
    INTEGER ID
    CHARACTER(LEN=40) NAME
  END TYPE EMPLOYEE
END MODULE
```

The following example shows a module containing an interface block:

```
MODULE ARRAY_CALCULATOR
  INTERFACE
    FUNCTION CALC_AVERAGE(D)
      REAL :: CALC_AVERAGE
      REAL, INTENT(IN) :: D(:)
    END FUNCTION
  END INTERFACE
END MODULE ARRAY_CALCULATOR
```

The following example shows a derived-type definition that is public with components that are private:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

This design allows you to change components of a type without affecting other program units that use the module.

If a derived type is needed in more than one program unit, the definition should be placed in a module and accessed by a USE statement whenever it is needed, as follows:

```
MODULE STUDENTS
  TYPE STUDENT_RECORD
  ...
  END TYPE
CONTAINS
  SUBROUTINE COURSE_GRADE(...)
  TYPE(STUDENT_RECORD) NAME
  ...
  END SUBROUTINE
END MODULE STUDENTS
...

PROGRAM SENIOR_CLASS
  USE STUDENTS
  TYPE(STUDENT_RECORD) ID
  ...
END PROGRAM
```

Program SENIOR_CLASS has access to type STUDENT_RECORD, because it uses module STUDENTS. Module procedure COURSE_GRADE also has access to type STUDENT_RECORD, because the derived-type definition appears in its host.

See Also

SUBMODULE
PUBLIC
PRIVATE
USE
Procedure Interfaces
Program Units and Procedures
PROTECTED Attribute and Statement

MODULE FUNCTION

Statement: *Indicates a separate module procedure.*

Example

```
submodule (M) A
contains
  real module function foo (arg) result(res)
    type(tt), intent(in) :: arg
    res = arg%r
  end function foo
end submodule A
```

See Also

Separate Module Procedures

MODULE PROCEDURE

Statement: *Identifies module procedures in an interface block that specifies a generic name.*

Example

```
!A program that changes non-default integers and reals
! into default integers and reals
   PROGRAM CHANGE_KIND
   USE Module1
   integer(2) in
   integer indef
   indef = DEFAULT(in)
   END PROGRAM

! procedures sub1 and sub2 defined as follows:
   MODULE Module1
    INTERFACE DEFAULT
      MODULE PROCEDURE Sub1, Sub2
    END INTERFACE
    CONTAINS
      FUNCTION Sub1(y)
        REAL(8) y
        sub1 = REAL(y)
      END FUNCTION
      FUNCTION Sub2(z)
        INTEGER Sub2
```

```
        INTEGER(2) z
        sub2 = INT(z)
      END FUNCTION
    END MODULE
```

See Also

INTERFACE
MODULE
Modules and Module Procedures
PROCEDURE

MODULE SUBROUTINE

Statement: *Indicates a separate module procedure.*

Example

```
submodule (M) A
contains
  real module subroutine FOO (arg)
    type(tt), intent(in) :: arg
    arg%r =1
  end subroutine FOO
end submodule A
```

See Also

Separate Module Procedures

MODULO

Elemental Intrinsic Function (Generic): *Returns the modulo of the arguments.*

Syntax

`result = MODULO (a,p)`

a	(Input) Must be of type integer or real.
p	(Input) Must have the same type and kind parameters as *a*. It must not have a value of zero.

Results

The result type is the same *a*. The result value depends on the type of *a*, as follows:

- If *a* is of type integer and *p* is not equal to zero, the value of the result is *a* - FLOOR(REAL(*a*) / REAL(*p*)) * *p*, that is, the *result* has a value such that *a* = *q* * *p* + *result* where *q* is an integer. The following also applies:
  - If *p* > 0, then 0 <= *result* < *p*
  - If *p* < 0, then *p* < *result* <= 0
- If *a* is of type real and *p* is not equal to zero, the value of the result is *a* - FLOOR(*a*/ *p*) * *p*.

If *p* is equal to zero (regardless of the type of *a*), the result is undefined.

Example

MODULO (7, 3) has the value 1.

MODULO (9, -6) has the value -3.

MODULO (-9, 6) has the value 3.

The following shows more examples:

```
INTEGER I
REAL R
I= MODULO(8, 5)         ! returns 3          Note: q=1
I= MODULO(-8, 5)        ! returns 2          Note: q=-2
I= MODULO(8, -5)        ! returns -2         Note: q=-2
I= MODULO(-8,-5)        ! returns -3         Note: q=1
R= MODULO(7.285, 2.35)  ! returns 0.2350001  Note: q=3
R= MODULO(7.285, -2.35) ! returns -2.115     Note: q=-4
```

See Also

MOD

MOVE_ALLOC

Intrinsic Subroutine (Generic): *Moves an allocation from one allocatable object to another. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

CALL MOVE_ALLOC (*from*,*to* [, *stat*, *errmsg*])

from	(Input; output) Can be of any type, type parameters, corank, and rank; it must be allocatable.
to	(Output) Must be type compatible with `from` and have the same rank and corank; it must be allocatable. `to` must be polymorphic if `from` is polymorphic. Each nondeferred type parameter of the declared type of `to` must have the same value as the corresponding type parameter of the declared type of `from`. For more information about type compatibility, see the description in CLASS.
stat	(Output, optional) Must be a non-coindexed integer scalar with a decimal exponent range of at least 4 (KIND=2) or greater.
errmsg	(Input; output; optional) Must be a non-coindexed default character variable.

When the execution of MOVE_ALLOC is successful, or if STAT is assigned the value STAT_FAILED_IMAGE, the following occurs:

- If `to` is currently allocated, it is deallocated.
- If `from` is allocated, `to` becomes allocated with the same type, type parameters, array bounds, and values as from.
- `from` is deallocated.

If `to` has the TARGET attribute, any pointer associated with `from` at the time of the call to MOVE_ALLOC becomes correspondingly associated with `to`. If `to` does not have the TARGET attribute, the pointer association status of any pointer associated with `from` on entry becomes undefined.

During execution of MOVE_ALLOC, the internal descriptor contents are copied from `from` to `to`, so that the storage pointed to by `to` is the storage that `from` used to point to.

Typically, MOVE_ALLOC is used to provide an efficient way to reallocate a variable to a larger size without copying the data twice.

A reference to MOVE_ALLOC that has a coarray from argument has an implicit synchronization of all active images of the current team. No image proceeds until all active images of the current team have completed execution of the reference. If the current team contains any images that have stopped or failed, an error condition occurs.

If the procedure reference is successful, *stat* becomes defined with the value zero.

If an error condition occurs and *stat* is not specified, error termination is initiated. If *stat* is present and the current team contains an image that has stopped, *stat* becomes defined with the value STAT_STOPPED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. Otherwise, if *stat* is present, the current team contains a failed image, and no other error condition occurs, *stat* becomes defined with the value STAT_FAILED_IMAGE from the intrinsic module ISO_FORTRAN_ENV. Otherwise, *stat* becomes defined with a positive integer value different from STAT_STOPPED_IMAGE and STAT_FAILED_IMAGE.

The definition status of *errmsg*, if present, remains unchanged if the execution of the reference is successful. If *errmsg* is present and an error condition occurs, *errmsg* becomes defined with an explanatory message describing the error.

Example

The following shows an example of how to increase the allocated size of X and keep the old values with only one copy of the old values. Using only assignment, a temporary variable named Y will be allocated and X assigned to Y. Then X will be deallocated and reallocated to be twice the size; then Y will be assigned to the first half of X. Finally, the temporary Y is deallocated.

```
! This program uses MOVE_ALLOC to make an allocated array X bigger and
! keep the old values of X in the variable X. Only one copy of the old values
! of X is needed.
    integer :: I, N = 2
    real, allocatable :: X(:), Y(:)
    allocate (X(N), Y(2*N))         ! Y is twice as big as X
    X = (/(I,I=1,N)/)       ! put "old values" into X
    Y = -1                  ! put different "old values" into Y
    print *, ' allocated of X is ', allocated (X)
    print *, ' allocated of Y is ', allocated (Y)
    print *, ' old X is ', X
    print *, ' old Y is ', Y
    Y (1:N) = X             ! copy all of X into the first half of Y
                            ! this is the only copying of values required
    print *, ' new Y is ', Y
    call move_alloc (Y, X)  ! X is now twice as big as it was, Y is
                            ! deallocated, the values were not copied from Y to X
    print *, ' allocated of X is ', allocated (X)
    print *, ' allocated of Y is ', allocated (Y)
    print *, ' new X is ', X
    end
```

The following shows the output for the above example:

```
 allocated of X is  T
 allocated of Y is  T
 old X is    1.000000       2.000000
 old Y is   -1.000000      -1.000000      -1.000000      -1.000000
 new Y is    1.000000       2.000000      -1.000000      -1.000000
 allocated of X is  T
 allocated of Y is  F
 new X is    1.000000       2.000000      -1.000000      -1.000000
```

See Also

ISO_FORTRAN_ENV Module

MOVETO, MOVETO_W (W*S)

Graphics Subroutines: *Move the current graphics position to a specified point. No drawing occurs.*

Module

USE IFQWIN

Syntax

CALL MOVETO (*x*,*y*,*s*)

CALL MOVETO_W (*wx*,*wy*,*ws*)

x, y — (Input) INTEGER(2). Viewport coordinates of the new graphics position.

s — (Output) Derived type `xycoord`. Viewport coordinates of the previous graphics position. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord  ! x coordinate
  INTEGER(2) ycoord  ! y coordinate
END TYPE xycoord
```

wx, wy — (Input) REAL(8). Window coordinates of the new graphics position.

ws — (Output) Derived type `wxycoord`. Viewport coordinates of the previous graphics position. The derived type `wxycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  DOUBLE PRECISION WX,WY
  STRUCTURE/WXYCOORD/
    DOUBLE PRECISION WX
    DOUBLE PRECISION WY
  END STRUCTURE
END TYPE wxycoord
```

MOVETO sets the current graphics position to the viewport coordinate (*x*, *y*). MOVETO_W sets the current graphics position to the window coordinate (*wx*, *wy*).

MOVETO and MOVETO_W assign the coordinates of the previous position to *s*.

Example

```
! Build as QuickWin or Standard Graphics ap.
USE IFQWIN
INTEGER(2) status, x, y
INTEGER(4) result
TYPE (xycoord) xy
RESULT = SETCOLORRGB(Z'FF0000') ! blue
x = 60
! Draw a series of lines DO y = 50, 92, 3
  CALL MOVETO(x, y, xy)
  status = LINETO(INT2(x + 20), y)
END DO
END
```

See Also

GETCURRENTPOSITION
LINETO
OUTGTEXT

MVBITS

Elemental Intrinsic Subroutine (Generic): *Copies a sequence of bits (a bit field) from one location to another. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL MVBITS (from,frompos,len,to,topos)`

from	(Input) Integer. Can be of any integer type. It represents the location from which a bit field is transferred.
frompos	(Input) Can be of any integer type; it must not be negative. It identifies the first bit position in the field transferred from *from*. *frompos* + *len* must be less than or equal to BIT_SIZE(*from*).
len	(Input) Can be of any integer type; it must not be negative. It identifies the length of the field transferred from *from*.
to	(Input; output) Can be of any integer type, but must have the same kind parameter as *from*. It represents the location to which a bit field is transferred. *to* is set by copying the sequence of bits of length *len*, starting at position *frompos* of *from* to position *topos* of *to*. No other bits of *to* are altered.
topos	(Input) Can be of any integer type; it must not be negative. It identifies the starting position (within *to*) for the bits being transferred. *topos* + *len* must be less than or equal to BIT_SIZE(*to*).

For more information on bit functions, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

You can also use the following specific routines:

BMVBITS	Arguments *from* and *to* must be INTEGER(1).
HMVBITS	Arguments *from* and *to* must be INTEGER(2).
IMVBITS	Arguments *from* and *to* must be INTEGER(2).
JMVBITS	Arguments *from* and *to* must be INTEGER(4).
KMVBITS	Arguments *from* and *to* must be INTEGER(8).

Example

If TO has the initial value of 6, its value after a call to MVBITS(7, 2, 2, TO, 0) is 5.

The following shows another example:

```
INTEGER(1) :: from = 13  ! 00001101
INTEGER(1) :: to = 6     ! 00000110
CALL MVBITS(from, 2, 2, to, 0) ! returns to = 00000111
END
```

See Also

BIT_SIZE
IBCLR
IBSET
ISHFT
ISHFTC

NAMELIST

Statement: *Associates a name with a list of variables. This group name can be referenced in some input/output operations.*

Syntax

NAMELIST */group/ var-list*[[,] */group/ var-list*]...

group	Is the name of the group.
var-list	Is a list of variables (separated by commas) that are to be associated with the preceding group name. The variables can be of any data type.

Description

The namelist group name is used by namelist I/O statements instead of an I/O list. The unique group name identifies a list whose entities can be modified or transferred.

A variable can appear in more than one namelist group.

Each variable in *var-list* must be accessed by use or host association, or it must have its type, type parameters, and shape explicitly or implicitly specified in the same scoping unit. If the variable is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

You cannot specify an assumed-size array in a namelist group.

Only the variables specified in the namelist can be read or written in namelist I/O. It is not necessary for the input records in a namelist input statement to define every variable in the associated namelist.

The order of variables in the namelist controls the order in which the values appear on namelist output. Input of namelist values can be in any order.

If the group name has the PUBLIC attribute, no item in the variable list can have the PRIVATE attribute.

The group name can be specified in more than one NAMELIST statement in a scoping unit. The variable list following each successive appearance of the group name is treated as a continuation of the list for that group name.

Example

In the following example, D and E are added to the variables A, B, and C for group name LIST:

```
NAMELIST /LIST/ A, B, C

NAMELIST /LIST/ D, E
```

In the following example, two group names are defined:

```
CHARACTER*30 NAME(25)
NAMELIST /INPUT/ NAME, GRADE, DATE /OUTPUT/ TOTAL, NAME
```

Group name INPUT contains variables NAME, GRADE, and DATE. Group name OUTPUT contains variables TOTAL and NAME.

The following shows another example:

```
NAMELIST /example/ i1, l1, r4, r8, z8, z16, c1, c10, iarray
! The corresponding input statements could be:
&example
i1  = 11
l1  = .TRUE.
r4  = 24.0
r8  = 28.0d0
z8  = (38.0, 0.0)
z16 = (316.0d0, 0.0d0)
c1  = 'A'
c10 = 'abcdefghij'
iarray(8) = 41, 42, 43
/
```

A sample program, NAMELIST.F90, is included in the `<install-dir>/samples` subdirectory.

See Also

READ
WRITE
Namelist Specifier
Namelist Input
Namelist Output

NARGS

Inquiry Intrinsic Function (Specific): *Returns the total number of command-line arguments, including the command. This function cannot be passed as an actual argument.*

Syntax

```
result = NARGS( )
```

Results

The result type is INTEGER(4). The result is the number of command-line arguments, including the command. For example, NARGS returns 4 for the command-line invocation of `PROG1 -g -c -a`.

Example

```
   INTEGER(2) result
   result = RUNQQ('myprog', '-c -r')
   END

!  MYPROG.F90 responds to command switches -r, -c,
!  and/or -d
INTEGER(4) count, num, i, status
CHARACTER(80) buf
REAL r1 / 0.0 /
```

```
COMPLEX c1 / (0.0,0.0) /
REAL(8) d1 / 0.0 /

num = 5
count = NARGS()
DO i = 1, count-1
   CALL GETARG(i, buf, status)
   IF (status .lt. 0) THEN
     WRITE (*,*) 'GETARG error - exiting'
     EXIT
   END IF
   IF (buf(2:status) .EQ.'r') THEN
     r1 = REAL(num)
     WRITE (*,*) 'r1 = ',r1
   ELSE IF (buf(2:status) .EQ.'c') THEN
     c1 = CMPLX(num)
     WRITE (*,*) 'c1 = ', c1
   ELSE IF (buf(2:status) .EQ.'d') THEN
     d1 = DBLE(num)
     WRITE (*,*) 'd1 = ', d1
   ELSE
     WRITE(*,*) 'Invalid command switch: ', buf (1:status)
   END IF
END DO
END
```

See Also

GETARG
IARGC
COMMAND_ARGUMENT_COUNT
GET_COMMAND
GET_COMMAND_ARGUMENT

NEAREST

Elemental Intrinsic Function (Generic): *Returns the nearest different number (representable on the processor) in a given direction.*

Syntax

`result = NEAREST (x, s)`

x	(Input) Must be of type real.
s	(Input) Must be of type real and nonzero.

Results

The result type and kind are the same as *x*. The result has a value equal to the machine representable number that is different from and nearest to *x*, in the direction of the infinity with the same sign as *s*.

Example

If 3.0 and 2.0 are REAL(4) values, NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$, which equals approximately 3.0000002. (For more information on the model for REAL(4), see Model for Real Data.

The following shows another example:

```
REAL(4) r1
REAL(8) r2, result
r1 = 3.0
result = NEAREST (r1, -2.0)
WRITE(*,*) result           ! writes 2.999999761581421

! When finding nearest to REAL(8), can't see
! the difference unless output in HEX
r2 = 111502.07D0
result = NEAREST(r2, 2.0)
WRITE(*,'(1x,Z16)') result  ! writes 40FB38E11EB851ED
result = NEAREST(r2, -2.0)
WRITE(*,'(1x,Z16)') result  ! writes 40FB38E11EB851EB
END
```

See Also

EPSILON

NEW_LINE

Inquiry Intrinsic Function (Generic): *Returns a new line character.*

Syntax

`result = NEW_LINE(a)`

a	(Input) Must be of type default character. It may be a scalar or an array.

Results

The result is a character scalar of length one with the same kind type parameter as a.

The result value is the ASCII newline character ACHAR(10).

NINT

Elemental Intrinsic Function (Generic): *Returns the nearest integer to the argument.*

Syntax

`result = NINT (a[,kind])`

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *a* is greater than zero, NINT(*a*) has the value INT(*a*+ 0.5); if *a* is less than or equal to zero, NINT(*a*) has the value INT(*a*- 0.5).

Specific Name	Argument Type	Result Type
ININT	REAL(4)	INTEGER(2)
NINT [1, 2]	REAL(4)	INTEGER(4)
KNINT	REAL(4)	INTEGER(8)
IIDNNT	REAL(8)	INTEGER(2)
IDNINT [2, 3]	REAL(8)	INTEGER(4)
KIDNNT	REAL(8)	INTEGER(8)
IIQNNT	REAL(16)	INTEGER(2)
IQNINT[2, 4]	REAL(16)	INTEGER(4)
KIQNNT	REAL(16)	INTEGER(8)

[1]Or JNINT.

[2]The setting of compiler options specifying integer size can affect NINT, IDNINT, and IQNINT.

[3]Or JIDNNT. For compatibility with older versions of Fortran, IDNINT can also be specified as a generic function.

[4]Or JIQNNT. For compatibility with older versions of Fortran, IQNINT can also be specified as a generic function.

Example

NINT (3.879) has the value 4.

NINT (-2.789) has the value -3.

The following shows another example:

```
INTEGER(4) i1, i2
i1 = NINT(2.783) ! returns 3
i2 = IDNINT(-2.783D0) ! returns -3
```

See Also

ANINT
INT

NLSEnumCodepages (W*S)

NLS Function: *Returns an array containing the codepages supported by the system, with each array element describing one valid codepage.*

Module

USE IFNLS

Syntax

ptr=> NLSEnumCodepages( )

Results

The result is a pointer to an array of codepages, with each element describing one supported codepage.

NOTE
After use, the pointer returned by NLSEnumCodepages should be deallocated with the DEALLOCATE statement.

See Also
NLSEnumLocales
DEALLOCATE

NLSEnumLocales (W*S)
NLS Function: *Returns an array containing the language and country combinations supported by the system, in which each array element describes one valid combination.*

Module
USE IFNLS

Syntax
```
ptr=> NLSEnumLocales( )
```

Results
The result is a pointer to an array of locales, in which each array element describes one supported language and country combination. Each element has the following structure:

```
TYPE NLS$EnumLocale
  CHARACTER*(NLS$MaxLanguageLen)  Language
  CHARACTER*(NLS$MaxCountryLen)   Country
  INTEGER(4)                      DefaultWindowsCodepage
  INTEGER(4)                      DefaultConsoleCodepage
END TYPE
```

If the application is a Windows or QuickWin application, NLS$DefaultWindowsCodepage is the codepage used by default for the given language and country combination. If the application is a console application, NLS $DefaultConsoleCodepage is the codepage used by default for the given language and country combination.

NOTE
After use, the pointer returned by NLSEnumLocales should be deallocated with the DEALLOCATE statement.

See Also
NLSEnumCodepages
DEALLOCATE

NLSFormatCurrency (W*S)
NLS Function: *Returns a correctly formatted currency string for the current locale.*

Module
USE IFNLS

Syntax

```
result = NLSFormatCurrency (outstr,instr[,flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted currency for the current locale. If *outstr* is longer than the formatted currency, it is blank-padded.
intstr	(Input) Character*(*). Number string to be formatted. Can contain only the characters '0' through '9', one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.
flags	(Input; optional) INTEGER(4). If specified, modifies the currency conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are: • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr*(bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *instr* has an illegal value

Example

```
USE IFNLS
CHARACTER(40) str
INTEGER(4) i
i = NLSFormatCurrency(str, "1.23")
print *, str                              ! prints $1.23
i = NLSFormatCurrency(str, "1000000.99")
print *, str                              ! prints $1,000,000.99
i = NLSSetLocale("Spanish", "Spain")
i = NLSFormatCurrency(str, "1.23")
print *, str                              ! prints 1 Pts
i = NLSFormatCurrency(str, "1000000.99")
print *, str                              ! prints 1.000.001 Pts
```

See Also

NLSFormatNumber
NLSFormatDate
NLSFormatTime

NLSFormatDate (W*S)

NLS Function: *Returns a correctly formatted string containing the date for the current locale.*

Module

USE IFNLS

Syntax

```
result = NLSFormatDate (outstr [, intime] [, flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted date for the current locale. If *outstr* is longer than the formatted date, it is blank-padded.
intime	(Input; optional) INTEGER(4). If specified, date to be formatted for the current locale. Must be an integer date such as the packed time created with PACKTIMEQQ. If you omit *intime*, the current system date is formatted and returned in *outstr*.
flags	(Input; optional) INTEGER(4). If specified, modifies the date conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are: • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides • NLS$UseAltCalendar - Use the locale's alternate calendar • NLS$LongDate - Use local long date format • NLS$ShortDate - Use local short date format

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *intime* has an illegal value

Example

```
USE IFNLS
INTEGER(4) i
CHARACTER(40) str
i = NLSFORMATDATE(str, FLAGS=NLS$NORMAL)           ! 8/1/10
i = NLSFORMATDATE(str, FLAGS=NLS$USEALTCALENDAR)   ! 8/1/10
i = NLSFORMATDATE(str, FLAGS=NLS$LONGDATE)         ! Sunday, August 1, 2010
i = NLSFORMATDATE(str, FLAGS=NLS$SHORTDATE)        ! 8/1/10
END
```

See Also

NLSFormatTime
NLSFormatCurrency
NLSFormatNumber

NLSFormatNumber (W*S)

NLS Function: *Returns a correctly formatted number string for the current locale.*

Module

USE IFNLS

Syntax

```
result = NLSFormatNumber (outstr, instr [, flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted number for the current locale. If *outstr* is longer than the formatted number, it is padded with blanks.
instr	(Input) Character*(*). Number string to be formatted. Can only contain the characters '0' through '9', one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.
flags	(Input; optional) INTEGER(4). If specified, modifies the number conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are: • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *instr* has an illegal value

Example

```
USE IFNLS
CHARACTER(40) str
INTEGER(4) i
i = NLSFormatNumber(str, "1.23")
print *, str                            ! prints 1.23
i = NLSFormatNumber(str, "1000000.99")
print *, str                            ! prints 1,000,000.99
i = NLSSetLocale("Spanish", "Spain")
i = NLSFormatNumber(str, "1.23")
print *, str                            ! prints 1,23
i = NLSFormatNumber(str, "1000000.99")
print *, str                            ! prints 1.000.000,99
END
```

See Also

NLSFormatTime
NLSFormatCurrency
NLSFormatDate

NLSFormatTime (W*S)

NLS Function: *Returns a correctly formatted string containing the time for the current locale.*

Module

USE IFNLS

Syntax

```
result = NLSFormatTime (outstr [, intime] [, flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted time for the current locale. If *outstr* is longer than the formatted time, it is blank-padded.
intime	(Input; optional) INTEGER(4). If specified, time to be formatted for the current locale. Must be an integer time such as the packed time created with PACKTIMEQQ. If you omit *intime*, the current system time is formatted and returned in outstr.
flags	(Input; optional) INTEGER(4). If specified, modifies the time conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are: • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides • NLS$NoMinutesOrSeconds - Do not return minutes or seconds • NLS$NoSeconds - Do not return seconds • NLS$NoTimeMarker - Do not add a time marker string • NLS$Force24HourFormat - Return string in 24 hour format

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *intime* has an illegal value

Example

```
USE IFNLS
INTEGER(4) i
CHARACTER(20) str
i = NLSFORMATTIME(str, FLAGS=NLS$NORMAL)              ! 11:38:28 PM
i = NLSFORMATTIME(str, FLAGS=NLS$NOMINUTESORSECONDS)  ! 11 PM
i = NLSFORMATTIME(str, FLAGS=NLS$NOTIMEMARKER)        ! 11:38:28 PM
i = NLSFORMATTIME(str, FLAGS=IOR(NLS$FORCE24HOURFORMAT,               &
&                      NLS$NOSECONDS))          ! 23:38 PM
END
```

See Also

NLSFormatCurrency
NLSFormatDate
NLSFormatNumber

NLSGetEnvironmentCodepage (W*S)

NLS Function: *Returns the codepage number for the system (Window) codepage or the console codepage.*

Module

USE IFNLS

Syntax

```
result = NLSGetEnvironmentCodepage (flags)
```

flags	(Input) INTEGER(4). Tells the function which codepage number to return. Available values (defined in IFNLS.F90) are: • NLS$ConsoleEnvironmentCodepage - Gets the codepage for the console • NLS$WindowsEnvironmentCodepage - Gets the current Windows codepage

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, it returns one of the following error codes:

- NLS$ErrorInvalidFlags - *flags* has an illegal value.
- NLS$ErrorNoConsole - There is no console associated with the given application. So, operations with the console codepage are not possible.

See Also

NLSSetEnvironmentCodepage

NLSGetLocale (W*S)

NLS Subroutine: *Returns the current language, country, or codepage.*

Module

USE IFNLS

Syntax

CALL NLSGetLocale ([*language*] [,*country*] [,*codepage*])

language	(Output; optional) Character*(*). Current language.
country	(Output; optional) Character*(*). Current country.
codepage	(Output; optional) INTEGER(4). Current codepage.

NLSGetLocale returns a valid codepage in *codepage*. It does not return one of the NLS$... symbolic constants that can be used with NLSSetLocale.

Example

```
USE IFNLS
CHARACTER(50) cntry, lang
INTEGER(4)    code
CALL NLSGetLocale (lang, cntry, code)    ! get all three
CALL NLSGetLocale (CODEPAGE = code)      ! get the codepage
CALL NLSGetLocale (COUNTRY = cntry, CODEPAGE =code) ! get country
                                                    ! and codepage
```

See Also

NLSSetLocale

NLSGetLocaleInfo (W*S)

NLS Function: *Returns information about the current locale.*

Module

USE IFNLS

Syntax

`result = NLSGetLocaleInfo (type,outstr)`

type	(Input) INTEGER(4). NLS parameter requested. A list of parameter names is provided in NLS LocaleInfo Parameters.
outstr	(Output) Character*(*). Parameter setting for the current locale. All parameter settings placed in *outstr* are character strings, even numbers. If a parameter setting is numeric, the ASCII representation of the number is used. If the requested parameter is a date or time string, an explanation of how to interpret the format in *outstr* is provided in NLS Date and Time Format.

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr* if successful, or if *outstr* has 0 length, the number of characters required to hold the requested information. Otherwise, the result is one of the following error codes (defined in `IFNLS.F90`):

- NLS$ErrorInvalidInput - *type* has an illegal value.
- NLS$ErrorInsufficientBuffer - The *outstr* buffer was too small, but was not 0 (so that the needed size would be returned).

The NLS$LI parameters are used for the argument *type* and select the locale information returned by NLSGetLocaleInfo in *outstr*. You can perform an inclusive OR with NLS$NoUserOverride and any NLS$LI parameter. This causes NLSGetLocaleInfo to bypass any user overrides and always return the system default value.

The following table lists and describes the NLS$LI parameters in alphabetical order.

NLS LocaleInfo Parameters

Parameter	Description
NLS$LI_ICALENDARTYPE	Specifies which type of calendar is currently being used: 1 - Gregorian (as in United States) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China 5 - Tangun Era (Korea)
NLS$LI_ICENTURY	Specifies whether to use full 4-digit century for the short date only: 0 - Two-digit year 1 - Full century
NLS$LI_ICOUNTRY	The country code, based on international phone codes, also referred to as IBM country codes.
NLS$LI_ICURRDIGITS	Number of decimal digits for the local monetary format.
NLS$LI_ICURRENCY	Determines how positive currency is represented:

Parameter	Description
	0 - Puts currency symbol in front with no separation: $1.1 1 - Puts currency symbol in back with no separation: 1.1$ 2 - Puts currency symbol in front with single space after: $ 1.1 3 - Puts currency symbol in back with single space before: 1.1 $
NLS$LI_IDATE	Short Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day
NLS$LI_IDAYLZERO	Specifies whether to use leading zeros in day fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_IDEFAULTANSICODEPAGE	ANSI code page associated with this locale.
NLS$LI_IDEFAULTCOUNTRY	Country code for the principal country in this locale. This is provided so that partially specified locales can be completed with default values.
NLS$LI_IDEFAULTLANGUAGE	Language ID for the principal language spoken in this locale. This is provided so that partially specified locales can be completed with default values.
NLS$LI_IDEFAULTOEMCODEPAGE	OEM code page associated with the locale.
NLS$LI_IDIGITS	The number of decimal digits.
NLS$LI_IFIRSTDAYOFWEEK	Specifies which day is considered first in a week: 0 - SDAYNAME1 1 - SDAYNAME2 2 - SDAYNAME3 3 - SDAYNAME4 4 - SDAYNAME5 5 - SDAYNAME6 6 - SDAYNAME7
NLS$LI_IFIRSTWEEKOFYEAR	Specifies which week of the year is considered first: 0 - Week containing 1/1 1 - First full week following 1/1 2 - First week containing at least 4 days
NLS$LI_IINTLCURRDIGITS	Number of decimal digits for the international monetary format.

Parameter	Description
NLS$LI_ILANGUAGE	An ID indicating the language.
NLS$LI_ILDATE	Long Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day
NLS$LI_ILZERO	Determines whether to use leading zeros in decimal fields: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_IMEASURE	This value is 0 if the metric system (S.I.) is used and 1 for the U.S. system of measurements.
NLS$LI_IMONLZERO	Specifies whether to use leading zeros in month fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_INEGCURR	Determines how negative currency is represented: 0 ($1.1) 1 -$1.1 2 $-1.1 3 $1.1- 4 (1.1$) 5 -1.1$ 6 1.1-$ 7 1.1$- 8 -1.1 $ (space before $) 9 -$ 1.1 (space after $) 10 1.1 $- (space before $) 11 $ 1.1- (space after $) 12 $ -1.1 (space after $) 13 1.1- $ (space before $) 14 ($ 1.1) (space after $) 15 (1.1 $) (space before $)
NLS$LI_INEGNUMBER	Determines how negative numbers are represented: 0 - Puts negative numbers in parentheses: (1.1) 1 - Puts a minus sign in front: -1.1 2 - Puts a minus sign followed by a space in front: - 1.1 3 - Puts a minus sign after: 1.1-

Parameter	Description
	4 - Puts a space then a minus sign after: 1.1 -
NLS$LI_INEGSEPBYSPACE	1 if the monetary symbol is separated by a space from a negative amount; otherwise, 0.
NLS$LI_INEGSIGNPOSN	Determines the formatting index for negative values. Same values as for NLS$LI_IPOSSIGNPOSN.
NLS$LI_INEGSYMPRECEDES	1 if the monetary symbol precedes, 0 if it follows a negative amount.
NLS$LI_IOPTIONALCALENDAR	Specifies which additional calendar types are valid and available for this locale. This can be a null separated list of all valid optional calendars: 0 - No additional types valid 1 - Gregorian (localized) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China 5 - Tangun Era (Korea)
NLS$LI_IPOSSEPBYSPACE	1 if the monetary symbol is separated by a space from a positive amount; otherwise, 0.
NLS$LI_IPOSSIGNPOSN	Determines the formatting index for positive values: 0 - Parenthesis surround the amount and the monetary symbol 1 - The sign string precedes the amount and the monetary symbol 2 - The sign string follows the amount and the monetary symbol 3 - The sign string immediately precedes the monetary symbol 4 - The sign string immediately follows the monetary symbol
NLS$LI_IPOSSYMPRECEDES	1 if the monetary symbol precedes, 0 if it follows a positive amount.
NLS$LI_ITIME	Time format: 0 - Use 12-hour format 1 - Use 24-hour format
NLS$LI_ITLZERO	Determines whether to use leading zeros in time fields: 0 - Use no leading zeros 1 - Use leading zeros for hours
NLS$LI_S1159	String for the AM designator.
NLS$LI_S2359	String for the PM designator.

Parameter	Description
NLS$LI_SABBREVCTRYNAME	The abbreviated name of the country as per ISO Standard 3166.
NLS$LI_SABBREVDAYNAME1 - NLS$LI_SABBREVDAYNAME7	Native abbreviated name for each day of the week. 1 = Mon, 2 = Tue, etc.
NLS$LI_SABBREVLANGNAME	The abbreviated name of the language, created by taking the 2-letter language abbreviation as found in ISO Standard 639 and adding a third letter as appropriate to indicate the sublanguage.
NLS$LI_SABBREVMONTHNAME1 - NLS$LI_SABBREVMONTHNAME13	Native abbreviated name for each month. 1 = Jan, 2 = Feb, etc. 13 = the 13th month, if it exists in the locale.
NLS$LI_SCOUNTRY	The full localized name of the country.
NLS$LI_SCURRENCY	The string used as the local monetary symbol. Cannot be set to digits 0-9.
NLS$LI_SDATE	Character(s) for the date separator. Cannot be set to digits 0-9.
NLS$LI_SDAYNAME1 - NLS$LI_SDAYNAME7	Native name for each day of the week. 1 = Monday, 2 = Tuesday, etc.
NLS$LI_SDECIMAL	The character(s) used as decimal separator. This is restricted such that it cannot be set to digits 0 - 9.
NLS$LI_SENGCOUNTRY	The full English name of the country. This will always be restricted to characters that map into the ASCII 127 character subset.
NLS$LI_SENGLANGUAGE	The full English name of the language from the ISO Standard 639. This will always be restricted to characters that map into the ASCII 127 character subset.
NLS$LI_SGROUPING	Sizes for each group of digits to the left of the decimal. An explicit size is needed for each group; sizes are separated by semicolons. If the last value is 0 the preceding value is repeated. To group thousands, specify "3;0".
NLS$LI_SINTLSYMBOL	Three characters of the International monetary symbol specified in ISO 4217 "Codes for the Representation of Currencies and Funds", followed by the character separating this string from the amount.
NLS$LI_SLANGUAGE	The full localized name of the language.
NLS$LI_SLIST	Character(s) used to separate list items, for example, comma in many locales.
NLS$LI_SLONGDATE	Long Date formatting string for this locale. The string returned may contain a string within single quotes (' '). Any characters within single quotes should be left as is. The d, M and y should have the day, month, and year substituted, respectively.

Parameter	Description
NLS$LI_SMONDECIMALSEP	The character(s) used as monetary decimal separator. This is restricted such that it cannot be set to digits 0-9.
NLS$LI_SMONGROUPING	Sizes for each group of monetary digits to the left of the decimal. If the last value is 0, the preceding value is repeated. To group thousands, specify "3;0".
NLS$LI_SMONTHNAME1 - NLS$LI_SMONTHNAME13	Native name for each month. 1 = January, 2 = February, etc. 13 = the 13th month, if it exists in the locale.
NLS$LI_SMONTHOUSANDSEP	The character(s) used as monetary separator between groups of digits left of the decimal. Cannot be set to digits 0-9.
NLS$LI_SNATIVECTRYNAME	The native name of the country.
NLS$LI_SNATIVEDIGITS	The ten characters that are the native equivalent to the ASCII 0-9.
NLS$LI_SNATIVELANGNAME	The native name of the language.
NLS$LI_SNEGATIVESIGN	String value for the negative sign. Cannot be set to digits 0-9.
NLS$LI_SPOSITIVESIGN	String value for the positive sign. Cannot be set to digits 0-9.
NLS$LI_SSHORTDATE	Short Date formatting string for this locale. The d, M and y should have the day, month, and year substituted, respectively. See NLS Date and Time Format for explanations of the valid strings.
NLS$LI_STHOUSAND	The character(s) used as separator between groups of digits left of the decimal. This is restricted such that it cannot be set to digits 0 - 9.
NLS$LI_STIME	Character(s) for the time separator. Cannot be set to digits 0-9.
NLS$LI_STIMEFORMAT	Time formatting string. See NLS Date and Time Format for explanations of the valid strings.

When NLSGetLocaleInfo (*type*, *outstr*) returns information about the date and time formats of the current locale, the value returned in *outstr* can be interpreted according to the following tables. Any text returned within a date and time string that is enclosed within single quotes should be left in the string in its exact form; that is, do not change the text or the location within the string.

Day

The day can be displayed in one of four formats using the letter "d". The following table shows the four variations:

d	Day of the month as digits without leading zeros for single-digit days
dd	Day of the month as digits with leading zeros for single-digit days
ddd	Day of the week as a three-letter abbreviation (SABBREVDAYNAME)
dddd	Day of the week as its full name (SDAYNAME)

Month

The month can be displayed in one of four formats using the letter "M". The uppercase "M" distinguishes months from minutes. The following table shows the four variations:

M	Month as digits without leading zeros for single-digit months
MM	Month as digits with leading zeros for single-digit months
MMM	Month as a three-letter abbreviation (SABBREVMONTHNAME)
MMMM	Month as its full name (SMONTHNAME)

Year

The year can be displayed in one of three formats using the letter "y". The following table shows the three variations:

y	Year represented by only the last digit
yy	Year represented by only the last two digits
yyyy	Year represented by the full 4 digits

Period/Era

The period/era string is displayed in a single format using the letters "gg".

gg	Period/Era string

Time

The time can be displayed in one of many formats using the letter "h" or "H" to denote hours, the letter "m" to denote minutes, the letter "s" to denote seconds and the letter "t" to denote the time marker. The following table shows the numerous variations of the time format. Lowercase "h" denotes the 12 hour clock and uppercase "H" denotes the 24 hour clock. The lowercase "m" distinguishes minutes from months.

h	Hours without leading zeros for single-digit hours (12 hour clock)
hh	Hours with leading zeros for single-digit hours (12 hour clock)
H	Hours without leading zeros for single-digit hours (24 hour clock)
HH	Hours with leading zeros for single-digit hours (24 hour clock)
m	Minutes without leading zeros for single-digit minutes
mm	Minutes with leading zeros for single-digit minutes
s	Seconds without leading zeros for single-digit seconds
ss	Seconds with leading zeros for single-digit seconds
t	One-character time marker string
tt	Multicharacter time marker string

Example

```
USE IFNLS
INTEGER(4) strlen
CHARACTER(40) str
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
```

```
print *, str    ! prints Monday if language is English
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str    ! prints Tuesday if language is English
```

See Also

NLSGetLocale
NLSFormatDate
NLSFormatTime
NLSSetLocale

NLSSetEnvironmentCodepage (W*S)

NLS Function: *Sets the codepage for the current console. The specified codepage affects the current console program and any other programs launched from the same console. It does not affect other open consoles or any consoles opened later.*

Module

USE IFNLS

Syntax

`result = NLSSetEnvironmentCodepage (codepage,flags)`

codepage	(Input) INTEGER(4). Number of the codepage to set as the console codepage.
flags	(Input) INTEGER(4). Must be set to NLS $ConsoleEnvironmentCodepage.

Results

The result type is INTEGER(4). The result is zero if successful. Otherwise, returns one of the following error codes defined in `IFNLS.F90`:

- NLS$ErrorInvalidCodepage - *codepage* is invalid or not installed on the system
- NLS$ErrorInvalidFlags - *flags* is not valid
- NLS$ErrorNoConsole - There is no console associated with the given applicatio. So operations, with the console codepage are not possible

The *flags* argument must be a NLS$ConsoleEnvironmentCodepage; it cannot be a NLS $WindowsEnvironmentCodepage. NLSSetEnvironmentCodepage does not affect the Windows* codepage.

See Also

NLSGetEnvironmentCodepage

NLSSetLocale (W*S)

NLS Function: *Sets the current language, country, or codepage.*

Module

USE IFNLS

Syntax

`result = NLSSetLocale (language[,country] [,codepage])`

language	(Input) Character*(*). One of the languages supported by the Windows* NLS APIs.
country	(Input; optional) Character*(*). If specified, characterizes the language further. If omitted, the default country for the language is set.
codepage	(Input; optional) INTEGER(4). If specified, codepage to use for all character-oriented NLS functions. Can be any valid supported codepage or one of the following predefined values defined in IFNLS.F90: • NLS$CurrentCodepage - The codepage is not changed. Only the language and country settings are altered by the function. • NLS$ConsoleEnvironmentCodepage - The codepage is changed to the default environment codepage currently in effect for console programs. • NLS$ConsoleLanguageCodepage - The codepage is changed to the default console codepage for the language and country combination specified. • NLS$WindowsEnvironmentCodepage - The codepage is changed to the default environment codepage currently in effect for Windows programs. • NLS$WindowsLanguageCodepage - The codepage is changed to the default Windows codepage for the language and country combination specified. If you omit *codepage*, it defaults to NLS$WindowsLanguageCodepage. At program startup, NLS$WindowsEnvironmentCodepage is used to set the codepage.

Results

The result type is INTEGER(4). The result is zero if successful. Otherwise, one of the following error codes (defined in IFNLS.F90) may be returned:

- NLS$ErrorInvalidLanguage - *language* is invalid or not supported
- NLS$ErrorInvalidCountry - *country* is invalid or is not valid with the language specified
- NLS$ErrorInvalidCodepage - *codepage* is invalid or not installed on the system

NOTE
NLSSetLocale works on installed locales only. Many locales are supported, but they must be installed through the system Control Panel/International menu.

When doing mixed-language programming with Fortran and C, calling NLSSetLocale with a codepage other than the default environment Windows codepage causes the codepage in the C run-time library to change by calling C's setmbcp( ) routine with the new codepage. Conversely, changing the C run-time library codepage does not change the codepage in the Fortran NLS library.

Calling NLSSetLocale has no effect on the locale used by C programs. The locale set with C's setlocale( ) routine is independent of NLSSetLocale.

Calling NLSSetLocale with the default environment console codepage, NLS$ConsoleEnvironmentCodepage, causes an implicit call to the Windows API SetFileApisToOEM( ). Calling NLSSetLocale with any other codepage causes a call to SetFileApisToANSI( ).

See Also

NLSGetLocale

NOFREEFORM

Statement: *Specifies that source code is in fixed-form format.*

For more information, see FREEFORM and NOFREEFORM.

NOFUSION

General Compiler Directive: *Prevents a loop from fusing with adjacent loops.*

Syntax

```
!DIR$ NOFUSION
```

The NOFUSION directive lets you fine tune your program on a loop-by-loop basis.

This directive should be placed immediately before the DO statement of the loop that should not be fused.

Example

Consider the following example that demonstrates use of the NOFUSION directive:

```
     subroutine sub (b,a,n)
     real a(n), b(n)
     do i=1,n
       a(i) = a(i) + b(i)
     enddo
!DIR$ NOFUSION
     do i=1,n
       a(i) = a(i) + 1
     enddo
     end
```

The following shows the same example, but it uses Standard Fortran array assignments, which allow implicit arrays:

```
     subroutine sub (b,a,n)
     real a(n), b(n)
     a = a + b
!DIR$ NOFUSION
     a = a + 1
     end
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements

NON_RECURSIVE

Keyword: *Specifies that a subroutine or function does not call itself directly or indirectly.*

For more information, see RECURSIVE and NON_RECURSIVE.

NOOPTIMIZE

Statement: *Disables optimizations.*

For more information, see OPTIMIZE and NOOPTIMIZE.

NOPREFETCH

Statement: *Disables data prefetching.*

For more information, see PREFETCH and NOPREFETCH.

NORM2

Transformational Intrinsic Function (Generic):
Returns the L_2 norm of an array.

Syntax

```
result = NORM2 (x [, dim])
```

x	(Input) Must be a real array.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 <= *dim* <= *n*, where *n* is the rank of *x*. The corresponding actual argument must not be an optional dummy argument.

Results

The result is of the same type and kind parameters as *x*.

The result is a scalar if *dim* is omitted; otherwise, the result has rank *n* - 1 and shape [d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n], where *n* is the rank of *x* and [d_1, d_2, ..., d_n] is the shape of *x*.

The result of NORM2 (*x*) has a value equal to a processor-dependent approximation to the generalized L_2 norm of *x*, which is the square root of the sum of the squares of the elements of *x*.

The result of NORM2 (*x*, DIM=*dim*) has a value equal to that of NORM2 (*x*) if *x* has rank one.

Otherwise, the value of element (s_1, s_2, .., s_{dim-1}, s_{dim+1}, ... s_n) of the result is equal to NORM2 (*x* (s_1, s_2, .., s_{dim-1}, :, s_{dim+1}, ... s_n)).

It is recommended that the processor compute the result without undue overflow or underflow.

Example

The value of NORM2 ([3.0, 4.0]) is 5.0 (approximately).

If X has the value:

```
[ 1.0  2.0 ]
[ 3.0  4.0 ]
```

then the value of NORM2 (X, DIM=1) is approximately [3.162, 4.472] and the value of NORM2 (X, DIM=2) is approximately [2.236,5.0].

NOSTRICT

Statement: *Enables language features not found in the language standard specified on the command line.*

For more information, see STRICT and NOSTRICT.

NOT

Elemental Intrinsic Function (Generic): *Returns the logical complement of the argument.*

Syntax

```
result = NOT (i)
```

i (Input) Must be of type integer.

Results

The result type and kind are the same as *i*. The result value is obtained by complementing *i* bit-by-bit according to the following truth table:

```
INOT (I)
  1      0
  0      1
```

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BNOT	INTEGER(1)	INTEGER(1)
INOT[1]	INTEGER(2)	INTEGER(2)
JNOT	INTEGER(4)	INTEGER(4)
KNOT	INTEGER(8)	INTEGER(8)

[1]Or HNOT.

Example

If I has a value equal to 10101010 (base 2), NOT (I) has the value 01010101 (base 2).

The following shows another example:

```
INTEGER(2) i(2), j(2)
i = (/4, 132/)          ! i(1) = 0000000000000100
                        ! i(2) = 0000000010000100
j = NOT(i)              ! returns (-5,-133)
                        ! j(1) = 1111111111111011
                        ! j(2) = 1111111101111011
```

See Also

BTEST
IAND
IBCHNG
IBCLR
IBSET
IEOR
IOR
ISHA
ISHC
ISHL
ISHFT

ISHFTC

NOUNROLL

Statement: *Disables the unrolling of a DO loop.*

For more information, see UNROLL and NOUNROLL.

NOUNROLL_AND_JAM

Statement: *Hints to the compiler to disable loop unrolling and jamming.*

For more information, see UNROLL_AND_JAM and NOUNROLL_AND_JAM.

NOVECTOR

Statement: *Disables vectorization of a DO loop.*

For more information, see VECTOR and NOVECTOR.

NOWAIT Clause

Parallel Directive Clause: *Specifies that threads may resume execution before the execution of the region completes.*

Syntax

```
NOWAIT
```

When you specify this clause, it removes the synchronization barrier implied at the end of the region.

Note that NOWAIT can also be specified as a keyword in several directives.

At most one NOWAIT clause or keyword can appear in a directive that allows the clause or keyword.

NULL

Transformational Intrinsic Function (Generic): *Initializes a pointer as disassociated when it is declared.*

Syntax

```
result = NULL ([ mold])
```

mold	(Optional) Must be a pointer; it can be of any type. Its pointer association status can be associated, disassociated, or undefined. If its status is associated, the target does not have to be defined with a value.

Results

The result type and kind are the same as *mold*, if present; otherwise, it is determined as follows:

If NULL ( ) Appears...	Type is Determined From...
On the right side of pointer assignment	The pointer on the left side
As initialization for an object in a declaration	The object

If NULL ( ) Appears...	Type is Determined From...
As default initialization for a component	The component
In a structure constructor	The corresponding component
As an actual argument	The corresponding dummy argument
In a DATA statement	The corresponding pointer object

The result is a pointer with disassociated association status.

Caution
If you use module IFWIN or IFWINTY, you will have a name conflict if you use the NULL intrinsic. To avoid this problem, rename the integer parameter constant NULL to something else; for example:

```
USE IFWIN, NULL0 => NULL
```

This example lets you use both NULL0 and NULL( ) in the same program unit with no conflict.

Example

Consider the following:

```
INTEGER, POINTER :: POINT1 => NULL( )
```

This statement defines the initial association status of POINT1 to be disassociated.

NULLIFY

Statement: *Disassociates a pointer from a target.*

Syntax

NULLIFY (*pointer-object*[,*pointer-object*]...)

pointer-object	Is a structure component or the name of a variable; it must be a pointer (have the POINTER attribute).

Description

The initial association status of a pointer is undefined. You can use NULLIFY to initialize an undefined pointer, giving it disassociated status. Then the pointer can be tested using the intrinsic function ASSOCIATED.

Example

The following is an example of the NULLIFY statement:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR_A(:), PTR_B(:)
PTR_A => TAR
PTR_B => TAR
...
NULLIFY(PTR_A)
```

After these statements are executed, PTR_A will have disassociated status, while PTR_B will continue to be associated with variable TAR.

The following shows another example:

```
!    POINTER2.F90    Pointing at a Pointer and Target
!DIR$ FIXEDFORMLINESIZE:80

     REAL, POINTER :: arrow1 (:)
     REAL, POINTER :: arrow2 (:)
     REAL, ALLOCATABLE, TARGET :: bullseye (:)

     ALLOCATE (bullseye (7))
     bullseye = 1.
     bullseye (1:7:2) = 10.
     WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye

     arrow1 => bullseye
     WRITE (*,'(/1x,a,7f8.0)') 'pointer',arrow1

     arrow2 => arrow1
     IF (ASSOCIATED(arrow2)) WRITE (*,'(/a/)') ' ARROW2 is pointed.'
     WRITE (*,'(1x,a,7f8.0)') 'pointer',arrow2

     NULLIFY (arrow2)
     IF (.NOT.ASSOCIATED(arrow2)) WRITE (*,'(/a/)') ' ARROW2 is not pointed.'
     WRITE (*,'( 1x,a,7f8.0)') 'pointer',arrow1
     WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye

     END
```

See Also

ALLOCATE
ASSOCIATED
DEALLOCATE
POINTER
TARGET
NULL
Pointer Assignments
Dynamic Allocation

NUM_IMAGES

Transformational Intrinsic Function (Generic):
Returns the number of images.

Syntax

```
result = NUM_IMAGES()
```

Results

The result type is default integer. The result is the number of images.

You can specify a compiler option or environment variable to modify the number of images. If both are specified, the environment variable setting overrides the compiler option setting.

A compiler option must be specified to enable coarrays. If it is not specified and you use this intrinsic function, an error occurs.

Example

In the following example, image 1 is used to read data. The other images then copy the data:

```
REAL :: R[*]

IF (THIS_IMAGE()==1) THEN
   READ (7,*) R
   DO I = 2, NUM_IMAGES()
      R[I] = R
   END DO
END IF
SYNC ALL
```

O to P

O to P

OBJCOMMENT

General Compiler Directive: *Specifies a library search path in an object file.*

Syntax

```
!DIR$ OBJCOMMENT LIB: library
```

library	Is a character constant specifying the name and, if necessary, the path of the library that the linker is to search.

The linker searches for the library named in OBJCOMMENT as if you named it on the command line, that is, before default library searches. You can place multiple library search directives in the same source file. Each search directive appears in the object file in the order it is encountered in the source file.

If the OBJCOMMENT directive appears in the scope of a module, any program unit that uses the module also contains the directive, just as if the OBJCOMMENT directive appeared in the source file using the module.

If you want to have the OBJCOMMENT directive in a module, but do not want it in the program units that use the module, place the directive outside the module that is used.

Example

```
!  MOD1.F90
MODULE a
!DIR$ OBJCOMMENT LIB: "opengl32.lib"
END MODULE a

!  MOD2.F90
!DIR$ OBJCOMMENT LIB: "graftools.lib"
MODULE b
!
END MODULE b

!  USER.F90
PROGRAM go
   USE a    ! library search contained in MODULE a
            !   included here
   USE b    ! library search not included
END
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
Equivalent Compiler Options

OPEN

Statement: *Connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection.*

Syntax

`OPEN ([UNIT=] io-unit[, FILE= name] [, ERR= label] [, IOMSG=msg-var] [, IOSTAT=i-var],slist)`

io-unit — Is an external unit specifier.

name — Is a character or numeric expression specifying the name of the file to be connected. For more information, see FILE Specifier.

label — Is the label of the branch target statement that receives control if an error occurs. For more information, see Branch Specifiers.

msg-var — Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs. For more information, see I/O Message Specifier.

i-var — Is a scalar integer variable that is defined as a positive integer (the number of the error message) if an error occurs, a negative integer if an end-of-file record is encountered, and zero if no error occurs. For more information, see I/O Status Specifier.

slist — Is one or more of the following OPEN specifiers in the form *specifier= value* or *specifier* (each specifier can appear only once):

ACCESS	DECIMAL	NAME	ROUND
ACTION	DEFAULTFILE	NEWUNIT	SHARE
ASSOCIATEVARIABLE	DELIM	NOSHARED	SHARED
ASYNCHRONOUS	DISPOSE	ORGANIZATION	SIGN
BLANK	ENCODING	PAD	STATUS
BLOCKSIZE	FILE	POSITION	TITLE
BUFFERCOUNT	FORM	READONLY	TYPE
BUFFERED	IOFOCUS	RECL	USEROPEN
CARRIAGECONTROL	MAXREC	RECORDSIZE	
CONVERT	MODE	RECORDTYPE	

The OPEN specifiers and their acceptable values are summarized in the OPEN Statement Overview.

The control specifiers that can be specified in an OPEN statement are discussed in I/O Control List in the *Language Reference*.

Description

The control specifiers ([UNIT=] *io-unit*, ERR= *label*, and IOSTAT= *i-var*) and OPEN specifiers can appear anywhere within the parentheses following OPEN. However, if the UNIT specifier is omitted, the *io-unit* must appear first in the list.

If you specify BLANK=, DECIMAL=, PAD=, ROUND=, or SIGN=, you must also specify FMT= or NML=.

If you specify ID=, you must also specify ASYNCHRONOUS='YES'.

Specifier values that are scalar numeric expressions can be any integer or real expression. The value of the expression is converted to integer data type before it is used in the OPEN statement.

If the NEWUNIT= specifier does not appear, an *io-unit* must be specified. If the keyword UNIT= is omitted, the *io-unit* must be first in the control list.

If the NEWUNIT= specifier appears, an *io-unit* must not be specified.

If the NEWUNIT= specifier appears, either the FILE= specifier or the STATUS=SCRATCH specifier must also appear.

Only one unit at a time can be connected to a file, but multiple OPENs can be performed on the same unit. If an OPEN statement is executed for a unit that already exists, the following occurs:

- If FILE is not specified, or FILE specifies the same file name that appeared in a previous OPEN statement, the current file remains connected.

  If the file names are the same, the values for the BLANK, CARRIAGECONTROL, CONVERT, DELIM, DISPOSE, ERR, IOSTAT, and PAD specifiers can be changed. Other OPEN specifier values cannot be changed, and the file position is unaffected.
- If FILE specifies a different file name, the previous file is closed and the new file is connected to the unit.

The ERR and IOSTAT specifiers from any previously executed OPEN statement have no effect on any currently executing OPEN statement. If an error occurs, no file is opened or created.

Secondary operating system messages do not display when IOSTAT is specified. To display these messages, remove IOSTAT or use a platform-specific method.

Example

You can specify character values at run time by substituting a character expression for a specifier value in the OPEN statement. The character value can contain trailing blanks but not leading or embedded blanks; for example:

```
CHARACTER*6 FINAL /' '/
...
IF (expr) FINAL = 'DELETE'
OPEN (UNIT=1, STATUS='NEW', DISP=FINAL)
```

The following statement creates a new sequential formatted file on unit 1 with the default file name fort.1:

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

The following example opens the existing file /usr/users/someone/test.dat:

```
 OPEN (unit=10, DEFAULTFILE='/usr/users/someone/', FILE='test.dat',
1     FORM='FORMATTED', STATUS='OLD')
```

The following example opens a new file:

```
! Prompt user for a filename and read it:
CHARACTER*64 filename
WRITE (*, '(A\)') ' enter file to create: '
READ (*, '(A)') filename
! Open the file for formatted sequential access as unit 7.
! Note that the specified access need not have been specified,
! since it is the default (as is "formatted").
OPEN (7, FILE = filename, ACCESS = 'SEQUENTIAL', STATUS = 'NEW')
```

The following example opens an existing file called DATA3.TXT:

```
! Open a file created by an editor, "DATA3.TXT", as unit 3:
OPEN (3, FILE = 'DATA3.TXT')
```

See Also

READ
WRITE
CLOSE
FORMAT
INQUIRE
OPEN Statement Overview

OPTIONAL

Statement and Attribute: *Permits dummy arguments to be omitted in a procedure reference.*

Syntax

The OPTIONAL attribute can be specified in a type declaration statement or an OPTIONAL statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] OPTIONAL [, *att-ls*] :: *d-arg*[, *d-arg*]...

Statement:

OPTIONAL [::] *d-arg*[, *d-arg*] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
d-arg	Is the name of a dummy argument.

Description

The OPTIONAL attribute can only appear in the scoping unit of a subprogram or an interface body, and can only be specified for dummy arguments. It cannot be specified for arguments that are passed by value.

A dummy argument is "present" if it associated with an actual argument that is itself present. A dummy argument that is not optional must be present. You can use the PRESENT intrinsic function to determine whether an optional dummy argument is associated with an actual argument.

To call a procedure that has an optional argument, you must use an explicit interface.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments. A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

Example

The following example shows a type declaration statement specifying the OPTIONAL attribute:

```
SUBROUTINE TEST(A)
REAL, OPTIONAL, DIMENSION(-10:2) :: A
END SUBROUTINE
```

The following is an example of the OPTIONAL statement:

```
SUBROUTINE TEST(A, B, L, X)
  OPTIONAL :: B
  INTEGER A, B, L, X
  IF (PRESENT(B)) THEN        ! Printing of B is conditional
    PRINT *, A, B, L, X       !   on its presence
  ELSE
    PRINT *, A, L, X
  ENDIF
END SUBROUTINE

INTERFACE
  SUBROUTINE TEST(ONE, TWO, THREE, FOUR)
    INTEGER ONE, TWO, THREE, FOUR
    OPTIONAL :: TWO
  END SUBROUTINE
END INTERFACE

INTEGER I, J, K, L
I = 1
J = 2
K = 3
L = 4

CALL TEST(I, J, K, L)            ! Prints:  1  2  3  4
CALL TEST(I, THREE=K, FOUR=L)    ! Prints:  1  3  4
END
```

Note that in the second call to subroutine TEST, the second positional (optional) argument is omitted. In this case, all following arguments must be keyword arguments.

The following shows another example:

```
SUBROUTINE ADD (a,b,c,d)
  REAL              a, b, d
  REAL, OPTIONAL :: c
  IF (PRESENT(c)) THEN
    d = a + b + c + d
  ELSE
    d = a + b + d
  END IF
END SUBROUTINE
```

Consider the following:

```
SUBROUTINE EX (a, b, c)
REAL, OPTIONAL :: b,c
```

This subroutine can be called with any of the following statements:

```
CALL EX (x, y, z)   !All 3 arguments are passed.
CALL EX (x)         !Only the first argument is passed.
CALL EX (x, c=z)    !The first optional argument is omitted.
```

Note that you *cannot* use a series of commas to indicate omitted optional arguments, as in the following example:

```
CALL EX (x,,z)   !Invalid statement.
```

This results in a compile-time error.

See Also

PRESENT
Argument Keywords in Intrinsic Procedures
Optional Arguments
Argument Association
Type Declarations
Compatible attributes

OPTIMIZE and NOOPTIMIZE

General Compiler Directive: *Enables or disables optimizations for the program unit.*

Syntax

```
!DIR$ OPTIMIZE[: n]

!DIR$ NOOPTIMIZE
```

n	Is the number denoting the optimization level. The number can be 0, 1, 2, or 3, which corresponds to compiler options `O0`, `O1`, `O2`, and `O3`. If *n* is omitted, the default is 2, which corresponds to option `O2`.

The OPTIMIZE and NOOPTIMIZE directives can only appear once at the top of a procedure program unit. A procedure program unit is a main program, an external subroutine or function, or a module. OPTIMIZE and NOOPTIMIZE cannot appear between program units or in a block data program unit. They do not affect any modules invoked with the USE statement in the program unit that contains them. They do affect CONTAINed procedures that do not include an explicit OPTIMIZE or NOOPTIMIZE directive.

NOOPTIMIZE is the same as OPTIMIZE:0. They are both equivalent to `-O0` (Linux* and macOS*) and `/Od` (Windows*).

The procedure is compiled with an optimization level equal to the smaller of *n* and the optimization level specified by the O compiler option on the command line. For example, if the procedure contains the directive NOOPTIMIZE and the program is compiled with compiler option `O3`, this procedure is compiled at `O0` while the rest of the program is compiled at `O3`.

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
O compiler option

OPTIONS Directive

General Compiler Directive: *Affects data alignment and warnings about data alignment. Also controls whether a target attribute is assigned to a section of program declarations.*

Syntax

```
!DIR$ OPTIONS option[option]
...
!DIR$ END OPTIONS
```

option — Is one (or more) of the following:

`/WARN=[NO]ALIGNMENT` — Controls whether warnings are issued by the compiler for data that is not naturally aligned. By default, you receive compiler messages when misaligned data is encountered (/WARN=ALIGNMENT).

`/[NO]ALIGN[= p]` — Controls alignment of fields in record structures and data items in common blocks. The fields and data items can be naturally aligned (for performance reasons) or they can be packed together on arbitrary byte boundaries.

p — Is a specifier with one of the following forms:

`[class=]rule`

`(class= rule,...)`

`ALL`

`NONE`

class — Is one of the following keywords:

- `COMMONS`: For common blocks
- `RECORDS`: For records
- `STRUCTURES`: A synonym for RECORDS

rule — Is one of the following keywords:

PACKED	Packs fields in records or data items in common blocks on arbitrary byte boundaries.
NATURAL	Naturally aligns fields in records and data items in common blocks on up to 64-bit boundaries (inconsistent with the Standard Fortran). This keyword causes the compiler to naturally align all data in a common block, including INTEGER(KIND=8), REAL(KIND=8), and all COMPLEX data.
STANDARD	Naturally aligns data items in

	common blocks on up to 32-bit boundaries (consistent with the Standard Fortran). This keyword only applies to common blocks; so, you can specify /ALIGN=COMMONS=STANDARD, but you cannot specify /ALIGN=STANDARD.
ALL	Is the same as specifying OPTIONS /ALIGN, OPTIONS /ALIGN=NATURAL, and OPTIONS /ALIGN=(RECORDS=NATURAL,COMMONS=NATURAL).
NONE	Is the same as specifying OPTIONS /NOALIGN, OPTIONS /ALIGN=PACKED, and OPTIONS /

ALIGN=(RECORDS=PACKED,COMMONS=PACKED).

The OPTIONS (and accompanying END OPTIONS) directives must come after OPTIONS, SUBROUTINE, FUNCTION, and BLOCK DATA statements (if any) in the program unit, and before the executable part of the program unit.

The OPTIONS directive supersedes the compiler option `align`.

For performance reasons, Intel® Fortran aligns local data items on natural boundaries. However, EQUIVALENCE, COMMON, RECORD, and STRUCTURE data declaration statements can force misaligned data. If /WARN=NOALIGNMENT is specified, warnings will not be issued if misaligned data is encountered.

NOTE
Misaligned data significantly increases the time it takes to execute a program. As the number of misaligned fields encountered increases, so does the time needed to complete program execution. Specifying /ALIGN (or compiler option `align`) minimizes misaligned data.

If you want aligned data in common blocks, do one of the following:

- Specify OPTIONS /ALIGN=COMMONS=STANDARD for data items up to 32 bits in length.
- Specify OPTIONS /ALIGN=COMMONS=NATURAL for data items up to 64 bits in length.
- Place source data declarations within the common block in descending size order, so that each data item is naturally aligned.

If you want packed, unaligned data in a record structure, do one of the following:

- Specify OPTIONS /ALIGN=RECORDS=PACKED.
- Place source data declarations in the record structure so that the data is naturally aligned.

Example

Consider the following:

```
! directives can be nested up to 100 levels
  !DIR$ OPTIONS /ALIGN=PACKED     ! Start of Group A
    declarations
  !DIR$ OPTIONS /ALIGN=RECO=NATU     ! Start of nested Group B
    more declarations
  !DIR$ END OPTIONS                  ! End of Group B
    still more declarations
  !DIR$ END OPTIONS                ! End of Group A
```

The OPTIONS specification for Group B only applies to RECORDS; common blocks within Group B will be PACKED. This is because COMMONS retains the previous setting (in this case, from the Group A specification).

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
align compiler option

OPTIONS Statement

Statement: *Overrides or confirms the compiler options in effect for a program unit.*

Syntax

OPTIONS *option*[*option*...]

option Is one of the following:

/ASSUME =	[NO]UNDERSCORE
/CHECK =	ALL
	[NO]BOUNDS
	NONE
/NOCHECK	
/CONVERT =	BIG_ENDIAN
	CRAY
	FDX
	FGX
	IBM
	LITTLE_ENDIAN
	NATIVE
	VAXD
	VAXG
/[NO]EXTEND_SOURCE	
/[NO]F77	
/[NO]I4	
/[NO]RECURSIVE	

Note that an option must always be preceded by a slash (/).

Some OPTIONS statement options are equivalent to compiler options.

The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, MODULE, and BLOCK DATA statements.

OPTIONS statement options override compiler options, but only until the end of the program unit for which they are defined. If you want to override compiler options in another program unit, you must specify the OPTIONS statement before that program unit.

Example

The following are valid OPTIONS statements:

```
OPTIONS /CHECK=ALL/F77
OPTIONS /I4
```

See Also

OPEN Statement CONVERT Method
OPTIONS Statement Method

For details on compiler options, see your *Compiler Options* reference

OR

Elemental Intrinsic Function (Generic): *Performs a bitwise inclusive OR on its arguments.*

See IOR.

Example

```
INTEGER i
i = OR(3, 10)   ! returns 11
```

ORDERED

OpenMP* Fortran Compiler Directive: *Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations, or, as a stand-alone directive, it specifies cross-iteration dependences in a doacross loop nest.*

Syntax

It takes one of the following forms:

Form 1:

```
!$OMP ORDERED [clause [[,] clause ]]
   block
!$OMP END ORDERED
```

Form 2:

```
!$OMP ORDERED clause [[[,] clause ]...]
```

Form 1 *clause*

Is an optional clause. It can be one of the following:

- `SIMD [keyword [, keyword]...]`

  Applies an ordered region inside a SIMD loop. Possible values are:

  - MONOTONIC (*list* [: *linear-step*])

    The *list* is a comma-separated list of one or more integer scalar variables. The *linear-step* is a positive, integer, scalar constant expression. The *linear-step* expression must be invariant (it must not be changed) during the execution of the associated construct. If *linear-step* is omitted, a default value of 1 is used.

    Multiple MONOTONIC keywords may appear; they are merged into a single MONOTONIC list. You cannot specify multiple MONOTONIC keywords with different *linear-step*s for the same variable.

    Each *list* item may have the POINTER attribute but not the ALLOCATABLE attribute. Each list item must comply with PRIVATE clause semantics.

- OVERLAP (*expr*)

  The *expr* is an integer expression. It specifies a block of code that has to be executed in scalar mode for overlapping loop index values and in parallel for different loop index values within the SIMD loop.

  The OVERLAP keyword can appear no more than once in an ORDERED directive.
- `THREADS`

  Applies an ordered region inside a PARALLEL DO loop. If no *clause* is specified, the directive behaves as if THREADS was specified.

At most one THREADS clause can appear in an ORDERED construct. At most one SIMD clause can appear in an ORDERED construct.

Form 2 *`clause`*

Is one of the following:

- DEPEND (SOURCE)
- DEPEND (SINK : vec)

  At most one DEPEND (SOURCE) clause can appear on an ORDERED construct.

  Either DEPEND (SINK : *vec*) clauses or DEPEND (SOURCE) clauses can appear in an ORDERED construct, but not both.

`block`

Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

The binding thread set for an ORDERED construct is the current team. An ordered region binds to the innermost enclosing loop region or the innermost enclosing SIMD region if the SIMD clause is present.

A doacross loop nest is a loop nest that has cross-iteration dependences. An iteration is dependent on one or more lexicographically earlier iterations. The ORDERED clause parameter on a loop directive identifies the loops associated with the doacross loop nest.

An ORDERED directive with no *clause* or with the THREADS clause specified can appear only in the dynamic extent of a DO or PARALLEL DO directive. The DO directive to which the ordered section binds *must* have the ORDERED clause specified.

An iteration of a loop using a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

One thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it can be guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel.

Ordered sections that bind to different DO directives are independent of each other.

If the SIMD clause is specified, the ordered regions encountered by any thread will use only a single SIMD lane to execute the ordered regions in the order of the loop iterations.

You can only specify the SIMD clause (!$OMP ORDERED SIMD) within an !$OMP SIMD loop or an !$OMP DECLARE SIMD procedure.

When a thread executing any subsequent iteration encounters an ORDERED construct with one or more DEPEND (SINK : *vec*) clauses, it waits until its dependences on all valid iterations specified by the DEPEND clauses are satisfied before it completes execution of the ORDERED region. A specific dependence is satisfied when a thread executing the corresponding iteration encounters an ORDERED construct with a DEPEND (SOURCE) clause.

For MONOTONIC (*list* : *linear-step*), the following rules apply:

- The value of the new *list* item on each iteration of the associated SIMD loop corresponds to the value of the original *list* item before entering the associated loop, plus the number of the iterations for which the conditional update happens prior to the current iteration, times *linear-step*.
- The value corresponding to the sequentially last iteration of the associated loop is assigned to the original *list* item.
- A *list* item must not be used in statements that lexically precede the ORDERED SIMD to which it is bound, that is, in the region from the OMP SIMD to the OMP ORDERED SIMD. A *list* item must not be modified outside the ORDERED SIMD to which it is bound, that is, in the region from the OMP END ORDERED to the OMP END SIMD.

Example

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
!$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO
      ...
      SUBROUTINE WORK(K)
!$OMP ORDERED
      WRITE(*,*) K
!$OMP END ORDERED
```

Ordered SIMD sections are useful for resolving cross-iteration data dependencies in otherwise data-parallel computations. For example, it may handle histogram updates such the following:

```
!$OMP SIMD
      DO I=0,N
        AMOUNT = COMPUTE_AMOUNT(I)
        CLUSTER = COMPUTE_CLUSTER(I)  ! Multiple I's may belong to the
                  !  same cluster within SIMD chunk
!$OMP ORDERED SIMD
        TOTALS(CLUSTER) = TOTALS(CLUSTER) + AMOUNT  ! Requires ordering to
                          !  process multiple updates
                          !  to the same cluster
!$OMP END ORDERED
      END DO
!$OMP END SIMD
```

The MONOTONIC keyword on the OMP ORDERED SIMD directive specifies that the body of a loop must be executed in the natural order of the loop iterations. In the following example, the block of code in the OMP ORDERED SIMD is executed in the ascending, monotonic order of the loop index I:

```
COUNT = 0
!$OMP SIMD
DO I = 1, N
  IF (COND(I)) THEN
!$OMP ORDERED SIMD MONOTONIC (COUNT:1)
    A(COUNT) = A(COUNT) + B(I)
    COUNT = COUNT + 1
    B(I) = C(COUNT)
!$OMP END ORDERED
  END IF
END DO
!$OMP END SIMD
```

In the following example, the OVERLAP keyword specifies that the block of code after the OMP ORDERED SIMD directive may contain overlap between the loop iterations:

```
!$OMP SIMD
DO I = 1, N
  INX = INDEX(I)
!$OMP ORDERED SIMD OVERLAP (INX)
  A(INX) = A(INX) + B(I)
!$OMP END ORDERED
END DO
!$OMP END SIMD
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

OUTGTEXT (W*S)

Graphics Subroutine: *In graphics mode, sends a string of text to the screen, including any trailing blanks.*

Module

USE IFQWIN

Syntax

```
CALL OUTGTEXT (text)
```

text	(Input) Character*(*). String to be displayed.

Text output begins at the current graphics position, using the current font set with SETFONT and the current color set with SETCOLORRGB or SETCOLOR. No formatting is provided. After it outputs the text, OUTGTEXT updates the current graphics position.

Before you call OUTGTEXT, you must call the INITIALIZEFONTS function.

Because OUTGTEXT is a graphics function, the color of text is affected by the SETCOLORRGB function, not by SETTEXTCOLORRGB.

Example

```
! build as a QuickWin App.
USE IFQWIN
INTEGER(2) result
INTEGER(4) i
TYPE (xycoord) xys

result = INITIALIZEFONTS()
result = SETFONT('t''Arial''h18w10pvib')
do i=1,6
   CALL MOVETO(INT2(0),INT2(30*(i-1)),xys)
   grstat=SETCOLOR(INT2(i))
   CALL OUTGTEXT('This should be ')
   SELECT CASE (i)
     CASE (1)
       CALL OUTGTEXT('Blue')
     CASE (2)
       CALL OUTGTEXT('Green')
```

```
     CASE (3)
       CALL OUTGTEXT('Cyan')
     CASE (4)
       CALL OUTGTEXT('Red')
     CASE (5)
       CALL OUTGTEXT('Magenta')
     CASE (6)
       CALL OUTGTEXT('Orange')
   END SELECT
end do
END
```

See Also

GETFONTINFO
GETGTEXTEXTENT
INITIALIZEFONTS
MOVETO
SETCOLORRGB
SETFONT
SETGTEXTROTATION

OUTTEXT (W*S)

Graphics Subroutine: *In text or graphics mode, sends a string of text to the screen, including any trailing blanks.*

Module

USE IFQWIN

Syntax

`CALL OUTTEXT (text)`

text (Input) Character*(*). String to be displayed.

Text output begins at the current text position in the color set with SETTEXTCOLORRGB or SETTEXTCOLOR. No formatting is provided. After it outputs the text, OUTTEXT updates the current text position.

To output text using special fonts, you must use the OUTGTEXT subroutine.

Example

```
USE IFQWIN
INTEGER(2) oldcolor
TYPE (rccoord) rc

CALL CLEARSCREEN($GCLEARSCREEN)
CALL SETTEXTPOSITION (INT2(1), INT2(5), rc)
oldcolor = SETTEXTCOLOR(INT2(4))
CALL OUTTEXT ('Hello, everyone')
END
```

See Also

OUTGTEXT
SETTEXTPOSITION
SETTEXTCOLORRGB

WRITE
WRAPON

PACK Directive

General Compiler Directive: *Specifies the memory alignment of derived-type items (and record structure items).*

Syntax

```
!DIR$ PACK[: {1 | 2 | 4 | 8} ]
```

Items of derived types, unions, and structures are aligned in memory on the smaller of two sizes: the size of the type of the item, or the current alignment setting. The current alignment setting can be 1, 2, 4, or 8 bytes. The default initial setting is 8 bytes (unless compiler option vms or align rec *n* bytes is specified). By reducing the alignment setting, you can pack variables closer together in memory.

The PACK directive lets you control the packing of derived-type or record structure items inside your program by overriding the current memory alignment setting.

For example, if PACK:1 is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted. If PACK:4 is specified, INTEGER(1), LOGICAL(1), and all character variables begin at the next available byte, whether odd or even. INTEGER(2) and LOGICAL(2) begin on the next even byte; all other variables begin on 4-byte boundaries.

If the PACK directive is specified without a number, packing reverts to the compiler option setting (if any), or the default setting of 8.

The directive can appear anywhere in a program before the derived-type definition or record structure definition. It cannot appear *inside* a derived-type or record structure definition.

Example

```
! Use 4-byte packing for this derived type
! Note PACK is used outside of the derived type definition
!DIR$ PACK:4
TYPE pair
   INTEGER a, b
END TYPE
! revert to default or compiler option
!DIR$ PACK
```

See Also

TYPE
STRUCTURE...END STRUCTURE
UNION...END UNION
General Compiler Directives
Syntax Rules for Compiler Directives
align:recnbytes compiler option
vms compiler option
Equivalent Compiler Options

PACK Function

Transformational Intrinsic Function (Generic): *Takes elements from an array and packs them into a rank-one array under the control of a mask.*

Syntax

```
result = PACK (array,mask[,vector])
```

array	(Input) Must be an array. It may be of any data type.
mask	(Input) Must be of type logical and conformable with *array*. It determines which elements are taken from *array*.
vector	(Input; optional) Must be a rank-one array with the same type and type parameters as *array*. Its size must be at least *t*, where *t* is the number of true elements in *mask*. If *mask* is a scalar with value true, *vector* must have at least as many elements as there are in array. Elements in *vector* are used to fill out the result array if there are not enough elements selected by *mask*.

Results

The result is a rank-one array with the same type and kind parameters as *array*. If *vector* is present, the size of the result is that of *vector*. Otherwise, the size of the result is the number of true elements in *mask*, or the number of elements in *array* (if *mask* is a scalar with value true).

Elements in *array* are processed in array element order to form the result array. Element i of the result is the element of *array* that corresponds to the ith true element of *mask*. If *vector* is present and has more elements than there are true values in *mask*, any result elements that are empty (because they were not true according to *mask*) are set to the corresponding values in *vector*.

Example

N is the array

```
 [ 0  8  0 ]
 [ 0  0  0 ]
 [ 7  0  0 ].
```

PACK (N, MASK=N .NE. 0, VECTOR=(/1, 3, 5, 9, 11, 13/)) produces the result (7, 8, 5, 9, 11, 13).

PACK (N, MASK=N .NE. 0) produces the result (7, 8).

The following shows another example:

```
INTEGER array(2, 3), vec1(2), vec2(5)
LOGICAL mask (2, 3)
array = RESHAPE((/7, 0, 0, -5, 0, 0/), (/2, 3/))
mask = array .NE. 0
!  array is  7  0 0  and mask is  T F F
!            0 -5 0               F T F
VEC1 = PACK(array, mask)      ! returns ( 7, -5 )
VEC2 = PACK(array, array .GT. 0, VECTOR= (/1,2,3,4,5/))
!  returns ( 7, 2, 3, 4, 5 )
```

See Also

UNPACK

PACKTIMEQQ

Portability Subroutine: *Packs time and date values.*

Module

USE IFPORT

Syntax

```
CALL PACKTIMEQQ (timedate,iyr,imon,iday,ihr,imin,isec)
```

timedate	(Output) INTEGER(4). Packed time and date information.
iyr	(Input) INTEGER(2). Year (*xxxx*AD).
imon	(Input) INTEGER(2). Month (1 - 12).
iday	(Input) INTEGER(2). Day (1 - 31)
ihr	(Input) INTEGER(2). Hour (0 - 23)
imin	(Input) INTEGER(2). Minute (0 - 59)
isec	(Input) INTEGER(2). Second (0 - 59)

The input values are interpreted as being in the time zone set on the local computer and following the daylight savings rules for that time zone.

The packed time is the number of seconds since 00:00:00 Greenwich mean time, January 1, 1970. Because packed time values can be numerically compared, you can use PACKTIMEQQ to work with relative date and time values. Use UNPACKTIMEQQ to unpack time information. SETFILETIMEQQ uses packed time.

Example

```
USE IFPORT
INTEGER(2) year, month, day, hour, minute, second,   &
           hund
INTEGER(4) timedate
CALL GETDAT (year, month, day)
CALL GETTIM (hour, minute, second, hund)
CALL PACKTIMEQQ (timedate, year, month, day, hour,   &
                 minute, second)
END
```

See Also

UNPACKTIMEQQ
SETFILETIMEQQ
GETFILEINFOQQ
TIME portability routine

PARALLEL Directive (OpenMP* API)

OpenMP* Fortran Compiler Directive: *Defines a parallel region.*

Syntax

```
!$OMP PARALLEL [clause[[,] clause] ... ]
   block
!$OMP END PARALLEL
```

clause	Is one or more of the following: • COPYIN (list) • DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)

- FIRSTPRIVATE (list)
- IF ([PARALLEL:] scalar-logical-expression)
- NUM_THREADS (*scalar_integer_expression*)

  Specifies the number of threads to be used in a parallel region. The *scalar_integer_expression* must evaluate to a positive scalar integer value. Only a single NUM_THREADS clause can appear in the directive.
- PRIVATE (list)
- PROC_BIND (MASTER | CLOSE | SPREAD)

  Specifies a method for mapping the threads in the team to the "places" in the current partition.

  Once a thread is assigned to a place, the OpenMP* implementation should not move it to another place.

  MASTER instructs the execution environment to assign every thread in the team to the same place as the master thread. CLOSE instructs the execution environment to assign the threads to places close to the place of the parent thread. SPREAD creates a sparse distribution for a team of T threads among the P places of the parent's place partition.

  For CLOSE and SPREAD, the threads with the smallest numbers are assigned starting with the place of the master thread. For CLOSE, threads are packed into consecutive places within the parent's partition. For SPREAD, the partition of the master thread is sub-divided and threads in the team are assigned round-robin to those subpartitions.

  Only a single PROC_BIND clause can appear in the directive.
- REDUCTION (reduction-identifier : list)
- SHARED (list)

block Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

The PARALLEL and END PARALLEL directive pair must appear in the same routine in the executable section of the code.

The END PARALLEL directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution at the end of a parallel region.

The number of threads in the team can be controlled by the NUM_THREADS clause, the environment variable OMP_NUM_THREADS, or by calling the run-time library routine OMP_SET_NUM_THREADS from a serial portion of the program.

NUM_THREADS supersedes the OMP_SET_NUM_THREADS routine, which supersedes the OMP_NUM_THREADS environment variable. Subsequent parallel regions, however, are not affected unless they have their own NUM_THREADS clauses.

Once specified, the number of threads in the team remains constant for the duration of that parallel region.

If the dynamic threads mechanism is enabled by an environment variable or a library routine, then the number of threads requested by the NUM_THREADS clause is the maximum number to use in the parallel region.

The code contained within the dynamic extent of the parallel region is executed on each thread, and the code path can be different for different threads.

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always serialized and executed by a team of one thread.

Example

You can use the PARALLEL directive in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array X upon which to work based on the thread number:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM( )
      NP = OMP_GET_NUM_THREADS( )
      IPOINTS = NPOINTS/NP
      CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

Assuming you previously used the environment variable OMP_NUM_THREADS to set the number of threads to six, you can change the number of threads between parallel regions as follows:

```
      CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
...
!$OMP END PARALLEL
      CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
...
!$OMP END PARALLEL DO
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
OpenMP* Run-time Library Routines for Fortran
PARALLEL DO
SHARED Clause
Parallel Processing Model for information about Binding Sets

PARALLEL and NOPARALLEL Loop Directives

General Compiler Directives: *PARALLEL facilitates auto-parallelization by assisting the compiler's dependence analysis of the immediately following DO loop. NOPARALLEL prevents this auto-parallelization.*

Syntax

```
!DIR$ PARALLEL [clause[[,] clause] ... ]

!DIR$ NOPARALLEL
```

clause — Is one or more of the following:

- ALWAYS [ASSERT]
- FIRSTPRIVATE (*list*)

Provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in a FIRSTPRIVATE list are subject to PRIVATE clause semantics. In addition, private (local) copies of each variable in the different iterations are initialized to the value the variable had upon entering the parallel loop.

- LASTPRIVATE (*list*)

  Provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause semantics. In addition, once the parallel loop is exited, each variable has the value that resulted from the sequentially last iteration of the parallel loop.
- NUM_THREADS (*scalar_integer_expression*)

  Specifies the number of threads to be used to parallelize the immediately following DO loop. The *scalar_integer_expression* must evaluate to a positive scalar integer value. Only a single NUM_THREADS clause can appear in the directive. Once specified, the number of threads used remains constant for the duration of that parallelized DO loop.
- PRIVATE (*list*)

  Declares specified variables to be private to each thread in a team. The declared variables become private to a task.

list Is one or more items in the form: var [:expr]…. Each list item must be separated by a comma.

var Is a scalar or array variable. The following rules apply:

- Assumed-size arrays cannot appear in a PRIVATE clause.
- A variable that is part of another variable (for example, as an array or structure element) cannot appear in a PRIVATE clause.
- A variable that appears in a PRIVATE clause must either be definable, or an allocatable array. This restriction does not apply to the FIRSTPRIVATE clause.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, cannot appear in a PRIVATE clause.

expr Is an integer expression denoting the number of array elements to privatize. When *expr* is specified, *var* must be an array or a pointer variable. The following rules also apply:

- If *var* is an array, then only its first *expr* elements are privatized. If *expr* is omitted, the entire array is privatized.
- If *var* is a pointer, then the first *expr* elements are privatized (element size is provided by the pointer's target type). If *expr* is omitted, only the pointer variable itself is privatized.
- Program behavior is undefined if *expr* evaluates to a non-positive value, or if it exceeds the array size.

PARALLEL helps the compiler to resolve dependencies, facilitating auto-parallelization of the immediately following DO loop. It instructs the compiler to ignore dependencies that it assumes may exist and which would prevent correct parallelization in the loop. However, if dependencies are proven, they are not ignored.

In addition, PARALLEL ALWAYS overrides the compiler heuristics that estimate the likelihood that parallelization of a loop will increase performance. It allows a loop to be parallelized even if the compiler thinks parallelization may not improve performance. If the ASSERT keyword is added, the compiler will generate an error-level assertion message saying that the compiler analysis and cost model indicate that the loop cannot be parallelized.

NOPARALLEL prevents auto-parallelization of the immediately following DO loop.

These directives take effect only if you specify the compiler option that enables auto-parallelization.

Caution
The directive PARALLEL ALWAYS should be used with care. Overriding the heuristics of the compiler should only be done if you are absolutely sure the parallelization will improve performance.

Example

```
program main
parameter (n=100)
integer x(n), a(n), k
!DIR$ NOPARALLEL
   do i=1,n
     x(i) = i
   enddo
!DIR$ PARALLEL LASTPRIVATE (k)
   do i=1,n
     a( x(i) ) = i
     k = x(i)
   enddo
   print *, k     ! print 100, the value of x(n)
end
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements

PARALLEL DO

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a parallel region containing a single DO directive.*

Syntax

```
!$OMP PARALLEL DO [clause[[,] clause] ... ]
      do-loop
[!$OMP END PARALLEL DO]
```

clause	Can be any of the clauses accepted by the DO or PARALLEL directives.
do-loop	Is a DO iteration (a DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

You cannot branch out of a DO loop associated with a DO directive.

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.

Example

In the following example, the loop iteration variable is private by default and it is not necessary to explicitly declare it. The END PARALLEL DO directive is optional:

```
!$OMP PARALLEL DO
      DO I=1,N
          B(I) = (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END PARALLEL DO
```

The following example shows how to use the REDUCTION clause in a PARALLEL DO directive:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
      DO I=1,N
        CALL WORK(ALOCAL,BLOCAL)
        A = A + ALOCAL
        B = B + BLOCAL
      END DO
!$OMP END PARALLEL DO
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

PARALLEL DO SIMD

OpenMP* Fortran Compiler Directive: *Specifies a PARALLEL construct that contains one DO SIMD construct and no other statement.*

Syntax

```
!$OMP PARALLEL DO SIMD [clause[[,] clause] ... ]
      do-loop
[!$OMP END PARALLEL DO SIMD]
```

clause — Can be any of the clauses accepted by the PARALLEL, DO, or SIMD directives.

do-loop — Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

You cannot branch out of a DO loop associated with a DO SIMD directive.

If the END PARALLEL DO SIMD directive is not specified, an END PARALLEL DO SIMD directive is assumed at the end of *do-loop*.

You cannot specify NOWAIT in a PARALLEL DO SIMD directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

PARALLEL SECTIONS

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a parallel region containing a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.*

Syntax

```
!$OMP PARALLEL SECTIONS [clause[[,] clause] ...]
[!$OMP SECTION]
   block
[!$OMP SECTION
   block]...
!$OMP END PARALLEL SECTIONS
```

clause — Can be any of the clauses accepted by the PARALLEL or SECTIONS directives.

block — Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

The last section ends at the END PARALLEL SECTIONS directive.

Example

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL XAXIS
!$OMP SECTION
      CALL YAXIS
!$OMP SECTION
      CALL ZAXIS
!$OMP END PARALLEL SECTIONS
```

See Also

OpenMP Fortran Compiler Directives

Syntax Rules for Compiler Directives

PARALLEL WORKSHARE

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a parallel region containing a single WORKSHARE directive.*

Syntax

```
!$OMP PARALLEL WORKSHARE [clause[[,] clause] ... ]
   block
!$OMP END PARALLEL WORKSHARE
```

clause	Can be any of the clauses accepted by the PARALLEL directive.
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

PARAMETER

Statement and Attribute: *Defines a named constant.*

Syntax

The PARAMETER attribute can be specified in a type declaration statement or a PARAMETER statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] PARAMETER [, att-ls] :: c =expr[, c = expr] ...
```

Statement:

```
PARAMETER [( ]c= expr[, c= expr] ... [) ]
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
c	Is the name of the constant.
expr	Is a constant expression. It can be of any data type.

Description

The type, type parameters, and shape of the named constant are determined in one of the following ways:

- By an explicit type declaration statement in the same scoping unit.
- By the implicit typing rules in effect for the scoping unit. If the named constant is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

For example, consider the following statement:

```
PARAMETER (MU=1.23)
```

According to implicit typing, MU is of integer type, so MU=1. For MU to equal 1.23, it should previously be declared REAL in a type declaration or be declared in an IMPLICIT statement.

A named array constant defined by a PARAMETER statement must have its rank specified in a previous specification statement.

A named constant must not appear in a format specification or as the character count for Hollerith constants. For compilation purposes, writing the name is the same as writing the value.

If the named constant is used as the length specifier in a CHARACTER declaration, it must be enclosed in parentheses.

The name of a constant cannot appear as part of another constant, although it can appear as either the real or imaginary part of a complex constant.

You can only use the named constant within the scoping unit containing the defining PARAMETER statement.

Any named constant that appears in the constant expression must have been defined previously in the same type declaration statement (or in a previous type declaration statement or PARAMETER statement), or made accessible by use or host association.

An entity with the PARAMETER attribute must not be a variable, a coarray, or a procedure.

Omission of the parentheses in a PARAMETER statement is an extension controlled by compiler option `altparam`. In this form, the type of the name is taken from the form of the constant rather than from implicit or explicit typing of the name.

Example

The following example shows a type declaration statement specifying the PARAMETER attribute:

```
REAL, PARAMETER :: C = 2.9979251, Y = (4.1 / 3.0)
```

The following is an example of the PARAMETER statement:

```
REAL(4) PI, PIOV2
REAL(8) DPI, DPIOV2
LOGICAL FLAG
CHARACTER*(*) LONGNAME
PARAMETER (PI=3.1415927, DPI=3.141592653589793238D0)
PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)
PARAMETER (FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS')
```

The following shows implicit-shape arrays declared using a PARAMETER attribute and PARAMETER statements:

```
INTEGER, PARAMETER :: R(*) = [1,2,3]
REAL :: M (2:*, -1:*)
PARAMETER (M = RESHAPE ([R,R], [3,2]))
```

The following shows other examples:

```
! implicit integer type
PARAMETER (nblocks = 10)
!
! implicit real type
IMPLICIT REAL (L-M)
PARAMETER (loads = 10.0, mass = 32.2)
!
! typed by PARAMETER statement
! Requires compiler option
PARAMETER mass = 47.3, pi = 3.14159
PARAMETER bigone = 'This constant is larger than forty characters'
!
! PARAMETER in attribute syntax
REAL, PARAMETER :: mass=47.3, pi=3.14159, loads=10.0, mass=32.2
```

See Also

DATA
Type Declarations
Compatible attributes
Constant Expressions
IMPLICIT
Alternative syntax for the PARAMETER statement
altparam compiler option

PARITY

Transformational Intrinsic Function (Generic): *Reduces an array by using an exclusive OR (.NEQV.) operation.*

Syntax

`result = PARITY (mask[, dim])`

mask	(Input) Must be an array of type logical.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 <= *dim* <= *n*, where *n* is the rank of *mask*. The corresponding actual argument must not be an optional dummy argument.

Results

The result has the same type and kind parameters as *mask*. It is scalar if *dim* does not appear; otherwise, the result has rank *n* - 1 and shape [d_1, d_2, ..., d_{dim-1}, d_{dim+1} , ..., d_n] where [d_1, d_2,..., d_n] is the shape of *mask*.

The result of PARITY(*mask*) has the value .TRUE. if an odd number of the elements of mask are true; otherwise, .FALSE..

If *mask* has rank one, PARITY(*mask*, *dim*) is equal to PARITY(*mask*). Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1} , ..., s_n) of PARITY(*mask*, *dim*) is equal to PARITY (*mask* (s_1, s_2, ..., s_{dim-1} , :, s_{dim+1} , ..., s_n)).

Example

If A is the array

```
[ T  F  T ]
[ T  T  F ]
```

and T has the value .TRUE. and F has the value .FALSE., then

PARITY (A, DIM=1) has the value [F, T, T] and PARITY (A, DIM=2) has the value [F, F].

PASSDIRKEYSQQ (W*S)

QuickWin Function: *Determines the behavior of direction and page keys in a QuickWin application.*

Module

USE IFQWIN

Syntax

```
result = PASSDIRKEYSQQ (val)
```

val

(Input) INTEGER(4) or LOGICAL(4).

A value of .TRUE. causes direction and page keys to be input as normal characters (the PassDirKeys flag is turned on). A value of .FALSE. causes direction and page keys to be used for scrolling.

The following constants, defined in IFQWIN.F90, can be used as integer arguments:

- PASS_DIR_FALSE - Turns off any special handling of direction keys. They are not passed to the program by GETCHARQQ.
- PASS_DIR_TRUE - Turns on special handling of direction keys. That is, they are passed to the program by GETCHARQQ.
- PASS_DIR_INSDEL - INSERT and DELETE are also passed to the program by GETCHARQQ
- PASS_DIR_CNTRLC - Only needed for a QuickWin application, but harmless if used with a Standard Graphics application that already passes CTRL+C.

  This value allows CTRL+C to be passed to a QuickWin program by GETCHARQQ if the following is true: the program must have removed the File menu EXIT item by using DELETEMENUQQ.

  This value also passes direction keys and INSERT and DELETE.

Results

The return value indicates the previous setting of the PassDirKeys flag.

The return data type is the same as the data type of *val*; that is, either INTEGER(4) or LOGICAL(4).

When the PassDirKeys flag is turned on, the mouse must be used for scrolling since the direction and page keys are treated as normal input characters.

The PASSDIRKEYSQQ function is meant to be used primarily with the GETCHARQQ and INCHARQQ functions. Do not use normal input statements (such as READ) with the PassDirKeys flag turned on, unless your program is prepared to interpret direction and page keys.

Example

```
use IFQWIN

logical*4 res
character*1 ch, ch1

Print *,"Type X to exit, S to scroll, D to pass Direction keys"

123 continue
ch = getcharqq( )
! check for escapes
! 0x00 0x?? is a function key
! 0xE0 0x?? is a direction key
if (ichar(ch) .eq. 0) then
  ch1 = getcharqq()
  print *,"function key follows escape = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
  goto 123
else if (ichar(ch) .eq. 224) then
```

```
  ch1 = getcharqq()
  print *,"direction key follows escape = ",ichar(ch),"  ",ichar(ch1),"  ",ch1
  goto 123
else
  print *,ichar(ch),"  ",ch

  if(ch .eq. 'S') then
    res = passdirkeysqq(.false.)
    print *, "Entering Scroll mode ",res
  endif

  if(ch .eq. 'D')  then
    res = passdirkeysqq(.true.)
    print *, "Entering Direction keys mode ",res
  endif

  if(ch .ne. 'X')  go to 123

endif
end
```

The following example uses an integer constant as an argument to PASSDIRKEYSQQ:

```
c======================================================================
c
c dirkeys4.for
c
c======================================================================
c
c       Compile/Load Input Line for Standard Graphics Full Screen Window
c
c       ifort /libs:qwins dirkeys4.for
c
c       Compile/Load Input Line for QuickWin Graphics
c
c       ifort /libs:qwin dirkeys4.for
c
c  Program to illustrate how to get almost every character
c  from the keyboard in QuickWin or Standard Graphics mode.
c  Comment out the deletemenu line for Standard Graphics mode.
c
c  If you are doing a standard graphics application,
c  control C will come in as a Z'03' without furtherc  effort.
c
c  In a QuickWin application,  The File menu Exit item must
c  be deleted, and PassDirKeysQQ called with PASS_DIR_CNTRLC
c  to get control C.
c
c======================================================================
      use IFQWIN

      integer(4) status

      character*1 key1,key2,ch1

      write(*,*) 'Initializing'

c-----don't do this for a Standard Grapics application
```

```
c     remove File menu Exit item.
      status = deletemenuqq(1,3) ! stop QuickWin from getting control C

c-----set up to pass all keys to window including control c.
      status  = passdirkeysqq(PASS_DIR_CNTRLC)
c==============================================================
c
c     read and print characters
c
c==============================================================

   10 key1 = getcharqq()

c-----first check for control+c
      if(ichar(key1) .eq. 3) then
        write(*,*) 'Control C Received'
        write(*,*) "Really want to quit?"
        write(*,*) "Type Y <cr> to exit, or any other char <cr> to continue."
        read(*,*) ch1
        if(ch1.eq."y" .or. ch1.eq."Y") goto 30
        goto 10
      endif

      if(ichar(key1).eq.0) then ! function key?
      key2 = getcharqq()
      write(*,15) ichar(key1),ichar(key2),key2
   15 format(1x,2i12,1x,a1,' function key')
      else
      if(ichar(key1).eq.224) then ! direction key?
      key2 = getcharqq()
      write(*,16) ichar(key1),ichar(key2),key2
   16 format(1x,2i12,1x,a1,' direction key')
      else
      write(*,20) key1,ichar(key1)  ! normal key
   20 format(1x,a1,i11)
      endif
      endif
      go to 10
   30 stop
      end
```

See Also

GETCHARQQ
INCHARQQ

PAUSE

Statement: *Temporarily suspends program execution and lets you execute operating system commands during the suspension. The PAUSE statement is a deleted feature in the Fortran Standard. Intel® Fortran fully supports features deleted in the Fortran Standard.*

Syntax

```
PAUSE [pause-code]
```

pause-code (Optional) Is an optional message. It can be either of the following:

- A scalar character constant of type default character.
- A string of up to six digits; leading zeros are ignored. (Standard Fortran limits digits to five.)

If you specify *pause-code*, the PAUSE statement displays the specified message and then displays the default prompt.

If you do not specify *pause-code*, the system displays the following default message:

```
    FORTRAN PAUSE
```

The following prompt is then displayed:

- On Windows* systems:

```
Fortran Pause - Enter command<CR> or <CR> to continue.
```

- On Linux* and macOS* systems:

```
PAUSE prompt>
```

For alternate methods of pausing while reading from and writing to a device, see READ and WRITE.

Effect on Windows* Systems

The program waits for input on `stdin`. If you enter a blank line, execution resumes at the next executable statement.

Anything else is treated as a DOS command and is executed by a system( ) call. The program loops, letting you execute multiple DOS commands, until a blank line is entered. Execution then resumes at the next executable statement.

Effect on Linux* and macOS* Systems

The effect of PAUSE differs depending on whether the program is a foreground or background process, as follows:

- If a program is a foreground process, the program is suspended until you enter the CONTINUE command. Execution then resumes at the next executable statement.

  Any other command terminates execution.
- If a program is a background process, the behavior depends on `stdin`, as follows:
  - If `stdin`is redirected from a file, the system displays the following (after the pause code and prompt):

```
To continue from background, execute 'kill -15 n'
```

    In this message, `n`is the process id of the program.
  - If `stdin`is not redirected from a file, the program becomes a suspended background job, and you must specify `fg`to bring the job into the foreground. You can then enter a command to resume or terminate processing.

Example

The following examples show valid PAUSE statements:

```
 PAUSE 701
 PAUSE 'ERRONEOUS RESULT DETECTED'
```

The following shows another example:

```
 CHARACTER*24 filename
 PAUSE 'Enter DIR to see available files or press RETURN'   &
&' if you already know filename.'
```

```
READ(*,'(A\)') filename
OPEN(1, FILE=filename)
. . .
```

See Also

STOP
SYSTEM
Deleted and Obsolescent Language Features

PEEKCHARQQ

Run-Time Function: *Checks the keystroke buffer for a recent console keystroke and returns .TRUE. if there is a character in the buffer or .FALSE. if there is not.*

Module

USE IFCORE

Syntax

```
result = PEEKCHARQQ( )
```

Results

The result type is LOGICAL(4). The result is .TRUE. if there is a character waiting in the keyboard buffer; otherwise, .FALSE..

To find out the value of the key in the buffer, call GETCHARQQ. If there is no character waiting in the buffer when you call GETCHARQQ, GETCHARQQ waits until there is a character in the buffer. If you call PEEKCHARQQ first, you prevent GETCHARQQ from halting your process while it waits for a keystroke. If there is a keystroke, GETCHARQQ returns it and resets PEEKCHARQQ to .FALSE..

Example

```
USE IFCORE
LOGICAL(4) pressed / .FALSE. /
DO WHILE (.NOT. pressed)
  WRITE(*,*) ' Press any key'
  pressed = PEEKCHARQQ ( )
END DO
END
```

See Also

GETCHARQQ
GETSTRQQ
FGETC
GETC

PERROR

Run-Time Subroutine: *Sends a message to the standard error stream, preceded by a specified string, for the last detected error.*

Module

USE IFCORE

Syntax

```
CALL PERROR (string)
```

string	(Input) Character*(*). Message to precede the standard error message.

The string sent is the same as that given by GERROR.

Example

```
USE IFCORE
character*24 errtext
errtext = 'In my opinion, '
. . .
! any error message generated by errtext is
!    preceded by 'In my opinion, '
Call PERROR (errtext)
```

See Also

GERROR
IERRNO

PIE, PIE_W (W*S)

Graphics Functions: *Draw a pie-shaped wedge in the current graphics color.*

Module

USE IFQWIN

Syntax

```
result = PIE (i,x1,y1,x2,y2,x3,y3,x4,y4)
```

```
result = PIE_W (i, wx1, wy1, wx2, wy2, wx3, y3,wx4, wy4)
```

i	(Input) INTEGER(2). Fill flag. One of the following symbolic constants defined in `IFQWIN.F90`: • $GFILLINTERIOR - Fills the figure using the current color and fill mask. • $GBORDER - Does not fill the figure.
x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
x3, y3	(Input) INTEGER(2). Viewport coordinates of start vector.
x4, y4	(Input) INTEGER(2). Viewport coordinates of end vector.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

wx3, wy3	(Input) REAL(8). Window coordinates of start vector.
wx4, wy4	(Input) REAL(8). Window coordinates of end vector.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the pie is clipped or partially out of bounds, the pie is considered successfully drawn and the return is 1. If the pie is drawn completely out of bounds, the return is 0.

The border of the pie wedge is drawn in the current color set by SETCOLORRGB.

The PIE function uses the viewport-coordinate system. The center of the arc is the center of the bounding rectangle, which is specified by the viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*).

The PIE_W function uses the window-coordinate system. The center of the arc is the center of the bounding rectangle specified by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*). The arc starts where it intersects an imaginary line extending from the center of the arc through (*wx3*, *wy3*). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*wx4*, *wy4*).

The fill flag option $GFILLINTERIOR is equivalent to a subsequent call to FLOODFILLRGB using the center of the pie as the starting point and the current graphics color (set by SETCOLORRGB) as the fill color. If you want a fill color different from the boundary color, you cannot use the $GFILLINTERIOR option. Instead, after you have drawn the pie wedge, change the current color with SETCOLORRGB and then call FLOODFILLRGB. You must supply FLOODFILLRGB with an interior point in the figure you want to fill. You can get this point for the last drawn pie or arc by calling GETARCINFO.

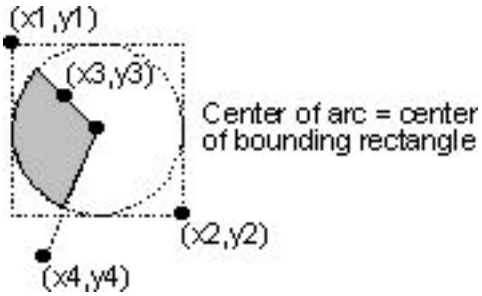
If you fill the pie with FLOODFILLRGB, the pie must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE
The PIE routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Pie routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Pie.

Example

```
! build as Graphics App.
USE IFQWIN
INTEGER(2) status, dummy
INTEGER(2) x1, y1, x2, y2, x3, y3, x4, y4
x1 = 80; y1 = 50
x2 = 180; y2 = 150
x3 = 110; y3 = 80
x4 = 90; y4 = 180

status = SETCOLOR(INT2(4))
dummy = PIE( $GFILLINTERIOR, x1, y1, x2, y2, &
                             x3, y3, x4, y4)
END
```

This figure shows the coordinates used to define PIE and PIE_W:



See Also

SETCOLORRGB
SETFILLMASK
SETLINESTYLE
FLOODFILLRGB
GETARCINFO
ARC
ELLIPSE
GRSTATUS
LINETO
POLYGON
RECTANGLE

POINTER - Fortran

Statement and Attribute: *Specifies that an object or a procedure is a pointer (a dynamic variable). A pointer does not contain data, but points to a scalar or array variable where data is stored. A pointer has no initial storage set aside for it; memory storage is created for the pointer as a program runs.*

Syntax

The POINTER attribute can be specified in a type declaration statement or a POINTER statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] POINTER [, *att-ls*] :: *ptr*[(*d-spec*)][, *ptr*[(*d-spec*)]]...

Statement:

POINTER [::]*ptr*[(*d-spec*)][, *ptr*[(*d-spec*)]] ...

type-spec	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ptr	Is the name of the pointer. The pointer cannot be declared with the INTENT or PARAMETER attributes.
d-spec	(Optional) Is a deferred-shape specification (: [, :] ...). Each colon represents a dimension of the array.

Description

No storage space is created for a data pointer until it is allocated with an ALLOCATE statement or until it is assigned to a allocated target.

Each pointer has an association status, which tells whether the pointer is currently associated with a target object. When a pointer is initially declared, its status is undefined. You can use the ASSOCIATED intrinsic function to find the association status of a pointer if the pointer's association status is defined.

Entities with the POINTER attribute can be associated with different data objects or procedures during execution of a program.

A data pointer must not be referenced or defined unless it is pointer associated with a target object that can be referenced or defined. A procedure pointer must not be referenced unless it is pointer associated with a target procedure.

If the data pointer is an array, and it is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

A pointer cannot be specified in an EQUIVALENCE or NAMELIST statement. A pointer in a DATA statement can only be associated with NULL().

A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

An entity with the POINTER attribute must not have the ALLOCATABLE, INTRINSIC, or TARGET attribute, and it must not be a coarray.

Fortran pointers are *not* the same as integer pointers. For more information, see the POINTER - Integer statement.

Example

The following example shows type declaration statements specifying the POINTER attribute:

```
TYPE(SYSTEM), POINTER :: CURRENT, LAST
REAL, DIMENSION(:,:), POINTER :: I, J, REVERSE
```

The following is an example of the POINTER statement:

```
TYPE(SYSTEM) :: TODAYS
POINTER :: TODAYS, A(:,:)
```

The following shows another example:

```
REAL, POINTER :: arrow (:)
REAL, ALLOCATABLE, TARGET :: bullseye (:,:)

! The following statement associates the pointer with an unused
! block of memory.

ALLOCATE (arrow (1:8), STAT = ierr)
IF (ierr.eq.0) WRITE (*,'(/1x,a)') 'ARROW allocated'
arrow = 5.
WRITE (*,'(1x,8f8.0/)') arrow
ALLOCATE (bullseye (1:8,3), STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'BULLSEYE allocated'
bullseye = 1.
bullseye (1:8:2,2) = 10.
WRITE (*,'(1x,8f8.0)') bullseye

! The following association breaks the association with the first
! target, which being unnamed and unassociated with other pointers,
! becomes lost. ARROW acquires a new shape.

arrow => bullseye (2:7,2)
```

```
WRITE (*,'(/1x,a)') 'ARROW is repointed & resized, all the 5s are lost'
WRITE (*,'(1x,8f8.0)') arrow

NULLIFY (arrow)
IF (.NOT.ASSOCIATED(arrow)) WRITE (*,'(/a/)') ' ARROW is not pointed'

DEALLOCATE (bullseye, STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'Deallocation successful.'
END
```

See Also

ALLOCATE
ASSOCIATED
DEALLOCATE
NULLIFY
TARGET
Deferred-Shape Arrays
Pointer Assignments
Pointer Association
Pointer Arguments
NULL
Integer POINTER statement
Type Declarations
Compatible attributes

POINTER - Integer

Statement: *Establishes pairs of objects and pointers, in which each pointer contains the address of its paired object. This statement is different from the Fortran POINTER statement.*

Syntax

```
POINTER (pointer,pointee) [,(pointer,pointee)] . . .
```

pointer	Is a variable whose value is used as the address of the *pointee*.
pointee	Is a variable; it can be an array name or array specification. It can also be a procedure named in an EXTERNAL statement or in a specific (non-generic) procedure interface block.

The following are *pointer* rules and behavior:

- Two pointers can have the same value, so pointer aliasing is allowed.
- When used directly, a pointer is treated like an integer variable. On IA-32 architecture, a pointer occupies one numeric storage unit, so it is a 32-bit quantity (INTEGER(4)). On Intel® 64 architecture, a pointer occupies two numeric storage units, so it is a 64-bit quantity (INTEGER(8)).
- A pointer cannot be a pointee.
- A pointer cannot appear in an ASSIGN statement and cannot have the following attributes:

ALLOCATABLE	INTRINSIC	POINTER
EXTERNAL	PARAMETER	TARGET

A pointer can appear in a DATA statement with integer literals only.

- Integers can be converted to pointers, so you can point to absolute memory locations.
- A pointer variable cannot be declared to have any other data type.
- A pointer cannot be a function return value.
- You can give values to pointers by doing the following:
  - Retrieve addresses by using the LOC intrinsic function (or the %LOC built-in function)
  - Allocate storage for an object by using the MALLOC intrinsic function (or by using malloc(3f) on Linux* and macOS* systems)

  For example:

```
Using %LOC:                  Using MALLOC:

INTEGER I(10)                INTEGER I(10)
INTEGER I1(10) /10*10/       POINTER (P,I)
POINTER (P,I)                P = MALLOC(40)
P = %LOC(I1)                 I = 10
I(2) = I(2) + 1              I(2) = I(2) + 1
```

- The value in a pointer is used as the pointee's base address.

The following are *pointee* rules and behavior:

- A pointee is not allocated any storage. References to a pointee look to the current contents of its associated pointer to find the pointee's base address.
- A pointee cannot be data-initialized or have a record structure that contains data-initialized fields.
- A pointee can appear in only one integer POINTER statement.
- A pointee array can have fixed, adjustable, or assumed dimensions.
- A pointee cannot appear in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement, and it cannot have the following attributes:

ALLOCATABLE	OPTIONAL	SAVE
AUTOMATIC	PARAMETER	STATIC
INTENT	POINTER	

- A pointee cannot be:
  - A dummy argument
  - A function return value
  - A record field or an array element
  - Zero-sized
  - An automatic object
  - The name of a generic interface block
- If a pointee is of derived type, it must be of sequence type.

Example

```
POINTER (p, k)
INTEGER j(2)

! This has the same effect as j(1) = 0, j(2) = 5
p = LOC(j)
k = 0
p = p + SIZEOF(k) ! 4 for 4-byte integer
k = 5
```

See Also

POINTER - Fortran

LOC

MALLOC
FREE

POLYBEZIER, POLYBEZIER_W (W*S)

Graphics Functions: *Draw one or more Bezier curves.*

Module

USE IFQWIN

Syntax

`result = POLYBEZIER (lppoints,cpoints)`

`result = POLYBEZIER_W (lppoints,cpoints)`

lppoints (Input) Derived type `xycoord`. Array of derived types defining the endpoints and the control points for each Bezier curve. The derived type `xycoord` is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

cpoints (Input) INTEGER(2). Number of points in *lppoints*.

Results

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

A Bezier curve is based on fitting a cubic curve to four points. The first point is the starting point, the next two points are control points, and last point is the ending point. The starting point must be given for the first curve; subsequent curves use the ending point of the previous curve as their starting point. So, *cpoints* should contain 4 for one curve, 7 for 2 curves, 10 for 3 curves, and so forth.

POLYBEZIER does not use or change the current graphics position.

Example

```
Program Bezier
use IFQWIN
! Shows how to use POLYBEZIER, POLYBEZIER_W,
! POLYBEZIERTO, and POLYBEZIERTO_W,
TYPE(xycoord)  lppoints(31)
TYPE(wxycoord) wlppoints(31)
TYPE(xycoord)  xy
TYPE(wxycoord) wxy
integer(4) i
integer(2) istat, orgx, orgy
real(8) worgx, worgy
i = setcolorrgb(Z'00FFFFFF') ! graphic to black
i = settextcolorrgb(Z'00FFFFFF') ! text to black
i = setbkcolorrgb(Z'00000000') ! background to white
call clearscreen($GCLEARSCREEN)
orgx = 20
orgy = 20
lppoints(1)%xcoord = 1+orgx
lppoints(1)%ycoord = 1+orgy
```

```
lppoints(2)%xcoord = 30+orgx
lppoints(2)%ycoord = 120+orgy
lppoints(3)%xcoord = 150+orgx
lppoints(3)%ycoord = 60+orgy
lppoints(4)%xcoord = 180+orgx
lppoints(4)%ycoord = 180+orgy
istat = PolyBezier(lppoints, 4)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
do i = 1,4,2
call moveto(lppoints(i)%xcoord,lppoints(i)%ycoord,xy)
istat = lineto(lppoints(i+1)%xcoord,lppoints(i+1)%ycoord)
end do
read(*,*)
worgx = 50.0
worgy = 50.0
wlppoints(1)%wx = 1.0+worgx
wlppoints(1)%wy = 1.0+worgy
wlppoints(2)%wx = 30.0+worgx
wlppoints(2)%wy = 120.0+worgy
wlppoints(3)%wx = 150.0+worgx
wlppoints(3)%wy = 60.0+worgy
wlppoints(4)%wx = 180.0+worgx
wlppoints(4)%wy = 180.0+worgy
i = setcolorrgb(Z'000000FF') ! graphic to red
istat = PolyBezier_W(wlppoints, 4)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
do i = 1,4,2
call moveto_w(wlppoints(i)%wx,wlppoints(i)%wy,wxy)
istat = lineto_w(wlppoints(i+1)%wx,wlppoints(i+1)%wy)
end do
read(*,*)
orgx = 80
orgy = 80
! POLYBEZIERTO uses the current graphics position
! as its initial starting point so we start the
! array with the first first control point.
! lppoints(1)%xcoord = 1+orgx ! need to move to this
! lppoints(1)%ycoord = 1+orgy
lppoints(1)%xcoord = 30+orgx
lppoints(1)%ycoord = 120+orgy
lppoints(2)%xcoord = 150+orgx
lppoints(2)%ycoord = 60+orgy
lppoints(3)%xcoord = 180+orgx
lppoints(3)%ycoord = 180+orgy
i = setcolorrgb(Z'0000FF00') ! graphic to green
call moveto(1+orgx,1+orgy,xy)
istat = PolyBezierTo(lppoints, 3)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
```

```
! point.   It is also tangent to the line from
! the second control point to the end point.
call moveto(1+orgx,1+orgy,xy)
istat = lineto(lppoints(1)%xcoord,lppoints(1)%ycoord)
call moveto(lppoints(2)%xcoord,lppoints(2)%ycoord,xy)
istat = lineto(lppoints(3)%xcoord,lppoints(3)%ycoord)
read(*,*)
worgx = 110.0
worgy = 110.0
!wlppoints(1)%wx = 1.0+worgx
!wlppoints(1)%wy = 1.0+worgy
wlppoints(1)%wx = 30.0+worgx
wlppoints(1)%wy = 120.0+worgy
wlppoints(2)%wx = 150.0+worgx
wlppoints(2)%wy = 60.0+worgy
wlppoints(3)%wx = 180.0+worgx
wlppoints(3)%wy = 180.0+worgy
call moveto_w(1.0+worgx,1.0+worgy,wxy)
i = setcolorrgb(Z'00FF0000') ! graphic to blue
istat = PolyBezierTo_W(wlppoints, 3)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
call moveto_w(1.0+worgx,1.0+worgy,wxy)
istat = lineto_w(wlppoints(1)%wx,wlppoints(1)%wy)
call moveto_w(wlppoints(2)%wx,wlppoints(2)%wy,wxy)
istat = lineto_w(wlppoints(3)%wx,wlppoints(3)%wy)
read(*,*)
END PROGRAM Bezier
```

See Also

POLYBEZIERTO, POLYBEZIERTO_W

POLYBEZIERTO, POLYBEZIERTO_W (W*S)

Graphics Functions: *Draw one or more Bezier curves.*

Module

USE IFQWIN

Syntax

`result = POLYBEZIERTO (lppoints,cpoints)`

`result = POLYBEZIERTO_W (lppoints,cpoints)`

lppoints

(Input) Derived type `xycoord`. Array of derived types defining the endpoints and the control points for each Bezier curve. The derived type `xycoord` is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

cpoints (Input) INTEGER(2). Number of points in *ppoints* or *lppoints*.

Results

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

A Bezier curve is based on fitting a cubic curve to four points. The first point is the starting point, the next two points are control points, and last point is the ending point. The starting point is the current graphics position as set by MOVETO for the first curve; subsequent curves use the ending point of the previous curve as their starting point. So, *cpoints* should contain 3 for one curve, 6 for 2 curves, 9 for 3 curves, and so forth.

POLYBEZIERTO moves the current graphics position to the ending point of the last curve drawn.

Example

See the example in POLYBEZIER, POLYBEZIER_W (W*S).

See Also

POLYBEZIER, POLYBEZIER_W
MOVETO, MOVETO_W

POLYGON, POLYGON_W (W*S)

Graphics Functions: *Draw a polygon using the current graphics color, logical write mode, and line style.*

Module

USE IFQWIN

Syntax

```
result = POLYGON (control,lppoints,cpoints)
result = POLYGON_W (control,lppoints,cpoints)
```

control (Input) INTEGER(2). Fill flag. One of the following symbolic constants defined in `IFQWIN.F90`:

- $GFILLINTERIOR - Draws a solid polygon using the current color and fill mask.
- $GBORDER - Draws the border of a polygon using the current color and line style.

lppoints (Input) Derived type `xycoord`. Array of derived types defining the polygon vertices in viewport coordinates. The derived type `xycoord` is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

cpoints (Input) INTEGER(2). Number of polygon vertices.

Results

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

The border of the polygon is drawn in the current graphics color, logical write mode, and line style, set with SETCOLORRGB, SETWRITEMODE, and SETLINESTYLE, respectively. The POLYGON routine uses the viewport-coordinate system (expressed in `xycoord` derived types), and the POLYGON_W routine uses real-valued window coordinates (expressed in `xycoord` types).

The argument *lppoints* is an array whose elements are `xycoord` derived types. Each element specifies one of the polygon's vertices. The argument *cpoints* is the number of elements (the number of vertices) in the *lppoints* array.

Note that POLYGON draws between the vertices in their order in the array. Therefore, when drawing outlines, skeletal figures, or any other figure that is not filled, you need to be careful about the order of the vertices. If you don't want lines between some vertices, you may need to repeat vertices to make the drawing backtrack and go to another vertex to avoid drawing across your figure. Also, POLYGON draws a line from the last specified vertex back to the first vertex.

If you fill the polygon using FLOODFILLRGB, the polygon must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE
The POLYGON routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Polygon routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Polygon.

Example

```
! Build as a Graphics App.

! Draw a skeletal box
    USE IFQWIN
    INTEGER(2) status
    TYPE (xycoord) poly(12)

! Set up box vertices in order they will be drawn, &
!    repeating some to avoid unwanted lines across box

    poly(1)%xcoord = 50
    poly(1)%ycoord = 80
    poly(2)%xcoord = 85
    poly(2)%ycoord = 35
    poly(3)%xcoord = 185
    poly(3)%ycoord = 35
    poly(4)%xcoord = 150
    poly(4)%ycoord = 80
    poly(5)%xcoord = 50
    poly(5)%ycoord = 80
    poly(6)%xcoord = 50
    poly(6)%ycoord = 180
    poly(7)%xcoord = 150
    poly(7)%ycoord = 180
    poly(8)%xcoord = 185
    poly(8)%ycoord = 135
    poly(9)%xcoord = 185
    poly(9)%ycoord = 35
    poly(10)%xcoord = 150
    poly(10)%ycoord = 80
    poly(11)%xcoord = 150
    poly(11)%ycoord = 180
```

```
    poly(12)%xcoord = 150
    poly(12)%ycoord = 80

    status = SETCOLORRGB(Z'0000FF')
    status = POLYGON($GBORDER, poly, INT2(12))
    END
```

See Also

SETCOLORRGB
SETFILLMASK
SETLINESTYLE
FLOODFILLRGB
GRSTATUS
LINETO
RECTANGLE
SETWRITEMODE

POLYLINEQQ (W*S)

Graphics Function: *Draws a line between each successive x, y point in a given array.*

Module

USE IFQWIN

Syntax

`result = POLYLINEQQ (points,cnt)`

points — (Input) An array of `DF_POINT` objects. The derived type `DF_POINT` is defined in `IFQWIN.F90` as:

```
type DF_POINT
  sequence
  integer(4) x
  integer(4) y
end type DF_POINT
```

cnt — (Input) INTEGER(4). Number of elements in the *points* array.

Results

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, zero.

POLYLINEQQ uses the viewport-coordinate system.

The lines are drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

The current graphics position is not used or changed as it is in the LINETO function.

Example

```
! Build for QuickWin or Standard Graphics
USE IFQWIN
TYPE(DF_POINT) points(12)
integer(4) result
integer(4) cnt, i
```

```
! load the points
do i = 1,12,2
  points(i).x =20*i
  points(i).y =10
  points(i+1).x =20*i
  points(i+1).y =60
end do

! A sawtooth pattern will appear in the upper left corner
result = POLYLINEQQ(points, 12)
end
```

See Also

LINETO
LINETOAREX
SETCOLORRGB
SETLINESTYLE
SETWRITEMODE

POPCNT

Elemental Intrinsic Function (Generic): *Returns the number of 1 bits in the integer argument.*

Syntax

```
result = POPCNT (i)
```

i	(Input) Must be of type integer or logical.

Results

The result type and kind are the same as *i*. The result value is the number of 1 bits in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPCNT(I) is 5.

POPPAR

Elemental Intrinsic Function (Generic): *Returns the parity of the integer argument.*

Syntax

```
result = POPPAR (i)
```

i	(Input) Must be of type integer or logical.

Results

The result type and kind are the same as *i*. The result value is 1 if there are an odd number of 1 bits in the binary representation of the integer I. The result value is zero if there are an even number.

POPPAR(*i*) is the same as 1 .AND. POPCNT(*i*).

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPPAR(I) is 1.

PRECISION

Inquiry Intrinsic Function (Generic): *Returns the decimal precision in the model representing real numbers with the same kind parameter as the argument.*

Syntax

```
result = PRECISION (x)
```

x	(Input) Must be of type real or complex; it can be scalar or array valued.

Results

The result is a scalar of type default integer. The result has the value INT((DIGITS(*x*) - 1) * LOG10(RADIX(*x*))). If RADIX(*x*) is an integral power of 10, 1 is added to the result.

Example

If X is a REAL(4) value, PRECISION(X) has the value 6. The value 6 is derived from INT ((24-1) * LOG10 (2.)) = INT (6.92...). For more information on the model for REAL(4), see Model for Real Data.

PREFETCH and NOPREFETCH

General Compiler Directives: *PREFETCH hints to the compiler to prefetch data into closer levels of cache. Prefetching data can minimize the effects of memory latency. NOPREFETCH disables data prefetching. These directives give fine-level control to the programmer to influence the prefetches generated by the compiler.*

Syntax

```
!DIR$ PREFETCH [var1[: hint1[: distance1]] [,var2[: hint2[: distance2]]]...]
```

```
!DIR$ PREFETCH *: hint[: distance]
```

```
!DIR$ NOPREFETCH [var1[,var2]...]
```

var	Is an optional memory reference.
hint	Is an optional integer constant expression with the integer value 0, 1, 2, or 3. These are the same as the defined values FOR_K_PREFETCH_T0, FOR_K_PREFETCH_T1, FOR_K_PREFETCH_T2, or FOR_K_PREFETCH_NTA for *hint* in the intrinsic subroutine MM_PREFETCH. To use this argument, you must also specify *var*.
distance	Is an optional integer constant expression with a value greater than 0. It indicates the number of (possibly vectorized) loop iterations ahead of which a prefetch is issued, before the corresponding load or store instruction. To use this argument, you must also specify *var* and *hint*.

To use these directives, compiler option `[q or Q]opt-prefetch` must be set. Note that this option is turned on by default if the compiler general optimization level is O2 or higher.

This directive affects the DO loop it precedes.

If you specify PREFETCH *, the *hint* and optional *distance* are used to prefetch all array accesses in the DO loop.

If you specify NOPREFETCH with no arguments, the following occurs:

- All arrays accessed in the DO loop will NOT be prefetched.
- It negates all other PREFETCH directives for the following DO loop.

If a loop includes expression A(j), placing !DIR$ PREFETCH A:0:d in front of the loop instructs the compiler to insert a vprefetch0 instruction for A within the loop that is d iterations ahead.

The PREFETCH directive takes precedence over the `[q or Q]opt-prefetch-distance` options.

The variables in a NOPREFETCH or PREFETCH directive take precedence over PREFETCH *.

A NOPREFETCH directive with no arguments negates all other PREFETCH directives for the following DO loop.

Example

```
! Issue no prefetches for A1
! Issue vector prefetch from L2 and higher caches for B with a distance
!   of 16 vectorized iterations ahead
! Issue vector prefetch from L1 and higher caches for B with a distance
!   of 4 vectorized iterations ahead
!DIR$ NOPREFETCH A1
!DIR$ PREFETCH B:1:16
!DIR$ PREFETCH B:0:4
   DO J = 1,N
      A1(J) = B(J-1) + B(J+1)
   END DO
```

In the following example, array A will be prefetched with hint 0 and distance 5, arrays B and C will be prefetched with hint 1 and distance 10, and array D will not be prefetched:

```
!DIR$ PREFETCH *:1:10
!DIR$ PREFETCH A:0:5
!DIR$ NOPREFETCH D
    DO J = 1, N
       A (J) = B (J) + C (J) + D (J)
    END DO
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives

MM_PREFETCH

o compiler option

`qopt-prefetch-distance, Qopt-prefetch-distance` compiler option

PRESENT

Inquiry Intrinsic Function (Generic): *Returns whether or not an optional dummy argument is present, that is, whether it has an associated actual argument.*

Syntax

```
result = PRESENT (a)
```

a	(Input) Must be an argument of the current procedure and must have the OPTIONAL attribute. An explicit interface for the current procedure must be visible to its caller; for more information, see Procedure Interfaces.

Results

The result is a scalar of type default logical. The result is .TRUE. if *a* is present; otherwise, the result is .FALSE..

Example

Consider the following:

```
MODULE MYMOD
CONTAINS
SUBROUTINE CHECK (X, Y)
  REAL X, Z
  REAL, OPTIONAL :: Y
  ...
  IF (PRESENT (Y)) THEN
    Z = Y
  ELSE
     Z = X * 2
  END IF
END SUBROUTINE CHECK
END MODULE MYMOD
...
USE MYMOD
CALL CHECK (15.0, 12.0) ! Causes Z to be set to 12.0
CALL CHECK (15.0) ! Causes Z to be set to 30.0
```

The following shows another example:

```
CALL who( 1, 2 ) ! prints "A present" "B present"
CALL who( 1 ) ! prints "A present"
CALL who( b = 2 ) ! prints "B present"
CALL who( ) ! prints nothing
CONTAINS
  SUBROUTINE who( a, b )
    INTEGER(4), OPTIONAL :: a, b
    IF (PRESENT(a)) PRINT *,'A present'
    IF (PRESENT(b)) PRINT *,'B present'
  END SUBROUTINE who
END
```

See Also

OPTIONAL
Optional Arguments

PRINT

Statement: *Displays output on the screen. TYPE is a synonym for PRINT. All forms and rules for the PRINT statement also apply to the TYPE statement.*

Syntax

The PRINT statement is the same as a formatted, sequential WRITE statement, except that the PRINT statement must never transfer data to user-specified I/O units. You can override this restriction by using environment variable FOR_PRINT.

A PRINT statement takes one of the following forms:

Formatted:

```
PRINT form[, io-list]
```

Formatted - List-Directed:

```
PRINT *[, io-list]
```

Formatted - Namelist:

```
PRINT nml
```

form	Is the nonkeyword form of a format specifier (no FMT=).
io-list	Is an I/O list.
*	Is the format specifier indicating list-directed formatting.
nml	Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

Example

In the following example, one record (containing four fields of data) is printed to the implicit output device:

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=', A, 'JOB=', A)
```

The following shows another example:

```
!   The following statements are equivalent:
    PRINT     '(A11)', 'Abbottsford'
    WRITE (*, '(A11)') 'Abbottsford'
    TYPE      '(A11)', 'Abbottsford'
```

See Also

PUTC
READ
WRITE
FORMAT
Data Transfer I/O Statements
File Operation I/O Statements
PRINT as a value in CLOSE

PRIORITY

Parallel Directive Clause: *Specifies that the generated tasks have the indicated priority for execution.*

Syntax

```
PRIORITY (priority-value)
```

priority-value	Provides a hint for the priority of task execution order. *priority-value* must evaluate to a non-negative scalar integer value.

Among all tasks ready to be executed, higher priority tasks (those with a higher numerical value of *priority-value*) are recommended to execute before lower priority tasks. A program that relies on task execution order being determined by this *priority-value* may have unspecified behavior.

At most one PRIORITY clause can appear in the directive.

If this clause is not specified, tasks generated by the construct have a task priority of zero (the default).

PRIVATE Clause

Parallel Directive Clause: *Declares one or more variables to be private to each thread in a team.*

Syntax

PRIVATE (*list*)

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

This clause allows each thread to have its own copy of the variable and makes it a local variable to the thread.

The following occurs when variables are declared in a PRIVATE clause:

- A new object of the same type is declared once for each thread in the team (or for each implicit task in a region) and it is used by each thread (or task) inside the scope of the directive construct instead of the original variable. The new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread upon entering the construct and the corresponding shared variable is undefined when the parallel construct is exited. Within a parallel, worksharing, or task region, the initial status of a private pointer is undefined.

The value and allocation status of the original variable will change only in the following cases:

- If it is accessed and modified by means of a pointer
- If it is accessed in the region but outside of the construct
- As a side effect of directives or clauses
- If accessed and modified via construct association.

If a variable has the ALLOCATABLE attribute, the following rules apply:

- If the variable is not currently allocated, the new list item will have an initial state of "unallocated".
- If the variable is allocated, the new list item will have an initial state of allocated with the same array bounds.

A variable that appears in a PRIVATE clause may be storage-associated with other variables when the PRIVATE clause is encountered by constructs such as EQUIVALENCE or COMMON. If A is a variable appearing in a PRIVATE clause and B is a variable that is storage-associated with A, then the following applies:

- The contents, allocation, and association status of B are undefined on entry to the parallel or task region.
- Any definition of A, or any definition of its allocation or association status, causes the contents, allocation, and association status of B to become undefined.
- Any definition of B, or any definition of its allocation or association status, causes the contents, allocation, and association status of A to become undefined.

- A list item that appears in a private clause can be a *selector* in an ASSOCIATE construct. If the ASSOCIATE construct association is established before entry to a parallel region, the association between the associate *name* and the original PRIVATE list item will be retained in the parallel region.

The following are restrictions for the PRIVATE clause:

- A variable that is part of another variable (as an array or structure element) must not appear in a PRIVATE clause.
- A variable that appears in a PRIVATE clause must either be definable or it must be an allocatable array. This restriction does not apply to the FIRSTPRIVATE clause.
- Assumed-size arrays must not appear in a PRIVATE clause.
- A dummy argument that is a pointer with the INTENT (IN) attribute must not appear in a PRIVATE clause. This restriction does not apply to the FIRSTPRIVATE clause.
- Variables that appear in NAMELIST statements, in variable format expressions, and in expressions for statement function definitions, must not appear in a PRIVATE clause.

Variables in a *list* can appear in other clauses as follows:

- Variables that appear in a PRIVATE, FIRSTPRIVATE, or REDUCTION clause in a parallel construct can also appear in a PRIVATE clause in an enclosed parallel, task, or worksharing construct.
- Variables that appear in a PRIVATE or FIRSTPRIVATE clause in a task construct can also appear in a PRIVATE clause in an enclosed parallel or task construct.
- Variables that appear in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause in a worksharing construct can also appear in a PRIVATE clause in an enclosed parallel or task construct.

NOTE
Variables that are used as counters for explicit or implicit DO loops or FORALL commands, or common blocks that are specified to be THREADPRIVATE become automatically private to each thread, even though they are not explicitly included inside a PRIVATE clause at the beginning of the scope of the parallel region.

Example

Consider the following:

```
!$OMP PARALLEL PRIVATE(A, B)
```

In this simple case, each thread will have its own copy of variables A and B. The variables can have different values in each thread because the variables are local to the thread.

See Also

FIRSTPRIVATE clause
LASTPRIVATE clause

PRIVATE Statement

Statement and Attribute: *Specifies that entities in a module can be accessed only within the module itself.*

Syntax

The PRIVATE attribute can be specified in a type declaration statement or a PRIVATE statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] PRIVATE [, *att-ls*] :: *entity*[, *entity*]...

Statement:

PRIVATE [[::] *entity*[, *entity*] ...]

type Is a data type specifier.

att-ls Is an optional list of attribute specifiers.

entity Is one of the following:

- A variable name
- A procedure name
- A derived type name
- A named constant
- A namelist group name
- An OpenMP* reduction-identifier

In statement form, an entity can also be one of the following:

- A module name
- A generic name
- A defined operator
- A defined assignment
- A defined I/O generic specification

Description

The PRIVATE attribute can only appear in the scoping unit of a module.

Only one PRIVATE statement without an entity list is permitted in the scoping unit of a module; it sets the default accessibility of all entities in the module and any entities accessed from a module by USE association whose names do not appear in a PUBLIC statement.

A module name can appear at most once in all the PUBLIC or PRIVATE statements in a scoping unit. If a module name is specified in a PRIVATE statement, the module name must have appeared in a USE statement within the same scoping unit.

A PRIVATE attribute can be specified on the TYPE statement of a derived-type definition in a module. This specifies the derived-type definition is not accessible outside the module or the module's descendants.

A PRIVATE statement without an entity-list can appear in the component-definition part of a derived-type definition, specifying the default accessibility of all components as PRIVATE. A PRIVATE statement with no entity-list can appear in the type-bound-procedure-part of a derived-type definition specifying the default accessibility of all type-bound procedures of that type as PRIVATE. In such cases, the default accessibility of components and type-bound procedures can be overridden by explicitly declaring a component or type-bound procedure PUBLIC.

A PRIVATE attribute can be specified in the statement declaring a component or a type-bound procedure of a derived type. This specifies that the component or type-bound procedure is not accessible outside the module or its descendants.

If no PRIVATE statements are specified in a module, the default is PUBLIC accessibility. Entities with PUBLIC accessibility can be accessed from outside the module by means of a USE statement.

If a derived type is declared PRIVATE in a module, its components are also PRIVATE. The derived type and its components are accessible to any subprograms within the defining module and the module's descendants through host association, but they are not accessible from outside the module.

If the derived type is declared PUBLIC in a module, but its components are declared PRIVATE, any scoping unit accessing the module though use association (or host association) can access the derived-type definition, but not its components.

If a module procedure has a dummy argument or a function result of a type that has PRIVATE accessibility, the module procedure must have PRIVATE accessibility. If the module has a generic identifier, it must also be declared PRIVATE.

If a procedure has a generic identifier, the accessibility of the procedure's specific name is independent of the accessibility of its generic identifier. One can be declared PRIVATE and the other PUBLIC.

The accessibility of the components of a type is independent of the accessibility of the type name. The following combinations are possible:

- A private type name with a private component
- A public type name with a public component
- A private type name with a public component
- A public type name with a private component

The accessibility of a type does not affect, and is not affected by, the accessibility of its components and type-bound procedures. If a type definition is private, then the type name, and thus the structure constructor for the type, are accessible only within the module containing the definition.

Example

The following examples show type declaration statements specifying the PUBLIC and PRIVATE attributes:

```
REAL,  PRIVATE  :: A, B, C
INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the PUBLIC and PRIVATE statements:

```
MODULE SOME_DATA
  REAL ALL_B
  PUBLIC ALL_B
  TYPE RESTRICTED_DATA
    REAL LOCAL_C(50)
  END TYPE RESTRICTED_DATA
  PRIVATE RESTRICTED_DATA
END MODULE
```

The following derived-type declaration statement indicates that the type is restricted to the module:

```
TYPE, PRIVATE  :: DATA
  ...
END TYPE DATA
```

The following example shows a PUBLIC type with PRIVATE components:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER, can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

The following shows another example:

```
 !LENGTH in module VECTRLEN calculates the length of a 2-D vector.
 !The module contains both private and public procedures
    MODULE VECTRLEN
      PRIVATE SQUARE
      PUBLIC LENGTH
     CONTAINS
     SUBROUTINE LENGTH(x,y,z)
        REAL,INTENT(IN) x,y
```

```
      REAL,INTENT(OUT) z
      CALL SQUARE(x,y)
      z = SQRT(x + y)
      RETURN
   END SUBROUTINE
   SUBROUTINE SQUARE(x1,y1)
      REAL x1,y1
      x1 = x1**2
      y1 = y1**2
      RETURN
   END SUBROUTINE
  END MODULE
```

See Also

MODULE
PUBLIC
TYPE
Defining Generic Names for Procedures
USE
Use and Host Association
Type Declarations
Compatible attributes

PROCEDURE

Statement: *Declares procedure pointers, dummy procedures, and external procedures.*

Syntax

PROCEDURE ([*proc-interface*]) [[, *proc-attr-spec*]... ::] *proc-decl-list*

proc-interface

(Optional) Is the name of an interface, an intrinsic type specifier, or a derived-type TYPE statement.

If an interface name is specified, it must be the name of an abstract interface, a procedure that has an explicit interface, or a procedure pointer. It must have been previously declared and it cannot be the same as a keyword that specifies an intrinsic type.

If *proc-interface* is a type specifier, the declared procedures or procedure pointers are functions that have implicit interfaces and the specified result type. If a type is specified for an external function, its function definition must specify the same result type and type parameters.

proc-attr-spec

(Optional) Is one of the following attributes:

- PUBLIC
- PRIVATE
- BIND (C [, NAME = *init-expr*])
  This is also called a *language-binding-spec*. The *init-expr* is a scalar character constant expression of default character kind. If NAME= is specified, there can only be one *proc-decl* item, which cannot have the POINTER attribute or be a dummy procedure.
- INTENT(INOUT)

	• OPTIONAL • POINTER • PROTECTED • SAVE Each *proc-attr-spec* gives the corresponding attribute to all procedures declared in that statement. If a procedure entity has the INTENT attribute or SAVE attribute, it must also have the POINTER attribute.
proc-decl-list	Is one or more of the following: `procedure-name` [=> *null-init*] where *null-init* is a reference to intrinsic function NULL with no arguments. If => *null-init* appears, the procedure must have the POINTER attribute. `procedure-name` is the name of a nonintrinsic procedure

Description

A PROCEDURE statement declares nonintrinsic procedures (procedure pointers, dummy procedures, internal procedures, and external procedures). If the procedure is not a procedure pointer, a module procedure, or an internal procedure, it specifies the EXTERNAL attribute for the procedure.

You cannot use the PROCEDURE statement to identify a BLOCK DATA subprogram.

If => *null-init* appears, it specifies that the initial association status of the corresponding procedure entity is disassociated. It also implies the SAVE attribute.

You can also declare procedures by using an EXTERNAL statement or a procedure component definition statement.

If the BIND attribute is specified for a procedure, each dummy argument must be:

- An interoperable procedure or a variable that is interoperable
- Assumed shape
- Assumed rank
- Assumed type
- Of assumed-character length or has the ALLOCATABLE or POINTER attribute

If the BIND attribute is specified for a function, the function result must be an interoperable scalar variable.

A Fortran procedure is interoperable if it has the BIND attribute; that is, if its interface is specified with a *language-binding-spec*.

The ALLOCATABLE or POINTER attribute must not be specified for a default-initialized dummy argument of a procedure that has a BIND attribute.

A dummy argument of a procedure that has a BIND attribute must not have both the OPTIONAL and VALUE attributes.

A variable that is a dummy argument of a procedure that has a BIND attribute must be of interoperable type or assumed type.

When a procedure whose interface has the BIND attribute is called, any actual argument corresponding to a dummy argument with the ALLOCATABLE or POINTER attribute is passed by C descriptor. Similarly, if a Fortran procedure with a BIND attribute has such dummy arguments, they are received by C descriptor.

Example

Consider the following:

```
PROCEDURE (NAME_TEMP) :: NAME37
```

The above declares `NAME37` to be a procedure with an identical interface to that of `NAME_TEMP`.

The following are equivalent:

```
PROCEDURE (INTEGER) X
INTEGER, EXTERNAL :: X
```

Consider the following:

```
ABSTRACT INTERFACE
  FUNCTION SIM_FUNC (X)
    REAL, INTENT (IN) :: X
    REAL :: SIM_FUNC
  END FUNCTION SIM_FUNC
END INTERFACE

INTERFACE
  SUBROUTINE SUB2 (X)
    REAL, INTENT (IN) :: X
  END SUBROUTINE SUB2
END INTERFACE
```

The following shows external and dummy procedures with explicit interfaces:

```
PROCEDURE (SIM_FUNC) :: VMAC, KAPPA
PROCEDURE (SUB2) :: PRINGLES
```

The following shows procedure pointers with an explicit interface, one initialized to NULL():

```
PROCEDURE (SIM_FUNC), POINTER :: P1, R1 => NULL()
PROCEDURE (SIM_FUNC), POINTER :: KAPPA_POINTER
```

The following shows a derived type with a procedure pointer component:

```
TYPE MEMO_TYPE
  PROCEDURE (SIM_FUNC), POINTER :: COMPONENT
END TYPE MEMO_TYPE
```

The following shows a variable of the above type:

```
TYPE (MEMO_TYPE) :: STRUCTA
```

The following shows an external or dummy function with an implicit interface:

```
PROCEDURE (INTEGER) :: PHIL0
```

See Also

Procedure Interfaces
Type-Bound Procedures
ABSTRACT INTERFACE
INTERFACE
EXTERNAL attribute

PROCESSOR Clause

Parallel Directive Clause: *Tells the compiler to create a vector version of the routine for the specified processor. When running on a processor that does not match "cpuid", a scalar version will be invoked multiple times based on vector length.*

Syntax

`PROCESSOR (cpuid)`

cpuid

Is one of the following:

`atom`	Intel Atom® processors with Supplemental Streaming SIMD Extensions 3 (SSSE3)
`atom_sse4_2`	Intel Atom® processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2)
`atom_sse4_2_movbe`	Intel Atom® processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) with MOVBE instructions enabled
`broadwell`	This is a synonym for `core_5th_gen_avx`.
`core_2_duo_sse4_1`	Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture processors with Intel® Streaming SIMD Extensions 4.1 (Intel® SSE4.1)
`core_2_duo_ssse3`	Intel® Core™2 Duo processors with Intel® Supplemental Streaming SIMD Extensions 3 (SSSE3)
`core_2nd_gen_avx`	2nd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX)

`core_3rd_gen_avx`	3rd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX) including the RDRND instruction
`core_4th_gen_avx`	4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction
`core_4th_gen_avx_tsx`	4th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDRND instruction, and support for Intel® Transactional Synchronization Extensions (Intel® TSX)
`core_5th_gen_avx`	5th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDSEED and Multi-Precision Add-Carry Instruction Extensions (ADX) instructions
`core_5th_gen_avx_tsx`	5th generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions 2 (Intel® AVX2) including the RDSEED and Multi-Precision Add-Carry Instruction Extensions (ADX) instructions, and support for Intel® Transactional Synchronization Extensions (Intel® TSX)
`core_aes_pclmulqdq`	Intel® Core™ processors with support for Advanced Encryption Standard (AES) instructions and carry-less multiplication instruction
`core_i7_sse4_2`	Intel® Core™ i7 processors with Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2) instructions
`generic`	Other Intel processors for IA-32 or Intel® 64 architecture or compatible processors not provided by Intel Corporation

`haswell`	This is a synonym for `core_4th_gen_avx`.
`pentium`	Intel® Pentium® processor
`pentium_4`	Intel® Pentium® 4 processor
`pentium_4_sse3`	Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3) instructions, Intel® Core™ Duo processors, Intel® Core™ Solo processors
`pentium_ii`	Intel® Pentium® II processors
`pentium_iii`	Intel® Pentium® III processors
`pentium_iii_no_xmm_regs`	Intel® Pentium® III processors with no XMM registers
`pentium_m`	Intel® Pentium® M processors
`pentium_mmx`	Intel® Pentium® processors with MMX™ technology
`pentium_pro`	Intel® Pentium® Pro processors
`skylake`	Intel® microarchitecture code name Skylake. This keyword targets the Client CPU *without* support for Intel® AVX-512 instructions.

`skylake_avx512`	Intel® microarchitecture code name Skylake. This keyword targets the Server CPU *with* support for Intel® AVX-512 instructions.

The vector version of the routine that is created by the compiler is not affected by processor options specified on the command line.

Multiple PROCESSOR clauses cause a syntax error.

See Also

DECLARE SIMD
ATTRIBUTES VECTOR

PRODUCT

Transformational Intrinsic Function (Generic):
Returns the product of all the elements in an entire array or in a specified dimension of an array.

Syntax

`result = PRODUCT (array[,dim] [,mask])`

array	(Input) Must be an array of type integer, real, or complex.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If PRODUCT(*array*) is specified, the result is the product of all elements of *array*. If *array* has size zero, the result is 1.
- If PRODUCT(*array*, MASK= *mask*) is specified, the result is the product of all elements of *array* corresponding to true elements of *mask*. If *array* has size zero, or every element of *mask* has the value .FALSE., the result is 1.

The following rules apply if *dim* is specified:

- If *array* has rank one, the value is the same as PRODUCT(*array*[,MASK= *mask*]).
- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- The value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of PRODUCT(*array*, *dim*[, *mask*]) is equal to PRODUCT(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK= *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

Example

PRODUCT ((/2, 3, 4/)) returns the value 24 (the product of 2 * 3 * 4). PRODUCT ((/2, 3, 4/), DIM=1) returns the same result.

PRODUCT (C, MASK=C .LT. 0.0) returns the product of the negative elements of C.

A is the array

```
  [ 1  4  7 ]
  [ 2  3  5 ].
```

PRODUCT (A, DIM=1) returns the value (2, 12, 35), which is the product of all elements in each column. 2 is the product of 1 * 2 in column 1. 12 is the product of 4 * 3 in column 2, and so forth.

PRODUCT (A, DIM=2) returns the value (28, 30), which is the product of all elements in each row. 28 is the product of 1 * 4 * 7 in row 1. 30 is the product of 2 * 3 * 5 in row 2.

If *array* has shape (2, 2, 2), *mask* is omitted, and *dim* is 1, the result is an array result with shape (2, 2) whose elements have the following values.

Resultant array element	Value
result(1, 1)	*array*(1, 1, 1) * *array*(2, 1, 1)
result(2, 1)	*array*(1, 2, 1) * *array*(2, 2, 1)
result(1, 2)	*array*(1, 1, 2) * *array*(2, 1, 2)
result(2, 2)	*array*(1, 2, 2) * *array*(2, 2, 2)

The following shows another example:

```
INTEGER array (2, 3)
INTEGER AR1(3), AR2(2)
array = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2,3/))
! array is  1 2 3
!           4 5 6
AR1 = PRODUCT(array, DIM = 1) ! returns [ 4 10 18 ]
AR2 = PRODUCT(array, MASK = array .LT. 6, DIM = 2)
                                ! returns [ 6 20 ]
END
```

See Also

SUM

PROGRAM

Statement: *Identifies the program unit as a main program and gives it a name.*

Syntax

```
[PROGRAM name]
   [specification-part]
   [execution-part]
[CONTAINS
   internal-subprogram-part]
END[PROGRAM [name]]
```

name — Is the name of the program.

specification-part — Is one or more specification statements, except for the following:

- INTENT (or its equivalent attribute)

	• OPTIONAL (or its equivalent attribute) • PUBLIC and PRIVATE (or their equivalent attributes) An automatic object must not appear in a specification statement. If a SAVE statement is specified, it has no effect.
execution-part	Is one or more executable constructs or statements, except for ENTRY or RETURN statements.
internal-subprogram-part	Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Description

The PROGRAM statement is optional. Within a program unit, a PROGRAM statement can be preceded only by comment lines or an OPTIONS statement.

The END statement is the only required part of a program. If a name follows the END statement, it must be the same as the name specified in the PROGRAM statement.

The program name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A main program must not reference itself (either directly or indirectly).

Example

The following is an example of a main program:

```
PROGRAM TEST
  INTEGER C, D, E(20,20)     ! Specification part
  CALL SUB_1                 ! Executable part
...
CONTAINS
  SUBROUTINE SUB_1           ! Internal subprogram
  ...
  END SUBROUTINE SUB_1
END PROGRAM TEST
```

The following shows another example:

```
PROGRAM MyProg
PRINT *, 'hello world'
END
```

PROTECTED

Statement and Attribute: *Specifies limitations on the use of module entities.*

Syntax

The PROTECTED attribute can be specified in a type declaration statement or a PROTECTED statement, and takes one of the following forms:

Type Declaration Statement:

`type, [att-ls, ] PROTECTED [, att-ls] :: entity[, entity] ...`

Statement:

`PROTECTED [::]entity[, entity] ...`

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
entity	Is the name of an entity in a module.

The PROTECTED attribute can only appear in the specification part of a module.

The PROTECTED attribute can only be specified for a procedure pointer or named variable that is not in a common block.

A non-pointer object that has the PROTECTED attribute and is accessed by use association can not appear in a variable definition or as the target in a pointer assignment statement.

A pointer object that has the PROTECTED attribute and is accessed by use association must not appear as any of the following:

- A *pointer-object* in a NULLIFY statement
- A *pointer-object* in a pointer assignment statement
- An *object* in an ALLOCATE or DEALLOCATE statement
- An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

The following restrictions apply outside of the module in which the entity has been given the PROTECTED attribute:

- A non-pointer entity may not be defined or redefined.
- A pointer entity may not have its association status changed through the pointer.

Example

The following example shows a type declaration statement specifying the PROTECTED attribute:

```
INTEGER, PROTECTED :: D, E
```

Consider the following example:

```
MODULE counter_mod
  INTEGER, PROTECTED :: current = 0
  CONTAINS

  INTEGER FUNCTION next()
  current = current + 1    ! current can be modified here
  next = current
  RETURN
  END FUNCTION next
END MODULE counter_mod

PROGRAM test_counter
USE counter_mod
PRINT *, next( )             ! Prints 1
current = 42                 ! Error: variable is protected
END PROGRAM test_counter
```

See Also

Modules and Module Procedures
Type Declarations
Compatible attributes
Pointer Assignments

PSECT

General Compiler Directive: *Modifies characteristics of a common block.*

Syntax

```
!DIR$ PSECT /common-name/ a[,a] ...
```

common-name

Is the name of the common block. The slashes (/) are required.

a

Is one of the following:

- ALIGN= *val* or ALIGN= *keyword*

  Specifies minimum alignment for the common block. ALIGN only has an effect when specified on Windows* and Linux* systems.

  The *val* is a constant ranging from 0 through 6 on Windows* systems and 0 through 4 on Linux* systems. The specified number is interpreted as a power of 2. The value of the expression is the alignment in bytes.

  The *keyword* is one of the following:

Keyword	Equivalent to *val*
BYTE	0
WORD	1
LONG	2
QUAD	3
OCTA	4
PAGE	On IA-32 architecture: range is 0 through 13 for Windows*; 0 through 12 for Linux*

- [NO]WRT

  Determines whether the contents of a common block can be modified during program execution.

If one program unit changes one or more characteristics of a common block, all other units that reference that common block must also change those characteristics in the same way.

The defaults are ALIGN=OCTA and WRT.

See Also

General Compiler Directives

Syntax Rules for Compiler Directives

PUBLIC

Statement and Attribute: *Specifies that entities in a module can be accessed from outside the module by specifying a USE statement.*

Syntax

The PUBLIC attribute can be specified in a type declaration statement or a PUBLIC statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] PUBLIC [, *att-ls*] :: *entity* [, *entity*]...

Statement:

PUBLIC [[::] *entity* [, *entity*] ...]

type — Is a data type specifier.

att-ls — Is an optional list of attribute specifiers.

entity — Is one of the following:

- A variable name
- A procedure name
- A derived type name
- A named constant
- A namelist group name
- An OpenMP* reduction-identifier

In statement form, an entity can also be one of the following:

- A module name
- A generic name
- A defined operator
- A defined assignment
- A defined I/O generic specification

Description

The PUBLIC attribute can only appear in the scoping unit of a module.

Only one PUBLIC statement without an entity list is permitted in the scoping unit of a module; it sets the default accessibility of all entities in the module and any entities accessed from a module by use association.

If no PRIVATE statements are specified in a module, the default is PUBLIC accessibility for all entities in the module and entities accessible from other modules by use association.

A module name can appear at most once in all the PUBLIC or PRIVATE statements in a scoping unit. If a module name is specified in a PUBLIC statement, the module name must have appeared in a USE statement within the same scoping unit.

A PUBLIC attribute can be specified on the TYPE statement of a derived-type definition in a module. This specifies that the derived-type definition is accessible by USE association where the module is used.

A PUBLIC attribute can be specified in the statement declaring a component or a type-bound procedure of a derived type. This specifies that the component or type-bound procedure is accessible by USE association where the module is used, even if the default accessibility of the components and type-bound procedures of the derived type has been set to PRIVATE in the definition of the derived type.

If a derived type is declared PUBLIC in a module, but its components are declared PRIVATE, any scoping unit accessing the module though use association (or host association) can access the derived-type definition, but not its components.

If a module procedure has a dummy argument or a function result of a type that has PRIVATE accessibility, the module procedure must have PRIVATE accessibility. If the module procedure has a generic identifier, it must also be declared PRIVATE.

If a procedure has a generic identifier, the accessibility of the procedure's specific name is independent of the accessibility of its generic identifier. One can be declared PRIVATE and the other PUBLIC.

The accessibility of the components of a type is independent of the accessibility of the type name. The following combinations are possible:

- A private type name with a private component
- A public type name with a public component
- A private type name with a public component
- A public type name with a private component

The accessibility of a type does not affect, and is not affected by, the accessibility of its components and type-bound procedures. If a type definition is private, then the type name, and thus the structure constructor for the type, are accessible only within the module containing the derived-type definition or in the module's descendants.

Example

The following examples show type declaration statements specifying the PUBLIC and PRIVATE attributes:

```
REAL,  PRIVATE  :: A, B, C
INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the PUBLIC and PRIVATE statements:

```
MODULE SOME_DATA
  REAL ALL_B
  PUBLIC ALL_B
  TYPE RESTRICTED_DATA
    REAL LOCAL_C(50)
  END TYPE RESTRICTED_DATA
  PRIVATE RESTRICTED_DATA
END MODULE
```

The following example shows a PUBLIC type with PRIVATE components:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER, can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

The following shows another example:

```
 ! LENGTH in module VECTRLEN calculates the length of a 2-D vector.
 ! The module contains both private and public procedures
    MODULE VECTRLEN
      PRIVATE SQUARE
      PUBLIC LENGTH
     CONTAINS
     SUBROUTINE LENGTH(x,y,z)
        REAL,INTENT(IN) x,y
        REAL,INTENT(OUT) z
        CALL SQUARE(x,y)
        z = SQRT(x + y)
        RETURN
```

```
    END SUBROUTINE
    SUBROUTINE SQUARE(x1,y1)
       REAL x1,y1
       x1 = x1**2
       y1 = y1**2
       RETURN
    END SUBROUTINE
  END MODULE
```

See Also

PRIVATE
MODULE
TYPE
Defining Generic Names for Procedures
USE
Use and Host Association
Type Declarations
Compatible attributes

PURE

Keyword: *Asserts that a user-defined procedure has no side effects.*

Description

This kind of procedure is specified by using the prefix PURE or the prefix ELEMENTAL without the prefix IMPURE in a FUNCTION or SUBROUTINE statement.

A pure procedure has no side effects. It has no effect on the state of the program, except for the following:

- For functions: It returns a value.
- For subroutines: It modifies INTENT(OUT) and INTENT(INOUT) parameters.

The following intrinsic and library procedures are implicitly pure:

- All intrinsic functions
- The elemental intrinsic subroutine MVBITS
- The intrinsic subroutine MOVE_ALLOC
- Intrinsic module procedures that are specified to be pure

A dummy argument or a procedure pointer may be specified to be pure. A type-bound procedure that is bound to a pure procedure is also pure.

A statement function is pure only if all functions that it references are pure.

Except for procedure arguments and pointer arguments, the following intent must be specified in the specification part of the procedure for all dummy arguments:

- For functions: INTENT(IN) or the VALUE attribute
- For subroutines: any INTENT (IN, OUT, or INOUT) or the VALUE attribute

A local variable declared in a pure procedure (including variables declared in any internal procedure) must not:

- Specify the SAVE attribute
- Be initialized in a type declaration statement or a DATA statement

The following variables have restricted use in pure procedures (and any internal procedures):

- Global variables

- Dummy arguments with INTENT(IN) (or no declared intent)
- Objects that are storage associated with any part of a global variable

They must not be used in any context that does either of the following:

- Causes their value to change. For example, they must not be used as:
  - The left side of an assignment statement or pointer assignment statement
  - An actual argument associated with a dummy argument with INTENT(OUT), INTENT(INOUT), or the POINTER attribute
  - An index variable in a DO or FORALL statement, or an implied-DO clause
  - The variable in an ASSIGN statement
  - An input item in a READ statement
  - An internal file unit in a WRITE statement
  - An object in an ALLOCATE, DEALLOCATE, or NULLIFY statement
  - An IOSTAT or SIZE specifier in an I/O statement, or the STAT specifier in a ALLOCATE or DEALLOCATE statement
- Creates a pointer to that variable. For example, they must not be used as:
  - The target in a pointer assignment statement
  - The right side of an assignment to a derived-type variable (including a pointer to a derived type) if the derived type has a pointer component at any level

A pure procedure must not contain the following:

- Any external I/O statement (including a READ or WRITE statement whose I/O unit is an external file unit number or *)
- A PAUSE statement
- A STOP statement
- An image control statement
- An OpenMP* directive

A pure procedure can be used in contexts where other procedures are restricted; for example:

- It can be called directly in a FORALL statement or be used in the mask expression of a FORALL statement.
- It can be called from a pure procedure. Pure procedures can only call other pure procedures, including one referenced by means of a defined operator, defined assignment, or finalization.
- It can be passed as an actual argument to a pure procedure.

If a procedure is used in any of these contexts, its interface must be explicit and it must be declared pure in that interface.

Example

Consider the following:

```
PURE FUNCTION DOUBLE(X)
  REAL, INTENT(IN) :: X
  DOUBLE = 2 * X
END FUNCTION DOUBLE
```

The following shows another example:

```
PURE INTEGER FUNCTION MANDELBROT(X)
  COMPLEX, INTENT(IN) :: X
  COMPLEX__:: XTMP
  INTEGER__:: K
  ! Assume SHARED_DEFS includes the declaration
  ! INTEGER ITOL
  USE SHARED_DEFS

  K = 0
```

```
  XTMP = -X
  DO WHILE (ABS(XTMP) < 2.0 .AND. K < ITOL)
    XTMP = XTMP**2 - X
    K = K + 1
  END DO
  MANDELBROT = K
END FUNCTION
```

The following shows the preceding function used in an interface block:

```
INTERFACE
  PURE INTEGER FUNCTION MANDELBROT(X)
    COMPLEX, INTENT(IN) :: X
  END FUNCTION MANDELBROT
END INTERFACE
```

The following shows a FORALL construct calling the MANDELBROT function to update all the elements of an array:

```
FORALL (I = 1:N, J = 1:M)
  A(I,J) = MANDELBROT(COMPLX((I-1)*1.0/(N-1), (J-1)*1.0/(M-1))
END FORALL
```

See Also

FUNCTION
SUBROUTINE
FORALL
ELEMENTAL prefix

PUTC

Portability Function: *Writes a character to Fortran external unit number 6.*

Module

USE IFPORT

Syntax

`result = PUTC (char)`

char (Input) Character. Character to be written to external unit 6.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code.

Example

```
use IFPORT
integer(4) i4
character*1 char1
do i = 1,26
  char1 = char(123-i)
  i4 = putc(char1)
  if (i4.ne.0) iflag = 1
enddo
```

See Also

GETC
WRITE
PRINT
FPUTC

PUTIMAGE, PUTIMAGE_W (W*S)

Graphics Subroutines: *Transfer the image stored in memory to the screen.*

Module

USE IFQWIN

Syntax

```
CALL PUTIMAGE (x,y,image,action)
```

```
CALL PUTIMAGE_W (wx,wy,image,action)
```

x, y — (Input) INTEGER(2). Viewport coordinates for upper-left corner of the image when placed on the screen.

wx, wy — (Input) REAL(8). Window coordinates for upper-left corner of the image when placed on the screen.

image — (Input) INTEGER(1). Array of single-byte integers. Stored image buffer.

action — (Input) INTEGER(2). Interaction of the stored image with the existing screen image. One of the following symbolic constants defined in IFQWIN.F90:

- $GAND - Forms a new screen display as the logical AND of the stored image and the existing screen display. Points that have the same color in both the existing screen image and the stored image remain the same color, while points that have different colors are joined by a logical AND.
- $GOR - Superimposes the stored image onto the existing screen display. The resulting image is the logical OR of the image.
- $GPRESET - Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by GETIMAGE, producing a negative image.
- $GPSET - Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by GETIMAGE.
- $GXOR - Causes points in the existing screen image to be inverted wherever a point exists in the stored image. This behavior is like that of a cursor. If you perform an exclusive OR of an image with the background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The $GXOR constant is a special mode often used for animation.
- In addition, the following ternary raster operation constants can be used (described in the online documentation for the Windows* API BitBlt):

- $GSRCCOPY (same as $GPSET)
- $GSRCPAINT (same as $GOR)
- $GSRCAND (same as $GAND)
- $GSRCINVERT (same as $GXOR)
- $GSRCERASE
- $GNOTSRCCOPY (same as $GPRESET)
- $GNOTSRCERASE
- $GMERGECOPY
- $GMERGEPAINT
- $GPATCOPY
- $GPATPAINT
- $GPATINVERT
- $GDSTINVERT
- $GBLACKNESS
- $GWHITENESS

PUTIMAGE places the upper-left corner of the image at the viewport coordinates (*x*, *y*). PUTIMAGE_W places the upper-left corner of the image at the window coordinates (*wx*, *wy*).

Example

```
! Build as a Graphics App.
USE IFQWIN
INTEGER(1), ALLOCATABLE :: buffer(:)
INTEGER(2) status, x
INTEGER(4) imsize

status = SETCOLOR(INT2(4))

! draw a circle
status = ELLIPSE($GFILLINTERIOR,INT2(40),INT2(55),    &
                 INT2(70),INT2(85))
imsize = IMAGESIZE (INT2(39),INT2(54),INT2(71), &
                 INT2(86))
ALLOCATE (buffer(imsize))
CALL GETIMAGE(INT2(39),INT2(54),INT2(71),INT2(86),    &
              buffer)

! copy a row of circles beneath it
DO x = 5 , 395, 35
   CALL PUTIMAGE(x, INT2(90), buffer, $GPSET)
END DO
DEALLOCATE(buffer)
END
```

See Also

GETIMAGE
GRSTATUS
IMAGESIZE

PXF(type)GET

POSIX Subroutine: *Gets the value stored in a component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXF(type)GET (jhandle,compname,value,ierror)
CALL PXF(type)GET (jhandle,compname,value,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFINTGET
REAL	REAL(4)	PXFREALGET
LGCL	LOGICAL(4)	PXFLGCLGET
STR	CHARACTER*(*)	PXFSTRGET
CHAR	CHARACTER(1)	PXFCHARGET
DBL	REAL(8)	PXFDBLGET
INT8	INTEGER(8)	PXFINT8GET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to retrieve data from.

value (Output) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). Stores the value of the component (or field).

ilen (Output) INTEGER(4). This argument can only be used when *(type)* is STR (PXFSTRGET). Stores the length of the returned string.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXF(type)GET subroutines retrieve the value from component (or field) *compname* of the structure associated with handle *jhandle* into variable *value*.

Example

See the example in PXFTIMES (which shows PXFINTGET and PXFINT8GET)

See Also

PXF(type)SET

PXF(type)SET

POSIX Subroutine: *Sets the value of a component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXF(type)SET (jhandle,compname,value,ierror)
CALL PXF(type)SET (jhandle,compname,value,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFINTSET
REAL	REAL(4)	PXFREALSET
LGCL	LOGICAL(4)	PXFLGCLSET
STR	CHARACTER*(*)	PXFSTRSET
CHAR	CHARACTER(1)	PXFCHARSET
DBL	REAL(8)	PXFDBLSET
INT8	INTEGER(8)	PXFINT8SET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to write data to.

value (Input) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). The value for the component (or field).

ilen (Input) INTEGER(4). This argument can only be used when *(type)* is STR (PXFSTRSET). The length of the string in *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Example

See the example in PXFSTRUCTCREATE (which shows PXFSTRSET)

See Also

PXF(type)GET

PXFA(type)GET

POSIX Subroutine: *Gets the array values stored in a component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFA(type)GET (jhandle,compname,value,ialen,ierror)
CALL PXFA(type)GET (jhandle,compname,value,ialen,ilen,ierror) ! syntax when (type) is
STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFAINTGET
REAL	REAL(4)	PXFAREALGET
LGCL	LOGICAL(4)	PXFALGCLGET
STR	CHARACTER*(*)	PXFASTRGET
CHAR	CHARACTER(1)	PXFACHARGET
DBL	REAL(8)	PXFADBLGET
INT8	INTEGER(8)	PXFAINT8GET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to retrieve data from.

value (Output) An array, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type of the array is INTEGER(4). Stores the value of the component (or field).

ialen (Input) INTEGER(4). The size of array *value*.

ilen (Output) INTEGER(4). This argument can only be used when *(type)* is STR (PXFASTRGET). An array that stores the lengths of elements of array *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFA(type)GET subroutines are similar to the PXF(type)GET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFA(type)GET subroutines are used, the entire array is accessed (read from the component or field) as a unit.

See Also

PXFA(type)SET
PXF(type)GET

PXFA(type)SET

POSIX Subroutine: *Sets the value of an array component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFA(type)SET (jhandle, compname,value,ialen,ierror)
```

```
CALL PXFA(type)SET (jhandle,compname,value,ialen,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFAINTSET
REAL	REAL(4)	PXFAREALSET
LGCL	LOGICAL(4)	PXFALGCLSET
STR	CHARACTER*(*)	PXFASTRSET
CHAR	CHARACTER(1)	PXFACHARSET
DBL	REAL(8)	PXFADBLSET
INT8	INTEGER(8)	PXFAINT8SET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to write data to.

value (Input) An array, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type of the array is INTEGER(4). The value for the component (or field).

ialen (Input) INTEGER(4). The size of array *value*.

ilen (Input) INTEGER(4). This argument can only be used when *(type)* is STR (PXFASTRSET). An array that specifies the lengths of elements of array *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFA(type)GET subroutines are similar to the PXF(type)GET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFA(type)GET subroutines are used, the entire array is accessed (read from the component or field) as a unit.

See Also

PXFA(type)GET
PXF(type)SET

PXFACCESS

POSIX Subroutine: *Determines the accessibility of a file.*

Module

USE IFPOSIX

Syntax

```
CALL PXFACCESS (path,ilen,iamode,ierror)
```

path (Input) Character. The name of the file.

ilen (Input) INTEGER(4). The length of the *path* string.

iamode (Input) INTEGER(4). One or more of the following:

0	Checks for existence of the file.
1[1]	Checks for execute permission.
2	Checks for write access.
4	Checks for read access.
6	Checks for read/write access.

[1]L*X only

ierror (Output) INTEGER(4). The error status.

If access is permitted, the result value is zero; otherwise, an error code. Possible error codes are:

- -1: A bad parameter was passed.
- ENOENT: The named directory does not exist.
- EACCES: Access requested was denied.

On Windows* systems, if the name given is a directory name, the function only checks for existence. All directories have read/write access on Windows* systems.

PXFALARM

POSIX Subroutine: *Schedules an alarm.*

Module

USE IFPOSIX

Syntax

```
CALL PXFALARM (iseconds,isecleft,ierror)
```

iseconds (Input) INTEGER(4). The number of seconds before the alarm signal should be delivered.

isecleft (Output) INTEGER(4). The number of seconds remaining until any previously scheduled alarm signal is due to be delivered.

It is set to zero if there was no previously scheduled alarm signal.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFALARM subroutine arranges for a SIGALRM signal to be delivered to the process in seconds *iseconds*.

On Linux* and macOS* systems, SIGALRM is a reserved defined constant that is equal to 14. You can use any other routine to install the signal handler. You can get SIGALRM and other signal values by using PXFCONST or IPXFCONST.

On Windows* systems, the SIGALRM feature is not supported, but the POSIX library has an implementation you can use. You can provide a signal handler for SIGALRM by using PXFSIGACTION.

See Also
PXFCONST
IPXFCONST
PXFSIGACTION

PXFCALLSUBHANDLE
POSIX Subroutine: *Calls the associated subroutine.*

Module
USE IFPOSIX

Syntax
`CALL PXFCALLSUBHANDLE (jhandle2,ival,ierror)`

jhandle2	(Input) INTEGER(4). A handle to the subroutine.
ival	(Input) INTEGER(4). The argument to the subroutine.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFCALLSUBHANDLE subroutine, when given a subroutine handle, calls the associated subroutine.

PXFGETSUBHANDLE should be used to obtain a subroutine handle.

> **NOTE**
> The subroutine cannot be a function, an intrinsic, or an entry point, and must be defined with exactly one integer argument.

See Also
PXFGETSUBHANDLE

PXFCFGETISPEED (L*X, M*X)
POSIX Subroutine: *Returns the input baud rate from a termios structure.*

Module
USE IFPOSIX

Syntax
`CALL PXFCFGETISPEED (jtermios,iospeed,ierror)`

jtermios	(Input) INTEGER(4). A handle of structure `termios`.
iospeed	(Output) INTEGER(4). The returned value of the input baud rate from the structure associated with handle *jtermios*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE
To get a handle for an instance of the `termios` structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also
PXFSTRUCTCREATE
PXFCFSETISPEED

PXFCFGETOSPEED (L*X, M*X)
POSIX Subroutine: *Returns the output baud rate from a termios structure.*

Module
USE IFPOSIX

Syntax
`CALL PXFCFGETOSPEED (jtermios,iospeed,ierror)`

jtermios	(Input) INTEGER(4). A handle of structure `termios`.
iospeed	(Output) INTEGER(4). The returned value of the output baud rate from the structure associated with handle *jtermios*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE
To get a handle for an instance of the `termios` structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also
PXFSTRUCTCREATE
PXFCFSETOSPEED

PXFCFSETISPEED (L*X, M*X)
POSIX Subroutine: *Sets the input baud rate in a termios structure.*

Module
USE IFPOSIX

Syntax
`CALL PXFCFSETISPEED (jtermios,ispeed,ierror)`

jtermios	(Input) INTEGER(4). A handle of structure `termios`.
ispeed	(Input) INTEGER(4). The value of the input baud rate for the structure associated with handle *jtermios*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE
To get a handle for an instance of the `termios` structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also
PXFSTRUCTCREATE
PXFCFGETISPEED

PXFCFSETOSPEED (L*X, M*X)
POSIX Subroutine: *Sets the output baud rate in a termios structure.*

Module
USE IFPOSIX

Syntax
`CALL PXFCFSETOSPEED (jtermios,ispeed,ierror)`

jtermios (Input) INTEGER(4). A handle of structure `termios`.

ispeed (Input) INTEGER(4). The value of the output baud rate for the structure associated with handle *jtermios*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE
To get a handle for an instance of the `termios` structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also
PXFSTRUCTCREATE
PXFCFGETOSPEED

PXFCHDIR
POSIX Subroutine: *Changes the current working directory.*

Module
USE IFPOSIX

Syntax
`CALL PXFCHDIR (path,ilen,ierror)`

path	(Input) Character. The directory to be changed to.
ilen	(Input) INTEGER(4). The length of the *path* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also
PXFMKDIR

PXFCHMOD
POSIX Subroutine: *Changes the ownership mode of the file.*

Module
USE IFPOSIX

Syntax
`CALL PXFCHMOD (path,ilen,imode,ierror)`

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of the *path* string.
imode	(Input) INTEGER(4). The ownership mode of the file. On Windows* systems, see your Microsoft* Visual C++ Installation in the `\include` directory under `sys\stat.h` for the values of *imode*. On Linux* and macOS* systems, use octal file-access mode.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

> **NOTE**
> On Linux* and macOS* systems, you must have sufficient ownership permissions, such as being the owner of the file or having read/write access of the file.

PXFCHOWN (L*X, M*X)
POSIX Subroutine: *Changes the owner and group of a file.*

Module
USE IFPOSIX

Syntax
`CALL PXFCHOWN (path,ilen,iowner,igroup,ierror)`

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of the *path* string.

iowner (Input) INTEGER(4). The owner UID.

igroup (Input) INTEGER(4). The group GID.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFCLEARENV

POSIX Subroutine: *Clears the process environment.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCLEARENV (ierror)
```

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

After a call to PXFCLEARENV, no environment variables are defined.

PXFCLOSE

POSIX Subroutine: *Closes the file associated with the descriptor.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCLOSE (fd,ierror)
```

fd (Input) INTEGER(4). A file descriptor.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

PXFOPEN

PXFCLOSEDIR

POSIX Subroutine: *Closes the directory stream.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCLOSEDIR (idirid,ierror)
```

idirid	(Input) INTEGER(4). The directory ID obtained from PXFOPENDIR.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFCLOSEDIR subroutine closes the directory associated with *idirid*.

See Also
PXFOPENDIR

PXFCONST
POSIX Subroutine: *Returns the value associated with a constant.*

Module
USE IFPOSIX

Syntax
CALL PXFCONST (*constname*,*ival*,*ierror*)

constname	(Input) Character. The name of one of the following constants: • STDIN_UNIT • STDOUT_UNIT • STDERR_UNIT • EINVAL • ENONAME • ENOHANDLE • EARRAYLEN • ENOENT • ENOTDIR • EACCES The constants beginning with E signify various error values for the system variable errno.
ival	(Output) INTEGER(4). The returned value of the constant.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, it is set to -1.

For more information on these constants, see your Microsoft* Visual C++ documentation (Windows* systems) or the errno.h file (Linux* and macOS* systems).

See Also
PXFISCONST

PXFCREAT
POSIX Subroutine: *Creates a new file or rewrites an existing file.*

Module
USE IFPOSIX

Syntax

```
CALL PXFCREAT (path,ilen,imode,ifildes,ierror)
```

path	(Input) Character. The pathname of the file.
ilen	(Input) INTEGER(4). The length of *path* string.
imode	(Input) INTEGER(4). The mode of the newly created file. On Windows* systems, see your Microsoft* Visual C++ documentation for permitted mode values. On Linux* and macOS* systems, use octal file-access mode.
ifildes	(Output) INTEGER(4). The returned file descriptor.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFCTERMID (L*X, M*X)

POSIX Subroutine: *Generates a terminal pathname.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCTERMID (s,ilen,ierror)
```

s	(Output) Character. The returned pathname of the terminal.
ilen	(Output) INTEGER(4). The length of the returned value in the *s* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror*is set to zero; otherwise, an error code.

This subroutine returns a string that refers to the current controlling terminal for the current process.

PXFDUP, PXFDUP2

POSIX Subroutine: *Duplicates an existing file descriptor.*

Module

USE IFPOSIX

Syntax

```
CALL PXFDUP (ifildes,ifid,ierror)
CALL PXFDUP2 (ifildes,ifildes2,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor to duplicate.
ifid	(Output) INTEGER(4). The returned new duplicated file descriptor.
ifildes2	(Input) INTEGER(4). The number for the new file descriptor.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFDUP subroutine creates a second file descriptor for an opened file.

The PXFDUP2 subroutine copies the file descriptor associated with *ifildes*. Integer number *ifildes2* becomes associated with this new file descriptor, but the value of *ifildes2* is not changed.

PXFE(type)GET

POSIX Subroutine: *Gets the value stored in an array element component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFE(type)GET (jhandle, compname,index,value,ierror)
```

```
CALL PXFE(type)GET (jhandle,compname,index,value,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFEINTGET
REAL	REAL(4)	PXFEREALGET
LGCL	LOGICAL(4)	PXFELGCLGET
STR	CHARACTER*(*)	PXFESTRGET
CHAR	CHARACTER(1)	PXFECHARGET
DBL	REAL(8)	PXFEDBLGET
INT8	INTEGER(8)	PXFEINT8GET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to retrieve data from.

index (Input) INTEGER(4). The index of the array element to get data for.

value (Output) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). Stores the value of the component (or field).

ilen (Output) INTEGER(4). This argument can only be used when *(type)* is STR (PXFESTRGET). Stores the length of the returned string.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFE(type)GET subroutines are similar to the PXF(type)GET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFE(type)GET subroutines are used, the array element with index *index* is accessed (read from the component or field).

See Also

PXFE(type)SET
PXF(type)GET

PXFE(type)SET

POSIX Subroutine: *Sets the value of an array element component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFE(type)SET (jhandle, compname,index,value,ierror)
```

```
CALL PXFE(type)SET (jhandle,compname,index,value,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFEINTSET
REAL	REAL(4)	PXFEREALSET
LGCL	LOGICAL(4)	PXFELGCLSET
STR	CHARACTER*(*)	PXFESTRSET
CHAR	CHARACTER(1)	PXFECHARSET
DBL	REAL(8)	PXFEDBLSET
INT8	INTEGER(8)	PXFEINT8SET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to write data to.

index (Input) INTEGER(4). The index of the array element to write data to.

value (Input) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). The value for the component (or field).

ilen (Input) INTEGER(4). This argument can only be used when *(type)* is STR (PXFESTRSET). The length of the string *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFE(type)SET subroutines are similar to the PXF(type)SET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFE(type)SET subroutines are used, the array element with index *index* is accessed (written to the component or field).

See Also

PXFE(type)GET
PXF(type)SET

PXFEXECV

POSIX Subroutine: *Executes a new process by passing command-line arguments.*

Module

USE IFPOSIX

Syntax

CALL PXFEXECV (*path*,*lenpath*,*argv*,*lenargv*,*iargc*,*ierror*)

path	(Input) Character. The path to the new executable process.
lenpath	(Input) INTEGER(4). The length of *path* string.
argv	(Input) An array of character strings. Contains the command-line arguments to be passed to the new process.
lenargv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *argv*.
iargc	(Input) INTEGER(4). The number of command-line arguments
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFEXECV subroutine executes a new executable process (file) by passing command-line arguments specified in the argv array. If execution is successful, no return is made to the calling process.

See Also

PXFEXECVE
PXFEXECVP

PXFEXECVE

POSIX Subroutine: *Executes a new process by passing command-line arguments.*

Module

USE IFPOSIX

Syntax

CALL PXFEXECVE (*path*,*lenpath*,*argv*,*lenargv*,*iargc*,*env*,*lenenv*,*ienvc*,*ierror*)

path	(Input) Character. The path to the new executable process.

lenpath	(Input) INTEGER(4). The length of *path* string.
argv	(Input) An array of character strings. Contains the command-line arguments to be passed to the new process.
lenargv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *argv*.
iargc	(Input) INTEGER(4). The number of command-line arguments.
env	(Input) An array of character strings. Contains the environment settings for the new process.
lenenv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *env*.
ienvc	(Input) INTEGER(4). The number of environment settings in *env*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFEXECVE subroutine executes a new executable process (file) by passing command-line arguments specified in the *argv* array and environment settings specified in the *env* array.

See Also

PXFEXECV
PXFEXECVP

PXFEXECVP

POSIX Subroutine: *Executes a new process by passing command-line arguments.*

Module

USE IFPOSIX

Syntax

`CALL PXFEXECVP (file,lenfile,argv,lenargv,iargc,ierror)`

file	(Input) Character. The filename of the new executable process.
lenfile	(Input) INTEGER(4). The length of *file* string.
argv	(Input) An array of character strings. Contains the command-line arguments to be passed to the new process.
lenargv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *argv*.
iargc	(Input) INTEGER(4). The number of command-line arguments.
ierror	(Input) Character. The filename of the new executable process.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFEXECVP subroutine executes a new executable process(file) by passing command-line arguments specified in the*argv* array. It uses the PATH environment variable to find the file to execute.

See Also

PXFEXECV
PXFEXECVE

PXFEXIT, PXFFASTEXIT

POSIX Subroutine: *Exits from a process.*

Module

USE IFPOSIX

Syntax

`CALL PXFEXIT (istatus)`

`CALL PXFFASTEXIT (istatus)`

istatus (Input) INTEGER(4). The exit value.

The PXFEXIT subroutine terminates the calling process. It calls, in last-in-first-out (LIFO) order, the functions registered by C runtime functions `atexit` and `onexit`, and flushes all file buffers before terminating the process. The *istatus* value is typically set to zero to indicate a normal exit and some other value to indicate an error.

The PXFFASTEXIT subroutine terminates the calling process without processing `atexit` or `onexit`, and without flushing stream buffers.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
   print *," the child process will be born"
   call PXFFORK(IPID, IERROR)
   call PXFGETPID(IPID_RET,IERROR)
   if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
   else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if
   end if
10 FORMAT (A,Z)
end program
```

PXFFCNTL (L*X, M*X)

POSIX Subroutine: *Manipulates an open file descriptor.*

Module

USE IFPOSIX

Syntax

CALL PXFFCNTL (*ifildes*,*icmd*,*iargin*,*iargout*,*ierror*)

ifildes	(Input) INTEGER(4). A file descriptor.
icmd	(Input) INTEGER(4). Defines an action for the file descriptor.
iargin	(Input; output) INTEGER(4). Interpretation of this argument depends on the value of *icmd*.
iargout	(Output) INTEGER(4). Interpretation of this argument depends on the value of *icmd*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFFCNTL is a multi-purpose subroutine that causes an action to be performed on a file descriptor. The action, defined in *icmd*, can be obtained by using the values of predefined macros in C header fcntl.h, or by using PXFCONST or IPXFCONST with one of the following constant names:

Constant	Action
F_DUPFD	Returns into *iargout* the lowest available unopened file descriptor greater than or equal to *iargin*. The new file descriptor refers to the same open file as *ifildes* and shares any locks. The system flag FD_CLOEXEC for the new file descriptor is cleared so the new descriptor will not be closed on a call to PXFEXEC subroutine.
F_GETFD	Returns into *iargout* the value of system flag FD_CLOEXEC associated with *ifildes*. In this case, *iargin* is ignored.
F_SETFD	Sets or clears the system flag FD_CLOEXEC for file descriptor *ifildes*. The PXFEXEC family of functions will close all file descriptors with the FD_CLOEXEC flag set. The value for FD_CLOEXEC is obtained from argument *iargin*.
F_GETFL	Returns the file status flags for file descriptor *ifildes*. Unlike F_GETFD, these flags are associated with the file and shared by all descriptors. A combination of the following flags, which are symbolic names for PXFCONST or IPXFCONST, can be returned:

Constant	Action
	• O_APPEND - Specifies the file is opened in append mode. • O_NONBLOCK - Specifies when the file is opened, it does not block waiting for data to become available. • O_RDONLY - Specifies the file is opened for reading only. • O_RDWR - Specifies the file is opened for both reading and writing. • O_WRONLY - Specifies the file is opened for writing only.
F_SETFL	Sets the file status flags from *iargin* for file descriptor *ifildes*. Only O_APPEND or O_NONBLOCK flags can be modified. In this case, *iargout* is ignored.
F_GETLK	Gets information about a lock. Argument *iargin* must be a handle of structure `flock`. This structure is taken as the description of a lock for the file. If there is a lock already in place that would prevent this lock from being locked, it is returned to the structure associated with handle *iargin*. If there are no locks in place that would prevent the lock from being locked, field l_type in the structure is set to the value of the constant with symbolic name F_UNLCK.
F_SETLK	Sets or clears a lock. Argument *iargin* must be a handle of structure `flock`. The lock is set or cleared according to the value of structure field l_type. If the lock is busy, an error is returned.
F_SETLKW	Sets or clears a lock, but causes the process to wait if the lock is busy. Argument *iargin* must be a handle of structure `flock`. The lock is set or cleared according to the value of structure field l_type. If the lock is busy, PXFCNTL waits for an unlock.

NOTE
To get a handle for an instance of the `flock` structure, use PXFSTRUCTCREATE with the string 'flock' for the structure name.

See Also

PXFSTRUCTCREATE
IPXFCONST
PXFCONST

PXFFDOPEN

POSIX Subroutine: *Opens an external unit.*

Module

USE IFPOSIX

Syntax

CALL PXFFDOPEN (*ifildes*,*iunit*,*access*,*ierror*)

ifildes (Input) INTEGER(4). The file descriptor of the opened file.

iunit (Input) INTEGER(4). The Fortran logical unit to connect to file descriptor *ifildes*.

access (Input) Character. A character string that specifies the attributes for the Fortran unit. The string must consist of one or more of the following keyword/value pairs. Keyword/value pairs should be separated by a comma, and blanks are ignored.

Keyword	Possible Values	Description	Default
'NEWLINE'	'YES' or 'NO'	I/O type	'YES'
'BLANK'	'NULL' or 'ZERO'	Interpretation of blanks	'NULL'
'STATUS'	'OLD', 'SCRATCH', or 'UNKNOWN'	File status at open	'UNKNOWN'
'FORM'	'FORMATTED' or 'UNFORMATTED'	Format type	'FORMATTED'

Keywords should be separated from their values by the equals ('=') character; for example:

```
call PXFDOPEN (IFILDES, IUNIT, 'BLANK=NULL, STATUS=UNKNOWN',
IERROR)
```

ierror (Output) INTEGER(4). The error status.

The PXFFDOPEN subroutine connects an external unit identified by *iunit* to a file descriptor *ifildes*. If unit is already connected to a file, the file should be closed before using PXFFDOPEN.

NOTE
On Windows* systems, the default value of the POSIX/IO flag is 0, which causes PXFFDOPEN to return an error.

To prevent this, call subroutine PXFPOSIXIO and set the value of the POSIX/IO flag to 1.

See Also

PXFPOSIXIO

PXFFFLUSH

POSIX Subroutine: *Flushes a file directly to disk.*

Module

USE IFPOSIX

Syntax

CALL PXFFFLUSH (*lunit*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFFLUSH subroutine writes any buffered output to the file connected to unit *lunit*.

PXFFGETC

POSIX Subroutine: *Reads a character from a file.*

Module

USE IFPOSIX

Syntax

CALL PXFFGETC (*lunit*,*char*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
char	(Input) Character. The character to be read.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFGETC subroutine reads a character from a file connected to unit *lunit*.

See Also

PXFFPUTC

PXFFILENO

POSIX Subroutine: *Returns the file descriptor associated with a specified unit.*

Module

USE IFPOSIX

Syntax

CALL PXFFILENO (*lunit*,*fd*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
fd	(Output) INTEGER(4). The returned file descriptor.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. Possible error codes are:

- EINVAL: *lunit* is not an open unit.
- EBADF: *lunit* is not connected with a file descriptor.

The PXFFILENO subroutine returns in *fd* the file descriptor associated with *lunit*.

NOTE
On Windows* systems, the default value of the POSIX/IO flag is 0, which prevents OPEN from connecting a unit to a file descriptor and causes PXFFILENO to return an error.

To prevent this, call subroutine PXFPOSIXIO and set the value of the POSIX/IO flag to 1. This setting allows a connection to a file descriptor.

NOTE
The file-descriptor used by POSIX is not the same as the corresponding file handle used by Windows, so it does not have the same value.

See Also
PXFPOSIXIO

PXFFORK (L*X, M*X)
POSIX Subroutine: *Creates a child process that differs from the parent process only in its PID.*

Module
USE IFPOSIX

Syntax
`CALL PXFFORK (ipid,ierror)`

ipid (Output) INTEGER(4). The returned PID of the new child process.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFORK subroutine creates a child process that differs from the parent process only in its PID. If successful, the PID of the child process is returned in the parent's thread of execution, and a zero is returned in the child's thread of execution. Otherwise, a -1 is returned in the parent's context and no child process is created.

Example
```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
```

```
    print *," My child's pid is", IPID_RET
    call PXFGETPPID(IPID_RET,IERROR)
    print *," The pid of my parent is",IPID_RET
    print *," Now I have exited with code 0xABCD"
    call PXFEXIT(Z'ABCD')
  else
    print *," I am a parent process"
    print *," My parent pid is ", IPID_RET
    print *," I am creating the process with pid", IPID
    print *," Now I am waiting for the end of the child process"
    call PXFWAIT(ISTAT, IPID_RET, IERROR)
    print *," The child with pid ", IPID_RET," has exited"
    if( PXFWIFEXITED(ISTAT) ) then
      print *, " The child exited normally"
      istat_ret = IPXFWEXITSTATUS(ISTAT)
      print 10," The low byte of the child exit code is", istat_ret
    end if
  end if
10 FORMAT (A,Z)
end program
```

See Also

IPXWEXITSTATUS

PXFFPATHCONF

POSIX Subroutine: *Gets the value for a configuration option of an opened file.*

Module

USE IFPOSIX

Syntax

CALL PXFFPATHCONF (*ifildes*,*name*,*ival*,*ierror*)

ifildes	(Input) INTEGER(4). The file descriptor of the opened file.
name	(Input) INTEGER(4). The configurable option.
ival	(Output) INTEGER(4). The value of the configurable option.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFPATHCONF subroutine gets a value for the configuration option named for the opened file with descriptor *ifildes*.

The configuration option, defined in *name*, can be obtained by using PXFCONST or IPXFCONST with one of the following constant names:

Constant	Action
_PC_LINK_MAX	Returns the maximum number of links to the file. If *ifildes* refers to a directory, then the value applies to the whole directory.

Constant	Action
_PC_MAX_CANON[1]	Returns the maximum length of a formatted input line; the file descriptor *ifildes* must refer to a terminal.
_PC_MAX_INPUT[1]	Returns the maximum length of an input line; the file descriptor *ifildes* must refer to a terminal.
_PC_NAME_MAX	Returns the maximum length of a filename in *ifildes* that the process is allowed to create.
_PC_PATH_MAX	Returns the maximum length of a relative pathname when *ifildes* is the current working directory.
_PC_PIPE_BUF	Returns the size of the pipe buffer; the file descriptor *ifildes* must refer to a pipe or FIFO.
_PC_CHOWN_RESTRICTED[1]	Returns nonzero if PXFCHOWN may not be used on this file. If *ifildes* refers to a directory, then this applies to all files in that directory.
_PC_NO_TRUNC[1]	Returns nonzero if accessing filenames longer than _POSIX_NAME_MAX will generate an error.
_PC_VDISABLE[1]	Returns nonzero if special character processing can be disabled; the file descriptor *ifildes* must refer to a terminal.

[1]L*X, M*X

On Linux* and macOS* systems, the corresponding macros are defined in <unistd.h>. The values for *name* can be obtained by using PXFCONST or IPXFCONST when passing the string names of predefined macros in <unistd.h>. The following table shows the corresponding macro names for the above constants:

Constant	Corresponding Macro
_PC_LINK_MAX	_POSIX_LINK_MAX
_PC_MAX_CANON	_POSIX_MAX_CANON
_PC_MAX_INPUT	_POSIX_MAX_INPUT
_PC_NAME_MAX	_POSIX_NAME_MAX
_PC_PATH_MAX	_POSIX_PATH_MAX
_PC_PIPE_BUF	_POSIX_PIPE_BUF
_PC_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED
_PC_NO_TRUNC	_POSIX_NO_TRUNC
_PC_VDISABLE	_POSIX_VDISABLE

See Also

IPXFCONST
PXFCONST
PXFPATHCONF

PXFFPUTC

POSIX Subroutine: *Writes a character to a file.*

Module

USE IFPOSIX

Syntax

CALL PXFFPUTC (*lunit*,*char*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
char	(Input) Character. The character to be written.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. A possible error code is EEND if the end of the file has been reached.

The PXFFPUTC subroutine writes a character to the file connected to unit *lunit*.

See Also

PXFFGETC

PXFFSEEK

POSIX Subroutine: *Modifies a file position.*

Module

USE IFPOSIX

Syntax

CALL PXFFSEEK (*lunit*,*ioffset*,*iwhence*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
ioffset	(Input) INTEGER(4). The number of bytes away from *iwhence* to place the pointer.
iwhence	(Input) INTEGER(4). The position within the file. The value must be one of the following constants (defined in `stdio.h`):

SEEK_SET = 0	Offset from the beginning of the file.
SEEK_CUR = 1	Offset from the current position of the file pointer.
SEEK_END = 2	Offset from the end of the file.

ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. Possible error codes are:

- EINVAL: No file is connected to *lunit*, *iwhence* is not a proper value, or the resulting offset is invalid.
- ESPIPE: *lunit* is a pipe or FIFO.
- EEND: The end of the file has been reached.

The PXFFSEEK subroutine modifies the position of the file connected to unit *lunit*.

PXFFSTAT

POSIX Subroutine: *Gets a file's status information.*

Module

USE IFPOSIX

Syntax

`CALL PXFFSTAT (ifildes,jstat,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor for an opened file.
jstat	(Input) INTEGER(4). A handle of structure `stat`.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFSTAT subroutine puts the status information for the file associated with *ifildes* into the structure associated with handle *jstat*.

> **NOTE**
> To get a handle for an instance of the `stat` structure, use PXFSTRUCTCREATE with the string 'stat' for the structure name.

See Also

PXFSTRUCTCREATE

PXFFTELL

POSIX Subroutine: *Returns the relative position in bytes from the beginning of the file.*

Module

USE IFPOSIX

Syntax

`CALL PXFFTELL (lunit,ioffset,ierror)`

lunit	(Input) INTEGER(4). A Fortran logical unit.
ioffset	(Output) INTEGER(4). The returned relative position in bytes from the beginning of the file.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFGETARG

POSIX Subroutine: *Gets the specified command-line argument.*

Module

USE IFPOSIX

Syntax

CALL PXFGETARG (*argnum*,*str*,*istr*,*ierror*)

argnum	(Input) INTEGER(4). The number of the command-line argument.
str	(Output) Character. The returned string value.
istr	(Output) INTEGER(4). The length of the returned string; it is zero if an error occurs.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFGETARG subroutine places the command-line argument with number *argnum* into character string *str*. If *argnum* is equal to zero, the value of the argument returned is the command name of the executable file.

See Also

IPXFARGC

PXFGETC

POSIX Subroutine: *Reads a character from standard input unit 5.*

Module

USE IFPOSIX

Syntax

CALL PXFGETC (*nextcar*,*ierror*)

nextcar	(Output) Character. The returned character that was read.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

PXFPUTC

PXFGETCWD

POSIX Subroutine: *Returns the path of the current working directory.*

Module

USE IFPOSIX

Syntax

CALL PXFGETCWD (*buf*,*ilen*,*ierror*)

buf	(Output) Character. The returned pathname of the current working directory.
ilen	(Output) INTEGER(4). The length of the returned pathname.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. A possible error code is EINVAL if the size of *buf* is insufficient.

PXFGETEGID (L*X, M*X)

POSIX Subroutine: *Gets the effective group ID of the current process.*

Module

USE IFPOSIX

Syntax

CALL PXFGETEGID (*iegid*,*ierror*)

iegid	(Output) INTEGER(4). The returned effective group ID.
ierror	(Output) INTEGER(4). The error status.

Description

If successful, *ierror* is set to zero; otherwise, an error code.

The effective ID corresponds to the set ID bit on the file being executed.

PXFGETENV

POSIX Subroutine: *Gets the setting of an environment variable.*

Module

USE IFPOSIX

Syntax

CALL PXFGETENV (*name*,*lenname*,*value*,*lenval*,*ierror*)

name	(Input) Character. The name of the environment variable.
lenname	(Input) INTEGER(4). The length of *name*.
value	(Output) Character. The returned value of the environment variable.
lenval	(Output) INTEGER(4). The returned length of *value*. If an error occurs, it returns zero.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

PXFSETENV

PXFGETEUID (L*X, M*X)

POSIX Subroutine: *Gets the effective user ID of the current process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETEUID (ieuid,ierror)`

ieuid	(Output) INTEGER(4). The returned effective user ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The effective ID corresponds to the set ID bit on the file being executed.

PXFGETGID (L*X, M*X)

POSIX Subroutine: *Gets the real group ID of the current process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETGID (igid,ierror)`

igid	(Output) INTEGER(4). The returned real group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The real ID corresponds to the ID of the calling process.

Example

See the example in PXFGETGROUPS

See Also

PXFSETGID

PXFGETGRGID (L*X, M*X)

POSIX Subroutine: *Gets group information for the specified GID.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETGRGID (jgid,jgroup,ierror)`

jgid	(Input) INTEGER(4). The group ID to retrieve information about.
jgroup	(Input) INTEGER(4). A handle of structure group.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFGETGRGID subroutine stores the group information from /etc/group for the entry that matches the group GID *jgid* in the structure associated with handle *jgroup*.

NOTE
To get a handle for an instance of the groupstructure, use PXFSTRUCTCREATE with the string 'group' for the structure name.

Example

See the example in PXFGETGROUPS

See Also

PXFSTRUCTCREATE

PXFGETGRNAM (L*X, M*X)

POSIX Subroutine: *Gets group information for the named group.*

Module

USE IFPOSIX

Syntax

CALL PXFGETGRNAM (*name*,*ilen*,*jgroup*,*ierror*)

name	(Input) Character. The name of the group to retrieve information about.
ilen	(Input) INTEGER(4). The length of the *name* string.
jgroup	(Input) INTEGER(4). A handle of structure group.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFGETGRNAM subroutine stores the group information from /etc/group for the entry that matches the group name name in the structure associated with handle *jgroup*.

NOTE
To get a handle for an instance of the group structure, use PXFSTRUCTCREATE with the string 'group' for the structure name.

See Also

PXFSTRUCTCREATE

PXFGETGROUPS (L*X, M*X)

POSIX Subroutine: *Gets supplementary group IDs.*

Module

USE IFPOSIX

Syntax

CALL PXFGETGROUPS (*igidsetsize*,*igrouplist*,*ngroups*,*ierror*)

igidsetsize	(Input) INTEGER(4). The number of elements in the *igrouplist* array.
igrouplist	(Output) INTEGER(4). The array that has the returned supplementary group IDs.
ngroups	(Output) INTEGER(4). The total number of supplementary group IDs for the process.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFGETGROUPS subroutine returns, up to size *igidsetsize*, the supplementary group IDs in array *igrouplist*. It is unspecified whether the effective group ID of the calling process is included in the returned list. If the size is zero, the list is not modified, but the total number of supplementary group IDs for the process is returned.

Example

```
program test5
  use ifposix
  implicit none
  integer(4) number_of_groups, ierror, isize,i, igid
  integer(4),allocatable,dimension(:):: igrouplist
  integer(JHANDLE_SIZE) jgroup

  ! Get total number of groups in system
  ! call PXFGETGROUPS with 0
  call PXFGETGROUPS(0, igrouplist, number_of_groups, ierror)
  if(ierror.NE.0) STOP 'Error: first call of PXFGETGROUPS fails'
  print *," The number of groups in system ", number_of_groups

  ! Get Group IDs
  isize = number_of_groups
  ALLOCATE( igrouplist(isize))
  call PXFGETGROUPS(isize, igrouplist, number_of_groups, ierror)
  if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     STOP 'Error: first call of PXFGETGROUPS fails'
  end if

  print *," Create an instance for structure 'group' "
  call PXFSTRUCTCREATE("group",jgroup, ierror)
  if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     STOP 'Error: PXFSTRUCTCREATE failed to create an instance of group'
  end if
```

```
  do i=1,  number_of_groups
     call PXFGETGRGID( igrouplist(i), jgroup, ierror)
     if(ierror.NE.0) then
       DEALLOCATE(igrouplist)
       call PXFSTRUCTFREE(jgroup, ierror)
       print *,'Error: PXFGETGRGID failed for i=',i," gid=", igrouplist(i)
       STOP 'Abnormal termination'
     end if
     call PRINT_GROUP_INFO(jgroup)
  end do

  call PXFGETGID(igid,ierror)
  if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     call PXFSTRUCTFREE(jgroup, ierror)
     print *,'Error: PXFGETGID failed'
     STOP 'Abnormal termination'
  end if
  call PXFGETGRGID( igid, jgroup, ierror)
  if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     call PXFSTRUCTFREE(jgroup, ierror)
     print *,"Error: PXFGETGRGID failed for  gid=", igid
     STOP 'Abnormal termination'
  end if

  call PRINT_GROUP_INFO(jgroup)
  DEALLOCATE(igrouplist)
  call PXFSTRUCTFREE(jgroup, ierror)
  print *," Program will normal terminated"
  call PXFEXIT(0)
end
```

PXFGETLOGIN

POSIX Subroutine: *Gets the name of the user.*

Module

USE IFPOSIX

Syntax

CALL PXFGETLOGIN (*s*,*ilen*,*ierror*)

s	(Output) Character. The returned user name.
ilen	(Output) INTEGER(4). The length of the string stored in *s*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFGETPGRP (L*X, M*X)

POSIX Subroutine: *Gets the process group ID of the calling process.*

Module

USE IFPOSIX

Syntax

CALL PXFGETPGRP (*ipgrp*,*ierror*)

ipgrp	(Output) INTEGER(4). The returned process group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Each process group is a member of a session and each process is a member of the session in which its process group is a member.

PXFGETPID

POSIX Subroutine: *Gets the process ID of the calling process.*

Module

USE IFPOSIX

Syntax

CALL PXFGETPID (*ipid*,*ierror*)

ipid	(Output) INTEGER(4). The returned process ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
```

```
    end if
  end if
10 FORMAT (A,Z)
end program
```

PXFGETPPID

POSIX Subroutine: *Gets the process ID of the parent of the calling process.*

Module

USE IFPOSIX

Syntax

CALL PXFGETPPID (*ippid*,*ierror*)

ippid	(Output) INTEGER(4). The returned process ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if
  end if
10 FORMAT (A,Z)
end program
```

PXFGETPWNAM (L*X, M*X)

POSIX Subroutine: *Gets password information for a specified name.*

Module

USE IFPOSIX

Syntax

CALL PXFGETPWNAM (*name*,*ilen*,*jpasswd*,*ierror*)

name	(Input) Character. The login name of the user to retrieve information about. For example, a login name might be "jsmith", while the actual name is "John Smith".
ilen	(Input) INTEGER(4). The length of the *name* string.
jpasswd	(Input) INTEGER(4). A handle of structure *compnam*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFGETPWNAM subroutine stores the user information from /etc/passwd for the entry that matches the user name *name* in the structure associated with handle *jpasswd*.

NOTE
To get a handle for an instance of the compnam structure, use PXFSTRUCTCREATE with the string 'compnam' for the structure name.

See Also

PXFSTRUCTCREATE

PXFGETPWUID (L*X, M*X)

POSIX Subroutine: *Gets password information for a specified UID.*

Module

USE IFPOSIX

Syntax

CALL PXFGETPWUID (*iuid*,*jpasswd*,*ierror*)

iuid	(Input) INTEGER(4). The user ID to retrieve information about.
jpasswd	(Output) INTEGER(4). A handle of structure compnam.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFGETPWUID subroutine stores the user information from /etc/passwd for the entry that matches the user ID *iuid* in the structure associated with handle *jpasswd*.

NOTE
To get a handle for an instance of the compnam structure, use PXFSTRUCTCREATE with the string 'compnam' for the structure name.

See Also
PXFSTRUCTCREATE

PXFGETSUBHANDLE
POSIX Subroutine: *Returns a handle for a subroutine.*

Module
USE IFPOSIX

Syntax
```
CALL PXFGETSUBHANDLE (sub,jhandle1,ierror)
```

sub	(Input) The Fortran subroutine to get a handle for.
jhandle1	(Output) INTEGER(4). The returned handle for the subroutine.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

> **NOTE**
> The argument *sub* cannot be a function, an intrinsic, or an entry point, and must be defined with exactly one integer argument.

PXFGETUID (L*X, M*X)
POSIX Subroutine: *Gets the real user ID of the current process.*

Module
USE IFPOSIX

Syntax
```
CALL PXFGETUID (iuid,ierror)
```

iuid	(Output) INTEGER(4). The returned real user ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The real ID corresponds to the ID of the calling process.

See Also
PXFSETUID

PXFISATTY (L*X, M*X)
POSIX Subroutine: *Tests whether a file descriptor is connected to a terminal.*

Module

USE IFPOSIX

Syntax

`CALL PXFISATTY (ifildes,isatty,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor.
isatty	(Output) LOGICAL(4). The returned value.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

If file descriptor *ifildes* is open and connected to a terminal, *isatty* returns .TRUE.; otherwise, .FALSE..

See Also

PXFOPEN

PXFISBLK

POSIX Function: *Tests for a block special file.*

Module

USE IFPOSIX

Syntax

`result = PXFISBLK (m)`

m	(Input) INTEGER(4). The value of the `st_mode`component (field) in the structure `stat`.

Results

The result type is logical. If the file is a block special file, the result value is .TRUE.; otherwise, .FALSE..

See Also

PXFISCHR

PXFISCHR

POSIX Function: *Tests for a character file.*

Module

USE IFPOSIX

Syntax

`result = PXFISCHR (m)`

m	(Input) INTEGER(4). The value of the `st_mode`component (field) in the structure `stat`.

Results

The result type is logical. If the file is a character file, the result value is .TRUE.; otherwise, .FALSE..

See Also
PXFISBLK

PXFISCONST
POSIX Function: *Tests whether a string is a valid constant name.*

Module
USE IFPOSIX

Syntax
```
result = PXFISCONST (s)
```

s	(Input) Character. The name of the constant to test.

Results
The result type is logical. The PXFISCONST function confirms whether the argument is a valid constant name that can be passed to functions PXFCONST and IPXFCONST. It returns .TRUE. only if IPXFCONST will return a valid value for name *s*.

See Also
IPXFCONST
PXFCONST

PXFISDIR
POSIX Function: *Tests whether a file is a directory.*

Module
USE IFPOSIX

Syntax
```
result = PXFISDIR (m)
```

m	(Input) INTEGER(4).The value of the `st_mode` component (field) in the structure `stat`.

Results
The result type is logical. If the file is a directory, the result value is .TRUE.; otherwise, .FALSE..

PXFISFIFO
POSIX Function: *Tests whether a file is a special FIFO file.*

Module
USE IFPOSIX

Syntax
```
result = PXFISFIFO (m)
```

m (Input) INTEGER(4). The value of the st_modecomponent (field) in the structure stat.

Results

The result type is logical.

The PXFISFIFO function tests whether the file is a special FIFO file created by PXFMKFIFO. If the file is a special FIFO file, the result value is .TRUE.; otherwise, .FALSE..

See Also

PXFMKFIFO
PXFISREG

PXFISREG

POSIX Function: *Tests whether a file is a regular file.*

Module

USE IFPOSIX

Syntax

```
result = PXFISREG (m)
```

m (Input) INTEGER(4). The value of the st_mode component (field) in the structure stat.

Results

The result type is logical. If the file is a regular file, the result value is .TRUE.; otherwise, .FALSE..

See Also

PXFMKFIFO
PXFISFIFO

PXFKILL

POSIX Subroutine: *Sends a signal to a specified process.*

Module

USE IFPOSIX

Syntax

```
CALL PXFKILL (ipid,isig,ierror)
```

ipid (Input) INTEGER(4). The process to kill. It is determined by one of the following values:

> 0	Kills the specific process.
< 0	Kills all processes in the group.
== 0	Kills all processes in the group except special processes.

	== pid_t-1	Kills all processes.

isig	(Input) INTEGER(4). The value of the signal to be sent.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFKILL subroutine sends a signal with value *isig* to a specified process. On Windows* systems, only the *ipid* for the current process can be used.

PXFLINK (L*X, M*X)

POSIX Subroutine: *Creates a link to a file or a directory.*

Module

USE IFPOSIX

Syntax

CALL PXFLINK (*existing*,*lenexist*,*new*,*lennew*,*ierror*)

existing	(Input) Character. The path to the file or directory that you want to link to.
lenexist	(Input) INTEGER(4). The length of the *existing* string.
new	(Input) Character. The name of the new link file.
lennew	(Input) INTEGER(4). The length of the *new* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFLINK subroutine creates a new link (also known as a hard link) to an existing file or directory.

This new name can be used exactly as the old one for any operation. Both names refer to the same entity (so they have the same permissions and ownership) and it is impossible to tell which name was the "original".

PXFLOCALTIME

POSIX Subroutine: *Converts a given elapsed time in seconds to local time.*

Module

USE IFPOSIX

Syntax

CALL PXFLOCALTIME (*isecnds*,*iatime*,*ierror*)

isecnds	(Input) INTEGER(4). The elapsed time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

iatime (Output) INTEGER(4). One-dimensional array with 9 elements containing numeric time data. The elements of *iatime* are returned as follows:

Element	Value
iatime(1)	Seconds (0-59)
iatime(2)	Minutes (0-59)
iatime(3)	Hours (0-23)
iatime(4)	Day of month (1-31)
iatime(5)	Month (1-12)
iatime(6)	Gregorian year (for example, 1990)
iatime(7)	Day of week (0-6, where 0 is Sunday)
iatime(8)	Day of year (1-366)
iatime(9)	Daylight savings flag (1 if daylight savings time is in effect; otherwise, 0)

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFLOCALTIME subroutine converts the time (in seconds since epoch) in the *isecnds* argument to the local date and time as described by the array *iatime* above.

PXFLSEEK

POSIX Subroutine: *Positions a file a specified distance in bytes.*

Module

USE IFPOSIX

Syntax

`CALL PXFLSEEK (ifildes,ioffset,iwhence,iposition,ierror)`

ifildes (Input) INTEGER(4). A file descriptor.

ioffset (Input) INTEGER(4). The number of bytes to move.

iwhence (Input) INTEGER(4). The starting position. The value must be one of the following:

- SEEK_SET = 0

  Sets the offset to *ioffset* bytes.
- SEEK_CUR = 1

  Sets the offset to its current location plus *ioffset* bytes.

- SEEK_END = 2

  Sets the offset to the size of the file plus *ioffset* bytes.

iposition — (Output) INTEGER(4). The ending position; the resulting offset location as measured in bytes from the beginning of the file.

ierror — (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFLSEEK subroutine repositions the offset of file descriptor *ifildes* to the argument *ioffset* according to the value of argument *iwhence*.

PXFLSEEK allows the file offset to be set beyond the end of the existing end-of-file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

PXFMKDIR

POSIX Subroutine: *Creates a new directory.*

Module

USE IFPOSIX

Syntax

`CALL PXFMKDIR (path,ilen,imode,ierror)`

path — (Input) Character. The path for the new directory.

ilen — (Input) INTEGER(4). The length of *path* string.

imode (L*X only) — (Input) INTEGER(4). The mode mask. Octal file-access mode.

ierror — (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

PXFRMDIR
PXFCHDIR

PXFMKFIFO (L*X, M*X)

POSIX Subroutine: *Creates a new FIFO.*

Module

USE IFPOSIX

Syntax

`CALL PXFMKFIFO (path,ilen,imode,ierror)`

path — (Input) Character. The path for the new FIFO.

ilen — (Input) INTEGER(4). The length of *path* string.

imode	(Input) INTEGER(4). The mode mask; specifies the FIFO's permissions. Octal file-access mode.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFMKFIFO subroutine creates a FIFO special file with name *path*. A FIFO special file is similar to a pipe, except that it is created in a different way. Once a FIFO special file is created, any process can open it for reading or writing in the same way as an ordinary file.

However, the FIFO file has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks it until some other process opens the same FIFO for writing, and vice versa.

See Also

PXFISFIFO

PXFOPEN

POSIX Subroutine: *Opens or creates a file.*

Module

USE IFPOSIX

Syntax

CALL PXFOPEN (*path*,*ilen*,*iopenflag*,*imode*,*ifildes*,*ierror*)

path	(Input) Character. The path of the file to be opened or created.
ilen	(Input) INTEGER(4). The length of *path* string.
iopenflag	(Input) INTEGER(4). The flags for the file. (For possible constant names that can be passed to PXFCONST or IPXFCONST, see below.)
imode	(Input) INTEGER(4). The permissions for a new file. This argument should always be specified when *iopenflag*=O_CREAT; otherwise, it is ignored. (For possible permissions, see below.)
ifildes	(Output) INTEGER(4). The returned file descriptor for the opened or created file.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

For *iopenflag*, you should specify one of the following constant values:

- O_RDONLY (read only)
- O_WRONLY (write only)
- O_RDWR (read and write)

In addition, you can also specify one of the following constant values by using a bitwise inclusive OR (IOR):

Value	Action
O_CREAT	Creates a file if the file does not exist.

Value	Action
O_EXCL	When used with O_CREAT, it causes the open to fail if the file already exists. In this case, a symbolic link exists, regardless of where it points to.
O_NOCTTY[1]	If *path* refers to a terminal device, it prevents it from becoming the process's controlling terminal even if the process does not have one.
O_TRUNC	If the file already exists, it is a regular file, and *imode* allows writing (its value is O_RDWR or O_WRONLY), it causes the file to be truncated to length 0.
O_APPEND	Opens the file in append mode. Before each write, the file pointer is positioned at the end of the file, as if with PXFLSEEK.
O_NONBLOCK (or O_NDELAY)[1]	When possible, opens the file in non-blocking mode. Neither the open nor any subsequent operations on the file descriptor that is returned will cause the calling process to wait. This mode need not have any effect on files other than FIFOs.
O_SYNC	Opens the file for synchronous I/O. Any writes on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware.
O_NOFOLLOW[1]	If *path* is a symbolic link, it causes the open to fail.
O_DIRECTORY[1]	If *path* is not a directory, it causes the open to fail.
O_LARGEFILE[1]	On 32-bit systems that support the Large Files System, it allows files whose sizes cannot be represented in 31 bits to be opened.
O_BINARY[2]	Opens the file in binary (untranslated) mode.
O_SHORT_LIVED[2]	Creates the file as temporary. If possible, it does not flush to the disk.
O_TEMPORARY[2]	Creates the file as temporary. The file is deleted when last file handle is closed.
O_RANDOM[2]	Specifies primarily random access from the disk.
O_SEQUENTIAL[2]	Specifies primarily sequential access from the disk.
O_TEXT[2]	Opens the file in text (translated) mode. [3]

[1]L*X only

[2]W*S

[2]W*S

[3]For more information, see "Text and Binary Modes" in the Visual C++* programmer's guide.

Argument *imode* specifies the permissions to use if a new file is created. The permissions only apply to future accesses of the newly created file. The value for *imode* can be any of the following constant values (which can be obtained by using PXFCONST or IPXFCONST):

Value	Description
S_IRWXU	00700 user (file owner) has read, write and execute permission.
S_IRUSR, S_IREAD	00400 user has read permission.
S_IWUSR, S_IWRITE	00200 user has write permission.
S_IXUSR, S_IEXEC	00100 user has execute permission.
S_IRWXG[1]	00070 group has read, write and execute permission.
S_IRGRP[1]	00040 group has read permission.
S_IWGRP[1]	00020 group has write permission.
S_IXGRP[1]	00010 group has execute permission.
S_IRWXO[1]	00007 others have read, write and execute permission.
S_IROTH[1]	00004 others have read permission.
S_IWOTH[1]	00002 others have write permission.
S_IXOTH[1]	00001 others have execute permission.
[1]L*X only	

Example

The following call opens a file for writing only and if the file does not exist, it is created:

```
call PXFOPEN( "OPEN.OUT", &
               8, &
               IOR( IPXFCONST(O_WRONLY), IPXFCONST(O_CREAT) ), &
               IOR( IPXFCONST(S_IREAD), IPXFCONST(S_IWRITE) ) )
```

See Also

PXFCLOSE
IPXFCONST
PXFCONST

PXFOPENDIR

POSIX Subroutine: *Opens a directory and associates a stream with it.*

Module

USE IFPOSIX

Syntax

CALL PXFOPENDIR (*dirname*,*lendirname*,*opendirid*,*ierror*)

dirname	(Input) Character. The directory name.
lendirname	(Input) INTEGER(4). The length of *dirname* string.
opendirid	(Output) INTEGER(4). The returned ID for the directory.
ierror	(Output) INTEGER(4). The error status. If successful, *ierror* is set to zero; otherwise, an error code.

This subroutine opens a directory pointed to by the *dirname* argument and returns the ID of the directory into *opendirid*. After the call, this ID can be used by functions PXFREADDIR, PXFREWINDDIR, PXFCLOSEDIR.

See Also

PXFCLOSEDIR
PXFREADDIR
PXFREADDIR

PXFPATHCONF

POSIX Subroutine: *Gets the value for a configuration option of an opened file.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPATHCONF (path,ilen,name,ival,ierror)
```

path	(Input) Character. The path to the opened file.
ilen	(Input) INTEGER(4). The length of *path*.
name	(Input) INTEGER(4). The configurable option.
ival	(Input) INTEGER(4). The value of the configurable option.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFPATHCONF subroutine gets a value for the configuration option named for the opened file with path *path*.

The configuration option, defined in *name*, can be obtained by using PXFCONST or IPXFCONST with one of the following constant names:

Constant	Action
_PC_LINK_MAX	Returns the maximum number of links to the file. If *path* refers to a directory, then the value applies to the whole directory.
_PC_MAX_CANON[1]	Returns the maximum length of a formatted input line; the *path* must refer to a terminal.
_PC_MAX_INPUT[1]	Returns the maximum length of an input line; the *path* must refer to a terminal.

Constant	Action
_PC_NAME_MAX	Returns the maximum length of a filename in *path* that the process is allowed to create.
_PC_PATH_MAX	Returns the maximum length of a relative pathname when *path* is the current working directory.
_PC_PIPE_BUF	Returns the size of the pipe buffer; the *path* must refer to a FIFO.
_PC_CHOWN_RESTRICTED[1]	Returns nonzero if PXFCHOWN may not be used on this file. If *path* refers to a directory, then this applies to all files in that directory.
_PC_NO_TRUNC[1]	Returns nonzero if accessing filenames longer than _POSIX_NAME_MAX will generate an error.
_PC_VDISABLE[1]	Returns nonzero if special character processing can be disabled; the *path* must refer to a terminal.

[1]L*X, M*X

On Linux* and macOS* systems, the corresponding macros are defined in <unistd.h>. The values for *name* can be obtained by using PXFCONST or IPXFCONST when passing the string names of predefined macros in <unistd.h>. The following table shows the corresponding macro names for the above constants:

Constant	Corresponding Macro
_PC_LINK_MAX	_POSIX_LINK_MAX
_PC_MAX_CANON	_POSIX_MAX_CANON
_PC_MAX_INPUT	_POSIX_MAX_INPUT
_PC_NAME_MAX	_POSIX_NAME_MAX
_PC_PATH_MAX	_POSIX_PATH_MAX
_PC_PIPE_BUF	_POSIX_PIPE_BUF
_PC_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED
_PC_NO_TRUNC	_POSIX_NO_TRUN C
_PC_VDISABLE	_POSIX_VDISABLE

See Also

IPXFCONST
PXFCONST
PXFFPATHCONF

PXFPAUSE

POSIX Subroutine: *Suspends process execution.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPAUSE (ierror)
```

ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFPAUSE subroutine causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.

PXFPIPE (L*X, M*X)

POSIX Subroutine: *Creates a communications pipe between two processes.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPIPE (ireadfd,iwritefd,ierror)
```

ireadfd	(Output) INTEGER(4). The file descriptor for reading.
iwritefd	(Output) INTEGER(4). The file descriptor for writing.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFPIPE subroutine returns a pair of file descriptors, pointing to a pipe inode, and places them into *ireadfd* for reading and into *iwritefd* for writing.

PXFPOSIXIO

POSIX Subroutine: *Sets the current value of the POSIX I/O flag.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPOSIXIO (new,old,ierror)
```

new	(Input) INTEGER(4). The new value for the POSIX I/O flag.
old	(Output) INTEGER(4). The previous value of the POSIX I/O flag.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

This subroutine sets the current value of the Fortran POSIX I/O flag and returns the previous value of the flag. The initial state of the POSIX I/O flag is unspecified.

If a file is opened with a Fortran OPEN statement when the value of the POSIX I/O flag is 1, the unit is accessed as if the records are newline delimited, even if the file does not contain records that are delimited by a new line character.

If a file is opened with a Fortran OPEN statement when the value of the POSIX I/O flag is zero, a connection to a file descriptor is not assumed and the records in the file are not required to be accessed as if they are newline delimited.

PXFPUTC

POSIX Subroutine: *Outputs a character to logical unit 6 (stdout).*

Module

USE IFPOSIX

Syntax

CALL PXFPUTC (*ch*,*ierror*)

ch	(Input) Character. The character to be written.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. A possible error code is EEND if the end of the file has been reached.

See Also

PXFGETC

PXFREAD

POSIX Subroutine: *Reads from a file.*

Module

USE IFPOSIX

Syntax

CALL PXFREAD (*ifildes*,*buf*,*nbyte*,*nread*,*ierror*)

ifildes	(Input) INTEGER(4). The file descriptor of the file to be read from.
buf	(Output) Character. The buffer that stores the data read from the file.
nbyte	(Input) INTEGER(4). The number of bytes to read.
nread	(Output) INTEGER(4). The number of bytes that were read.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFREAD subroutine reads *nbyte* bytes from the file specified by *ifildes* into memory in *buf*. The subroutine returns the total number of bytes read into *nread*. If no error occurs, the value of *nread* will equal the value of *nbyte*.

See Also

PXFWRITE

PXFREADDIR

POSIX Subroutine: *Reads the current directory entry.*

Module

USE IFPOSIX

Syntax

`CALL PXFREADDIR (idirid,jdirent,ierror)`

idirid	(Input) INTEGER(4). The ID of a directory obtained from PXFOPENDIR.
jdirent	(Output) INTEGER(4). A handle of structure `dirent`.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFREADDIR subroutine reads the entry of the directory associated with *idirid* into the structure associated with handle *jdirent*.

> **NOTE**
> To get a handle for an instance of the `dirent` structure, use PXFSTRUCTCREATE with the string 'dirent' for the structure name.

See Also

PXFOPENDIR
PXFREWINDDIR

PXFRENAME

POSIX Subroutine: *Changes the name of a file.*

Module

USE IFPOSIX

Syntax

`CALL PXFRENAME (old,lenold,new,lennew,ierror)`

old	(Input) Character. The name of the file to be renamed.
lenold	(Input) INTEGER(4). The length of *old* string.
new	(Input) Character. The new file name.
lennew	(Input) INTEGER(4). The length of *new* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFREWINDDIR

POSIX Subroutine: *Resets the position of the stream to the beginning of the directory.*

Module

USE IFPOSIX

Syntax

CALL PXFREWINDDIR (*idirid*,*ierror*)

idirid	(Input) INTEGER(4). The ID of a directory obtained from PXFOPENDIR.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFRMDIR

POSIX Subroutine: *Removes a directory.*

Module

USE IFPOSIX

Syntax

CALL PXFRMDIR (*path*,*ilen*,*ierror*)

path	(Input) Character. The directory to be removed. It must be empty.
ilen	(Input) INTEGER(4). The length of *path* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

PXFMKDIR
PXFCHDIR

PXFSETENV

POSIX Subroutine: *Adds a new environment variable or sets the value of an environment variable.*

Module

USE IFPOSIX

Syntax

CALL PXFSETENV (*name*,*lenname*,*new*,*lennew*,*ioverwrite*,*ierror*)

name	(Input) Character. The name of the environment variable.
lenname	(Input) INTEGER(4). The length of *name*.

new	(Input) Character. The value of the environment variable.
lennew	(Input) INTEGER(4). The length of *new*.
ioverwrite	(Input) INTEGER(4). A flag indicating whether to change the value of the environment variable if it exists.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

If *name* does not exist, PXFSETENV adds it with value*new*.

If *name* exists, PXFSETENV sets its value to *new* if*ioverwrite* is a nonzero number. If *ioverwrite* is zero, the value of *name* is not changed.

If *lennew* is equal to zero, PXFSETENV sets the value of the environment variable to a string equal to *new* after removing any leading or trailing blanks.

Example

```
program test2
use ifposix
character*10 name, new
integer lenname, lennew, ioverwrite, ierror
name = "FOR_NEW"
lenname = 7
new = "ON"
lennew = 2
ioverwrite = 1

CALL PXFSETENV (name, lenname, new, lennew, ioverwrite, ierror)
print *, "name= ", name
print *, "lenname= ", lenname
print *, "new= ", lenname
print *, "lennew= ", lenname
print *, "ierror= ", ierror
end
```

See Also

PXFGETENV

PXFSETGID (L*X, M*X)

POSIX Subroutine: *Sets the effective group ID of the current process.*

Module

USE IFPOSIX

Syntax

CALL PXFSETGID (*igid*,*ierror*)

igid	(Input) INTEGER(4). The group ID.
ierror	(Output) INTEGER(4). The error status. If successful, *ierror* is set to zero; otherwise, an error code.

If the caller is the superuser, the real and saved group ID's are also set. This feature allows a program other than root to drop all of its group privileges, do some un-privileged work, and then re-engage the original effective group ID in a secure manner.

Caution
If the user is root then special care must be taken. PXFSETGID checks the effective gid of the caller. If it is the superuser, all process-related group ID's are set to gid. After this has occurred, it is impossible for the program to regain root privileges.

See Also
PXFGETGID

PXFSETPGID (L*X, M*X)
POSIX Subroutine: *Sets the process group ID.*

Module
USE IFPOSIX

Syntax
`CALL PXFSETPGID (ipid,ipgid,ierror)`

ipid	(Input) INTEGER(4). The process group ID to change.
ipgid	(Input) INTEGER(4). The new process group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSETPGID subroutine sets the process group ID of the process specified by *ipid* to *ipgid*.

If *ipid* is zero, the process ID of the current process is used. If *ipgid* is zero, the process ID of the process specified by *ipid* is used.

PXFSETPGID can be used to move a process from one process group to another, but both process groups must be part of the same session. In this case, *ipgid* specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

PXFSETSID (L*X, M*X)
POSIX Subroutine: *Creates a session and sets the process group ID.*

Module
USE IFPOSIX

Syntax
`CALL PXFSETSID (isid,ierror)`

isid	(Output) INTEGER(4). The session ID.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSETSID subroutine creates a new session if the calling process is not a process group leader.

The calling process is the leader of the new session and the process group leader for the new process group. The calling process has no controlling terminal.

The process group ID and session ID of the calling process are set to the PID of the calling process. The calling process will be the only process in this new process group and in this new session.

PXFSETUID (L*X, M*X)

POSIX Subroutine: *Sets the effective user ID of the current process.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSETUID (iuid,ierror)
```

iuid (Output) INTEGER(4). The session ID.

ierror (Output) INTEGER(4). The user status.

If successful, *ierror* is set to zero; otherwise, an error code.

If the effective user ID of the caller is root, the real and saved user ID's are also set. This feature allows a program other than root to drop all of its user privileges, do some un-privileged work, and then re-engage the original effective user ID in a secure manner.

Caution
If the user is root then special care must be taken. PXFSETUID checks the effective uid of the caller. If it is the superuser, all process-related user ID's are set to uid. After this has occurred, it is impossible for the program to regain root privileges.

See Also

PXFGETUID

PXFSIGACTION (L*X, M*X)

POSIX Subroutine: *Changes the action associated with a specific signal. It can also be used to examine the action of a signal.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSIGACTION (isig,jsigact,josigact,ierror)
```

isig	(Input) INTEGER(4). The signal number whose action should be changed.
jsigact	(Input) INTEGER(4). A handle of structure sigaction. Specifies the new action for signal *isig*.
josigact	(Output) INTEGER(4). A handle of structure sigaction.Stores the previous action for signal *isig*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The signal specified in isig can be any valid signal except SIGKILL and SIGSTOP.

If *jsigact* is nonzero, the new action for signal *isig* is installed from the structure associated with handle *jsigact*. If *josigact* is nonzero, the previous action of the specified signal is saved in the structure associated with handle *josigact* where it can be examined.

NOTE
To get a handle for an instance of the sigaction structure, use PXFSTRUCTCREATE with the string 'sigaction' for the structure name.

See Also
PXFSTRUCTCREATE

PXFSIGADDSET (L*X, M*X)
POSIX Subroutine: *Adds a signal to the signal set.*

Module
USE IFPOSIX

Syntax
```
CALL PXFSIGADDSET (jsigset,isigno,ierror)
```

jsigset	(Input) INTEGER(4). A handle of structure sigset. This is the set to add the signal to.
isigno	(Input) INTEGER(4). The signal number to add to the set.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGADDSET subroutine adds signal number *isigno* to the set of signals associated with handle *jsigset*. This set of signals is used by PXFSIGACTION as field sa_mask in structure sigaction. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure sigaction).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure sigaction.

NOTE
To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also
PXFSTRUCTCREATE
PXFSIGDELSET
PXFSIGACTION

PXFSIGDELSET (L*X, M*X)
POSIX Subroutine: *Deletes a signal from the signal set.*

Module
USE IFPOSIX

Syntax
`CALL PXFSIGDELSET (jsigset,isigno,ierror)`

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set to delete the signal from.
isigno	(Input) INTEGER(4). The signal number to delete from the set.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGDELSET subroutine removes signal number *isigno* from the set of signals associated with handle *jsigset*. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

NOTE
To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also
PXFSTRUCTCREATE
PXFSIGADDSET
PXFSIGACTION

PXFSIGEMPTYSET (L*X, M*X)
POSIX Subroutine: *Empties a signal set.*

Module
USE IFPOSIX

Syntax

```
CALL PXFSIGEMPTYSET (jsigset,ierror)
```

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set to empty.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, nonzero.

The PXFSIGEMPTYSET subroutine initializes the signal set associated with handle *jsigset* to empty; all signals are excluded from the set. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

> **NOTE**
> To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

PXFSTRUCTCREATE
PXFSIGFILLSET
PXFSIGACTION

PXFSIGFILLSET (L*X, M*X)

POSIX Subroutine: *Fills a signal set.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSIGFILLSET (jsigset,ierror)
```

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set to fill.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGFILLSET subroutine initializes the signal set associated with handle *jsigset* to full; all signals are included into the set. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

> **NOTE**
> To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

PXFSTRUCTCREATE
PXFSIGEMPTYSET
PXFSIGACTION

PXFSIGISMEMBER (L*X, M*X)

POSIX Subroutine: *Tests whether a signal is a member of a signal set.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGISMEMBER (jsignset,isigno,ismember,ierror)`

jsignset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set the signal will be tested in.
isigno	(Input) INTEGER(4). The signal number to test for membership.
ismember	(Output) Logical. The returned result.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGISMEMBER subroutine tests whether *isigno* is a member of the set associated with handle *jsignset*. If the signal is a member of the set, *ismember* is set to .TRUE.; otherwise, .FALSE.. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

> **NOTE**
> To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

PXFSTRUCTCREATE
PXFSIGACTION

PXFSIGPENDING (L*X, M*X)

POSIX Subroutine: *Examines pending signals.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGPENDING (jsigset,ierror)`

jsigset	(Input) INTEGER(4). A handle of structure `sigaction`. The signals to examine.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGPENDING subroutine is used to examine pending signals (ones that have been raised while blocked). The signal mask of the pending signals is stored in the signal set associated with handle *jsigset*.

PXFSIGPROCMASK (L*X, M*X)

POSIX Subroutine: *Changes the list of currently blocked signals.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGPROCMASK (ihow,jsigset,josigset,ierror)`

ihow	(Input) INTEGER(4). Defines the action for *jsigset*.
jsigset	(Input) INTEGER(4). A handle of structure `sigset`. The signals to examine.
josigset	(Input) INTEGER(4). A handle of structure `sigset`. Stores the previous mask of blocked signals.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The argument *ihow* indicates the way in which the set is to be changed, and consists of one of the following constant names:

Constant [1]	Action
SIG_BLOCK	The resulting set of blocked signals will be the union of the current signal set and the *jsigset* signal set.
SIG_UNBLOCK	The resulting set of blocked signals will be the current set of blocked signals with the signals in *jsigset* removed. It is legal to attempt to unblock a signal that is not blocked.
SIG_SETMASK	The resulting set of blocked signals will be the *jsigset* signal set.

[1]These names can be used in PXFCONST or IPXFCONST.

If *josigset* is non-zero, the previous value of the signal mask is stored in the structure associated with handle *josigset*.

See Also

IPXFCONST
PXFCONST

PXFSIGSUSPEND (L*X, M*X)

POSIX Subroutine: *Suspends the process until a signal is received.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGSUSPEND (jsigset,ierror)`

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. Specifies a set of signals.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFSIGSUSPEND temporarily replaces the signal mask for the process with that given by the structure associated with the *jsigset* handle; it then suspends the process until a signal is received.

PXFSLEEP

POSIX Subroutine: *Forces the process to sleep.*

Module

USE IFPOSIX

Syntax

`CALL PXFSLEEP (iseconds,isecleft,ierror)`

iseconds	(Input) INTEGER(4). The number of seconds to sleep.
isecleft	(Output) INTEGER(4). The number of seconds left to sleep.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSLEEP subroutine forces the current process to sleep until seconds *iseconds* have elapsed or a signal arrives that cannot be ignored.

PXFSTAT

POSIX Subroutine: *Gets a file's status information.*

Module

USE IFPOSIX

Syntax

`CALL PXFSTAT (path,ilen,jstat,ierror)`

path	(Input) Character. The path to the file.

ilen	(Input) INTEGER(4). The length of *path* string.
jstat	(Input) INTEGER(4). A handle of structure `stat`.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSTAT subroutine puts the status information for the file specified by path into the structure associated with handle *jstat*.

> **NOTE**
> To get a handle for an instance of the `stat` structure, use PXFSTRUCTCREATE with the string 'stat' for the structure name.

See Also
PXFSTRUCTCREATE

PXFSTRUCTCOPY
POSIX Subroutine: *Copies the contents of one structure to another.*

Module
USE IFPOSIX

Syntax
`CALL PXFSTRUCTCOPY (structname,jhandle1,jhandle2,ierror)`

structname	(Input) Character. The name of the structure.
jhandle1	(Input) INTEGER(4). A handle to the structure to be copied.
jhandle2	(Input) INTEGER(4). A handle to the structure that will receive the copy.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Example
See the example in PXFSTRUCTCREATE

PXFSTRUCTCREATE
POSIX Subroutine: *Creates an instance of the specified structure.*

Module
USE IFPOSIX

Syntax
`CALL PXFSTRUCTCREATE (structname,jhandle,ierror)`

structname	(Input) Character. The name of the structure. As for any character string, the name must be specified in single or double quotes; for example, the structure `sigaction` would be specified as 'sigaction'. For more information on available structures, see the table below.
jhandle	(Output) INTEGER(4). The handle of the newly-created structure.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

If your application passes information to the system, you should call one of the PXF(type)SET subroutines. If your application needs to get information from the structure, you should call one of the PXF(type)GET subroutines.

The following table shows:

- The structures that are available in the Fortran POSIX library
- The fields within each structure
- The subroutines you must use to access the structure fields

Structure Name	Field Names	Subroutines for Access
sigset [1]	Fields are hidden.	PXFSIGEMPTYSET [1], PXFSIGFILLSET [1], PXFSIGADDSET [1], or PXFSIGDELSET [1]
sigaction	sa_handler	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	sa_mask	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	sa_flags	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
utsname	sysname	For all fields:
	nodename	PXFSTRGET
	release	
	version	
	machine	
tms	tms_utime	For all fields:
	tms_stime	PXFINTGET or PXFINT8GET
	tms_cutime	
	tms_cstime	
dirent	d_name	PXFSTRGET
stat	st_mode	For all fields:
	st_ino	PXFINTGET or PXFINT8GET
	st_dev	

Structure Name	Field Names	Subroutines for Access
	st_nlink	
	st_uid	
	st_gid	
	st_size	
	st_atime	
	st_mtime	
	st_ctime	
utimbuf	actime	For all fields:
	modtime	PXFINTGET or PXFINT8GET
flock [1]	l_type	For all fields:
	l_whence	PXFINTGET or PXFINT8GET
	l_start	
	l_len	
	l_pid	
termios [1]	c_iflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_oflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_cflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_lflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_cc	PXFAINTGET/PXFAINTSET or PXFAINT8GET/PXFAINT8SET
group [1]	gr_name	PXFSTRGET
	gr_gid	PXFINTGET or PXFINT8GET
	gr_nmem	PXFINTGET or PXFINT8GET
	gr_mem	PXFESTRGET
passwd [1]	pw_name	PXFSTRGET
	pw_uid	PXFINTGET or PXFINT8GET
	pw_gid	PXFINTGET or PXFINT8GET
	pw_dir	PXFSTRGET
	pw_shell	PXFSTRGET

[1] L*X only

As for any character string, you must use single or double quotes when specifying a field name in a PXF(type)GET or PXF(type)SET subroutine. For example, field name sysname (in structure `utsname`) must be specified as 'sysname'.

Example

```
program test4
  use ifposix
  implicit none
  integer(jhandle_size) jhandle1,jhandle2
  integer(4) ierror,ilen1

  print *," Create a first instance for structure 'utsname' "
  call PXFSTRUCTCREATE("utsname",jhandle1,ierror)
  if(ierror.NE.0) STOP 'Error: cannot create structure for jhandle1'

  print *," Create a second instance for structure 'utsname' "
  call PXFSTRUCTCREATE("utsname",jhandle2,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jhandle1,ierror)
     STOP 'test failed - cannot create structure for jhandle2'
  end if

  print *,"Fill the structure associated with jhandle1 with arbitrary data"
  call PXFSTRSET(jhandle1,"sysname","00000000000000",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component sysname for jhandle1')

  call PXFSTRSET(jhandle1,"Nodename","11111111111111",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component nodename for jhandle1')

  call PXFSTRSET(jhandle1,"RELEASE","22222222222222",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component release for jhandle1')

  call PXFSTRSET(jhandle1,"verSION","33333333333333",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component version for jhandle1')

  call PXFSTRSET(jhandle1,"machine","44444444444444",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component machine for jhandle1')

  print *,"Fill the structure associated with jhandle2 with arbitary data"
  call PXFSTRSET(jhandle2,"sysname","aaaaaaaaa",7,ierror)
  if(ierror.NE.0) call Error('Error: can't set component sysname for jhandle2')

  call PXFSTRSET(jhandle2,"Nodename","BBBBBBBBB  BBB",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component nodename for jhandle2')

  call PXFSTRSET(jhandle2,"RELEASE","cCCC cc-cccnc",12,ierror)
  if(ierror.NE.0) call Error('Error: can't set component release for jhandle2')

  call PXFSTRSET(jhandle2,"verSION","ddddd",1,ierror)
  if(ierror.NE.0) call Error('Error: can't set component version for jhandle2')

  call PXFSTRSET(jhandle2,"machine","eeeeeee",6,ierror)
  if(ierror.NE.0) call Error('Error: can't set component machine for jhandle2')

  print *,"Print contents of the structure associated with jhandle1"
  call PRINT_UTSNAME(jhandle1)

  print *,"Print contents of the structure associated with jhandle2"
  call PRINT_UTSNAME(jhandle2)

  print *,"Get operating system info into structure associated with jhandle1"
```

```
  call PXFUNAME(jhandle1,ierror)
  if(ierror.NE.0) call Error('Error: call to PXFUNAME has failed')

  print *,"Print contents of the structure associated with jhandle1"
  print*,"  returned from PXFUNAME"
  call PRINT_UTSNAME(jhandle1)

  print *,"Copy the contents of the structure associated with jhandle1"
  print *,"  into the structure associated with jhandle2"
  call PXFSTRUCTCOPY("utsname",jhandle1,jhandle2,ierror)
  if(ierror.NE.0) call Error('Error: can't copy jhandle1 contents into jhandle2')

  print *,"Print the contents of the structure associated with jhandle2."
  print *,"  It should be the same after copying."
  call PRINT_UTSNAME(jhandle2)

  print *,"Free memory for instance of structure associated with jhandle1"
  call PXFSTRUCTFREE(jhandle1,ierror)
  if(ierror.NE.0) STOP 'Error: can't free instance of structure for jhandle1'

  print *,"Free memory for instance of structure associated with jhandle2"
  call PXFSTRUCTFREE(jhandle2,ierror)
  if(ierror.NE.0) STOP 'Error: can't free instance of structure for jhandle2'

  print *,"Program terminated normally"
  call PXFEXIT(0)
end
```

See Also

PXFSTRUCTFREE
the example in PXFTIMES

PXFSTRUCTFREE

POSIX Subroutine: *Deletes the instance of a structure.*

Module

USE IFPOSIX

Syntax

CALL PXFSTRUCTFREE (*jhandle*,*ierror*)

jhandle	(Input) INTEGER(4). The handle of a structure.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSTRUCTFREE subroutine deletes the instance of the structure associated with handle *jhandle*.

Example

See the example in PXFSTRUCTCREATE, the example in PXFTIMES

PXFSYSCONF

POSIX Subroutine: *Gets values for system limits or options.*

Module

USE IFPOSIX

Syntax

CALL PXFSYSCONF (*name*,*ival*,*ierror*)

name	(Input) INTEGER(4). The system option you want information about.
ival	(Output) INTEGER(4). The returned value.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFSYSCONF lets you determine values for system limits or system options at runtime.

The value for *name* can be any of the following constants:

Constant	Description
_SC_ARG_MAX[1]	Indicates the maximum length of the arguments to the PXFEXEC family of routines.
_SC_CHILD_MAX[1]	Indicates the number of simultaneous processes per user ID.
_SC_CLK_TCK	Indicates the number of clock ticks per second.
_SC_STREAM_MAX[2]	Indicates the maximum number of streams that a process can have open at any time.
_SC_TZNAME_MAX	Indicates the maximum number of bytes in a timezone name.
_SC_OPEN_MAX	Indicates the maximum number of files that a process can have open at any time.
_SC_JOB_CONTROL[1]	Indicates whether POSIX-style job control is supported.
_SC_SAVED_IDS[1]	Indicates whether a process has a saved set-user-ID and a saved set-group-ID.
_SC_VERSION[1]	Indicates the year and month the POSIX.1 standard was approved in the format YYYYMML; the value 199009L indicates the most recent revision, 1990.
_SC_BC_BASE_MAX[1]	Indicates the maximum obase value accepted by the bc(1) utility.
_SC_BC_DIM_MAX[1]	Indicates the maximum value of elements that bc(1) permits in an array.

Constant	Description
_SC_BC_SCALE_MAX[1]	Indicates the maximum scale value allowed by bc(1).
_SC_BC_STRING_MAX[1]	Indicates the maximum length of a string accepted by bc(1).
_SC_COLL_WEIGHTS_MAX[1]	Indicates the maximum numbers of weights that can be assigned to an entry of the LC_COLLATE order keyword in the locale definition file.
_SC_EXPR_NEST_MAX[1,3]	Indicates the maximum number of expressions that can be nested within parentheses by expr(1).
_SC_LINE_MAX[1]	Indicates the maximum length of a utility's input line length, either from standard input or from a file. This includes the length for a trailing newline.
_SC_RE_DUP_MAX[1]	Indicates the maximum number of repeated occurrences of a regular expression when the interval notation \{m,n\} is used.
_SC_2_VERSION[1]	Indicates the version of the POSIX.2 standard; it is in the format YYYYMML.
_SC_2_DEV[1]	Indicates whether the POSIX.2 C language development facilities are supported.
_SC_2_FORT_DEV[1]	Indicates whether the POSIX.2 FORTRAN language development utilities are supported.
_SC_2_FORT_RUN[1]	Indicates whether the POSIX.2 FORTRAN runtime utilities are supported.
_SC_2_LOCALEDEF[1]	Indicates whether the POSIX.2 creation of locates via localedef(1) is supported.
_SC_2_SW_DEV[1]	Indicates whether the POSIX.2 software development utilities option is supported.
_SC_PAGESIZE (or _SC_PAGE_SIZE)	Indicates the size of a page (in bytes).
_SC_PHYS_PAGES[4]	Indicates the number of pages of physical memory. Note that it is possible for the product of this value and the value of _SC_PAGE_SIZE to overflow.
_SC_AVPHYS_PAGES[4]	Indicates the number of currently available pages of physical memory.

[1]L*X only

[2]The corresponding POSIX macro is STREAM_MAX.

[3]The corresponding POSIX macro is EXPR_NEST_MAX.

[4]L*X, W*32

The corresponding macros are defined in <bits/confname.h> on Linux* systems; <unistd.h> on macOS* systems. The values for argument *name* can be obtained by using PXFCONST or IPXFCONST when passing the string names of predefined macros in the appropriate .h file.

See Also

IPXFCONST
PXFCONST

PXFTCDRAIN (L*X, M*X)

POSIX Subroutine: *Waits until all output written has been transmitted.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCDRAIN (ifildes,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFTCFLOW (L*X, M*X)

POSIX Subroutine: *Suspends the transmission or reception of data.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCFLOW (ifildes,iaction,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
iaction	(Input) INTEGER(4). The action to perform.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTCFLOW subroutine suspends or resumes transmission or reception of data from the terminal referred to by *ifildes*. The action performed depends on the value of *iaction*, which must be one of the following constant names:

Constant [1]	Action
TCOOFF	Output is suspended.
TCOON	Output is resumed.
TCIOFF	A STOP character is transmitted. This should cause the terminal to stop transmitting data to the system.

Constant [1]	Action
TCION	A START character is transmitted. This should cause the terminal to resume transmitting data to the system.

[1]These names can be used in PXFCONST or IPXFCONST.

See Also
IPXFCONST
PXFCONST

PXFTCFLUSH (L*X, M*X)
POSIX Subroutine: *Discards terminal input data, output data, or both.*

Module
USE IFPOSIX

Syntax
```
CALL PXFTCFLUSH (ifildes,iaction,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
iaction	(Input) INTEGER(4). The action to perform.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The action performed depends on the value of *iaction*, which must be one of the following constant names:

Constant [1]	Action
TCIFLUSH	Discards all data that has been received but not read.
TCOFLUSH	Discards all data that has been written but not transmitted.
TCIOFLUSH	Discards both data received but not read and data written but not transmitted. (Performs TCIFLUSH and TCOFLUSH actions.)

[1]These names can be used in PXFCONST or IPXFCONST.

See Also
IPXFCONST
PXFCONST

PXFTCGETATTR (L*X, M*X)
POSIX Subroutine: *Reads current terminal settings.*

Module
USE IFPOSIX

Syntax

```
CALL PXFTCGETATTR (ifildes,jtermios,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
jtermios	(Output) INTEGER(4). A handle for structure `termios`. Stores the terminal settings.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE
To get a handle for an instance of the `termios`structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also
PXFSTRUCTCREATE
PXFTCSETATTR

PXFTCGETPGRP (L*X, M*X)
POSIX Subroutine: *Gets the foreground process group ID associated with the terminal.*

Module
USE IFPOSIX

Syntax
```
CALL PXFTCGETPGRP (ifildes,ipgid,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ipgid	(Output) INTEGER(4). The returned process group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also
PXFTCSETPGRP

PXFTCSENDBREAK (L*X, M*X)
POSIX Subroutine: *Sends a break to the terminal.*

Module
USE IFPOSIX

Syntax
```
CALL PXFTCSENDBREAK (ifildes,iduration,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.

iduration	(Input) INTEGER(4). Indicates how long the break should be.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTCSENDBREAK subroutine sends a break (a '\0' with a framing error) to the terminal associated with *ifildes*.

PXFTCSETATTR (L*X, M*X)

POSIX Subroutine: *Writes new terminal settings.*

Module

USE IFPOSIX

Syntax

CALL PXFTCSETATTR (*ifildes*,*ioptacts*,*jtermios*,*ierror*)

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ioptacts	(Input) INTEGER(4). Specifies when the terminal changes take effect.
jtermios	(Input) INTEGER(4). A handle for structure termios. Contains the new terminal settings.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTCSETATTR subroutine copies all terminal parameters from structure termiosinto the terminal associated with *ifildes*. When the terminal settings will change depends on the value of *ioptacts*, which must be one of the following constant names:

Constant [1]	Action
TCSANOW	The changes occur immediately.
TCSADRAIN	The changes occur after all output written to *ifildes* has been transmitted.
TCSAFLUSH	The changes occur after all output written to *ifildes* has been transmitted, and all input that had been received but not read has been discarded.

[1]These names can be used in PXFCONST or IPXFCONST.

NOTE
To get a handle for an instance of the termiosstructure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

PXFSTRUCTCREATE
PXFTCGETATTR

PXFTCSETPGRP (L*X, M*X)

POSIX Subroutine: *Sets the foreground process group ID associated with the terminal.*

Module

USE IFPOSIX

Syntax

`CALL PXFTCSETPGRP (ifildes,ipgid,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ipgid	(Input) INTEGER(4). The foreground process group ID for *ifildes*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

PXFTCGETPGRP

PXFTIME

POSIX Subroutine: *Returns the current system time.*

Module

USE IFPOSIX

Syntax

`CALL PXFTIME (itime,ierror)`

itime	(Output) INTEGER(4). The returned system time.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTIME subroutine returns the number of seconds since Epoch (00:00:00 UTC, January 1, 1970).

Example

See the example in PXFTIMES.

PXFTIMES

POSIX Subroutine: *Returns process times.*

Module

USE IFPOSIX

Syntax

`CALL PXFTIMES (jtms,itime,ierror)`

jtms	(Output) INTEGER(4). A handle of structure `tms`.

itime	(Output) INTEGER(4). The returned time since system startup.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTIMES subroutine fills the fields of structure `tms` associated with handle *jtms* with components of time that was spent by the current process. The structure fields are:

- tms_utime - User CPU time
- tms_stime - System CPU time
- tms_cutime - User time of child process
- tms_cstime - System time of child process

All members are measured in system clocks. The values can be converted to seconds by dividing by value *ival* returned from the following call:

```
PXFSYSCONF(IPXFCONST('_SC_CLK_TCK'), ival, ierror)
```

User time is the time charged for the execution of user instructions of the calling process. System time is the time charged for execution by the system on behalf of the calling process.

> **NOTE**
> To get a handle for an instance of the `tms` structure, use PXFSTRUCTCREATE with the string 'tms' for the structure name.

Example

```
program test_uname
  use ifposix
  implicit none
  integer(jhandle_size) jtms1, jtms2
  integer(4) ierror,i
  integer(4),parameter :: n=10000000
  integer(SIZEOF_CLOCK_T) itime,time1,time2, user_time1,user_time2
  integer(SIZEOF_CLOCK_T) system_time1,system_time2
  integer(4) clocks_per_sec, iname
  real(8) s, PI
  real(8) seconds_user, seconds_system

  print *,"Create a first instance for structure 'tms'"
  call PXFSTRUCTCREATE("tms",jtms1,ierror)
  if(ierror.NE.0) STOP 'Error: cannot create structure for handle jtms1'
  print *,"Create a second instance for structure 'tms'"
  call PXFSTRUCTCREATE("tms",jtms2,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     STOP 'Error: cannot create structure for handle jtms2'
  end if

  print *, 'Do some calculations'
  call PXFTIMES(jtms1, itime,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the first call of PXFTIMES fails'
  end if
```

```
  call PXFTIME(time1, ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the first call of PXFTIME fails'
  end if

  s = 0._8
  PI = atan(1._8)*4
  do i=0, n
    s = s + cos(i*PI/n)*sin(i*PI/n)
  end do
  print *," s=",s

  call PXFTIMES(jtms2, itime,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the second call of PXFTIMES fails'
  end if
  call PXFTIME(time2, ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the second call of PXFTIME fails'
  end if
!DIR$ IF DEFINED(_M_INNN)
     call PXFINT8GET(jtms1,"tms_utime",user_time1,ierror)
     call PXFINT8GET(jtms1,"tms_stime",system_time1,ierror)
     call PXFINT8GET(jtms2,"tms_utime",user_time2,ierror)
     call PXFINT8GET(jtms2,"tms_stime",system_time2,ierror)
!DIR$ ELSE
     call PXFINTGET(jtms1,"tms_utime",user_time1,ierror)
     call PXFINTGET(jtms1,"tms_stime",system_time1,ierror)
     call PXFINTGET(jtms2,"tms_utime",user_time2,ierror)
     call PXFINTGET(jtms2,"tms_stime",system_time2,ierror)
!DIR$ ENDIF

  iname = IPXFCONST("_SC_CLK_TCK")
  call PXFSYSCONF(iname,clocks_per_sec, ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the call of PXFSYSCONF fails'
  end if

  seconds_user = (user_time2 - user_time1)/DBLE(clocks_per_sec)
  seconds_system = (system_time2 - system_time1)/DBLE(clocks_per_sec)
  print *," The processor time of calculations:"
  print *," User code execution(in seconds):", seconds_user
  print *," Kernal code execution(in seconds):", seconds_system
  print *," Total processor time(in seconds):", seconds_user + seconds_system
  print *," Elapsed wall clock time(in seconds):", time2 - time1

  print *,"Free memory for instance of structure associated with jtms"
  call PXFSTRUCTFREE(jtms1,ierror)
  call PXFSTRUCTFREE(jtms2,ierror)
end program
```

See Also
PXFSTRUCTCREATE

PXFTTYNAME (L*X, M*X)
POSIX Subroutine: *Gets the terminal pathname.*

Module
USE IFPOSIX

Syntax
`CALL PXFTTYNAME (ifildes,s,ilen,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
s	(Output) Character. The returned terminal pathname.
ilen	(Output) INTEGER(4). The length of the string stored in *s*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFUCOMPARE
POSIX Subroutine: *Compares two unsigned integers.*

Module
USE IFPOSIX

Syntax
`CALL PXFUCOMPARE (i1,i2,icmpr,idiff)`

i1, *i2* — (Input) INTEGER(4). The two unsigned integers to compare.

icmpr — (Output) INTEGER(4). The result of the comparison; one of the following values:

-1	If *i1* < *i2*
0	If *i1* = *i2*
1	If *i1* > *i2*

idiff — (Output) INTEGER(4). The absolute value of the difference.

The PXFUCOMPARE subroutine compares two unsigned integers and returns the absolute value of their difference into *idiff*.

PXFUMASK
POSIX Subroutine: *Sets a new file creation mask and gets the previous one.*

Module

USE IFPOSIX

Syntax

CALL PXFUMASK (*icmask*,*iprevcmask*,*ierror*)

icmask	(Input) INTEGER(4). The new file creation mask.
iprevcmask	(Output) INTEGER(4). The previous file creation mask.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFUNAME

POSIX Subroutine: *Gets the operation system name.*

Module

USE IFPOSIX

Syntax

CALL PXFUNAME (*jutsname*,*ierror*)

jutsname	(Input) INTEGER(4). A handle of structure utsname.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFUNAME subroutine provides information about the operation system. The information is stored in the structure associated with handle *jutsname*.

Example

See the example in PXFSTRUCTCREATE

PXFUNLINK

POSIX Subroutine: *Removes a directory entry.*

Module

USE IFPOSIX

Syntax

CALL PXFUNLINK (*path*,*ilen*,*ierror*)

path	(Input) Character. The name of the directory entry to remove.
ilen	(Input) INTEGER(4). The length of *path* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFUTIME

POSIX Subroutine: *Sets file access and modification times.*

Module

USE IFPOSIX

Syntax

CALL PXFUTIME (*path*,*ilen*,*jutimbuf*,*ierror*)

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of *path* string.
jutimbuf	(Input) INTEGER(4). A handle of structure utimbuf.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFUTIME subroutine sets access and modification times for the file pointed to by path. The time values are retrieved from structure *utimbuf*.

PXFWAIT (L*X, M*X)

POSIX Subroutine: *Waits for a child process.*

Module

USE IFPOSIX

Syntax

CALL PXFWAIT (*istat*,*iretpid*,*ierror*)

istat	(Output) INTEGER(4). The returned status of the child process.
iretpid	(Output) INTEGER(4). The process ID of the stopped child process.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFWAIT subroutine suspends execution of the current process until a child has exited, or until a signal is delivered whose action terminates the current process or calls a signal handling routine. If the child has already exited by the time of the call (a "zombie" process), a return is immediately made. Any system resources used by the child are freed.

The subroutine returns in *iretpid* the value of the process ID of the child that exited, or zero if no child was available. The returned value in *istat* can be used in subroutines IPXFWEXITSTATUS, IPXFWSTOPSIG, IPXFWTERMSIG, PXFWIFEXITED, PXFWIFSIGNALLED, and PXFWIFSTOPPED.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
```

```
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if
  end if
10 FORMAT (A,Z)
end program
```

See Also

PXFWAITPID
IPXFWEXITSTATUS
IPXFWSTOPSIG
IPXFWTERMSIG
PXFWIFEXITED
PXFWIFSIGNALLED
PXFWIFSTOPPED

PXFWAITPID (L*X, M*X)

POSIX Subroutine: *Waits for a specific PID.*

Module

USE IFPOSIX

Syntax

CALL PXFWAITPID (*ipid*,*istat*,*ioptions*,*iretpid*,*ierror*)

ipid

(Input) INTEGER(4). The PID to wait for. One of the following values:

Value	Action
< -1	Specifies to wait for any child process whose process group ID is equal to the absolute value of *ipid*.
-1	Specifies to wait for any child process; this is the same behavior as PXFWAIT.

Value	Action
0	Specifies to wait for any child process whose process group ID is equal to that of the calling process.
> 0	Specifies to wait for the child whose process ID is equal to the value of *ipid*.

istat (Output) INTEGER(4). The returned status of the child process.

ioptions (Input) INTEGER(4). One or more of the following constant values (which can be passed to PXFCONST or IPXFCONST):

Value	Action
WNOHANG	Specifies to return immediately if no child process has exited.
WUNTRACED	Specifies to return for child processes that have stopped, and whose status has not been reported.

iretpid (Output) INTEGER(4). The PID of the stopped child process.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFWAITPID subroutine suspends execution of the current process until the child specified by *ipid* has exited, or until a signal is delivered whose action terminates the current process or calls a signal handling routine. If the child specified by *ipid* has already exited by the time of the call (a "zombie" process), a return is immediately made. Any system resources used by the child are freed.

The returned value in *istat* can be used in subroutines IPXFWEXITSTATUS, IPXFWSTOPSIG, IPXFWTERMSIG, PXFWIFEXITED, PXFWIFSIGNALLED, and PXFWIFSTOPPED.

See Also

PXFWAIT
IPXFWEXITSTATUS
IPXFWSTOPSIG
IPXFWTERMSIG
PXFWIFEXITED
PXFWIFSIGNALLED
PXFWIFSTOPPED

PXFWIFEXITED (L*X, M*X)

POSIX Function: *Determines if a child process has exited.*

Module

USE IFPOSIX

Syntax

```
result = PXFWIFEXITED (istat)
```

istat	(Output) INTEGER(4). The status of the child process (obtained from PXFWAIT or PXFWAITPID).

Results

The result type is logical. The result value is .TRUE. if the child process has exited normally; otherwise, .FALSE..

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if
  end if
10 FORMAT (A,Z)
end program
```

See Also

PXFWIFSIGNALED
PXFWIFSTOPPED

PXFWIFSIGNALED (L*X, M*X)

POSIX Function: *Determines if a child process has exited because of a signal.*

Module

USE IFPOSIX

Syntax

```
result = PXFWIFSIGNALED (istat)
```

istat	(Output) INTEGER(4). The status of the child process (obtained from PXFWAIT or PXFWAITPID).

Results

The result type is logical. The result value is .TRUE. if the child process has exited because of a signal that was not caught; otherwise, .FALSE..

See Also

PXFWIFEXITED
PXFWIFSTOPPED

PXFWIFSTOPPED (L*X, M*X)

POSIX Function: *Determines if a child process has stopped.*

Module

USE IFPOSIX

Syntax

```
result = PXFWIFSTOPPED (istat)
```

istat	(Output) INTEGER(4). The status of the child process (obtained from PXFWAIT or PXFWAITPID).

Results

The result type is logical. The result value is .TRUE. if the child process has stopped; otherwise, .FALSE..

See Also

PXFWIFEXITED
PXFWIFSIGNALED

PXFWRITE

POSIX Subroutine: *Writes to a file.*

Module

USE IFPOSIX

Syntax

```
CALL PXFWRITE (ifildes,buf,nbyte,nwritten,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor for the file to be written to.
buf	(Input) Character. The buffer that contains the data to write into the file.
nbyte	(Input) INTEGER(4). The number of bytes to write.
nwritten	(Output) INTEGER(4). The returned number of bytes written.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFWRITE subroutine writes *nbyte* bytes from the storage *buf* into a file specified by file descriptor *ifildes*. The subroutine returns the total number of bytes read into *nwritten*. If no error occurs, the value of *nwritten* will equal the value of *nbyte*.

See Also
PXFREAD

Q to R

Q to R

QCMPLX
Elemental Intrinsic Function (Specific): *Converts an argument to COMPLEX(16) type. This function cannot be passed as an actual argument.*

Syntax
```
result = QCMPLX (x[,y])
```

x	(Input) Must be of type integer, real, or complex.
y	(Input; optional) Must be of type integer or real. It must not be present if *x* is of type complex.

Results
The result type is COMPLEX(16) (or COMPLEX*32).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX(REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

QCMPLX(*x*, *y*) has the complex value whose real part is REAL(*x*, *kind*=16) and whose imaginary part is REAL(*y*, *kind*=16).

Example
QCMPLX (-3) has the value (-3.0Q0, 0.0Q0).

QCMPLX (4.1, 2.3) has the value (4.1Q0, 2.3Q0).

See Also
CMPLX
DCMPLX
FLOAT
INT
IFIX
REAL
SNGL

QEXT

Elemental Intrinsic Function (Generic): *Converts a number to quad precision (REAL(16)) type.*

Syntax

```
result = QEXT (a)
```

a (Input) Must be of type integer, real, or complex.

Results

The result type is REAL(16) (REAL*16). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If *a* is of type REAL(16), the result is the value of the *a* with no conversion (QEXT(a) = a).

If *a* is of type integer or real, the result has as much precision of the significant part of *a* as a REAL(16) value can contain.

If *a* is of type complex, the result has as much precision of the significant part of the real part of *a* as a REAL(16) value can contain.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(16)
	INTEGER(2)	REAL(16)
	INTEGER(4)	REAL(16)
	INTEGER(8)	REAL(16)
QEXT	REAL(4)	REAL(16)
QEXTD	REAL(8)	REAL(16)
	REAL(16)	REAL(16)
	COMPLEX(4)	REAL(16)
	COMPLEX(8)	REAL(16)
	COMPLEX(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

Example

QEXT (4) has the value 4.0 (rounded; there are 32 places to the right of the decimal point).

QEXT ((3.4, 2.0)) has the value 3.4 (rounded; there are 32 places to the right of the decimal point).

QFLOAT

Elemental Intrinsic Function (Generic): *Converts an integer to quad precision (REAL(16)) type.*

Syntax

```
result = QFLOAT (a)
```

a (Input) Must be of type integer.

Results

The result type is REAL(16) (REAL*16).

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

Example

QFLOAT (-4) has the value -4.0 (rounded; there are 32 places to the right of the decimal point).

QNUM

Elemental Intrinsic Function (Specific): *Converts a character string to a REAL(16) value. This function cannot be passed as an actual argument.*

Syntax

```
result = QNUM (i)
```

i (Input) Must be of type character.

Results

The result type is REAL(16). The result value is the real value represented by the character string *i*.

Example

QNUM ("-174.23") has the value -174.23 of type REAL(16).

QRANSET

Portability Subroutine: *Sets the seed for a sequence of pseudo-random numbers.*

Module

USE IFPORT

Syntax

```
CALL QRANSET (rseed)
```

rseed (Input) INTEGER(4). The reset value for the seed.

QREAL

Elemental Intrinsic Function (Specific): *Converts the real part of a COMPLEX(16) argument to REAL(16) type. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.*

Syntax

```
result = QREAL (a)
```

a	(Input) Must be of type COMPLEX(16) (or COMPLEX*32).

Results

The result type is quad-precision real (REAL(16) or REAL*16).

Example

QREAL ((2.0q0, 3.0q0)) has the value 2.0q0.

See Also

REAL
DREAL

QSORT

Portability Subroutine: *Performs a quick sort on an array of rank one.*

Module

USE IFPORT

Syntax

CALL QSORT (*array*,*len*,*isize*,*compar*)

array	(Input) Any type except assumed-length character. A one-dimensional array to be sorted. If the data type does not conform to one of the predefined interfaces for QSORT, you may have to create a new interface (see Note and Example below).
len	(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Number of elements in *array*.
isize	(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Size, in bytes, of a single element of *array*: • 4 if array is of type REAL(4) • 8 if array is of type REAL(8) or complex • 16 if array is of type COMPLEX(8)
compar	(Input) INTEGER(2). Name of a user-defined ordering function that determines sort order. The type declaration of *compar* takes the form: INTEGER(2) FUNCTION *compar*(*arg1*, *arg2*) where *arg1* and *arg2* have the same type as *array* (above) and are not assumed-length character. Assume-length characters should be wrapped in a derived type. After you have created an ordering scheme, implement your sorting function so that it returns the following: • Negative if arg1 should precede arg2 • Zero if arg1 is equivalent to arg2 • Positive if arg1 should follow arg2 Dummy argument *compar* must be declared as external.

In place of an INTEGER kind, you can specify the constant SIZEOF_SIZE_T, defined in `IFPORT.F90`, for argument *len* or *isize*. Use of this constant ensures correct compilation.

NOTE
If you use QSORT with different data types, your program must have a USE IFPORT statement so that all the calls work correctly. In addition, if you wish to use QSORT with a derived type or a type that is not in the predefined interfaces, you must include an overload for the generic subroutine QSORT. Examples of how to do this are in the portability module's source file, `IFPORT.F90`.

Example

```
! program showing how to call 'QSORT' on
! a user-defined type.
!
! Define the type to be shared.
!
module share_type
    type element_type
        integer       :: data
        character(10) :: key
    end type
end module

! Main program calls QSORT.
!
program main

    use, intrinsic :: iso_c_binding, only: c_size_t
    use IFPORT       ! To get QSORT
    use share_type   ! To get shared type

    ! Define an overload of the default QSORT signature
    ! with a signature using the shared type.
    !
    interface
        subroutine QSORT_element_types(array, len, isize, comp)
           use, intrinsic :: iso_c_binding, only:c_size_t
           use share_type
           type(element_type) array(len)
           integer(C_SIZE_T) len, isize
           integer(2), external :: comp
           !
           ! Hook the overload to the real thing but be careful
           ! to connect to the correct qsort: the Fortran one, not
           ! the C one!
           !
           ! We need to call the _Fortran_ qsort, not the _C_ one, or
           ! there will be errors from the 1-origin vs. 0-origin indexing
           ! and the row-major vs. column-major ordering.
           !
           ! The symptom is that "OrderCharCI" is called with pointer values
           ! which are outside the bounds of the array to be sorted.
           !
           !DIR$ IF DEFINED(_WIN64)
```

```
          !DIR$ ATTRIBUTES ALIAS:'QSORT' :: QSORT_element_types
          !DIR$ ELSE
          !DIR$ ATTRIBUTES ALIAS: '_QSORT' :: QSORT_element_types
          !DIR$ ENDIF
        end subroutine QSORT_element_types
    end interface

    type(element_type) :: c(7)

    integer(2), external :: OrderCharCI

    integer (C_SIZE_T) :: size_of_element, size_of_array
    ! Fill in the array to be sorted.  The data value is chosen so
    ! that the sorted array will have the values in numeric order.
    ! Thus we can check the result of the sort.
    !
    c(1)%key  = 'aisjdop'
    c(1)%data = 3
    c(2)%key  = '35djf2'
    c(2)%data = 1
    c(3)%key  = 'ss:ss'
    c(3)%data = 6
    c(4)%key  = 'MMhQQ'
    c(4)%data = 4
    c(5)%key  = 'mmHqq'
    c(5)%data = 5
    c(6)%key  = 'aaaa'
    c(6)%data = 2
    c(7)%key  = '["\/'
    c(7)%data = 7

    size_of_array   = size(c)         !  7
    size_of_element = sizeof(c(1))    ! 16

    write(*,*) '"C" is:'
    do i = 1, 7
        write(*,*) ' "', c(i)%key, '" value ', c(i)%data
    end do

    write(*,*) ' '
    write(*,*) 'size of C is            ', size_of_array, ' elements'
    write(*,*) 'size of element C(1) is ', size_of_element, ' bytes'
    write(*,*) 'len of key in C(1) is   ',   len(c(1)%key)
    write(*,*) ' '

    ! Call the overloaded QSORT routine.
    !
    Call QSort_element_types(C, size_of_array, size_of_element, OrderCharCI)

    write(*,*) 'Sorted "C" is '
    do i = 1, 7
         write(*,*) ' "', c(i)%key, '" value ', c(i)%data
    end do

end program main

! Computes order of character strings using a case insensitive ordering.
!
```

```
! Return -1 if C1 before C2, 0 if C1 = C2, and 1 if C1 after C2.
!
! Called first with the pair (2,3), then (1,2), then (1,3)...when passing
! character strings of length 10.
!
! Passing "element_type" objects, it's called first with the pair (1, <invalid>),
! and the second item has a address well before the beginning of "C".
!

function OrderCharCI(c1, c2)
    use share_type

    implicit none

    type(element_type), intent(in) :: c1 ! Character strings to be ordered.
    type(element_type), intent(in) :: c2 !

    ! Function result:
    !
    integer(2) :: OrderCharCI

    ! Locals:
    !
    character(10) :: c1L !} Local copies of c1 and c2.
    character(10) :: c2L !}

    integer :: i ! Loop index.

    write(*,*)'OrderCharCI, parameter C1 is "', c1%key, '" ', c1%data, ', len is ', len(c1%key)
    write(*,*)' len_trim is ', len_trim(c1%key)
    write(*,*) ' '

    ! SEGV on access to C2
    !
    write(*,*)'OrderCharCI, parameter C2 is "', c2%key, '" ', c2%data, ', len is ', len(c2%key)
    write(*,*)' len_trim is ', len_trim(c2%key)
    write(*,*) ' '
    c1L = c1%key
    c2L = c2%key

    write(*,*) 'about to start do loop'

    do i = 1, len_trim(C1L)
        if ('a' <= C1L(i:i) .and. c1L(i:i) <= 'z') c1L(i:i) = char(ichar(c1L(i:i)) - ichar('a')
+ ichar('A'))
    end do
    do i = 1, len_trim(C2L)
        if ('a' <= c2L(i:i) .and. c2L(i:i) <= 'z') c2L(i:i) = char(ichar(c2L(i:i)) - ichar('a')
+ ichar('A'))
    end do
    if (c1L == c2L) Then
        OrderCharCI = 0
        write(*,*) ' - equal'
    else if (c1L < c2L) Then
        OrderCharCI = -1
        write(*,*) ' - c1 is less'
    else
        OrderCharCI = 1
```

```
        write(*,*) ' - c1 is more'
    end if
end function OrderCharCI
```

The following shows another example:

```
PROGRAM SORTQ
   use, intrinsic :: iso_c_binding, only: c_size_t
   use IFPORT
   integer(2), external :: cmp_function
   integer(2) insort(26), i
   integer (C_SIZE_T) array_len, array_size
   array_len = 26
   array_size = 2
   do i=90,65,-1
     insort(i-64)=91 - i
   end do
   print *, "Before: "
   print *,insort
   CALL qsort(insort,array_len,array_size,cmp_function)
   print *, 'After: '
   print *, insort
END
!
   integer(2) function cmp_function(a1, a2)
   integer(2) a1, a2
   cmp_function=a1-a2
   end function
```

RADIX

Inquiry Intrinsic Function (Generic): *Returns the base of the model representing numbers of the same type and kind as the argument.*

Syntax

`result = RADIX (x)`

x	(Input) Must be of type integer or real; it can be scalar or array valued.

Results

The result is a scalar of type default integer. For an integer argument, the result has the value r (as defined in Model for Integer Data). For a real argument, the result has the value b (as defined in Model for Real Data).

Example

If X is a REAL(4) value, RADIX (X) has the value 2.

See Also

DIGITS
EXPONENT
FRACTION
Data Representation Models

RAISEQQ

Portability Function: *Sends a signal to the executing program.*

Module

USE IFPORT

Syntax

`result = RAISEQQ (sig)`

sig

(Input) INTEGER(4). Signal to raise. One of the following constants (defined in `IFPORT.F90`):

- SIG$ABORT - Abnormal termination
- SIG$FPE - Floating-point error
- SIG$ILL - Illegal instruction
- SIG$INT - CTRL+Csignal
- SIG$SEGV - Illegal storage access
- SIG$TERM - Termination request

If you do not install a signal handler (with SIGNALQQ, for example), when a signal occurs the system by default terminates the program with exit code 3.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

If a signal-handling routine for *sig* has been installed by a prior call to SIGNALQQ, RAISEQQ causes that routine to be executed. If no handler routine has been installed, the system terminates the program (the default action).

See Also

SIGNALQQ
SIGNAL
KILL

RAN

Nonelemental Intrinsic Function (Specific): *Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.*

Syntax

`result = RAN (i)`

i

(Input; output) Must be an INTEGER(4) variable or array element.

It should initially be set to a large, odd integer value. The RAN function stores a value in the argument that is later used to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs to obtain different random numbers.

Results

The result type is REAL(4). The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. It is set equal to the value associated with the argument *i*.

RAN is not a pure function.

Example

In RAN (I), if variable I has the value 3, RAN has the value 4.8220158E-05.

See Also

RANDOM
RANDOM_NUMBER

RAND, RANDOM

Portability Functions: *Return real random numbers in the range 0.0 to 1.0, not including the end points.*

Module

USE IFPORT

Syntax

```
result = RAND ([ iflag])
result = RANDOM (iflag)
```

iflag (Input) INTEGER(4). Optional for RAND. Controls the way the random number is selected.

Results

The result type is REAL(4). RAND and RANDOM return random numbers in the range 0.0 to 1.0, not including the end points.

Value of *iflag*	Selection process
1	The generator is restarted and the first random value is selected.
0	The next random number in the sequence is selected.
Otherwise	The generator is reseeded using *iflag*, restarted, and the first random value is selected.

When RAND is called without an argument, the following applies:

- The value of *iflag* is assumed to be 0.
- You must specify USE IFPORT.

There is no difference between RAND and RANDOM. Both functions are included to ensure portability of existing code that references one or both of them.

The intrinsic functions RANDOM_NUMBER and RANDOM_SEED provide the same functionality and they are the recommended functions to use when writing programs to generate random numbers.

You can use SRAND to restart the pseudorandom number generator used by RAND.

NOTE
RANDOM is available as a function or subroutine.

Example

The following example shows how to use both the RANDOM function and the RANDOM subroutine:

```
use ifport
real(4) ranval
call seed(1995) ! initialize
call random(ranval) ! get next random number
print *,ranval

ranval = random(1) ! initialize
ranval = random(0) ! get next random number

print *,ranval
end
```

See Also

RANDOM_NUMBER
RANDOM_SEED
SRAND

RANDOM Subroutine

Portability Subroutine: *Returns a pseudorandom number greater than or equal to zero and less than one from the uniform distribution.*

Module

USE IFPORT

Syntax

`CALL RANDOM (ranval)`

ranval	(Output) REAL(4). Pseudorandom number, 0 *ranval*< 1, from the uniform distribution.

A given seed always produces the same sequence of values from RANDOM.

If SEED is not called before the first call to RANDOM, RANDOM begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, pass the constant RND$TIMESEED (defined in `IFCORE.F90`) to SEED before the first call to RANDOM.

The portability routines DRAND, DRANDM, IRAND, IRANDM, RAN, RAND, and the RANDOM portability function and subroutine use the same algorithms and thus return the same answers. They are all compatible and can be used interchangeably. The algorithm used is a "Prime Modulus M Multiplicative Linear Congruential Generator," a modified version of the random number generator by Park and Miller in "Random Number Generators: Good Ones Are Hard to Find," CACM, October 1988, Vol. 31, No. 10.

Example

```
USE IFPORT
REAL(4) ran

CALL SEED(1995)
CALL RANDOM(ran)
```

The following example shows how to use both the RANDOM subroutine and the RANDOM function:

```
use ifport
real(4) ranval
! from libifcore.lib
call seed(1995)     ! initialize
! also from for_m_irand.c in libfor
call random(ranval) ! get next random number

print *,ranval
! from libifport.lib
ranval = random(1)  ! initialize
! same
ranval = random(0)  ! get next random number

print *,ranval
end
```

See Also

RANDOM_NUMBER
SEED
DRAND and DRANDM
IRAND and IRANDM
RAN
RAND

RANDOM_NUMBER

Intrinsic Subroutine: *Returns one pseudorandom number or an array of such numbers.*

Syntax

`CALL RANDOM_NUMBER (`*`harvest`*`)`

harvest	(Output) Must be of type real. It can be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution within the range 0 <= *x* < 1.

The seed for the pseudorandom number generator used by RANDOM_NUMBER can be set or queried with RANDOM_SEED. If RANDOM_SEED is not used, the processor sets the seed for RANDOM_NUMBER to a processor-dependent value.

The RANDOM_NUMBER generator uses two separate congruential generators together to produce a period of approximately 10**18, and produces real pseudorandom results with a uniform distribution in [0, 1). It accepts two integer seeds, the first of which is reduced to the range [1, 2147483562]. The second seed is reduced to the range [1, 2147483398]. This means that the generator effectively uses two 31-bit seeds.

The RANDOM_NUMBER generator does not produce subnormal numbers.

For more information on the algorithm, see the following:

- Communications of the ACM vol 31 num 6 June 1988, titled: Efficient and Portable Combined Random Number Generators by Pierre L'ecuyer.
- Springer-Verlag New York, N. Y. 2nd ed. 1987, titled: A Guide to Simulation by Bratley, P., Fox, B. L., and Schrage, L. E.

Example

Consider the following:

```
REAL Y, Z (5, 5)
! Initialize Y with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = Y)
CALL RANDOM_NUMBER (Z)
```

Y and Z contain uniformly distributed random numbers.

The following shows another example:

```
REAL x, array1 (5, 5)
CALL RANDOM_SEED()
CALL RANDOM_NUMBER(x)
CALL RANDOM_NUMBER(array1)
```

The following shows another example:

```
program testrand
  intrinsic random_seed, random_number
  integer size
  integer, allocatable :: seed(:), gseed(:), hiseed(:), zseed(:)
  real harvest(10)
  call random_seed(SIZE=size)
  print *,"size ",size
  allocate(seed(size),gseed(size),hiseed(size),zseed(size))
  hiseed = -1
  zseed = 0
  seed = 123456789
  seed(size) = 987654321
  call random_seed(PUT=hiseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"hiseed gseed", hiseed, gseed
  call random_seed(PUT=zseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"zseed gseed ", zseed, gseed
  call random_seed(PUT=seed(1:size))
  call random_seed(GET=gseed(1:size))
  call random_number(HARVEST=harvest)
  print *, "seed gseed ", seed, gseed
  print *, "harvest"
  print *, harvest
  call random_seed(GET=gseed(1:size))
  print *,"gseed after harvest ", gseed
end program testrand
```

See Also

RANF Intrinsic Procedure

RANDOM_SEED
RANDOM

SEED
DRAND and DRANDM
IRAND and IRANDM
RAN
RAND and RANDOM

RANDOM_SEED

Intrinsic Subroutine (Generic): *Changes or queries the seed (starting point) for the pseudorandom number generator used by intrinsic subroutine RANDOM_NUMBER. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL RANDOM_SEED ([size] [,put] [,get])
```

size	(Output; optional) Must be scalar and of type integer. Set to the number of integers (N) that the processor uses to hold the value of the seed.
put	(Input; optional) Must be an integer array of rank one and size greater than or equal to N. It is used to reset the value of the seed.
get	(Output; optional) Must be an integer array of rank one and size greater than or equal to N. It is set to the current value of the seed.

No more than one argument can be specified. If no argument is specified, a random number based on the date and time is assigned to the seed.

You can determine the size of the array the processor uses to store the seed by calling RANDOM_SEED with the *size* argument (see the second example below).

If RANDOM_SEED is called with no arguments, the seed is set to a different, unpredictable value on each call.

Example

Consider the following:

```
CALL RANDOM_SEED                        ! Processor initializes the
                                        !   seed randomly from the date
                                        !   and time
CALL RANDOM_SEED (SIZE = M)             ! Sets M to N
CALL RANDOM_SEED (PUT = SEED (1 : M))   ! Sets user seed
CALL RANDOM_SEED (GET = OLD  (1 : M))   ! Reads the current seed
```

The following shows another example:

```
INTEGER I
INTEGER, ALLOCATABLE :: new (:), old(:)
CALL RANDOM_SEED ( )  ! Processor reinitializes the seed
                      ! randomly from the date and time
CALL RANDOM_SEED (SIZE = I)  ! I is set to the size of
                             ! the seed array
ALLOCATE (new(I))
ALLOCATE (old(I))
CALL RANDOM_SEED (GET=old(1:I)) ! Gets the current seed
WRITE(*,*) old
new = 5
```

```
CALL RANDOM_SEED (PUT=new(1:I)) ! Sets seed from array
                                ! new
END
```

See Also

RANDOM_NUMBER
SEED
SRAND

RANDU

Intrinsic Subroutine (Generic): *Computes a pseudorandom number as a single-precision value. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL RANDU (i1,i2,x)`

i1, i2	(Input; output) Must be scalars of type INTEGER(2) or INTEGER(4). They contain the *seed* for computing the random number. These values are updated during the computation so that they contain the updated seed.
x	(Output) Must be a scalar of type REAL(4). This is where the computed random number is returned.

The result is returned in *x*, which must be of type REAL(4). The result value is a pseudorandom number in the range 0.0 to 1.0. The algorithm for computing the random number value is based on the values for *i1* and *i2*.

The result value is a pseudorandom number in the range 0.0 to 1.0. The algorithm for computing the random number value is based on the values for *i1* and *i2*.

If *i1* = 0 and *i2* = 0, the generator base is set as follows:

```
    x(n + 1) = 2**16 + 3
```

Otherwise, it is set as follows:

```
    x(n + 1) = (2**16 + 3) * x(n) mod 2**32
```

The generator base x(n + 1) is stored in *i1*, *i2*. The result is x(n + 1) scaled to a real value y(n + 1), for 0.0 <= y(n + 1) < 1.

Example

Consider the following:

```
REAL X
INTEGER(2) I, J
...
CALL RANDU (I, J, X)
```

If I and J are values 4 and 6, X has the value 5.4932479E-04.

RANF Intrinsic Procedure

Elemental Intrinsic Function (Generic): *Generates a random number between 0.0 and RAND_MAX. This function must not be passed as an actual argument. RANF can be used as an intrinsic procedure or as a portability routine. It is an intrinsic procedure unless you specify USE IFPORT.*

Syntax

```
result = RANF ()
```

Results

The result type is REAL(4). The result value is a single-precision pseudo-random number between 0.0 and (2**31) - 1.

The initial seed is set by the following:

```
CALL SRAND(ISEED)
```

where ISEED is type INTEGER(4).

The intrinsic function RANF generates a different sequence of random numbers than the RANF portability function generates for the same seed. The intrinsic function RANF used inside a loop can be vectorized into one call that returns four results, but the portability function RANF cannot be so optimized.

See Also

RANDOM_NUMBER

RANF portability routine

SRAND

RANF Portability Routine

Portability Function: *Generates a random number between 0.0 and RAND_MAX. RANF can be used as a portability routine or as an intrinsic procedure. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

```
result = RANF ( )
```

Results

The result type is REAL(4). The result value is a single-precision pseudo-random number between 0.0 and (2**31) - 1.

The initial seed is set by the following:

```
CALL SRAND(ISEED)
```

where ISEED is type INTEGER(4).

See Also

RANDOM_NUMBER

RANF intrinsic procedure

SRAND

RANGE

Inquiry Intrinsic Function (Generic): *Returns the decimal exponent range in the model representing numbers with the same kind parameter as the argument.*

Syntax

```
result = RANGE (x)
```

x	(Input) Must be of type integer, real, or complex; it can be scalar or array valued.

Results

The result is a scalar of type default integer.

For an integer argument, the result has the value INT(LOG10(HUGE(*x*))). For information on the integer model, see Model for Integer Data.

For a real or complex argument, the result has the value INT(MIN (LOG10(HUGE(*x*)), -LOG10(TINY(*x*)))). For information on the real model, see Model for Real Data.

Example

If X is a REAL(4) value, RANGE (X) has the value 37. (HUGE(X) = (1 - 2^{-24}) x $2^{128}$and TINY(X) = 2^{-126})

See Also

HUGE

TINY

RANGET

Portability Subroutine: *Returns the current seed.*

Module

USE IFPORT

Syntax

```
CALL RANGET (seed)
```

seed	(Output) INTEGER(4). The current seed value.

RANK

Inquiry Intrinsic Function (Generic): *Returns the rank of a data object.*

Syntax

```
result = RANK (a)
```

a (Input) Is a data object. It can be of any type.

Results

The result type is default integer scalar. The result value is the rank of *a*.

Example

If object C is an assumed-rank dummy argument and its associated argument is an array of rank 5, RANK(C) returns the value 5.

If D is an array declared DIMENSION (2, 3, 4), RANK(D) returned the value 3.

See Also

Assumed-Rank Specifications

RANSET

Portability Subroutine: *Sets the seed for the random number generator.*

Module

USE IFPORT

Syntax

```
CALL RANSET (seed)
```

seed (Input) REAL(4). The reset value for the seed.

READ Statement

Statement: *Transfers input data from external sequential, direct-access, or internal records.*

Syntax

Sequential

Formatted:

```
READ (eunit, format [, advance] [, asynchronous] [, blank] [, decimal] [, id] [, pad] [, pos] [, round] [, size] [, iostat] [, err] [, end] [, eor] [, iomsg]) [io-list]
```

```
READ form[, io-list]
```

Formatted - List-Directed:

```
READ (eunit, *[, asynchronous] [, blank] [, decimal] [, id] [, pad] [, pos] [, round] [, size] [, iostat] [, err] [, end] [, iomsg]) [io-list]
```

```
READ *[, io-list]
```

Formatted - Namelist:

```
READ (eunit, nml-group[, asynchronous] [, blank] [, decimal] [, id] [, pad] [, pos] [, round] [, size] [, iostat] [, err] [, end] [, iomsg])
```

```
READ nml
```

Unformatted:

```
READ (eunit [, asynchronous] [, id] [, pos] [, iostat] [, err][, end] [, iomsg]) [io-list]
```

Direct-Access
Formatted:

```
READ (eunit, format, rec [, asynchronous] [, blank] [, decimal] [, id] [, pad] [, pos] [, round] [, size] [, iostat] [, err] [, iomsg]) [io-list]
```

Unformatted:

```
READ (eunit, rec [, asynchronous] [, id] [, pos] [, iostat] [, err] [, iomsg]) [io-list]
```

Internal

```
READ (iunit, format [, nml-group] [, iostat] [, err] [, end] [, iomsg]) [io-list]
```

Internal Namelist

```
READ (iunit, nml-group [, iostat] [, err] [, end] [, iomsg]) [io-list]
```

eunit
Is an external unit specifier, optionally prefaced by UNIT=. UNIT= is required if *eunit* is not the first specifier in the list.

format
Is a format specifier. It is optionally prefaced by FMT= if *format* is the second specifier in the list and the first specifier indicates a logical or internal unit specifier *without* the optional keyword UNIT=.

For internal READs, an asterisk (*) indicates list-directed formatting. For direct-access READs, an asterisk is not permitted.

advance
Is an advance specifier (ADVANCE=*c-expr*). If the value of *c-expr* is 'YES', the statement uses advancing input; if the value is 'NO', the statement uses nonadvancing input. The default value is 'YES'.

asynchronous
Is an asynchronous specifier (ASYNCHRONOUS=*i-expr*). If the value of *i-expr* is 'YES', the statement uses asynchronous input; if the value is 'NO', the statement uses synchronous input. The default value is 'NO'.

blank
Is a blank control specifier (BLANK = blnk). If the value of blnk is 'NULL', all blanks are ignored. If the value is 'ZERO', all blanks are treated as zeros. The default value is 'NULL'.

decimal
Is a decimal mode specifier (DECIMAL=*dmode*) that evaluates to 'COMMA' or 'POINT'. The default value is 'POINT'.

id
Is an id specifier (ID=*id-var*). If ASYNCHRONOUS='YES' is specified and the operation completes successfully, the id specifier becomes defined with an implementation-dependent value that can be specified in a future WAIT or INQUIRE statement to identify the particular data transfer operation. If an error occurs, the id specifier variable becomes undefined.

pad
Is a blank pading specifier (PAD = pd). If the value of pad is 'YES', the record will be padded with blanks when necessary. If the value is 'NO', the record will not be padded with blanks. The default value is 'YES'.

pos	Is a pos specifier (POS=*p*) that indicates a file position in file storage units in a stream file (ACCESS='STREAM'). It can only be specified on a file opened for stream access. If omitted, the stream I/O occurs starting at the next file position after the current file position.
round	Is a rounding specifier (ROUND=*rmode*) that determines the I/O rounding mode for this READ statement. If omitted, the rounding mode is unchanged. Possible values are UP, DOWN, ZERO, NEAREST, COMPATIBLE or PROCESSOR_DEFINED.
size	Is a character count specifier (SIZE=*i-var*).
iostat	Is the name of a variable to contain the completion status of the I/O operation. Optionally prefaced by IOSTAT=.
err, end, eor	Are branch specifiers if an error (ERR=*label*), end-of-file (END=*label*), or end-of-record (EOR=*label*) condition occurs. EOR can only be specified for nonadvancing READ statements.
iomsg	Is an I/O message specifier (IOMSG=*msg-var*).
io-list	Is an I/O list: the names of the variables, arrays, array elements, or character substrings from which or to which data will be transferred. Optionally an implied-DO list. If an item in *io-list* is an expression that calls a function, that function must not execute an I/O statement or the EOF intrinsic function on the same external unit as *eunit*. If I/O is to or from a formatted device, *io-list* cannot contain derived-type variables, but it can contain components of derived types. If I/O is to a binary or unformatted device, *io-list* can contain either derived type components or a derived type variable.
form	Is the nonkeyword form of a format specifier (no FMT=).
*	Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)
nml-group	Is the namelist group specification for namelist I/O. Optionally prefaced by NML=. NML= is required if *nml-group* is not the second I/O specifier. For more information, see Namelist Specifier.
nml	Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist I/O.
rec	Is the cell number of a record to be accessed directly. It must be prefaced by REC=.
iunit	Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if *iunit* is not the first specifier in the list. It must be a character variable. It must not be an array section with a vector subscript.

If you specify EOR= or SIZE=, you must also specify ASYNCHRONOUS='NO'.

If you specify BLANK=, DECIMAL=, PAD=, or ROUND=, you must also specify FMT= or NML=.

If you specify ID=, you must also specify ASYNCHRONOUS='YES'.

Caution
The READ statement can disrupt the results of certain graphics text functions (such as SETTEXTWINDOW) that alter the location of the cursor. You can avoid the problem by getting keyboard input with the GETCHARQQ function and echoing the keystrokes to the screen using OUTTEXT. Alternatively, you can use SETTEXTPOSITION to control cursor location.

Example

```
DIMENSION ia(10,20)
! Read in the bounds for the array.
! Then read in the array in nested implied-DO lists
! with input format of 8 columns of width 5 each.
READ (6, 990) il, jl, ((ia(i,j), j = 1, jl), i =1, il)
990 FORMAT (2I5, /, (8I5))

! Internal read gives a variable string-represented numbers
CHARACTER*12 str
str = '123456'
READ (str,'(i6)') i

! List-directed read uses no specified format
REAL x, y
INTEGER i, j
READ (*,*) x, y, i, j
```

See Also
I/O Lists
I/O Control List
Forms for Sequential READ Statements
Forms for Direct-Access READ Statements
Forms and Rules for Internal READ Statements
PRINT
WRITE
I/O Formatting

REAL Directive
General Compiler Directive: *Specifies the default real kind.*

Syntax
```
!DIR$ REAL:{ 4 | 8 | 16 }
```

The REAL directive selects a size of 4 (KIND=4), 8 (KIND=8), or 16 (KIND=16) bytes for default real numbers. When the directive is in effect, all default real and complex variables are of the kind specified in the directive. Only numbers specified or implied as REAL without KIND are affected.

The REAL directive can appear only at the top of a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. REAL cannot appear at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

Example

```
REAL r              ! a 4-byte REAL
WRITE(*,*) KIND(r)
CALL REAL8( )
WRITE(*,*) KIND(r)  ! still a 4-byte REAL
                    !   not affected by setting in subroutine
END
SUBROUTINE REAL8( )
  !DIR$ REAL:8
  REAL s ! an 8-byte REAL
  WRITE(*,*) KIND(s)
END SUBROUTINE
```

See Also

REAL
COMPLEX
General Compiler Directives
Syntax Rules for Compiler Directives

REAL Function

Elemental Intrinsic Function (Generic): *Converts a value to real type.*

Syntax

`result = REAL (a[,kind])`

a	(Input) Must be of type integer, real, or complex, or a binary, octal, or hexadecimal literal constant.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

The result value depends on the type and absolute value of *a* as follows:

- If *a* is integer or real, the result is equal to an approximation of *a*. If *a* is complex, the result is equal to an approximation of the real part of *a*.
- If *a* is a binary, octal, or hexadecimal literal constant, the value of the result is the value whose bit sequence according to the model in Bit Model is the same as that of *a* as modified by padding or truncation according to the following:
  - If the length of the sequence of bits specified by *a* is less than the size in bits of a scalar variable of the same type and kind type parameter as the result, the binary, octal, or hexadecimal literal constant is treated as if it were extended to a length equal to the size in bits of the result by padding on the left with zero bits.
  - If the length of the sequence of bits specified by *a* is greater than the size in bits of a scalar variable of the same type and kind type parameter as the result, the binary, octal, or hexadecimal literal constant is treated as if it were truncated from the left to a length equal to the size in bits of the result.
  - If a binary, octal, or hexadecimal literal constant is truncated as an argument to intrinsic function REAL, the discarded bits must all be zero.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(4)
FLOATI	INTEGER(2)	REAL(4)
FLOAT [2, 3]	INTEGER(4)	REAL(4)
REAL [2]	INTEGER(4)	REAL(4)
FLOATK	INTEGER(8)	REAL(4)
	REAL(4)	REAL(4)
SNGL [2, 4]	REAL(8)	REAL(4)
SNGLQ	REAL(16)	REAL(4)
	COMPLEX(4)	REAL(4)
	COMPLEX(8)	REAL(8)

[1] These specific functions cannot be passed as actual arguments.

[2]The setting of compiler options specifying real size can affect FLOAT, REAL, and SNGL.

[3] Or FLOATJ. For compatibility with older versions of Fortran, FLOAT is generic, allowing any kind of INTEGER argument, and returning a default real result.

[4] For compatibility with older versions of Fortran, SNGL is generic, allowing any kind of REAL argument, and returning a default real result.

If the argument is a a binary, octal, or hexadecimal constant, the result is affected by the `assume old-boz` option. The default option setting, `noold-boz`, treats the argument as a bit string that represents a value of the data type of the intrinsic, that is, the bits are not converted. If setting `old-boz` is specified, the argument is treated as a signed integer and the bits are converted.

Example

REAL (-4) has the value -4.0.

REAL (Y) has the same kind parameter and value as the real part of complex variable Y.

If C is complex, C%RE is the same as REAL (C).

See Also

Binary, Octal, Hexadecimal, and Hollerith Constants
Model for Bit Data
DFLOAT
DREAL
DBLE
`assume` compiler option

REAL Statement

Statement: *Specifies the REAL data type.*

Syntax

```
REAL
REAL([KIND=] n)
```

```
REAL* n
DOUBLE PRECISION
```

n	Is a constant expression that evaluates to kind 4, 8 or 16.

Description

If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is default real.

Default real is affected by compiler options specifying real size and by the REAL directive.

The default KIND for DOUBLE PRECISION is affected by compiler option double-size. If this compiler option is not specified, default DOUBLE PRECISION is REAL(8).

No kind parameter is permitted for data declared with type DOUBLE PRECISION.

REAL(4) and REAL*4 (single precision) are the same data type. REAL(8), REAL*8, and DOUBLE PRECISION are the same data type.

Example

```
! type declarations with attribute specifiers
REAL (8), PARAMETER :: testval=50.d0
REAL, SAVE :: a(10), b(20,30)
REAL, PARAMETER :: x = 100.

! attribute statements to declare the same entities
REAL x, a, b, testval*8
DIMENSION a(10), b(20,30)
SAVE a, b
PARAMETER (x = 100., testval=50.d0)
```

See Also

DOUBLE PRECISION
REAL directive
Real Data Types
General Rules for Real Constants
REAL(4) Constants
REAL(8) or DOUBLE PRECISION Constants
Real and Complex Editing
Model for Real Data

RECORD

Statement: *Declares a record structure as an entity with a name.*

Syntax

```
RECORD /structure-name/record-namelist [, /structure-name/record-namelist]...
```

structure-name	Is the name of a previously declared structure.
record-namelist	Is a list of one or more variable names, array names, or array specifications, separated by commas. All of the records named in this list have the same structure and are allocated separately in memory.

You can use record names in COMMON and DIMENSION statements, but not in DATA or NAMELIST statements.

Records initially have undefined values unless you have defined their values in structure declarations.

STRUCTURE and RECORD constructs have been replaced by derived types, which should be used in writing new code. See Derived Data Types.

Example

```
STRUCTURE /address/
  LOGICAL*2    house_or_apt
  INTEGER*2    apt
  INTEGER*2    housenumber
  CHARACTER*30 street
  CHARACTER*20 city
  CHARACTER*2  state
  INTEGER*4    zip
END STRUCTURE

RECORD /address/ mailing_addr(20), shipping_addr(20)
```

The following shows another example:

```
RECORD /T1/ a, b, /T2/ c, /T3/ d, e, f
```

See Also

TYPE
MAP...END MAP
STRUCTURE...END STRUCTURE
UNION...END UNION
Record Structures

RECTANGLE, RECTANGLE_W (W*S)

Graphics Functions: *Draw a rectangle using the current graphics color, logical write mode, and line style.*

Module

USE IFQWIN

Syntax

`result = RECTANGLE (control, x1, y1, x2, y2)`

`result = RECTANGLE_W (control, wx1, wy1, wx2, wy2)`

control	(Input) INTEGER(2). Fill flag. One of the following symbolic constants defined in `IFQWIN.F90`: • $GFILLINTERIOR - Draws a solid figure using the current color and fill mask. • $GBORDER - Draws the border of a rectangle using the current color and line style.
x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of rectangle.

wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of rectangle.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0.

The RECTANGLE function uses the viewport-coordinate system. The viewport coordinates (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangle.

The RECTANGLE_W function uses the window-coordinate system. The window coordinates (*wx1*, *wy1*) and (*wx2*, *wy2*) are the diagonally opposed corners of the rectangle.

SETCOLORRGB sets the current graphics color. SETFILLMASK sets the current fill mask. By default, filled graphic shapes are filled solid with the current color.

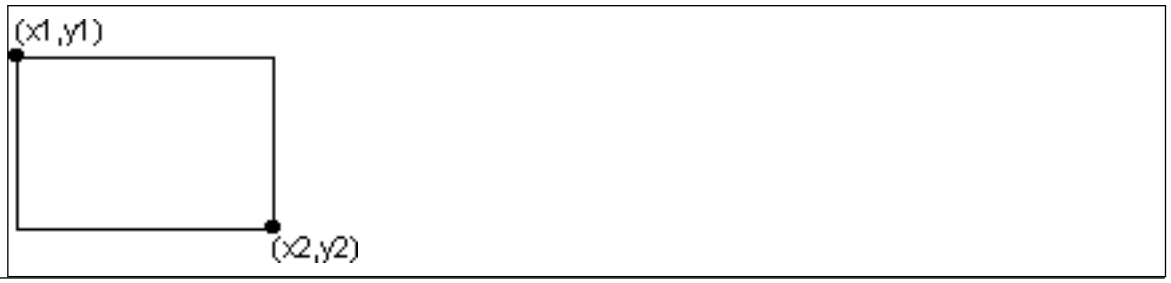
If you fill the rectangle using FLOODFILLRGB, the rectangle must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE
The RECTANGLE routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Rectangle routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Rectangle.

Example

This program draws the rectangle shown below.

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) dummy, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
dummy = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
END
```



See Also

SETFILLMASK
GRSTATUS
LINETO
POLYGON
FLOODFILLRGB
SETLINESTYLE
SETCOLOR

SETWRITEMODE

RECURSIVE and NON_RECURSIVE

Keywords: *RECURSIVE specifies that a subroutine or function can call itself directly or indirectly. NON_RECURSIVE specifies that a subroutine or function does not call itself directly or indirectly.*

Description

RECURSIVE or NON_RECURSIVE can be specified once in a FUNCTION or SUBROUTINE statement. They cannot both appear in the same statement.

Procedures not specified as RECURSIVE or NON_RECURSIVE are currently compiled as NON_RECURSIVE.

The default behavior can be changed by using the `-assume [no]recursion` option, or in an OPTIONS statement.

NOTE
The Fortran 2018 Standard specifies that the default mode is recursion; previous standards specified the default was no recursion. The default compilation mode will change to recursion in a future release.

If a function is directly recursive and array valued, and if the keyword RESULT is not specified in the FUNCTION statement, the result variable is the function name, and all occurrences of the function name in the executable part of the function are references to the function result variable.

A directly recursive function cannot have a declared type of CHARACTER if the character length is declared as *.

A procedure interface is always explicit within the subprogram that defines the procedure.

The keyword RECURSIVE *must* be specified if the compilation mode is set to non-recursive by a compiler option and if any of the following applies (directly or indirectly):

- The subprogram invokes itself.
- The subprogram invokes a subprogram defined by an ENTRY statement in the same subprogram.
- An ENTRY procedure in the same subprogram invokes one of the following:
  - Itself
  - Another ENTRY procedure in the same subprogram
  - The subprogram defined by the FUNCTION or SUBROUTINE statement

The keyword NON_RECURSIVE must be specified if the compilation mode is set to recursion by a compiler option and the procedure is not to be compiled for recursion.

Example

```
! RECURS.F90
!
  i = 0
  CALL Inc (i)
  END
  RECURSIVE SUBROUTINE Inc (i)
  i = i + 1
  CALL Out (i)
  IF (i.LT.20) CALL Inc (i)    ! This also works in OUT
  END SUBROUTINE Inc
```

```
  SUBROUTINE Out (i)
  WRITE (*,*) i
  END SUBROUTINE Out
```

See Also

ENTRY
FUNCTION
SUBROUTINE
OPTIONS
Program Units and Procedures
recursive compiler option

REDUCTION

Parallel Directive Clause: *Performs a reduction operation on the specified variables.*

Syntax

REDUCTION ([*reduction-modifier*,]*reduction-identifier* : *list*)

reduction-modifier	Is INSCAN. If present, an inclusive or exclusive scan computation is performed in the body of the loop (nest).
reduction-identifier	Is an identifier, user defined operator, or generic name which has appeared in an accessible DECLARE REDUCTION directive, or one of the predefined reduction operators or intrinsic names in the table of predefined *reduction-identifier*s below.
list	Is the name of one or more variables that are accessible to the scoping unit. Each *list* item must be definable. Each name must be separated by a comma. The type of each variable must be compatible with a type for which the *reduction-identifier* has an accessible definition. Assumed-size arrays, procedure pointers, and dummy arguments that are pointers with the INTENT (IN) attribute are not allowed. Array sections are allowed. For each list item, the number of copies is unspecified. The same list item cannot appear in both a REDUCTION and an IN_REDUCTION clause.

The INSCAN *reduction-modifier* is allowed only on a worksharing-loop, work-sharing SIMD loop, SIMD construct, a parallel work-sharing loop construct, or a parallel work-sharing SIMD construct. It indicates that in each iteration of the loop, a scan computation is performed over the updates to each list item. Each list item is made PRIVATE in the construct. Upon completion of the region, the value of the private copy of a list item from the last logical iteration of the loops of the construct is assigned to the original list item.

Each list item of a REDUCTION clause with the INSCAN *reduction-modifier* must appear as a list item in an INCLUSIVE or EXCLUSIVE clause on a SCAN directive contained in the loop (nest) enclosed in the construct. If INSCAN appears in one REDUCTION clause of a construct, all REDUCTION clauses of that construct must contain an INSCAN *reduction-modifier*. If an INSCAN *reduction-modifier* is applied to a construct that is combined with a TARGET construct, it is as if each list item also appears in a MAP clause with a *map-type* of TOFROM.

NOTE
Currently, ifort only supports SCAN directives on SIMD and TARGET SIMD construct directives.

Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in *list* is created for each thread as if the PRIVATE clause had been used. The private copy is initialized according to the *initializer-clause* of the *reduction-identifier*. A dummy argument that is a pointer with the INTENT (IN) attribute must not appear in a REDUCTION clause.

At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the *combiner* specified for the *reduction-identifier*. Reduction operations should all be associative (except for subtraction), and the compiler can freely reassociate the computation of the final value; the partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the construct containing the REDUCTION clause.

However, if the REDUCTION clause is used in a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all the threads complete the REDUCTION clause.

Any *list* item copies associated with the reduction must be initialized before they are accessed by the tasks participating in the reduction. After the end of the region, the original *list* item contains the result of the reduction.

An original *list* item with the POINTER attribute, or any pointer component of an original *list* item that is referenced, must be associated at entry to the construct that contains the REDUCTION clause. Also, the *list* item, or the pointer component of the *list* item, must not be deallocated, allocated, or pointer assigned within the region.

An original *list* item with the ALLOCATABLE attribute, or any allocatable component of an original *list* item that is referenced, must be in the allocated state at entry to the construct that contains the REDUCTION clause. Also, the *list* item, or the allocatable component of the *list* item, must be neither deallocated nor allocated within the region.

Any number of REDUCTION clauses can be specified on the directive, but a *list* item can appear only once in REDUCTION clauses for that directive.

If a *list* item is an array section, the following applies:

- The array section must specify contiguous storage and it cannot be a zero-length array section.
- Accesses to the elements of the array outside the specified array section result in unspecified behavior.
- The reduction clause will be applied to each separate element of the array section.
- Access to the elements of the array outside the specified array section result in unspecified behavior.

The following table lists predefined *reduction-identifier*s. These *reduction-identifier*s are implicitly defined and cannot be redefined in a DECLARE REDUCTION directive for the types shown here. The actual initialization value will be consistent with the data type of the reduction variable.

Predefined *reduction-identifiers*

reduction-identifier	Type	Combiner	Initializer
+	INTEGER, REAL, COMPLEX	omp_out = omp_out + omp_in	omp_priv = 0
*	INTEGER, REAL, COMPLEX	omp_out = omp_out * omp_in	omp_priv = 1

reduction-identifier	Type	Combiner	Initializer
-	INTEGER, REAL, COMPLEX	omp_out = omp_out + omp_in	omp_priv = 0
.AND.	LOGICAL	omp_out = omp_out .AND. omp_in	omp_priv = .TRUE.
.OR.	LOGICAL	omp_out = omp_out .OR. omp_in	omp_priv = .FALSE.
.EQV.	LOGICAL	omp_out = omp_out .EQV. omp_in	omp_priv = .TRUE.
.NEQV.	LOGICAL	omp_out = omp_out .NEQV. omp_in	omp_priv = .FALSE.
MAX	INTEGER, REAL	omp_out = max (omp_out, omp_in)	omp_priv =Smallest representable number
MIN	INTEGER, REAL	omp_out = min (omp_out, omp_in)	omp_priv = Largest representable number
IAND	INTEGER	omp_out = iand (omp_out, omp_in)	omp_priv = All bits set
IOR	INTEGER	omp_out = ior (omp_out, omp_in)	omp_priv = 0
IEOR	INTEGER	omp_out = ieor (omp_out, omp_in)	omp_priv = 0

If a directive allows REDUCTION clauses, the number you can specify is not limited. However, each variable name can appear only once in only one of the clauses.

NOTE
If a variable appears in a REDUCTION clause on a combined construct for which the first construct is TARGET, it is treated as if it had appeared in a MAP clause with a *map-type* of TOFROM.

Example

In the following program fragment, time will be the sum of the time spent in the do loop across all threads:

```
  use omp_lib
  integer i
  double precision t1, time
  call omp_set_num_threads(4)
  ...
!$omp do reduction(+:time)
  do i = 1, omp_get_num_threads()
    t1 = omp_get_wtime()
    ...
    time = omp_get_wtime() - t1
  end do
  !here time is equal to the total time across all threads
  ...
```

See Also

DECLARE REDUCTION
IN_REDUCTION
SCAN
TASK_REDUCTION

%REF

Built-in Function: *Changes the form of an actual argument. Passes the argument by reference. In Intel® Fortran, passing by reference is the default.*

Syntax

`%REF (a)`

a	(Input) An expression, record name, procedure name, array, character array section, or array element.

You must specify %REF in the actual argument list of a CALL statement or function reference. You cannot use it in any other context.

The following table lists the Intel® Fortran defaults for argument passing, and the allowed uses of %REF:

Actual Argument Data Type	Default	%REF
Expressions:		
Logical	REF	Yes
Integer	REF	Yes
REAL(4)	REF	Yes
REAL(8)	REF	Yes
REAL(16)	REF	Yes
COMPLEX(4)	REF	Yes
COMPLEX(8)	REF	Yes
COMPLEX(16)	REF	Yes
Character	See table note [1]	Yes
Hollerith	REF	No
Aggregate[2]	REF	Yes
Derived	REF	Yes
Array Name:		
Numeric	REF	Yes
Character	See table note [1]	Yes
Aggregate[2]	REF	Yes
Derived	REF	Yes

Actual Argument Data Type	Default	%REF
Procedure Name:		
Numeric	REF	Yes
Character	See table note [1]	Yes

[1]A character argument is passed by address and hidden length.

[2]In Intel® Fortran record structures

The %REF and %VAL functions override related !DIR$ ATTRIBUTE settings.

Example

```
CHARACTER(LEN=10) A, B
CALL SUB(A, %REF(B))
```

Variable A is passed by address and hidden length. Variable B is passed by reference.

Note that on Windows* systems, compiler option `iface` determines how the character argument for variable B is passed.

See Also

CALL
%VAL
%LOC
/iface compiler option

REGISTERMOUSEEVENT (W*S)

QuickWin Function: *Registers the application-supplied callback routine to be called when a specified mouse event occurs in a specified window.*

Module

USE IFQWIN

Syntax

`result = REGISTERMOUSEEVENT (unit,mouseevents,callbackroutine)`

unit

(Input) INTEGER(4). Unit number of the window whose callback routine on mouse events is to be registered.

mouseevents

(Input) INTEGER(4). One or more mouse events to be handled by the callback routine to be registered. Symbolic constants (defined in `IFQWIN.F90`) for the possible mouse events are:

- MOUSE$LBUTTONDOWN - Left mouse button down
- MOUSE$LBUTTONUP - Left mouse button up
- MOUSE$LBUTTONDBLCLK - Left mouse button double-click
- MOUSE$RBUTTONDOWN - Right mouse button down
- MOUSE$RBUTTONUP - Right mouse button up
- MOUSE$RBUTTONDBLCLK - Right mouse button double-click
- MOUSE$MOVE - Mouse moved

callbackroutine	(Input) Routine to be called on the specified mouse event in the specified window. It must be declared EXTERNAL.

Results

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer that can be one of the following:

- MOUSE$BADUNIT - The unit specified is not open, or is not associated with a QuickWin window.
- MOUSE$BADEVENT - The event specified is not supported.

For every BUTTONDOWN or BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double click interval, set in the system's CONTROL PANEL/MOUSE.

Example

The following example registers the routine CALCULATE, to be called when the user double-clicks the left mouse button while the mouse cursor is in the child window opened as unit 4:

```
USE IFQWIN
INTEGER(4) result
OPEN (4, FILE= 'USER')
...
result = REGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK, CALCULATE)
```

See Also

UNREGISTERMOUSEEVENT
WAITONMOUSEEVENT

REMAPALLPALETTERGB, REMAPPALETTERGB (W*S)

Graphics Functions: *REMAPALLPALETTERGB remaps a set of Red-Green-Blue (RGB) color values to indexes recognized by the video hardware.*
REMAPPALETTERGB remaps one color index to an RGB color value.

Module

USE IFQWIN

Syntax

`result = REMAPALLPALETTERGB (`*colors*`)`

`result = REMAPPALETTERGB (`*index*`, `*colors*`)`

colors	(Input) INTEGER(4). Ordered array of RGB color values to be mapped in order to indexes. Must hold 0-255 elements.
index	(Input) INTEGER(4). Color index to be reassigned an RGB color.
color	(Input) INTEGER(4). RGB color value to assign to a color index.

Results

The result type is INTEGER(4). REMAPALLPALETTERGB returns 0 if successful; otherwise, -1. REMAPPALETTERGB returns the previous color assigned to the index.

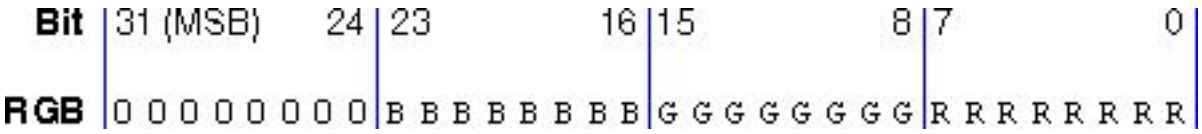
The REMAPALLPALETTERGB function remaps all of the available color indexes simultaneously (up to 236; 20 indexes are reserved by the operating system). The *colors* argument points to an array of RGB color values. The default mapping between the first 16 indexes and color values is shown in the following table. The 16 default colors are provided with symbolic constants in `IFQWIN.F90`.

Index	Color	Index	Color
0	$BLACK	8	$GRAY
1	$BLUE	9	$LIGHTBLUE
2	$GREEN	10	$LIGHTGREEN
3	$CYAN	11	$LIGHTCYAN
4	$RED	12	$LIGHTRED
5	$MAGENTA	13	$LIGHTMAGENTA
6	$BROWN	14	$YELLOW
7	$WHITE	15	$BRIGHTWHITE

The number of colors mapped can be fewer than 236 if the number of colors supported by the current video mode is fewer, but at most 236 colors can be mapped by REMAPALLPALETTERGB. Most Windows* graphics drivers support a palette of 256K colors or more, of which only a few can be mapped into the 236 palette indexes at a time. To access and use all colors on the system, bypass the palette and use direct RGB color functions such as such as SETCOLORRGB and SETPIXELSRGB.

Any RGB colors can be mapped into the 236 palette indexes. Thus, you could specify a palette with 236 shades of red.

In each RGB color value, each of the three colors, red, green and blue, is represented by an eight-bit value (2 hex digits). In the values you specify with REMAPALLPALETTERGB or REMAPPALETTERGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 11111111 (hex FF) the maximum for each of the three components. For example, Z'008080' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

Example

```
! Build as QuickWin or Standard Graphics App.

USE IFQWIN
INTEGER(4) colors(3)
INTEGER(2) status
colors(1) = Z'00FFFF' ! yellow
colors(2) = Z'FFFFFF' ! bright white
colors(3) = 0         ! black

status = REMAPALLPALETTERGB(colors)
status = REMAPPALETTERGB(INT2(47), Z'45A315')
END
```

See Also

SETBKCOLORRGB
SETCOLORRGB
SETBKCOLOR
SETCOLOR

RENAME

Portability Function: *Renames a file.*

Module

USE IFPORT

Syntax

```
result = RENAME (from,to)
```

from	(Input) Character*(*). Path of an existing file.
to	(Input) Character*(*). The new path for the file (see Caution note below).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code, such as:

- EACCES - The file or directory specified by *to* could not be created (invalid path). This error is also returned if the drive specified is not currently connected to a device.
- ENOENT - The file or path specified by *from* could not be found.
- EXDEV - Attempt to move a file to a different device.

> **Caution**
> This routine can cause data to be lost. If the file specified in *to* already exists, RENAME deletes the pre-existing file.
>
> It is possible to rename a file to itself without error.
>
> The paths can use forward (/) or backward (\) slashes as path separators and can include drive letters (if permitted by your operating system).

Example

```
use IFPORT
integer(4) istatus
character*12 old_name, new_name
print *, "Enter file to rename: "
read *, old_name
print *, "Enter new name: "
read *, new_name
ISTATUS = RENAME (old_name, new_name)
```

See Also

RENAMEFILEQQ

RENAMEFILEQQ

Portability Function: *Renames a file or directory.*

Module

USE IFPORT

Syntax

`result = RENAMEFILEQQ (oldname,newname)`

oldname	(Input) Character*(*). Current name of the file or directory to be renamed.
newname	(Input) Character*(*). New name of the file or directory.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can use RENAMEFILEQQ to move a file from one directory to another on the same drive by giving a different path in the *newname* parameter.

If the function fails, call GETLASTERRORQQ to determine the reason. One of the following errors can be returned:

- ERR$ACCES - Permission denied. The file's or directory's permission setting does not allow the specified access.
- ERR$EXIST - The file or directory already exists.
- ERR$NOENT - File or directory or path specified by *oldname* not found.
- ERR$XDEV - Attempt to move a file or directory to a different device.

Example

```
USE IFPORT
USE IFCORE
INTEGER(4) len
CHARACTER(80) oldname, newname
LOGICAL(4) result

WRITE(*,'(A, \)') ' Enter old name: '
len = GETSTRQQ(oldname)
WRITE(*,'(A, \)') ' Enter new name: '
len = GETSTRQQ(newname)
result = RENAMEFILEQQ(oldname, newname)
END
```

See Also

FINDFILEQQ
RENAME
GETLASTERRORQQ

REPEAT

Transformational Intrinsic Function (Generic):
Concatenates several copies of a string.

Syntax

`result = REPEAT (string,ncopies)`

string	(Input) Must be scalar and of type character.
ncopies	(Input) Must be scalar and of type integer. It must not be negative.

Results

The result is a scalar of type character and length *ncopies* x LEN(*string*). The kind parameter is the same as *string*. The value of the result is the concatenation of *ncopies* copies of *string*.

Example

REPEAT ('S', 3) has the value SSS.

REPEAT ('ABC', 0) has the value of a zero-length string.

The following shows another example:

```
CHARACTER(6) str
str = REPEAT('HO', 3)  ! returns HOHOHO
```

See Also

SPREAD

RESHAPE

Transformational Intrinsic Function (Generic):
Constructs an array with a different shape from the argument array.

Syntax

```
result = RESHAPE (source,shape[,pad] [,order])
```

source	(Input) Must be an array. It may be of any data type. It supplies the elements for the result array. Its size must be greater than or equal to PRODUCT(*shape*) if *pad* is omitted or has size zero.
shape	(Input) Must be an integer array of up to 31 elements, with rank one and constant size. It defines the shape of the result array. Its size must be positive; its elements must not have negative values.
pad	(Input; optional) Must be an array with the same type and kind parameters as *source*. It is used to fill in extra values if the result array is larger than *source*.
order	(Input; optional) Must be an integer array with the same shape as *shape*. Its elements must be a permutation of (1,2,...,n), where n is the size of *shape*. If *order* is omitted, it is assumed to be (1,2,...,n).

Results

The result is an array of shape *shape* with the same type and kind parameters as *source*. The size of the result is the product of the values of the elements of *shape*.

In the result array, the array elements of *source* are placed in the order of dimensions specified by *order*. If *order* is omitted, the array elements are placed in normal array element order.

The array elements of *source* are followed (if necessary) by the array elements of *pad* in array element order. If necessary, additional copies of *pad* follow until all the elements of the result array have values.

NOTE
In standard Fortran array element order, the first dimension varies fastest. For example, element order in a two-dimensional array would be (1,1), (2,1), (3,1) and so on. In a three-dimensional array, each dimension having two elements, the array element order would be (1,1,1), (2, 1, 1), (1, 2, 1), (2, 2, 1), (1, 1, 2), (2, 1, 2), (1, 2, 2), (2, 2, 2).

RESHAPE can be used to reorder a Fortran array to match C array ordering before the array is passed from a Fortran to a C procedure.

Example

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 3/)) has the value

```
 [ 3  5  7 ]
 [ 4  6  8 ].
```

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 4/), (/1, 1/), (/2, 1/)) has the value

```
 [ 3  4  5  6 ]
 [ 7  8  1  1 ].
```

The following shows another example:

```
INTEGER AR1( 2, 5)
REAL F(5,3,8)
REAL C(8,3,5)
AR1 = RESHAPE((/1,2,3,4,5,6/),(/2,5/),(/0,0/),(/2,1/))
! returns      1 2 3 4 5
!              6 0 0 0 0
!
! Change Fortran array order to C array order
C = RESHAPE(F, (/8,3,5/), ORDER = (/3, 2, 1/))
END
```

See Also

PACK
SHAPE
TRANSPOSE
Array Assignment Statements

RESULT

Keyword: *Specifies a name for a function result.*

Description

Normally, a function result is returned in the function's name, and all references to the function name are references to the function result.

However, if you use the RESULT keyword in a FUNCTION statement, you can specify a local variable name for the function result. In this case, all references to the function name are recursive calls, and the function name must not appear in specification statements.

The RESULT name must be different from the name of the function.

Example

The following shows an example of a recursive function specifying a RESULT variable:

```
RECURSIVE FUNCTION FACTORIAL(P) RESULT(L)
  INTEGER, INTENT(IN) :: P
  INTEGER L
  IF (P == 1) THEN
    L = 1
  ELSE
    L = P * FACTORIAL(P - 1)
  END IF
END FUNCTION
```

The following shows another example:

```
 recursive function FindSame(Aindex,Last,Used) &
&  result(FindSameResult)
 type(card) Last
 integer Aindex, i
 logical matched, used(5), FindSameResult
 if( Aindex > 5 ) then
   FindSameResult = .true.
   return
 endif
 ...
```

See Also

FUNCTION
ENTRY
RECURSIVE
Program Units and Procedures

RETURN

Statement: *Transfers control from a subprogram to the calling program unit.*

Syntax

RETURN [*expr*]

expr	Is a scalar expression that is converted to an integer value if necessary. The *expr* is only allowed in subroutines; it indicates an alternate return. (An alternate return is an obsolescent feature in Standard Fortran.)

Description

When a RETURN statement is executed in a function subprogram, control is transferred to the referencing statement in the calling program unit.

When a RETURN statement is executed in a subroutine subprogram, control is transferred to the first executable statement following the CALL statement that invoked the subroutine, or to the alternate return (if one is specified).

Example

The following shows how alternate returns can be used in a subroutine:

```
   CALL CHECK(A, B, *10, *20, C)
   ...
10 ...
20 ...
   SUBROUTINE CHECK(X, Y, *, *, C)
   ...
50   IF (X) 60, 70, 80
60   RETURN
70   RETURN 1
80   RETURN 2
   END
```

The value of X determines the return, as follows:

- If X < 0, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the calling program.
- If X = = 0, the first alternate return (RETURN 1) occurs and control is transferred to the statement identified with label 10.
- If X > 0, the second alternate return (RETURN 2) occurs and control is transferred to the statement identified with label 20.

Note that an asterisk (*) specifies the alternate return. An ampersand (&) can also specify an alternate return in a CALL statement, but not in a subroutine's dummy argument list.

The following shows another example:

```
    SUBROUTINE Loop
    CHARACTER in
10  READ (*, '(A)') in
    IF (in .EQ. 'Y') RETURN
    GOTO 10
!   RETURN implied by the following statement:
    END

! The following example shows alternate returns:
    CALL AltRet (i, *10, *20, *30)
    WRITE (*, *) 'normal return'
    GOTO 40
10  WRITE (*, *) 'I = 10'
    GOTO 40
20  WRITE (*, *) 'I = 20'
    GOTO 40
30  WRITE (*, *) 'I = 30'
40  CONTINUE
    END
    SUBROUTINE AltRet (i, *, *, *)
    IF (i .EQ. 10) RETURN 1
    IF (i .EQ. 20) RETURN 2
    IF (i .EQ. 30) RETURN 3
    END
```

In the above example, RETURN 1 specifies the list's first alternate-return label, which is a symbol for the actual argument *10 in the CALL statement. RETURN 2 specifies the second alternate-return label, and RETURN 3 specifies the third alternate-return label.

See Also

CALL

CASE

REWIND

Statement: *Positions a sequential or direct access file at the beginning of the file (the initial point). It takes one of the following forms:*

Syntax

```
REWIND ([UNIT=] io-unit[, ERR= label] [, IOMSG=msg-var] [, IOSTAT=i-var])

REWIND io-unit
```

io-unit	(Input) Is an external unit specifier.
label	Is the label of the branch target statement that receives control if an error occurs.
msg-var	(Output) Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs.
i-var	(Output)Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

The unit number must refer to a file on disk or magnetic tape, and the file must be open for sequential, direct, or append access.

If a REWIND is done on a direct access file, the NEXTREC specifier is assigned a value of 1.

If a file is already positioned at the initial point, a REWIND statement has no effect.

If a REWIND statement is specified for a unit that is not open, it has no effect.

Example

The following statement repositions the file connected to I/O unit 3 to the beginning of the file:

```
REWIND 3
```

Consider the following statement:

```
REWIND (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 at the beginning of the file. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

The following shows another example:

```
WRITE (7, '(I10)') int
REWIND (7)
READ (7, '(I10)') int
```

See Also

OPEN
READ
WRITE
Data Transfer I/O Statements
Branch Specifiers

REWRITE

Statement: *Rewrites the current record.*

Syntax

Formatted:

```
REWRITE (eunit, format[, iostat] [, err]) [io-list]
```

Unformatted:

```
REWRITE (eunit[, iostat][ , err]) [io-list]
```

eunit	Is an external unit specifier ([UNIT=]io-unit).
format	Is a format specifier ([FMT=]format).
iostat	Is a status specifier (IOSTAT=i-var).
err	Is a branch specifier (ERR=label) if an error condition occurs.
io-list	Is an I/O list.

Description

In the REWRITE statement, data (translated if formatted; untranslated if unformatted) is written to the current (existing) record in a file with direct access.

The current record is the last record accessed by a preceding, successful sequential or direct-access READ statement.

Between a READ and REWRITE statement, you should not specify any other I/O statement (except INQUIRE) on that logical unit. Execution of any other I/O statement on the logical unit destroys the current-record context and causes the current record to become undefined.

Only one record can be rewritten in a single REWRITE statement operation.

The output list (and format specification, if any) must not specify more characters for a record than the record size. (Record size is specified by RECLin an OPEN statement.)

If the number of characters specified by the I/O list (and format, if any) do not fill a record, blank characters are added to fill the record.

Example

In the following example, the current record (contained in the relative organization file connected to logical unit 3) is updated with the values represented by NAME, AGE, and BIRTH:

```
      REWRITE (3, 10, ERR=99) NAME, ,AGE, BIRTH
10    FORMAT (A16, I2, A8)
```

RGBTOINTEGER (W*S)

QuickWin Function: *Converts three integers specifying red, green, and blue color intensities into a four-byte RGB integer for use with RGB functions and subroutines.*

Module

USE IFQWIN

Syntax

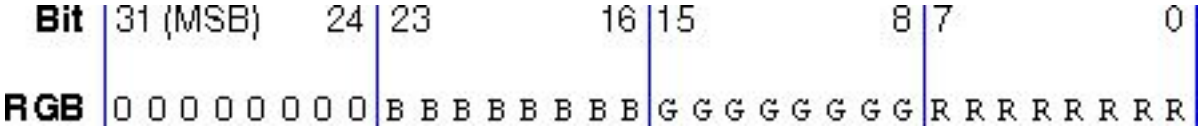
```
result = RGBTOINTEGER (red, green, blue)
```

red	(Input) INTEGER(4). Intensity of the red component of the RGB color value. Only the lower 8 bits of *red* are used.
green	(Input) INTEGER(4). Intensity of the green component of the RGB color value. Only the lower 8 bits of *green* are used.
blue	(Input) INTEGER(4). Intensity of the blue component of the RGB color value. Only the lower 8 bits of *blue* are used.

Results

The result type is INTEGER(4). The result is the combined RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value returned with RGBTOINTEGER, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

Example

```
! Build as a QuickWin App.
USE IFQWIN
INTEGER r, g, b, rgb, result
INTEGER(2) status
r = Z'F0'
g = Z'F0'
b = 0
rgb = RGBTOINTEGER(r, g, b)
result = SETCOLORRGB(rgb)
status = ELLIPSE($GFILLINTERIOR,INT2(40), INT2(55), &
                 INT2(90), INT2(85))
END
```

See Also

INTEGERTORGB
SETCOLORRGB
SETBKCOLORRGB
SETPIXELRGB
SETPIXELSRGB
SETTEXTCOLORRGB

RINDEX

Portability Function: *Locates the index of the last occurrence of a substring within a string.*

Module

USE IFPORT

Syntax

```
result = RINDEX (string,substr)
```

string (Input) Character*(*). Original string to search.

substr (Input) Character*(*). String to search for.

Results

The result type is INTEGER(4). The result is the starting position of the final occurrence of *substrg* in *string*. The result is zero if *substring* does not occur in *string*.

Example

```
USE IFPORT
character*80 mainstring
character*4 shortstr
integer(4) where
mainstring="Hello Hello Hello Hello There There There"
shortstr="Hello"
where=rindex(mainstring,shortstr)
! where is 19
```

See Also

INDEX

RNUM

Elemental Intrinsic Function (Specific): *Converts a character string to a REAL(4) value. This function cannot be passed as an actual argument.*

Syntax

```
result = RNUM (i)
```

i (Input) Must be of type character.

Results

The result type is REAL(4). The result value is the real value represented by the character string *i*.

Example

RNUM ("821.003") has the value 821.003 of type REAL(4).

RRSPACING

Elemental Intrinsic Function (Generic): *Returns the reciprocal of the relative spacing of model numbers near the argument value.*

Syntax

```
result = RRSPACING (x)
```

x	(Input) Must be of type real.

Results

The result type and kind are the same as *x*. The result has the value | *x** b $^{-e}$| x b ^{p}. Parameters *b*, *e*, *p* are defined in Model for Real Data.

Example

If -3.0 is a REAL(4) value, RRSPACING (-3.0) has the value 0.75 x 2 ^{24}.

The following shows another example:

```
REAL(4) res4
REAL(8) res8, r2
res4 = RRSPACING(3.0)  ! returns 1.258291E+07
res4 = RRSPACING(-3.0) ! returns 1.258291E+07
r2 = 487923.3
res8 = RRSPACING(r2)   ! returns 8.382458680573952E+015
END
```

See Also

SPACING
Data Representation Models

RSHIFT

Elemental Intrinsic Function (Generic): *Shifts the bits in an integer right by a specified number of positions. This is the same as specifying ISHFT with a negative shift.*

See Also

See ISHFT.

RTC

Portability Function: *Returns the number of seconds elapsed since a specific Greenwich mean time.*

Module

USE IFPORT

Syntax

```
result = RTC( )
```

Results

The result type is REAL(8). The result is the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

Example

```
USE IFPORT
real(8) s, s1, time_spent
INTEGER(4) i, j
s = RTC( )
call sleep(4)
```

```
s1 = RTC( )
time_spent = s1 - s
PRINT *, 'It took ',time_spent, 'seconds to run.'
```

See Also

DATE_AND_TIME

TIME portability routine

RUNQQ

Portability Function: *Executes another program and waits for it to complete.*

Module

USE IFPORT

Syntax

`result = RUNQQ (filename,commandline)`

filename	(Input) Character*(*). File name of a program to be executed.
commandline	(Input) Character*(*). Command-line arguments passed to the program to be executed.

Results

The result type is INTEGER(2). If the program executed with RUNQQ terminates normally, the exit code of that program is returned to the program that launched it. If the program fails, -1 is returned.

The RUNQQ function executes a new process for the operating system using the same path, environment, and resources as the process that launched it. The launching process is suspended until execution of the launched process is complete.

Example

```
USE IFPORT
INTEGER(2) result
result = RUNQQ('myprog', '-c -r')
END
```

See also the example in NARGS.

See Also

NARGS
SYSTEM
NARGS

S

S

SAME_TYPE_AS

Inquiry Intrinsic Function (Generic): *Inquires whether the dynamic type of one object is the same as the dynamic type of another object.*

Syntax

```
result = SAME_TYPE_AS (a , b)
```

a	(Input) Is an object of extensible type. If it is a pointer, it must not have an undefined association status.
b	(Input) Is an object of extensible type. If it is a pointer, it must not have an undefined association status.

Results

The result type is default logical scalar. The result is true only if the dynamic type of *a* is the same as the dynamic type of *b*.

SAVE

Statement and Attribute: *Causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram.*

Syntax

The SAVE attribute can be specified in a type declaration statement or a SAVE statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,] SAVE [, att-ls] :: entity[, entity ] ...
```

Statement:

```
SAVE [[::]entity [, entity ] ...]
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
entity	Is the name of an object, the name of a procedure pointer, or the name of a common block enclosed in slashes (/*common-block-name*/).

Description

In Intel® Fortran, certain variables are given the SAVE attribute, or not, by default. Variables are implicitly given the SAVE attribute depending on whether a program unit is compiled for recursion.

NOTE
In the following lists, "initialized" means that the variable has been given an initial value in a DATA statement, it has been initialized in a type declaration statement, it is of a derived type that is default initialized, or it has a component that is default initialized.

The following variables are *not* saved by default:

- Scalar variables that are local to a recursive procedure and are not initialized (see above Note)
- Arrays of any type that are local to a recursive procedure and are not initialized (see above Note)
- Variables that are declared AUTOMATIC
- Local variables with the ALLOCATABLE attribute

The following variables are saved by default:

- Variables in COMMON blocks
- Scalar variables not of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that are local to a non-recursive subprogram
- Non-scalar local variables of non-recursive subprograms
- Module or submodule variables
- Initialized (see above Note) variables
- RECORD variables that are data initialized by default initialization specified in its STRUCTURE declaration

Local variables that are not described in the preceding two lists are saved by default.

NOTE
Certain compiler options (such as options `[Q]save`, `assume norecursion`, and `auto`) and the use of OpenMP* features can change the defaults.

NOTE
Coarrays that do not have the ALLOCATABLE attribute and are local to the main program unit or to a procedure are not statically allocated, but are allocated at program startup. They are not deallocated until the program terminates.

To enhance portability and avoid possible compiler warning messages, Intel recommends that you use the SAVE statement to name variables whose values you want to preserve between subprogram invocations.

When a SAVE statement does not explicitly contain a list, all allowable items in the scoping unit are saved.

A SAVE statement cannot specify the following (their values cannot be saved):

- A blank common
- An object in a common block
- A procedure
- A dummy argument
- A function result
- An automatic object
- A PARAMETER (named) constant

Even though a common block can be included in a SAVE statement, individual variables within the common block can become undefined (or redefined) in another scoping unit.

If a common block is saved in any scoping unit of a program (other than the main program), it must be saved in every scoping unit in which the common block appears.

A SAVE statement has no effect in a main program.

Example

The following example shows a type declaration statement specifying the SAVE attribute:

```
SUBROUTINE TEST()
  REAL, SAVE :: X, Y
```

The following is an example of the SAVE statement:

```
SAVE A, /BLOCK_B/, C, /BLOCK_D/, E
```

The following shows another example:

```
  SUBROUTINE MySub
    COMMON /z/ da, in, a, idum(10)
    real(8) x,y
    ...

    SAVE x, y, /z/
! alternate declaration
    REAL(8), SAVE :: x, y
    SAVE /z/
```

See Also

COMMON
DATA
RECURSIVE and NON_RECURSIVE
MODULE
MODULE PROCEDURE
Type Declarations
Compatible attributes
SAVE value in CLOSE

SAVEIMAGE, SAVEIMAGE_W (W*S)

Graphics Functions: *Save an image from a specified portion of the screen into a Windows bitmap file.*

Module

USE IFQWIN

Syntax

`result = SAVEIMAGE (filename,ulxcoord,ulycoord, lrxcoord,lrycoord)`

`result = SAVEIMAGE_W (filename,ulwxcoord,ulwycoord, lrwxcoord,lrwycoord)`

filename	(Input) Character*(*). Path of the bitmap file.
ulxcoord, ulycoord	(Input) INTEGER(4). Viewport coordinates for upper-left corner of the screen image to be captured.
lrxcoord, lrycoord	(Input) INTEGER(4). Viewport coordinates for lower-right corner of the screen image to be captured.
ulwxcoord, ulwycoord	(Input) REAL(8). Window coordinates for upper-left corner of the screen image to be captured.

lrwxcoord, lrwycoord	(Input) REAL(8). Window coordinates for lower-right corner of the screen image to be captured.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The SAVEIMAGE function captures the screen image within a rectangle defined by the upper-left and lower-right screen coordinates and stores the image as a Windows bitmap file specified by *filename*. The image is stored with a palette containing the colors displayed on the screen.

SAVEIMAGE defines the bounding rectangle in viewport coordinates. SAVEIMAGE_W defines the bounding rectangle in window coordinates.

See Also

GETIMAGE, GETIMAGE_W
IMAGESIZE, IMAGESIZE_W
LOADIMAGE, LOADIMAGE_W
PUTIMAGE, PUTIMAGE_W

SCALE

Elemental Intrinsic Function (Generic): *Returns the value of the exponent part (of the model for the argument) changed by a specified value.*

Syntax

```
result = SCALE (x,i)
```

x	(Input) Must be of type real.
i	(Input) Must be of type integer.

Results

The result type and kind are the same as *x*. The result has the value *x* x b ^{i}. Parameter *b* is defined in Model for Real Data.

Example

If 3.0 is a REAL(4) value, SCALE (3.0, 2) has the value 12.0 and SCALE (3.0, 3) has the value 24.0.

The following shows another example:

```
REAL r
r = SCALE(5.2, 2)     ! returns 20.8
```

See Also

LSHIFT
Data Representation Models

SCAN Directive

OpenMP* Fortran Compiler Directive: *Specifies a scan computation that updates each list item in each iteration of the loop.*

Syntax

loop-associated-directive

```
do-loop-headers
   block
   !$OMP SCAN clause
   block
do-termination-statements
[end-loop-associated-directive]
```

loop-associated-directive	Is a SIMD or TARGET SIMD directive.
do-loop-headers	Is one or more Fortran DO statements.
block	Is a structured block (section) of statements or constructs.
clause	Is one of the following: • INCLUSIVE (*list*) • EXCLUSIVE (*list*) If an INCLUSIVE clause appears, an inclusive scan computation is performed for each *list* item in the clause. If an EXCLUSIVE clause appears, an exclusive scan computation is performed for each *list* item in the clause.
do-termination-statements	Is one or more Fortran DO-loop terminal statements, such as END DO, CONTINUE, or a terminal statement for a nonblock DO construct.
end-loop-associated-directive	Is an optional END SIMD or END TARGET SIMD directive.

The SCAN directive can appear in the body of a loop or loop nest that is in a worksharing construct, a worksharing-loop SIMD, or a SIMD construct.

There are two phases for each iteration of a SCAN loop, as follows:

- The input phase

  For each iteration of an INCLUSIVE scan loop, the statements that appear lexically prior to the SCAN directive constitute the input phase.

  For each iteration, except the last iteration of an EXCLUSIVE scan loop, the statements that lexically precede the directive constitute the input phase.
- The scan phase

  For each iteration of an INCLUSIVE scan loop, the statements that follow the directive constitute the scan phase.

  For each iteration, except the last iteration of an EXCLUSIVE scan loop, the statements that lexically follow the directive constitute the scan phase.

The last loop iteration does not have an input phase. For this iteration, all statements lexically preceding and following the directive constitute the scan phase. All computations that update a list item during an iteration are contained in the input phase. A statement that references a list item in the scan phase uses the result of the scan operation for that iteration.

For a given iteration, the result of a scan operation is calculated according to the last generalized prefixed sum ($PRESUM_{last}$) applied to the sequence of values given by the original value of the list item upon entry of the loop construct, and updated values given the list item in each of the iterations of the loop. The $PRESUM_{last}$ (*op*, a_1, . . . a_n) is defined for a binary operation *op* and a sequence of N values a_1, . . . a_n as follows:

- If N = 1, a_1

- If N > 1, *op* (PRESUM$_{last}$ (*op*, a_1, . . . a_k), PRESUM$_{last}$ (*op*, a_{k+1}, . . . a_n)) where 1 <= K <= N

At the beginning of the input phase for each iteration, the list item is initialized with the initializer value of the *reduction-identifier* specified by the REDUCTION clause on the innermost enclosing OpenMP* construct. For a given iteration, the update value of a list item is the value of the list item upon completion of the iteration's input phase.

If *orig_value* is the initial value of a list item upon entry to a worksharing-loop, worksharing SIMD, or SIMD construct, if *combiner* is the combiner for the *reduction-identifier* specified in the REDUCTION clause on the construct, and *update-value*$_i$ is the value for the list item for iteration i, then at the beginning of the scan phase of the first iteration, a list item of an INCLUSIVE clause on a SCAN directive is assigned the result of the operation PRESUM$_{last}$ (*combiner*, *orig_value*, *update-value*$_1$ . . . *update-value*$_i$). At the beginning of the scan phase of the first iteration, a list item of an EXCLUSIVE clause on a SCAN directive is assigned the value *orig_value*. At the beginning of the scan phase of each subsequent iteration, i > 1, a list item of an EXCLUSIVE clause on a SCAN directive is the result of the operation PRESUM$_{last}$ (*combiner*, *orig_value*, *update-value*$_1$ . . . *update-value*$_{i-1}$).

A worksharing-loop, worksharing-loop SIMD, or a SIMD construct that has a REDUCTION clause with the INSCAN modifier must contain exactly one SCAN directive in the loop body of the construct, and a list item in an INCLUSIVE or EXCLUSIVE clause must be a list item in the REDUCTION clause of the construct.

With the exception of dependencies for list items in the INCLUSIVE or EXCLUSIVE clause, cross-iteration dependencies across loop iterations are not permitted. Except for INCLUSIVE or EXCLUSIVE clause list-item dependencies, intra-iteration dependencies between a statement lexically preceding the SCAN directive and a statement lexically following a SCAN directive are not permitted.

NOTE
Currently, ifort only supports SCAN directives on SIMD and TARGET SIMD construct directives.

Example

The following contains an inclusive and an exclusive scan SIMD loop:

```
real,dimension(10)   :: a, b
real                 :: s
integer              :: i
...

do i = 1, 10
  a(i) = real (i)
end do
s = 0.0
! Inclusive scan
!$omp simd reduction (inscan, +:s)
do i = 1, 10
  s    = s + a(i)
  !$omp scan inclusive (s)
  b(i) = s
end do
print *, b

s = 0.0
! Exclusive scan
!$omp simd reduction (inscan, +:s)
do i = 1, 10
  b(i) = s
  !$omp scan exclusive (s)
```

```
   s    = s + a(i)
end do
print *, b
```

The first print statement prints the sequence 1.0 3.0 6.0 10.0 15.0 21.0 28.0 36.0 45.0 55.0. The second print statement prints the sequence 0.0 1.0 3.0 6.0 10.0 15.0 21.0 28.0 36.0 45.0.

See Also

REDUCTION clause
SIMD OpenMP* Fortran directive
TARGET SIMD
DO statement

SCAN Function

Elemental Intrinsic Function (Generic): *Scans a string for any character in a set of characters.*

Syntax

`result = SCAN (string, set [, back] [, kind])`

string	(Input) Must be of type character.
set	(Input) Must be of type character with the same kind parameter as *string*.
back	(Input; optional) Must be of type logical.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* is omitted (or is present with the value false) and *string* has at least one character that is in *set*, the value of the result is the position of the leftmost character of *string* that is in *set*.

If *back* is present with the value true and *string* has at least one character that is in *set*, the value of the result is the position of the rightmost character of *string* that is in *set*.

If no character of *string* is in *set* or the length of *string* or *set* is zero, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Example

SCAN ('ASTRING', 'ST') has the value 2.

SCAN ('ASTRING', 'ST', BACK=.TRUE.) has the value 3.

SCAN ('ASTRING', 'CD') has the value zero.

The following shows another example:

```
INTEGER i
INTEGER array(2)
i = SCAN ('FORTRAN', 'TR')                 ! returns 3
i = SCAN ('FORTRAN', 'TR', BACK = .TRUE.)  ! returns 5
i = SCAN ('FORTRAN', 'GHA')                ! returns 6
i = SCAN ('FORTRAN', 'ora')                ! returns 0
```

```
array = SCAN ((/'FORTRAN','VISUALC'/),(/'A', 'A'/))
                                      ! returns (6, 5)

! Note that when using SCAN with arrays, the string
! elements must be the same length. When using string
! constants, blank pad to make strings the same length.
! For example:

array = SCAN ((/'FORTRAN','MASM '/),(/'A', 'A'/))
                                      ! returns (6, 2)
END
```

See Also

VERIFY

SCANENV

Portability Subroutine: *Scans the environment for the value of an environment variable.*

Module

USE IFPORT

Syntax

CALL SCANENV (*envname*,*envtext*,*envvalue*)

envname	(Input) Character*(*). Contains the name of an environment variable you need to find the value for.
envtext	(Output) Character*(*). Set to the full text of the environment variable if found, or to ' ' if nothing is found.
envvalue	(Output) Character*(*). Set to the value associated with the environment variable if found or to ' ' if nothing is found.
	SCANENV scans for an environment variable that matches *envname* and returns the value or string it is set to.

SCROLLTEXTWINDOW (W*S)

Graphics Subroutine: *Scrolls the contents of a text window.*

Module

USE IFQWIN

Syntax

CALL SCROLLTEXTWINDOW (*rows*)

rows	(Input) INTEGER(2). Number of rows to scroll.

The SCROLLTEXTWINDOW subroutine scrolls the text in a text window (previously defined by SETTEXTWINDOW). The default text window is the entire window.

The *rows* argument specifies the number of lines to scroll. A positive value for *rows* scrolls the window up (the usual direction); a negative value scrolls the window down. Specifying a number larger than the height of the current text window is equivalent to calling CLEARSCREEN ($GWINDOW). A value of 0 for *rows* has no effect.

Example

```
!  Build as QuickWin or Standard Graphics app.
 USE IFQWIN INTEGER(2) row, istat
 CHARACTER(18) string
 TYPE (rccoord) oldpos

 CALL SETTEXTWINDOW (INT2(1), INT2(0), &
                     INT2(25), INT2(80))
 CALL CLEARSCREEN ( $GCLEARSCREEN )
 CALL SETTEXTPOSITION (INT2(1), INT2(1), oldpos)
 DO row = 1, 6
   string = 'Hello, World # '
   CALL SETTEXTPOSITION( row, INT2(1), oldpos )
   WRITE(string(15:16), '(I2)') row
   CALL OUTTEXT( string )
 END DO
 istat  = displaycursor($GCURSORON)
 WRITE(*,'(1x,A\)') 'Hit ENTER'
 READ (*,*)
 ! wait for ENTER
 !  Scroll window down 4 lines

 CALL SCROLLTEXTWINDOW(INT2( -4) )
 CALL SETTEXTPOSITION (INT2(10), INT2(18), oldpos)
 WRITE(*,'(2X,A\)') "Hit ENTER"
 READ( *,* ) ! wait for ENTER
 !  Scroll window up 5 lines
 CALL SCROLLTEXTWINDOW( INT2(5 ))
 END
```

See Also

CLEARSCREEN
GETTEXTPOSITION
GETTEXTWINDOW
GRSTATUS
OUTTEXT
SETTEXTPOSITION
SETTEXTWINDOW
WRAPON

SCWRQQ

Portability Subroutine: *Returns the floating-point processor control word.*

Module

USE IFPORT

Syntax

`CALL SCWRQQ (control)`

control (Output) INTEGER(2). Floating-point processor control word.

SCRWQQ performs the same function as the run-time subroutine GETCONTROLFPQQ, and is provided for compatibility.

Example

See the example in LCWRQQ.

See Also

GETCONTROLFPQQ
LCWRQQ

SECNDS Intrinsic Procedure

Elemental Intrinsic Function (Generic): *Provides the system time of day, or elapsed time, as a floating-point value in seconds. SECNDS can be used as an intrinsic function or as a portability routine. It is an intrinsic procedure unless you specify USE IFPORT.*

Syntax

This function must not be passed as an actual argument. It is not a pure function, so it cannot be referenced inside a FORALL construct.

```
result = SECNDS (x)
```

x (Input) Must be of type real.

Results

The result type and kind are the same as *x*.

If *x* is zero, the result value is the time in seconds since the most recent midnight (in local time).

If *x* is not zero, it is compared to *now*: the value that would be returned if *x* had been zero. If *x* is less than *now*, then the result value is *now* - *x*. If *x* is more than *now*, the function assumes that *x* is the result of a call to SECONDS made the previous day and returns the value between that time and *now*.

This function cannot detect a delay of more than one day between calls. For timing intervals longer than 24 hours, use DCLOCK.

This function does not account for daylight savings changes in either direction.

The value of SECNDS is accurate to 0.01 second, which is the resolution of the system clock.

The 24 bits of precision provide accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

Example

The following shows how to use SECNDS to perform elapsed-time computations:

```
C    START OF TIMED SEQUENCE
     T1 = SECNDS(0.0)

C    CODE TO BE TIMED
     ...
     DELTA = SECNDS(T1)      ! DELTA gives the elapsed time
```

See Also

DATE_AND_TIME

RTC

SYSTEM_CLOCK

TIME intrinsic procedure

SECNDS portability routine

DCLOCK

SECNDS Portability Routine

Portability Function: *Returns the number of seconds that have elapsed since midnight, less the value of its argument. SECNDS can be used as a portability function or as an intrinsic procedure. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

`result = SECNDS (time)`

time	(Input) REAL(4). Number of seconds, precise to a hundredth of a second (0.01), to be subtracted.

Results

The result type is REAL(4). The result value is the number of seconds that have elapsed since midnight, minus *time*, with a precision of a hundredth of a second (0.01).

To start the timing clock, call SECNDS with 0.0, and save the result in a local variable. To get the elapsed time since the last call to SECNDS, pass the local variable to SECNDS on the next call.

Example

```
USE IFPORT
REAL(4) s
INTEGER(4) i, j
s = SECNDS(0.0)
DO I = 1, 100000
   J = J + 1
END DO
s = SECNDS(s)
PRINT *, 'It took ',s, 'seconds to run.'
```

See Also

DATE_AND_TIME

RTC

SYSTEM_CLOCK

TIME portability routine

SECNDS intrinsic procedure

SECTIONS

OpenMP* Fortran Compiler Directive: *Specifies that the enclosed SECTION directives define blocks of code to be divided among threads in a team. Each section is executed once by a thread in the team.*

Syntax

```
!$OMP SECTIONS [clause[[,] clause] ... ]
[!$OMP SECTION]
   block
[!$OMP SECTION
   block]...
!$OMP END SECTIONS[NOWAIT]
```

clause	Is one of the following: • FIRSTPRIVATE (list) • LASTPRIVATE ([CONDITIONAL:] list) • PRIVATE (list) • REDUCTION (reduction-identifier : list)
block	Is a structured block (section) of statements or constructs. Any constituent section must also be a structured block. You cannot branch into or out of the block.

The binding thread set for a SECTIONS construct is the current team. A SECTIONS region binds to the innermost enclosing parallel region.

Each section of code is preceded by a SECTION directive, although the directive is optional for the first section. The SECTION directives must appear within the lexical extent of the SECTIONS and END SECTIONS directive pair.

The last section ends at the END SECTIONS directive. Threads that complete execution of their SECTIONs encounter an implied barrier at the END SECTIONS directive unless NOWAIT is specified.

SECTIONS directives must be encountered by all threads in a team or by none at all.

Example

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
      CALL XAXIS
!$OMP SECTION
      CALL YAXIS
!$OMP SECTION
      CALL ZAXIS
!$OMP END SECTIONS
!$OMP END PARALLEL
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

SEED

Portability Subroutine: *Changes the starting point of the pseudorandom number generator.*

Module

USE IFPORT

Syntax

`CALL SEED (iseed)`

iseed (Input) INTEGER(4). Starting point for RANDOM.

SEED uses *iseed* to establish the starting point of the pseudorandom number generator. A given seed always produces the same sequence of values from RANDOM.

If SEED is not called before the first call to RANDOM, RANDOM always begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, pass the constant RND$TIMESEED (defined in `IFPORT.F90`) to the SEED routine before the first call to RANDOM.

This routine is not thread-safe.

Example

```
USE IFPORT
REAL myrand
CALL SEED(7531)
CALL RANDOM(myrand)
```

See Also

RANDOM
RANDOM_SEED
RANDOM_NUMBER

SELECT CASE and END SELECT

Statement: *Transfers program control to a selected block of statements according to the value of a controlling expression.*

Example

```
CHARACTER*1 cmdchar
. . .
Files: SELECT CASE (cmdchar)
  CASE ('0')
    WRITE (*, *) "Must retrieve one to nine files"
  CASE ('1':'9')
    CALL RetrieveNumFiles (cmdchar)
  CASE ('A', 'a')
    CALL AddEntry
  CASE ('D', 'd')
    CALL DeleteEntry
```

```
  CASE ('H', 'h')
    CALL Help
  CASE DEFAULT
    WRITE (*, *) "Command not recognized; please re-enter"
END SELECT Files
```

See Also

See CASE.

SELECT RANK

Statement: *Marks the beginning of a SELECT RANK construct. The construct selects for execution at most one of its constituent blocks. The selection is based on the rank of an assumed-rank variable.*

Syntax

```
[name:] SELECT RANK ([ assoc-name => ] selector)
  [rank-case-stmt
    block]...
END SELECT [name]
```

name	(Optional) Is the name of the SELECT RANK construct.
assoc-name	(Optional) Is an identifier that becomes associated with the *selector*. It becomes the associating entity. The identifier name must be unique within the construct. If unspecified, the associate name for the construct is *selector*.
selector	Is the name of an assumed-rank array.
rank-case-stmt	(Optional) Is one of the following: • RANK (*scalar-int-const-expr*) [*name*] • RANK (*) [*name*] • RANK DEFAULT [*name*] The *scalar-int-const-expr* must be non-negative. The same rank value can only be specified in one *rank-case-stmt*. RANK DEFAULT and RANK (*) can be specified only once in the construct. RANK (*) cannot be specified if *selector* has the ALLOCATABLE or POINTER attribute.
block	(Optional) Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified at the beginning of a SELECT RANK statement, the same name must appear in the corresponding END SELECT statement. If a *rank-case-stmt* specifies a construct name, the corresponding SELECT RANK statement must specify the same name. The same construct name must not be used for different named constructs in the same scoping unit. If no name is specified at the beginning of a SELECT RANK statement, you cannot specify one following the END SELECT statement or any corresponding *rank-case-stmt*.

If no other *rank-case-stmt* of the construct matches the *selector*, a RANK DEFAULT statement, if present, identifies the block of code to be executed.

A SELECT RANK construct selects at most one *block* to be executed. During execution of that block, the associate name identifies an entity that is associated with the selector.

The following steps determine which block is selected for execution:

1. If the *selector* is argument associated with an assumed-size array, a RANK (*) statement identifies the block to be executed.
2. If the *selector* is *not* argument associated with an assumed-size array, a RANK (*scalar-int-const-exp*) statement identifies the block to be executed if the *selector* has that rank.
3. Otherwise, if there is a RANK DEFAULT statement, the block following that statement is executed.

A branch is allowed from within a block of a SELECT RANK construct to the END SELECT statement. Branches to the END SELECT from outside the construct are not allowed.

The associating entity in the block following a RANK DEFAULT statement is assumed rank and has the same attributes as the *selector*; it can used only in contexts that an assumed-rank entity can be used. The associating entity in the block following a RANK (*) statement is an assumed-size one dimensional array and lower bound 1, as if declared DIMENSION(1:*).

The associating entity in the block following a RANK (*scalar-int-const-exp*) statement has the rank specified. The lower bound is the lower bound of the *selector* for each corresponding dimension, and the upper bound is the upper bound of the *selector* for each corresponding dimension of the *selector*. The associating entity in a RANK(*) or a RANK (*scalar-int-const-exp*) block is a variable and may appear in variable definition contexts.

If the *selector* has the POINTER, TARGET or ALLOCATABLE attribute, the associating entity has the same attributes.

Example

The following example shows a SELECT RANK construct that initializes scalars, rank 1, and rank 2 arrays to zero. If the dummy argument rank is greater than 2, an error message is printed.

```
SUBROUTINE INITIALIZE (ARG)
  REAL :: ARG(..)
  SELECT RANK (ARG)
    RANK (0)   ! Scalar
      ARG = 0.0
    RANK (1)
      ARG(:) = 0.0
    RANK (2)
      ARG(:, :) = 0.0
    RANK DEFAULT
      PRINT *, "Subroutine initialize called with unexpected rank argument"
  END SELECT
  RETURN
END SUBROUTINE
```

The following example shows how to use a select rank to initialize assumed-size arrays of any rank to zero:

```
SUBROUTINE INITIALIZE (ARG, SIZE)
  REAL,CONTIGUOUS :: ARG(..)
  INTEGER         :: SIZE, I
  SELECT RANK (ARG)
    RANK (0)   ! Special case the scalar case
      ARG = 0.0
    RANK (*)
      DO I = 1, SIZE
        ARG(I) = 0.0
      END DO
  END SELECT
  RETURN
END SUBROUTINE
```

See Also

Construct Association

SELECT TYPE

Statement: *Marks the beginning of a SELECT TYPE construct. The construct selects for execution at most one of its constituent blocks. The selection is based on the dynamic type of a specified expression.*

Syntax

```
[name:] SELECT TYPE ([ assoc-name => ] selector)
  [type-guard-stmt
    block]...
END SELECT [name]
```

name	(Optional) Is the name of the SELECT TYPE construct.
assoc-name	(Optional) Is an identifier that becomes associated with the *selector*. It becomes the associating entity. The identifier name must be unique within the construct. If selector is not a named variable, *assoc-name* => must appear.
selector	Is an expression or variable. It must be polymorphic. It is evaluated when the SELECT TYPE statement is executed.
type-guard-stmt	(Optional) Is one of the following: • TYPE IS (type) [*name*] • CLASS IS (type) [*name*] • CLASS DEFAULT [*name*] CLASS DEFAULT can be specified only once in the construct. The same type and kind parameter values can only be specified in one TYPE IS statement and one CLASS IS statement.
type	Is an intrinsic type specifier or a derived-type specifier. It must specify that each length type parameter is assumed. It cannot be a sequence derived type or a type with the BIND attribute.
block	(Optional) Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified at the beginning of a SELECT TYPE statement, the same name must appear in the corresponding END SELECT statement. The same construct name must not be used for different named constructs in the same scoping unit. If no name is specified at the beginning of a SELECT TYPE statement, you cannot specify one following the END SELECT statement or any *type-guard-stmt*. If a *type-guard-stmt* specifies a construct name, it must match the construct name on the corresponding SELECT TYPE statement.

A branch to the END SELECT statement is allowed from within the SELECT TYPE construct, but it is not allowed from anywhere outside the construct.

Execution of a SELECT TYPE construct whose selector is not a variable causes the selector expression to be evaluated.

A SELECT TYPE construct selects at most one block to be executed. During execution of that block, the associate name identifies an entity that is associated with the selector. The entity associated with the associate name has the declared type and type parameters of the selector. The entity is polymorphic only if the selector is polymorphic.

The following steps determine which block is selected for execution:

1. If a TYPE IS statement matches the selector, the block following that statement is executed. A TYPE IS statement matches the selector if the dynamic type and type parameter values of the selector are the same as those specified by the statement.
2. Otherwise, if exactly one CLASS IS statement matches the selector, the block following that statement is executed. A CLASS IS statement matches the selector if the dynamic type of the selector is an extension of the type specified by the statement and the kind type parameters specified by the statement are the same as the corresponding type parameters of the dynamic type of the selector.
3. Otherwise, if several CLASS IS statements match the selector, one of these statements must specify a type that is an extension of all the types specified in the others. In this case, the block following that statement is executed.
4. Otherwise, if there is a CLASS DEFAULT statement, the block following that statement is executed.

Within the block following a TYPE IS statement, the associating entity is not polymorphic, it has the type named in the TYPE IS statement, and has the type parameters of the selector.

Within the block following a CLASS IS statement, the associating entity is polymorphic and has the declared type named in the CLASS IS statement. The type parameters of the associating entity are the corresponding type parameters of the selector.

Within the block following a CLASS DEFAULT statement, the associating entity is polymorphic and has the same declared type as the selector. The type parameters of the associating entity are those of the declared type of the selector.

If the declared type of the selector is M, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (M).

A *type-guard-stmt* cannot be a branch target statement. You can branch to an END SELECT statement only from within its SELECT TYPE construct.

Example

The following example shows a SELECT TYPE construct:

```
TYPE POINT
   REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: POINT_3D
   REAL :: Z
END TYPE POINT_3D
TYPE, EXTENDS(POINT) :: COLOR_POINT
   INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3D
TYPE(COLOR_POINT), TARGET :: CP
CLASS(POINT), POINTER :: P_OR_CP
P_OR_CP=> CP
SELECT TYPE ( AN => P_OR_CP )
CLASS IS ( POINT )
                  ! "CLASS ( POINT ) :: AN" is implied here
   PRINT *, AN%X, AN%Y     ! This block gets executed
TYPE IS ( POINT_3D )
```

```
                  ! "TYPE ( POINT_3D ) :: AN" is implied here
   PRINT *, AN%X, AN%Y, AN%Z
END SELECT
```

The following example uses declarations from the above example, but it omits the associate name AN:

```
P_OR_CP => P3D
SELECT TYPE ( P_OR_CP )
CLASS IS ( POINT )
                  ! "CLASS ( POINT ) :: P_OR_CP" is implied here
   PRINT *, P_OR_CP%X, P_OR_CP%Y
TYPE IS ( POINT_3D )
                  ! "TYPE ( POINT_3D ) :: P_OR_CP" is implied here
   PRINT *, P_OR_CP%X, P_OR_CP%Y, P_OR_CP%Z    ! This block gets executed
END SELECT
```

See Also

Construct Association
Additional Attributes Of Associate Names

SELECTED_CHAR_KIND

Transformational Intrinsic Function (Generic):
Returns the value of the kind type parameter of the character set named by the argument.

Syntax

`result = SELECTED_CHAR_KIND(name)`

name	(Input) Must be scalar and of type default character. Its value must be 'DEFAULT' or 'ASCII'.

Results

The result is a scalar of type default integer.

The result is a scalar of type default integer. The result value is 1 if NAME has the value 'DEFAULT' or 'ASCII'; otherwise, the result value is -1.

SELECTED_INT_KIND

Transformational Intrinsic Function (Generic):
Returns the value of the kind parameter of an integer data type.

Syntax

`result = SELECTED_INT_KIND (r)`

r	(Input) Must be scalar and of type integer.

Results

The result is a scalar of type default integer. The result has a value equal to the value of the kind parameter of the integer data type that represents all values *n* in the range of values *n* with $-10^r < n < 10^r$.

If no such kind type parameter is available on the processor, the result is -1. If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range. For more information, see Model for Integer Data.

Example

SELECTED_INT_KIND (6) = 4

The following shows another example:

```
i = SELECTED_INT_KIND(8)  ! returns 4
i = SELECTED_INT_KIND(3)  ! returns 2
i = SELECTED_INT_KIND(10) ! returns 8
i = SELECTED_INT_KIND(20) ! returns -1 because 10**20
                          !   is bigger than 2**63
```

See Also

SELECTED_REAL_KIND

SELECTED_REAL_KIND

Transformational Intrinsic Function (Generic):
Returns the value of the kind parameter of a real data type.

Syntax

`result = SELECTED_REAL_KIND ([p] [,r] [,radix])`

`p`	(Input; optional) Must be scalar and of type integer.
`r`	(Input; optional) Must be scalar and of type integer.
`radix`	(Input; optional) Must be scalar and of type integer.

At least argument `p` or `r` must be present.

Results

If `p` or `r` is absent, the result is as if the argument was present with the value zero. If `radix` is absent, there is no requirement on the radix of the selected kind.

The result is a scalar of type default integer. If both arguments are absent, the result is zero. Otherwise, the result has a value equal to a value of the kind parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least `p` digits, a decimal exponent range, as returned by the function RANGE, of at least `r`, and a radix, as returned by the function RADIX, of `radix`.

If no such kind type parameter is available on the processor, the result is as follows:

```
-1 if the precision is not available but the range is available
-2 if the exponent range is not available but the precision is available
-3 if neither is available
-4 if real types for the precision and the range are available separately but not together
-5 if no real type of the specified radix is available
```

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision. Intel® Fortran currently does not return -4 for any combination of `p` and `r`, and supports only a radix of 2. For more information, see Model for Real Data.

Example

SELECTED_REAL_KIND (6, 70) = 8

The following shows another example:

```
i = SELECTED_REAL_KIND(r=200, radix=2)  ! returns 8
i = SELECTED_REAL_KIND(13)     ! returns 8
i = SELECTED_REAL_KIND (100, 200)  ! returns -1
```

```
i = SELECTED_REAL_KIND (13, 5000)  ! returns -2
i = SELECTED_REAL_KIND (100, 5000) ! returns -3
i = SELECTED_REAL_KIND (13, radix=42)  ! returns -5
```

The following example gives a compile-time error:

```
i = SELECTED_REAL_KIND ()
```

See Also

SELECTED_INT_KIND
IEEE_SELECTED_REAL_KIND

SEQUENCE

Statement: *Preserves the storage order of a derived-type definition.*

Syntax

```
SEQUENCE
```

Description

The SEQUENCE statement allows derived types to be used in common blocks and to be equivalenced.

The SEQUENCE statement appears only as part of derived-type definitions. It causes the components of the derived type to be stored in the same sequence they are listed in the type definition. If you do not specify SEQUENCE, the physical storage order is not necessarily the same as the order of components in the type definition.

If a derived type is a sequence derived type, then any other derived type that includes it must also be a sequence type.

Example

```
!DIR$ PACK:1
TYPE NUM1_SEQ
  SEQUENCE
  INTEGER(2)::int_val
  REAL(4)::real_val
  LOGICAL(2)::log_val
END TYPE NUM1_SEQ
TYPE num2_seq
   SEQUENCE
   logical(2)::log_val
   integer(2)::int_val
   real(4)::real_val
end type num2_seq
type (num1_seq) num1
type (num2_seq) num2
character*8 t, t1
equivalence (num1,t)
equivalence (num2,t1)
num1%int_val=2
num1%real_val=3.5
num1%log_val=.TRUE.
t1(1:2)=t(7:8)
t1(3:4)=t(1:2)
t1(5:8)=t(3:6)
print *, num2%int_val, num2%real_val, num2%log_val
end
```

See Also

Derived Data Types
Data Types, Constants, and Variables

SETACTIVEQQ (W*S)

QuickWin Function: *Makes a child window active, but does not give it focus.*

Module

USE IFQWIN

Syntax

```
result = SETACTIVEQQ (unit)
```

unit	(Input) INTEGER(4). Unit number of the child window to be made active.

Results

The result type is INTEGER(4). The result is 1 if successful; otherwise, 0.

When a window is made active, it receives graphics output (from ARC, LINETO and OUTGTEXT, for example) but is not brought to the foreground and does not have the focus. If a window needs to be brought to the foreground, it must be given the focus. A window is given focus with FOCUSQQ, by clicking it with the mouse, or by performing I/O other than graphics on it, unless the window was opened with IOFOCUS='.FALSE.'. By default, IOFOCUS='.TRUE.', except for child windows opened as unit '*'.

The window that has the focus is always on top, and all other windows have their title bars grayed out. A window can have the focus and yet not be active and not have graphics output directed to it. Graphical output is independent of focus.

If IOFOCUS='.TRUE.', the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

See Also

GETACTIVEQQ
FOCUSQQ
INQFOCUSQQ

SETBKCOLOR (W*S)

Graphics Function: *Sets the current background color index for both text and graphics.*

Module

USE IFQWIN

Syntax

```
result = SETBKCOLOR (color)
```

color	(Input) INTEGER(4). Color index to set the background color to.

Results

The result type is INTEGER(4). The result is the previous background color index.

SETBKCOLOR changes the background color index for both text and graphics. The color index of text over the background color is set with SETTEXTCOLOR. The color index of graphics over the background color (used by drawing functions such as FLOODFILL and ELLIPSE) is set with SETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETBKCOLORRGB, SETCOLORRGB, and SETTEXTCOLORRGB.

Changing the background color index does not change the screen immediately. The change becomes effective when CLEARSCREEN is executed or when doing text input or output, such as with READ, WRITE, or OUTTEXT. The graphics output function OUTGTEXT does not affect the color of the background.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are GETBKCOLOR and SETBKCOLOR. The default background color index is 0, which is associated with black unless the user remaps the palette with REMAPPALETTERGB.

NOTE
The SETBKCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the SetBkColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$SetBkColor.

Example

```
USE IFQWIN
INTEGER(4) i
i = SETBKCOLOR(14)
```

See Also

SETBKCOLORRGB
GETBKCOLOR
REMAPALLPALETTERGB, REMAPPALETTERGB
SETCOLOR
SETTEXTCOLOR

SETBKCOLORRGB (W*S)

Graphics Function: *Sets the current background color to the given Red-Green-Blue (RGB) value.*

Module

USE IFQWIN

Syntax

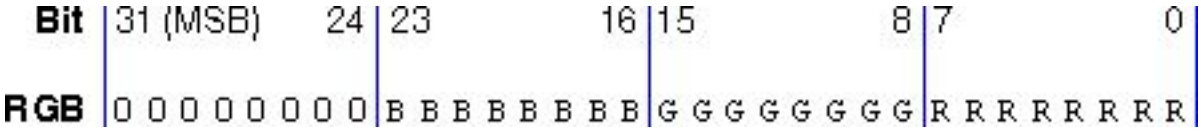
`result = SETBKCOLORRGB (color)`

color	(Input) INTEGER(4). RGB color value to set the background color to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous background RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETBKCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

The default background color is value 0, which is black. Changing the background color value does not change the screen immediately, but becomes effective when CLEARSCREEN is executed or when doing text input or output such as READ, WRITE, or OUTTEXT. The graphics output function OUTGTEXT does not affect the color of the background.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. The RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT) is set with SETTEXTCOLORRGB. The RGB color value of graphics over the background color (used by graphics functions such as ARC, OUTGTEXT, and FLOODFILLRGB) is set with SETCOLORRGB.

SETBKCOLORRGB (and the other RGB color selection functions SETCOLORRGB, and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(4) oldcolor
INTEGER(2) status, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
oldcolor = SETBKCOLORRGB(Z'FF0000') !blue
oldcolor = SETCOLORRGB(Z'FF') ! red
CALL CLEARSCREEN ($GCLEARSCREEN)
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

See Also

GETBKCOLORRGB
SETCOLORRGB
SETTEXTCOLORRGB
SETPIXELRGB
SETPIXELSRGB
SETBKCOLOR

SETCLIPRGN (W*S)

Graphics Subroutine: *Limits graphics output to part of the screen.*

Module

USE IFQWIN

Syntax

```
CALL SETCLIPRGN (x1,y1,x2,y2)
```

x1, y1	(Input) INTEGER(2). Physical coordinates for upper-left corner of clipping region.
x2, y2	(Input) INTEGER(2). Physical coordinates for lower-right corner of clipping region.

The SETCLIPRGN function limits the display of subsequent graphics output and font text output to that which fits within a designated area of the screen (the "clipping region"). The physical coordinates (x1, *y1*) and (*x2*, *y2*) are the upper-left and lower-right corners of the rectangle that defines the clipping region. The SETCLIPRGN function does not change the viewport-coordinate system; it merely masks graphics output to the screen.

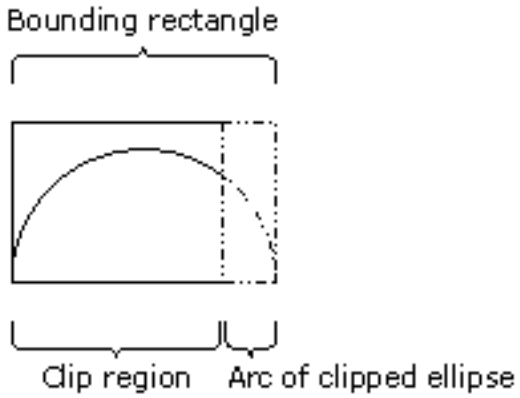
SETCLIPRGN affects graphics and font text output only, such as OUTGTEXT. To mask the screen for text output using OUTTEXT, use SETTEXTWINDOW.

Example

This program draws an ellipse lying partly within a clipping region, as shown below.

```
! Build as QuickWin or Standard Graphics ap.
USE IFQWIN
INTEGER(2) status, x1, y1, x2, y2
INTEGER(4) oldcolor
x1 = 10;  y1 = 50
x2 = 170; y2 = 150
! Draw full ellipse in white
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
oldcolor = SETCOLORRGB(Z'FF0000') !blue
WRITE(*,*) "Hit enter"
READ(*,*)
CALL CLEARSCREEN($GCLEARSCREEN) ! clear screen
CALL SETCLIPRGN( INT2(0), INT2(0), &
                 INT2(150), INT2(125))
! only part of ellipse inside clip region drawn now
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

The following shows the output of this program.



See Also

GETPHYSCOORD
GRSTATUS

SETTEXTWINDOW
SETVIEWORG
SETVIEWPORT
SETWINDOW

SETCOLOR (W*S)

Graphics Function: *Sets the current graphics color index.*

Module

USE IFQWIN

Syntax

```
result = SETCOLOR (color)
```

color (Input) INTEGER(2). Color index to set the current graphics color to.

Results

The result type is INTEGER(2). The result is the previous color index if successful; otherwise, -1.

The SETCOLOR function sets the current graphics color index, which is used by graphics functions such as ELLIPSE. The background color index is set with SETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB.

Example

```
USE IFQWIN
INTEGER(2) color, oldcolor
LOGICAL status
TYPE (windowconfig) wc

status = GETWINDOWCONFIG(wc)
color = wc%numcolors - 1
oldcolor = SETCOLOR(color)
END
```

See Also

SETCOLORRGB
GETCOLOR
REMAPPALETTERGB
SETBKCOLOR
SETTEXTCOLOR
SETPIXEL
SETPIXELS

SETCOLORRGB (W*S)

Graphics Function: *Sets the current graphics color to the specified Red-Green-Blue (RGB) value.*

Module

USE IFQWIN

Syntax

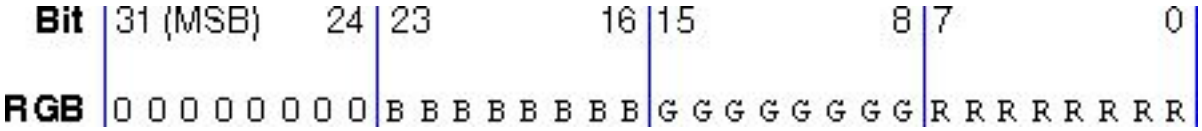
```
result = SETCOLORRGB (color)
```

color — (Input) INTEGER(4). RGB color value to set the current graphics color to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

SETCOLORRGB sets the RGB color value of graphics over the background color, used by the following graphics functions: ARC, ELLIPSE, FLOODFILL, LINETO, OUTGTEXT, PIE, POLYGON, RECTANGLE, and SETPIXEL. SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. SETTEXTCOLORRGB sets the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT).

SETCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB, and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) numfonts
INTEGER(4) oldcolor
TYPE (xycoord) xy
numfonts = INITIALIZEFONTS( )
oldcolor = SETCOLORRGB(Z'0000FF')   ! red
oldcolor = SETBKCOLORRGB(Z'00FF00') ! green
CALL MOVETO(INT2(200), INT2(100), xy)
CALL OUTGTEXT("hello, world")
END
```

See Also

SETBKCOLORRGB
SETTEXTCOLORRGB
GETCOLORRGB
ARC
ELLIPSE
FLOODFILLRGB

SETCOLOR
LINETO
OUTGTEXT
PIE
POLYGON
RECTANGLE
REMAPPALETTERGB
SETPIXELRGB
SETPIXELSRGB

SETCONTROLFPQQ

Portability Subroutine: *Sets the value of the floating-point processor control word.*

Module

USE IFPORT

Syntax

`CALL SETCONTROLFPQQ (controlword)`

controlword (Input) INTEGER(2). Floating-point processor control word.

The floating-point control word specifies how various exception conditions are handled by the floating-point math processor, sets the floating-point precision, and specifies the floating-point rounding mechanism used.

The control word can be any of the following constants (defined in `IFPORT.F90`):

Parameter name	Hex value	Description
FPCW$MCW_IC	Z'1000'	**Infinity control mask**
FPCW$AFFINE	Z'1000'	Affine infinity
FPCW$PROJECTIVE	Z'0000'	Projective infinity
FPCW$MCW_PC	Z'0300'	**Precision control mask**
FPCW$64	Z'0300'	64-bit precision
FPCW$53	Z'0200'	53-bit precision
FPCW$24	Z'0000'	24-bit precision
FPCW$MCW_RC	Z'0C00'	**Rounding control mask**
FPCW$CHOP	Z'0C00'	Truncate
FPCW$UP	Z'0800'	Round up
FPCW$DOWN	Z'0400'	Round down
FPCW$NEAR	Z'0000'	Round to nearest
FPCW$MCW_EM	Z'003F'	**Exception mask**
FPCW$INVALID	Z'0001'	Allow invalid numbers

Parameter name	Hex value	Description
FPCW$DENORMAL	Z'0002'	Allow subnormals (very small numbers)
FPCW$SUBNORMAL	Z'0002'	Allow subnormals (very small numbers)
FPCW$ZERODIVIDE	Z'0004'	Allow divide by zero
FPCW$OVERFLOW	Z'0008'	Allow overflow
FPCW$UNDERFLOW	Z'0010'	Allow underflow
FPCW$INEXACT	Z'0020'	Allow inexact precision

An exception is disabled if its control bit is set to 1. An exception is enabled if its control bit is cleared to 0.

Setting the floating-point precision and rounding mechanism can be useful if you are reusing old code that is sensitive to the floating-point precision standard used and you want to get the same results as on the old machine.

You can use GETCONTROLFPQQ to retrieve the current control word and SETCONTROLFPQQ to change the control word. Most users do not need to change the default settings. If you need to change the control word, always use SETCONTROLFPQQ to make sure that special routines handling floating-point stack exceptions and abnormal propagation work correctly.

NOTE
The Intel® Fortran exception handler allows for software masking of invalid operations, but does not allow the math chip to mask them. If you choose to use the software masking, be aware that this can affect program performance if you compile code written for Intel Fortran with another compiler.

Example

```
USE IFPORT
INTEGER(2) status, control, controlo
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
!     Save old control word
controlo = control
!     Clear all flags
control = control .AND. Z'0000'
!     Set new control to round up
control = control .OR. FPCW$UP
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

See Also

GETCONTROLFPQQ
GETSTATUSFPQQ
LCWRQQ
SCWRQQ
CLEARSTATUSFPQQ

SETDAT

Portability Function: *Sets the system date. This function is only available on Windows* and Linux* systems.*

Module

USE IFPORT

Syntax

`result = SETDAT (iyr, imon, iday)`

iyr	(Input) INTEGER(2) or INTEGER(4). Year (*xxxx*AD).
imon	(Input) INTEGER(2) or INTEGER(4). Month (1-12).
iday	(Input) INTEGER(2) or INTEGER(4). Day of the month (1-31).

Results

The result type is LOGICAL(4). The result is .TRUE. if the system date is changed; .FALSE. if no change is made.

Actual arguments of the function SETDAT can be any valid INTEGER(2) or INTEGER(4) expression.

All arguments must be of the same integer kind, that is, all must be INTEGER(2) or all must be INTEGER(4).

If INTEGER(2) arguments are passed, you must specify USE IFPORT.

Refer to your operating system documentation for the range of permitted dates.

> **NOTE**
> On Linux systems, you must have root privileges to execute this function.

Example

```
USE IFPORT
LOGICAL(4) success
success = SETDAT(INT2(1997+1), INT2(2*3), INT2(30))
END
```

See Also

GETDAT
GETTIM
SETTIM

SETENVQQ

Portability Function: *Sets the value of an existing environment variable, or adds and sets a new environment variable.*

Module

USE IFPORT

Syntax

`result = SETENVQQ (varname=value)`

varname=value	(Input) Character*(*). String containing both the name and the value of the variable to be added or modified. Must be in the form: *varname = value*, where *varname* is the name of an environment variable and *value* is the value being assigned to it.

Results

The result is of type LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Environment variables define the environment in which a program executes. For example, the LIB environment variable defines the default search path for libraries to be linked with a program.

SETENVQQ deletes any terminating blanks in the string. Although the equal sign (=) is an illegal character within an environment value, you can use it to terminate *value* so that trailing blanks are preserved. For example, the string `PATH= `sets *value* to `''`.

You can use SETENVQQ to remove an existing variable by giving a variable name followed by an equal sign with no value. For example, LIB= removes the variable LIB from the list of environment variables. If you specify a value for a variable that already exists, its value is changed. If the variable does not exist, it is created.

SETENVQQ affects only the environment that is local to the current process. You cannot use it to modify the command-level environment. When the current process terminates, the environment reverts to the level of the parent process. In most cases, this is the operating system level. However, you can pass the environment modified by SETENVQQ to any child process created by RUNQQ. These child processes get new variables and/or values added by SETENVQQ.

SETENVQQ uses the C runtime routine _putenv and GETENVQQ uses the C runtime routine getenv. From the C documentation:

getenv and _putenv use the copy of the environment pointed to by the global variable _environ to access the environment. getenv operates only on the data structures accessible to the run-time library and not on the environment segment created for the process by the operating system.

SETENVQQ and GETENVQQ will not work properly with the Windows* APIs SetEnvironmentVariable and GetEnvironmentVariable.

Example

```
USE IFPORT
LOGICAL(4) success
success = SETENVQQ("PATH=c:\mydir\tmp")
success = &
SETENVQQ("LIB=c:\mylib\bessel.lib;c:\math\difq.lib")
END
```

See Also

GETENVQQ
RUNQQ

SETERRORMODEQQ

Portability Subroutine: *Sets the prompt mode for critical errors that by default generate system prompts.*

Module

USE IFPORT

Syntax

CALL SETERRORMODEQQ (*pmode*)

pmode	(Input) LOGICAL(4). Flag that determines whether a prompt is displayed when a critical error occurs.

Certain I/O errors cause the system to display an error prompt. For example, attempting to write to a disk drive with the drive door open generates an "Abort, Retry, Ignore" message. When the system starts up, system error prompting is enabled by default (*pmode*= .TRUE.). You can also enable system error prompts by calling SETERRORMODEQQ with *pmode* set to ERR$HARDPROMPT (defined in IFPORT.F90).

If prompt mode is turned off, critical errors that normally cause a system prompt are silent. Errors in I/O statements such as OPEN, READ, and WRITE fail immediately instead of being interrupted with prompts. This gives you more direct control over what happens when an error occurs. For example, you can use the ERR= specifier to designate an executable statement to branch to for error handling. You can also take a different action than that requested by the system prompt, such as opening a temporary file, giving a more informative error message, or exiting.

You can turn off prompt mode by setting *pmode* to .FALSE. or to the constant ERR$HARDFAIL (defined in IFPORT.F90).

Note that SETERRORMODEQQ affects only errors that generate a system prompt. It does not affect other I/O errors, such as writing to a nonexistent file or attempting to open a nonexistent file with STATUS='OLD'.

Example

```
!PROGRAM 1
!  DRIVE B door open
OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
!  Generates a system prompt error here and waits for the user
!  to respond to the prompt before continuing
100   WRITE(*,*) ' Continuing'
END

! PROGRAM 2
!  DRIVE B door open
 USE IFPORT
 CALL SETERRORMODEQQ(.FALSE.)
 OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
!  Causes the statement at label 100 to execute
!  without system prompt
100   WRITE(*,*) ' Drive B: not available, opening      &
                  &alternative drive.'
 OPEN (10, FILE = 'C:\NOFILE.DAT')
 END
```

SETEXITQQ

QuickWin Function: *Sets a QuickWin application's exit behavior.*

Module

USE IFQWIN

Syntax

```
result = SETEXITQQ (exitmode)
```

exitmode (Input) INTEGER(4). Determines the program exit behavior. The following exit parameters are defined in `IFQWIN.F90`:

- QWIN$EXITPROMPT - Displays the following message box:

  `"Program exited with exit status X. Exit Window?"`

  where X is the exit status from the program.

  If Yes is entered, the application closes the window and terminates. If No is entered, the dialog box disappears and you can manipulate the windows as usual. You must then close the window manually.
- QWIN$EXITNOPERSIST - Terminates the application without displaying a message box.
- QWIN$EXITPERSIST - Leaves the application open without displaying a message box.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The default for both QuickWin and Standard Graphics applications is QWIN$EXITPROMPT.

Example

```
! Build as QuickWin Ap
USE IFQWIN
INTEGER(4) exmode, result

WRITE(*,'(1X,A,/)') 'Please enter the exit mode 1, 2   &
                     or 3 '
READ(*,*) exmode
SELECT CASE (exmode)
  CASE (1)
    result = SETEXITQQ(QWIN$EXITPROMPT)
  CASE (2)
    result = SETEXITQQ(QWIN$EXITNOPERSIST)
  CASE (3)
    result = SETEXITQQ(QWIN$EXITPERSIST)
  CASE DEFAULT
    WRITE(*,*) 'Invalid option - checking for bad     &
                return'
    IF(SETEXITQQ( exmode ) .NE. -1) THEN
      WRITE(*,*) 'Error not returned'
    ELSE
      WRITE(*,*) 'Error code returned'
    ENDIF
END SELECT
END
```

See Also

GETEXITQQ

SET_EXPONENT

Elemental Intrinsic Function (Generic): *Returns the value of the exponent part (of the model for the argument) set to a specified value.*

Syntax

```
result = SET_EXPONENT (x,i)
```

x	(Input) Must be of type real.
i	(Input) Must be of type integer.

Results

The result type and kind are the same as *x*. The result has the value *x* x b $^{i-e}$. Parameters *b* and *e* are defined in Model for Real Data. If *x* has the value zero, the result is zero.

Example

If 3.0 is a REAL(4) value, SET_EXPONENT (3.0, 1) has the value 1.5.

See Also

EXPONENT

Data Representation Models

SETFILEACCESSQQ

Portability Function: *Sets the file access mode for a specified file.*

Module

USE IFPORT

Syntax

```
result = SETFILEACCESSQQ (filename,access)
```

filename	(Input) Character*(*). Name of a file to set access for.
access	(Input) INTEGER(4). Constant that sets the access. Can be any combination of the following flags, combined by an inclusive OR (such asIOR or OR): • FILE$ARCHIVE - Marked as having been copied to a backup device. • FILE$HIDDEN - Hidden. The file does not appear in the directory list that you can request from the command console. • FILE$NORMAL - No special attributes (default). • FILE$READONLY - Write-protected. You can read the file, but you cannot make changes to it. • FILE$SYSTEM - Used by the operating system. The flags are defined in module `IFPORT.F90`.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

To set the access value for a file, add the constants representing the appropriate access.

Example

```
USE IFPORT
INTEGER(4) permit
LOGICAL(4) result

permit = 0   ! clear permit
permit = IOR(FILE$READONLY, FILE$HIDDEN)
result = SETFILEACCESSQQ ('formula.f90', permit)
END
```

See Also

GETFILEINFOQQ

SETFILETIMEQQ

Portability Function: *Sets the modification time for a specified file.*

Module

USE IFPORT

Syntax

`result = SETFILETIMEQQ (`*`filename`*`,`*`timedate`*`)`

filename	(Input) Character*(*). Name of a file.
timedate	(Input) INTEGER(4). Time and date information, as packed by PACKTIMEQQ.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The modification time is the time the file was last modified and is useful for keeping track of different versions of the file. The process that calls SETFILETIMEQQ must have write access to the file; otherwise, the time cannot be changed. If you set *timedate* to FILE$CURTIME (defined in `IFPORT.F90`), SETFILETIMEQQ sets the modification time to the current system time.

If the function fails, call GETLASTERRORQQ to determine the reason. It can be one of the following:

- ERR$ACCES - Permission denied. The file's (or directory's) permission setting does not allow the specified access.
- ERR$INVAL - Invalid argument; the *timedate* argument is invalid.
- ERR$MFILE - Too many open files (the file must be opened to change its modification time).
- ERR$NOENT - File or path not found.
- ERR$NOMEM - Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

Example

```
USE IFPORT
INTEGER(2) day, month, year
INTEGER(2) hour, minute, second, hund
INTEGER(4) timedate
LOGICAL(4) result

CALL GETDAT(year, month, day)
```

```
CALL GETTIM(hour, minute, second, hund)
CALL PACKTIMEQQ (timedate, year, month, day,    &
                 hour, minute, second)
result = SETFILETIMEQQ('myfile.dat', timedate)
END
```

See Also

PACKTIMEQQ
UNPACKTIMEQQ
GETLASTERRORQQ

SETFILLMASK (W*S)

Graphics Subroutine: *Sets the current fill mask to a new pattern.*

Module

USE IFQWIN

Syntax

`CALL SETFILLMASK (mask)`

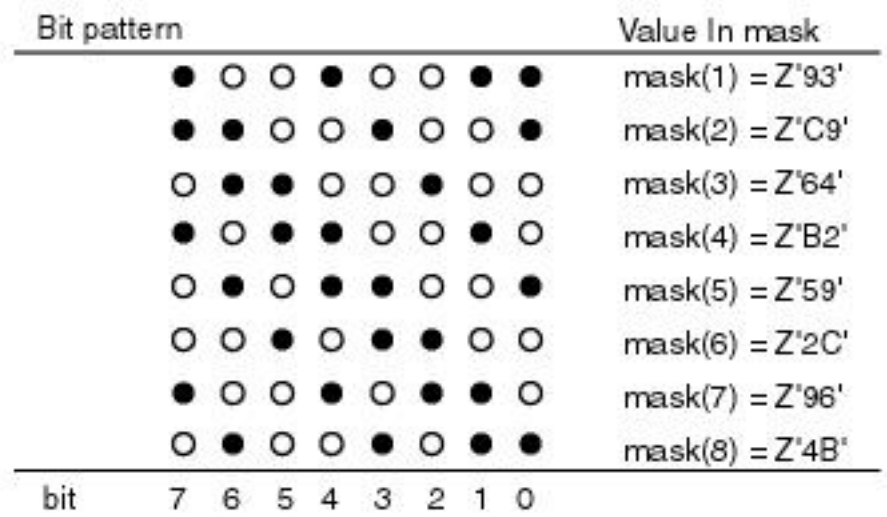
mask (Input) INTEGER(1). One-dimensional array of length 8.

There are 8 bytes in *mask*, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of *mask* becomes the top 8 bits of the pattern, and the eighth element (byte) of *mask* becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of zero are set to the current background color. The current graphics color is set with SETCOLORRGB or SETCOLOR. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with SETFILLMASK), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (FLOODFILLRGB, PIE, ELLIPSE, POLYGON, and RECTANGLE).

To change the current fill mask, determine the array of bytes that corresponds to the desired bit pattern and set the pattern with SETFILLMASK, as in the following example.

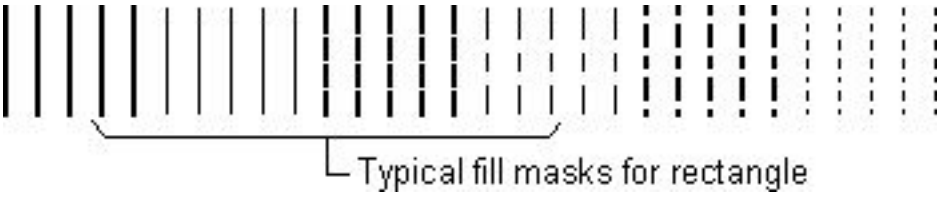


Example

This program draws six rectangles, each with a different fill mask, as shown below.

```
! Build as QuickWin or Standard Graphics Ap.
USE IFQWIN

INTEGER(1), TARGET :: style1(8) &
            /Z'18',Z'18',Z'18',Z'18',Z'18',Z'18',Z'18',Z'18'/
INTEGER(1), TARGET :: style2(8) &
            /Z'08',Z'08',Z'08',Z'08',Z'08',Z'08',Z'08',Z'08'/
INTEGER(1), TARGET :: style3(8) &
            /Z'18',Z'00',Z'18',Z'18',Z'18',Z'00',Z'18',Z'18'/
INTEGER(1), TARGET :: style4(8) &
            /Z'00',Z'08',Z'00',Z'08',Z'08',Z'08',Z'08',Z'08'/
INTEGER(1), TARGET :: style5(8) &
            /Z'18',Z'18',Z'00',Z'18',Z'18',Z'00',Z'18',Z'18'/
INTEGER(1), TARGET :: style6(8) &
            /Z'08',Z'00',Z'08',Z'00',Z'08',Z'00',Z'08',Z'00'/
INTEGER(1) oldstyle(8) ! Placeholder for old style
INTEGER loop
INTEGER(1), POINTER :: ptr(:)

CALL GETFILLMASK( oldstyle )
! Make 6 rectangles, each with a different fill
DO loop = 1, 6
  SELECT CASE (loop)
    CASE (1)
      ptr => style1
    CASE (2)
      ptr => style2
    CASE (3)
      ptr => style3
    CASE (4)
      ptr => style4
    CASE (5)
      ptr => style5
    CASE (6)
      ptr => style6
  END SELECT
  CALL SETFILLMASK( ptr)
  status = RECTANGLE($GFILLINTERIOR,INT2(loop*40+5), &
              INT2(90),INT2((loop+1)*40), INT2(110))
END DO

CALL SETFILLMASK( oldstyle )  ! Restore old style
READ (*,*)                    ! Wait for ENTER to be
                              ! pressed
END
```

The following shows the output of this program.



See Also

ELLIPSE
FLOODFILLRGB
GETFILLMASK
PIE
POLYGON
RECTANGLE

SETFONT (W*S)

Graphics Function: *Finds a single font that matches a specified set of characteristics and makes it the current font used by the OUTGTEXT function.*

Module

USE IFQWIN

Syntax

`result = SETFONT (`*`options`*`)`

options	(Input) Character*(*). String describing font characteristics (see below for details).

Results

The result type is INTEGER(2). The result is the index number (*x* as used in the `n`*x* option) of the font if successful; otherwise, -1.

The SETFONT function searches the list of available fonts for a font matching the characteristics specified in *options*. If a font matching the characteristics is found, it becomes the current font. The current font is used in all subsequent calls to the OUTGTEXT function. There can be only one current font.

The *options* argument consists of letter codes, as follows, that describe the desired font. The argument is neither case sensitive nor position sensitive.

`t'` *fontname*`'`	Name of the desired typeface. It can be any installed font.
`h`*y*	Character height, where *y* is the number of pixels.
`w`*x*	Select character width, where *x* is the number of pixels.
`f`	Select only a fixed-space font (do not use with the `p` characteristic).
`p`	Select only a proportional-space font (do not use with the `f` characteristic).
`v`	Select only a vector-mapped font (do not use with the `r` characteristic). Roman, Modern, and Script are examples of vector-mapped fonts, also called plotter fonts. True Type fonts (for example, Arial, Symbol, and Times New Roman) are not vector-mapped.

r	Select only a raster-mapped (bitmapped) font (do not use with the v characteristic). Courier, Helvetica, and Palatino are examples of raster-mapped fonts, also called screen fonts. True Type fonts are not raster-mapped.
e	Select the bold text format. This parameter is ignored if the font does not allow the bold format.
u	Select the underline text format. This parameter is ignored if the font does not allow underlining.
i	Select the italic text format. This parameter is ignored if the font does not allow italics.
b	Select the font that best fits the other parameters specified.
n*x*	Select font number *x*, where *x* is less than or equal to the value returned by the INTIALIZEFONTS function.

You can specify as many options as you want, except with n*x*, which should be used alone. If you specify options that are mutually exclusive (such as the pairs f/p or r/v), the SETFONT function ignores them. There is no error detection for incompatible parameters used with n*x*.

If the b option is specified and at least one font is initialized, SETFONT sets a font and returns 0 to indicate success.

In selecting a font, the SETFONT routine uses the following criteria, rated from highest precedence to lowest:

1. Pixel height
2. Typeface
3. Pixel width
4. Fixed or proportional font

You can also specify a pixel width and height for fonts. If you choose a nonexistent value for either and specify the b option, SETFONT chooses the closest match.

A smaller font size has precedence over a larger size. If you request Arial 12 with best fit, and only Arial 10 and Arial 14 are available, SETFONT selects Arial 10.

If you choose a nonexistent value for pixel height and width, the SETFONT function applies a magnification factor to a vector-mapped font to obtain a suitable font size. This automatic magnification does not apply if you specify the roption (raster-mapped font), or if you request a specific typeface and do not specify the b option (best-fit).

If you specify the n*x* parameter, SETFONT ignores any other specified options and supplies only the font number corresponding to *x*.

If a height is given, but not a width, SETFONT computes the width to preserve the correct font proportions.

If a width is given, but not a height, SETFONT uses a default height, which may vary from font type to font type. This may lead to characters that appear distorted, particularly when a very wide width is specified. This behavior is the same as that of the Windows* API CreateFontIndirect. A sample program is provided below showing you how to calculate the correct height for a given width.

The font functions affect only OUTGTEXT and the current graphics position; no other Fortran Graphics Library output functions are affected by font usage.

For each window you open, you must call INITIALIZEFONTS before calling SETFONT. INITIALIZEFONTS needs to be executed after each new child window is opened in order for a subsequent SETFONT call to be successful.

Example

```
!  Build as a Graphics ap.
USE IFQWIN
INTEGER(2) fontnum, numfonts
TYPE (xycoord) pos
numfonts = INITIALIZEFONTS ( )
!  Set typeface to Arial, character height to 18,
!  character width to 10, and italic
fontnum = SETFONT ('t''Arial''h18w10i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT('Demo text')
END
```

Another example follows:

```
! The following program shows you how to compute
!  an appropriate font height for a given font width
!
!  Build as a Graphics ap.
 USE IFQWIN
 INTEGER(2) fontnum, numfonts
 TYPE (xycoord) pos
 TYPE (rccoord) rcc
 TYPE (FONTINFO) info
 CHARACTER*11 str, str1
 CHARACTER*22 str2
 real rh
 integer h, inw
 str = "t'Arial'bih"
 str1= " "
 numfonts = INITIALIZEFONTS ( )
 ! Default both height and width.  This seems to work
 ! properly.  From this setting get the ratio between
 ! height and width.
 fontnum = SETFONT ("t'Arial'")
 ireturn = GETFONTINFO(info)
 rh =  real(info%pixheight)/real(info%avgwidth)

 ! Now calculate the height for a width of 40
 write(*,*) 'Input desired width:'
 read(*,*) inw
 h =int(inw*rh)
 write(str1,'(I3.3)') h
 str2 = str//str1
 print *,str2
 fontnum = SETFONT (str2)
 CALL MOVETO (INT2(10), INT2(50), pos)
 CALL OUTGTEXT('ABCDEFGabcdefg12345!@#$%')
 CALL MOVETO (INT2(10), INT2(50+10+h), pos)
 CALL OUTGTEXT('123456789012345678901234')
 ireturn = GETFONTINFO(info)
```

```
 call settextposition(4,1, rcc)
 print *,  info%avgwidth, info%pixheight
END
```

See Also

GETFONTINFO
GETGTEXTEXTENT
GRSTATUS
OUTGTEXT
INITIALIZEFONTS
SETGTEXTROTATION

SETGTEXTROTATION (W*S)

Graphics Subroutine: *Sets the orientation angle of the font text output in degrees. The current orientation is used in calls to OUTGTEXT.*

Module

USE IFQWIN

Syntax

CALL SETGTEXTROTATION (*degree-tenths*)

degree-tenths	(Input) INTEGER(4). Angle of orientation, in tenths of degrees, of the font text output.

The orientation of the font text output is set in tenths of degrees. Horizontal is 0°, and angles increase counterclockwise so that 900 (90°) is straight up, 1800 (180°) is upside down and left, 2700 (270°) is straight down, and so forth. If the user specifies a value greater than 3600 (360°), the subroutine takes a value equal to:

```
 MODULO (user-specified tenths of degrees, 3600)
```

Although SETGTEXTROTATION accepts arguments in tenths of degrees, only increments of one full degree differ visually from each other on the screen.

Bitmap fonts cannot be rotated; TruType fonts should be used instead.

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) fontnum, numfonts
INTEGER(4) oldcolor, deg
TYPE (xycoord) pos
numfonts = INITIALIZEFONTS ( )
fontnum = SETFONT ('t''Arial''h18w10i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT('Straight text')
deg = -1370
CALL SETGTEXTROTATION(deg)
oldcolor = SETCOLORRGB(Z'008080')
CALL OUTGTEXT('Slanted text')
END
```

See Also

GETGTEXTROTATION

SETLINESTYLE (W*S)

Graphics Subroutine: *Sets the current line style to a new line style.*

Module

USE IFQWIN

Syntax

`CALL SETLINESTYLE (mask)`

mask	(Input) INTEGER(2). Desired Quickwin line-style mask. (See the table below.)

The mask is mapped to the style that most closely equivalences the percentage of the bits in the mask that are set. The style produces lines that cover a certain percentage of the pixels in that line.

SETLINESTYLE sets the style used in drawing a line. You can choose from the following styles:

QuickWin Mask	Internal Windows* Style	Selection Criteria	Appearance
0xFFFF	PS_SOLID	16 bits on	_____
0xEEEE	PS_DASH	11 to 15 bits on	----------
0xECEC	PS_DASHDOT	10 bits on	-.-.-.-.-.-.-.-.-.
0xECCC	PS_DASHDOTDOT	9 bits on	-..-..-..-..-..-..-
0xAAAA	PS_DOT	1 to 8 bits on	.....
0x0000	PS_NULL	0 bits on	

SETLINESTYLE affects the drawing of straight lines as in LINETO, POLYGON, and RECTANGLE, but not the drawing of curved lines as in ARC, ELLIPSE, or PIE.

The current graphics color is set with SETCOLORRGB or SETCOLOR. SETWRITEMODE affects how the line is displayed.

Example

```
! Build as a Graphics ap.
 USE IFQWIN
 INTEGER(2)     status, style
 TYPE (xycoord) xy

 style = Z'FFFF'
 CALL SETLINESTYLE(style)
 CALL MOVETO(INT2(50), INT2(50), xy )
 status = LINETO(INT2(300), INT2(300))
 END
```

See Also

GETLINESTYLE
GRSTATUS

LINETO
POLYGON
RECTANGLE
SETCOLOR
SETWRITEMODE

SETLINEWIDTHQQ (W*S)

Graphics Subroutine: *Sets the width of a solid line drawn using any of the supported graphics functions.*

Module

USE IFQWIN

Syntax

```
CALL SETLINEWIDTHQQ (x)
```

x (Input) INTEGER(4). It can be any non-negative integer.

This subroutine sets the line width in pixels using the value that is passed as the argument.

SETLINEWIDTHQQ affects the drawing of straight lines using functions such as LINETO, POLYGON, LINETOAR, LINETOAREX, RECTANGLE, and it affects the drawing of curved lines using functions such as ARC, ELLIPSE, or PIE.

NOTE
The nWidth argument in the Windows* API CreatePen() is the width used to draw the lines or borders of a closed shape. A cosmetic pen can only have a width of 1 pixel. If you specify a higher width, it is ignored. A geometric pen can have a width of 1 or more pixels, but the line can only be solid or null. This means that if you specify the style as PS_DASH, PS_DOT, PS_DASHDOT, or PS_DASHDOTDOT, but set a width higher than 1 pixel, the line is drawn as PS_SOLID.

See Also

GETLINEWIDTHQQ (W*S)

Windows* API CreatePen in the Microsoft* MSDN documentation

SETMESSAGEQQ (W*S)

QuickWin Subroutine: *Changes QuickWin status messages, state messages, and dialog box messages.*

Module

USE IFQWIN

Syntax

```
CALL SETMESSAGEQQ (msg,id)
```

msg (Input) Character*(*). Message to be displayed. Must be a regular Fortran string, not a C string. Can include multibyte characters.

id

(Input) INTEGER(4). Identifier of the message to be changed. The following table shows the messages that can be changed and their identifiers:

Id	Message
QWIN$MSG_TERM	"Program terminated with exit code"
QWIN$MSG_EXITQ	"Exit Window"
QWIN$MSG_FINISHED	"Finished"
QWIN$MSG_PAUSED	"Paused"
QWIN$MSG_RUNNING	"Running"
QWIN$MSG_FILEOPENDLG	"Text Files(*.txt), *.txt; Data Files(*.dat), *.dat; All Files(*.*), *.*;"
QWIN$MSG_BMPSAVEDLG	"Bitmap Files(*.bmp), *.bmp; All Files(*.*), *.*;"
QWIN$MSG_INPUTPEND	"Input pending in"
QWIN$MSG_PASTEINPUTPEND	"Paste input pending"
QWIN$MSG_MOUSEINPUTPEND	"Mouse input pending in"
QWIN$MSG_SELECTTEXT	"Select Text in"
QWIN$MSG_SELECTGRAPHICS	"Select Graphics in"
QWIN$MSG_PRINTABORT	"Error! Printing Aborted."
QWIN$MSG_PRINTLOAD	"Error loading printer driver"
QWIN$MSG_PRINTNODEFAULT	"No Default Printer."
QWIN$MSG_PRINTDRIVER	"No Printer Driver."
QWIN$MSG_PRINTINGERROR	"Print: Printing Error."
QWIN$MSG_PRINTING	"Printing"
QWIN$MSG_PRINTCANCEL	"Cancel"
QWIN$MSG_PRINTINPROGRESS	"Printing in progress..."
QWIN$MSG_HELPNOTAVAIL	"Help Not Available for Menu Item"
QWIN$MSG_TITLETEXT	"Graphic"

QWIN$MSG_FILEOPENDLG and QWIN$MSG_BMPSAVEDLG control the text in file choosing dialog boxes and have the following syntax:

"file description, file designation"

You can change any string produced by QuickWin by calling SETMESSAGEQQ with the appropriate *id*. This includes status messages displayed at the bottom of a QuickWin application, state messages (such as "Paused"), and dialog box messages. These messages can include multibyte characters. To change menu messages, use MODIFYMENUSTRINGQQ.

Example

```
USE IFQWIN
print*, "Hello"
CALL SETMESSAGEQQ('Changed exit text', QWIN$MSG_EXITQ)
```

See Also

MODIFYMENUSTRINGQQ

SETMOUSECURSOR (W*S)

Quickwin Function: *Sets the shape of the mouse cursor for the window in focus.*

Module

USE IFQWIN

USE IFWIN

Syntax

`oldcursor = SETMOUSECURSOR (newcursor)`

newcursor	(Input) INTEGER(4). A Windows HCURSOR value. For many predefined shapes, LoadCursor(0, shape) is a convenient way to get a legitimate value. See the list of predefined shapes below. A value of zero prevents the cursor from being displayed.

Results

The result type is INTEGER(4). This is also an HCURSOR Value. The result is the previous cursor value.

The window in focus at the time SETMOUSECURSOR is called has its cursor changed to the specified value. Once changed, the cursor retains its shape until another call to SETMOUSECURSOR.

In Standard Graphics applications, units 5 and 6 (the default screen input and output units) are always considered to be in focus.

The following predefined values for cursor shapes are available:

Predefined Value	Cursor Shape
IDC_APPSTARTING	Standard arrow and small hourglass
IDC_ARROW	Standard arrow
IDC_CROSS	Crosshair
IDC_IBEAM	Text I-beam
IDC_ICON	Obsolete value
IDC_NO	Slashed circle
IDC_SIZE	Obsolete value; use IDC_SIZEALL

Predefined Value	Cursor Shape
IDC_SIZEALL	Four-pointed arrow
IDC_SIZENESW	Double-pointed arrow pointing northeast and southwest
IDC_SIZENS	Double-pointed arrow pointing north and south
IDC_SIZENWSE	Double-pointed arrow pointing northwest and southeast
IDC_SIZEWE	Double-pointed arrow pointing west and east
IDC_UPARROW	Vertical arrow
IDC_WAIT	Hour glass

A LoadCursor must be done on these values before they can be used by SETMOUSECURSOR.

Example

```
! Build as Standard Graphics or QuickWin
    use ifqwin
    use ifwin

    integer*4  cursor, oldcursor
    write(6,*) 'The cursor will now be changed to an hour glass shape'
    write(6,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_WAIT)
    oldcursor = SetMouseCursor(cursor)
    read(5,*)

    write(6,*) 'The cursor will now be changed to a cross-hair shape'
    write(6,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_CROSS)
    oldcursor = SetMouseCursor(cursor)
    read(5,*)

    write(6,*) 'The cursor will now be turned off'
    write(6,*) 'Hit <return> to see the next change'
    oldcursor = SetMouseCursor(0)
    read(5,*)

    write(6,*) 'The cursor will now be turned on'
    write(6,*) 'Hit <return> to see the next change'
    oldcursor = SetMouseCursor(oldcursor)
    read(5,*)

  stop
  end
```

SETPIXEL, SETPIXEL_W (W*S)

Graphics Functions: *Set a pixel at a specified location to the current graphics color index.*

Module

USE IFQWIN

Syntax

```
result = SETPIXEL (x,y)
result = SETPIXEL_W (wx, wy)
```

x, y (Input) INTEGER(2). Viewport coordinates for target pixel.

wx, wy (Input) REAL(8). Window coordinates for target pixel.

Results

The result type is INTEGER(2). The result is the previous color index of the target pixel if successful; otherwise, -1 (for example, if the pixel lies outside the clipping region).

SETPIXEL sets the specified pixel to the current graphics color index. The current graphics color index is set with SETCOLOR and retrieved with GETCOLOR. The non-RGB color functions (such as SETCOLOR and SETPIXELS) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function. SETPIXELRGB and SETPIXELRGB_W give access to the full color capacity of the system by using direct color values rather than indexes to a palette.

NOTE
The SETPIXEL routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the SetPixel routine by including the IFWIN module, you need to specify the routine name as MSFWIN$SetPixel.

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) status, x, y
status = SETCOLOR(INT2(2))
x = 10
! Draw pixels.
DO y = 50, 389, 3
   status = SETPIXEL( x, y )
   x = x + 2
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

See Also

SETPIXELRGB
GETPIXEL
SETPIXELS
GETPIXELS
GETCOLOR
SETCOLOR

SETPIXELRGB, SETPIXELRGB_W (W*S)

Graphics Functions: *Set a pixel at a specified location to the specified Red-Green-Blue (RGB) color value.*

Module

USE IFQWIN

Syntax

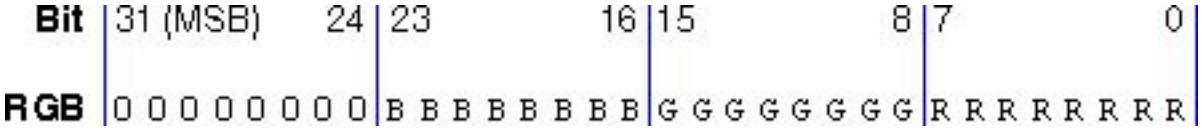
```
result = SETPIXELRGB (x,y,color)
result = SETPIXELRGB_W (x,y,color)
```

x, y	(Input) INTEGER(2). Viewport coordinates for target pixel.
wx, wy	(Input) REAL(8). Window coordinates for target pixel.
color	(Input) INTEGER(4). RGB color value to set the pixel to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous RGB color value of the pixel.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETPIXELRGB or SETPIXELRGB_W, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

If any of the pixels are outside the clipping region, those pixels are ignored.

SETPIXELRGB (and the other RGB color selection functions such as SETPIXELSRGB, SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) x, y
INTEGER(4) color
DO i = 10, 30, 10
  SELECT CASE (i)
    CASE(10)
      color = Z'0000FF'
    CASE(20)
```

```
      color = Z'00FF00'
    CASE (30)
      color = Z'FF0000'
  END SELECT
! Draw pixels.
  DO y = 50, 180, 2
    status = SETPIXELRGB( x, y, color )
    x      = x + 2
  END DO
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

See Also

GETPIXELRGB
GETPIXELSRGB
SETCOLORRGB
SETPIXELSRGB

SETPIXELS (W*S)

Graphics Subroutine: *Sets the color indexes of multiple pixels.*

Module

USE IFQWIN

Syntax

`CALL SETPIXELS (n, x, y, color)`

n	(Input) INTEGER(4). Number of pixels to set. Sets the number of elements in the other arguments.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels to set.
color	(Input) INTEGER(2). Array containing color indexes to set the pixels to.

SETPIXELS sets the pixels specified in the arrays *x* and *y* to the color indexes in *color*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to SETPIXELS with n less than 1 are also ignored. SETPIXELS is a much faster way to set multiple pixel color indexes than individual calls to SETPIXEL.

Unlike SETPIXELS, SETPIXELSRGB gives access to the full color capacity of the system by using direct color values rather than indexes to a palette. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a Graphics ap.
    USE IFQWIN
    INTEGER(2) color(9)
    INTEGER(2) x(9), y(9), i
    DO i = 1, 9
      x(i) = 20 * i
      y(i) = 10 * i
      color(i) = INT2(i)
    END DO
    CALL SETPIXELS(9, x, y, color)
    END
```

See Also

GETPIXELS
SETPIXEL
SETPIXELSRGB

SETPIXELSRGB (W*S)

Graphics Subroutine: *Sets multiple pixels to the given Red-Green-Blue (RGB) color.*

Module

USE IFQWIN

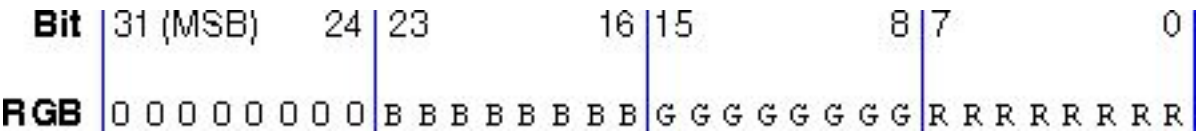
Syntax

CALL SETPIXELSRGB (*n*,*x*,*y*,*color*)

n	(Input) INTEGER(4). Number of pixels to be changed. Determines the number of elements in arrays *x* and *y*.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of the pixels to set.
color	(Input) INTEGER(4). Array containing the RGB color values to set the pixels to. Range and result depend on the system's display adapter.

SETPIXELSRGB sets the pixels specified in the arrays *x* and *y* to the RGB color values in *color*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three color values, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you set with SETPIXELSRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

A good use for SETPIXELSRGB is as a buffering form of SETPIXELRGB, which can improve performance substantially. The example code shows how to do this.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to SETPIXELSRGB with *n* less than 1 are also ignored.

SETPIXELSRGB (and the other RGB color selection functions such as SETPIXELRGB and SETCOLORRGB) sets colors to values chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Buffering replacement for SetPixelRGB and
! SetPixelRGB_W. This can improve performance by
! doing batches of pixels together.

USE IFQWIN
PARAMETER (I$SIZE = 200)
INTEGER(4) bn, bc(I$SIZE), status
INTEGER(2) bx(I$SIZE),by(I$SIZE)

bn = 0
DO i = 1, I$SIZE
  bn = bn + 1
  bx(bn) = i
  by(bn) = i
  bc(bn) = GETCOLORRGB()
  status = SETCOLORRGB(bc(bn)+1)
END DO
CALL SETPIXELSRGB(bn,bx,by,bc)
END
```

See Also

GETPIXELSRGB
SETPIXELRGB
GETPIXELRGB
SETPIXELS

SETTEXTCOLOR (W*S)

Graphics Function: *Sets the current text color index.*

Module

USE IFQWIN

Syntax

`result = SETTEXTCOLOR (index)`

index (Input) INTEGER(2). Color index to set the text color to.

Results

The result type is INTEGER(2). The result is the previous text color index.

SETTEXTCOLOR sets the current text color index. The default value is 15, which is associated with white unless the user remaps the palette. GETTEXTCOLOR returns the text color index set by SETTEXTCOLOR. SETTEXTCOLOR affects text output with OUTTEXT, WRITE, and PRINT.

The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use SETTEXTCOLORRGB, SETBKCOLORRGB, and SETCOLORRGB.

NOTE
The SETTEXTCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the SetTextColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$SetTextColor.

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) oldtc
oldtc = SETTEXTCOLOR(INT2(2)) ! green
WRITE(*,*) "hello, world"
END
```

See Also

GETTEXTCOLOR
REMAPPALETTERGB
SETCOLOR
SETTEXTCOLORRGB

SETTEXTCOLORRGB (W*S)

Graphics Function: *Sets the current text color to the specified Red-Green-Blue (RGB) value.*

Module

USE IFQWIN

Syntax

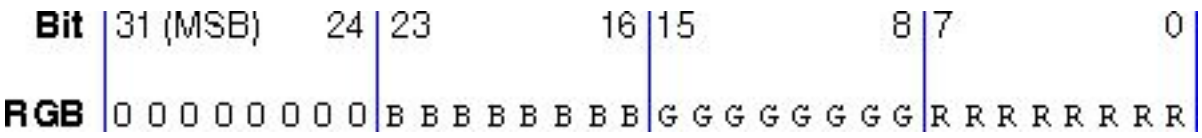
`result = SETTEXTCOLORRGB (color)`

color	(Input) INTEGER(4). RGB color value to set the text color to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous text RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

SETTEXTCOLORRGB sets the current text RGB color. The default value is Z'00FFFFFF', which is full-intensity white. SETTEXTCOLORRGB sets the color used by OUTTEXT, WRITE, and PRINT. It does not affect the color of text output with the OUTGTEXT font routine. Use SETCOLORRGB to change the color of font output.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. SETCOLORRGB sets the RGB color value of graphics over the background color, used by the graphics functions such as ARC, FLOODFILLRGB, and OUTGTEXT.

SETTEXTCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB and SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETTEXTCOLOR, SETBKCOLOR, and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(4) oldtc

oldtc = SETTEXTCOLORRGB(Z'000000FF')
WRITE(*,*) 'I am red'
oldtc = SETTEXTCOLORRGB(Z'0000FF00')
CALL OUTTEXT ('I am green'//CHAR(13)//CHAR(10))
oldtc = SETTEXTCOLORRGB(Z'00FF0000')
PRINT *, 'I am blue'
END
```

See Also

SETBKCOLORRGB
SETCOLORRGB
GETTEXTCOLORRGB
GETWINDOWCONFIG
OUTTEXT

SETTEXTCURSOR (W*S)

Graphics Function: *Sets the height and width of the text cursor (the caret) for the window in focus.*

Module

USE IFQWIN

Syntax

`result = SETTEXTCURSOR (newcursor)`

newcursor	(Input) INTEGER(2). The leftmost 8 bits specify the width of the cursor, and the rightmost 8 bits specify the height of the cursor. These dimensions can range from 1 to 8, and represent a fraction of the current character cell size. For example:

- Z'0808' - Specifies the full character cell; this is the default size.
- Z'0108' - Specifies 1/8th of the character cell width, and 8/8th (or all) of the character cell height.

If either of these dimensions is outside the range 1 to 8, it is forced to 8.

Results

The result type is INTEGER(2); it is the previous text cursor value in the same format as *newcursor*.

NOTE
After calling SETTEXTCURSOR, you must call DISPLAYCURSOR($GCURSORON) to actually see the cursor.

Example

```
use IFQWIN
integer(2) oldcur
integer(2) istat
type(rccoord) rc
open(10,file='user')
istat  = displaycursor($GCURSORON)
write(10,*) 'Text cursor is now character cell size, the default.'
read(10,*)
write(10,*) 'Setting text cursor to wide and low.'
oldcur = settextcursor(Z'0801')
istat  = displaycursor($GCURSORON)
read(10,*)
write(10,*) 'Setting text cursor to high and narrow.'
oldcur = settextcursor(Z'0108')
istat  = displaycursor($GCURSORON)
read(10,*)
write(10,*) 'Setting text cursor to a dot.'
oldcur = settextcursor(Z'0101')
istat  = displaycursor($GCURSORON)
read(10,*)
end
```

See Also

DISPLAYCURSOR

SETTEXTPOSITION (W*S)

Graphics Subroutine: *Sets the current text position to a specified position relative to the current text window.*

Module

USE IFQWIN

Syntax

CALL SETTEXTPOSITION (*row*,*column*,*t*)

row (Input) INTEGER(2). New text row position.

column (Input) INTEGER(2). New text column position.

t (Output) Derived type `rccoord`. Previous text position. The derived type `rccoord`is defined in `IFQWIN.F90` as follows:

```
TYPE rccoord
  INTEGER(2) row ! Row coordinate
  INTEGER(2) col ! Column coordinate
END TYPE rccoord
```

Subsequent text output with the OUTTEXT function (as well as standard console I/O statements, such as PRINT and WRITE) begins at the point (*row*, *column*).

Example

```
USE IFQWIN
TYPE (rccoord) curpos

WRITE(*,*) "Original text position"
CALL SETTEXTPOSITION (INT2(6), INT2(5), curpos)
WRITE (*,*) 'New text position'
END
```

See Also

CLEARSCREEN
GETTEXTPOSITION
OUTTEXT
SCROLLTEXTWINDOW
SETTEXTWINDOW
WRAPON

SETTEXTWINDOW (W*S)

Graphics Subroutine: *Sets the current text window.*

Module

USE IFQWIN

Syntax

`CALL SETTEXTWINDOW (r1,c1,r2,c2)`

r1, c1 (Input) INTEGER(2). Row and column coordinates for upper-left corner of the text window.

r2, c2 (Input) INTEGER(2). Row and column coordinates for lower-right corner of the text window.

SETTEXTWINDOW specifies a window in row and column coordinates where text output to the screen using OUTTEXT, WRITE, or PRINT will be displayed. You set the text location within this window with SETTEXTPOSITION.

Text is output from the top of the window down. When the window is full, successive lines overwrite the last line.

SETTEXTWINDOW does not affect the output of the graphics text routine OUTGTEXT. Use the SETVIEWPORT function to control the display area for graphics output.

Example

```
USE IFQWIN
TYPE (rccoord) curpos

CALL SETTEXTWINDOW(INT2(5), INT2(1), INT2(7), &
                   INT2(40))
CALL SETTEXTPOSITION (INT2(5), INT2(5), curpos)
WRITE(*,*) "Only two lines in this text window"
WRITE(*,*) "so this line will be overwritten"
WRITE(*,*) "by this line"
END
```

See Also

GETTEXTPOSITION
GETTEXTWINDOW
GRSTATUS
OUTTEXT
SCROLLTEXTWINDOW
SETTEXTPOSITION
SETVIEWPORT
WRAPON

SETTIM

Portability Function: *Sets the system time in your programs. This function is only available on Windows* and Linux* systems.*

Module

USE IFPORT

Syntax

`result = SETTIM (ihr, imin, isec, i100th)`

ihr	(Output) INTEGER(4) or INTEGER(2). Hour (0-23).
imin	(Output) INTEGER(4) or INTEGER(2). Minute (0-59).
isec	(Output) INTEGER(4) or INTEGER(2). Second (0-59).
i100th	(Output) INTEGER(4) or INTEGER(2). Hundredths of a second (0-99).

Results

The result type is LOGICAL(4). The result is .TRUE. if the system time is changed; .FALSE. if no change is made.

All arguments must be of the same integer kind, that is, all must be INTEGER(2) or all must be INTEGER(4).

If INTEGER(2) arguments are passed, you must specify USE IFPORT.

NOTE
On Linux systems, you must have root privileges to execute this function.

Example

```
USE IFPORT
LOGICAL(4) success
success = SETTIM(INT2(21),INT2(53+3),&
                 INT2(14*2),INT2(88))
END
```

See Also

GETDAT
GETTIM
SETDAT

SETVIEWORG (W*S)

Graphics Subroutine: *Moves the viewport-coordinate origin (0, 0) to the specified physical point.*

Module

USE IFQWIN

Syntax

CALL SETVIEWORG (*x*,*y*,*s*)

x, *y*

(Input) INTEGER(2). Physical coordinates of new viewport origin.

s

(Output) Derived type `xycoord`. Physical coordinates of the previous viewport origin. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord ! x-coordinate
  INTEGER(2) ycoord ! y-coordinate
END TYPE xycoord
```

The `xycoord`type variable *s*, defined in `IFQWIN.F90`, returns the physical coordinates of the previous viewport origin.

Example

```
USE IFQWIN
TYPE ( xycoord ) xy

CALL SETVIEWORG(INT2(30), INT2(30), xy)
```

See Also

GETCURRENTPOSITION
GETPHYSCOORD
GETVIEWCOORD
GETWINDOWCOORD
GRSTATUS
SETCLIPRGN
SETVIEWPORT

SETVIEWPORT (W*S)

Graphics Subroutine: *Redefines the graphics viewport by defining a clipping region in the same manner as SETCLIPRGN and then setting the viewport-coordinate origin to the upper-left corner of the region.*

Module

USE IFQWIN

Syntax

```
CALL SETVIEWPORT (x1,y1,x2,y2)
```

x1, y1	(Input) INTEGER(2). Physical coordinates for upper-left corner of viewport.
x2, y2	(Input) INTEGER(2). Physical coordinates for lower-right corner of viewport.

The physical coordinates (*x1*, *y1*) and (*x2*, *y2*) are the upper-left and lower-right corners of the rectangular clipping region. Any window transformation done with the SETWINDOW function is relative to the viewport, not the entire screen.

Example

```
USE IFQWIN
INTEGER(2) upx, upy
INTEGER(2) downx, downy

upx = 0
upy = 30
downx= 250
downy = 100
CALL SETVIEWPORT(upx, upy, downx, downy)
```

See Also

GETVIEWCOORD
GETPHYSCOORD
GRSTATUS
SETCLIPRGN
SETVIEWORG
SETWINDOW

SETWINDOW (W*S)

Graphics Function: *Defines a window bound by the specified coordinates.*

Module

USE IFQWIN

Syntax

```
result = SETWINDOW (finvert,wx1,wy1,wx2,wy2)
```

finvert	(Input) LOGICAL(2). Direction of increase of the y-axis. If *finvert* is .TRUE., the y-axis increases from the window bottom to the window top (as Cartesian coordinates). If *finvert* is .FALSE., the y-axis increases from the window top to the window bottom (as pixel coordinates).
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of window.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of window.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0 (for example, if the program that calls SETWINDOW is not in a graphics mode).

The SETWINDOW function determines the coordinate system used by all window-relative graphics routines. Any graphics routines that end in _W (such as ARC_W, RECTANGLE_W, and LINETO_W) use the coordinate system set by SETWINDOW.

Any window transformation done with the SETWINDOW function is relative to the viewport, not the entire screen.

An arc drawn using inverted window coordinates is not an upside-down version of an arc drawn with the same parameters in a noninverted window. The arc is still drawn counterclockwise, but the points that define where the arc begins and ends are inverted.

If *wx1* equals *wx2* or *wy1* equals *wy2*, SETWINDOW fails.

Example

```
USE IFQWIN
INTEGER(2) status
LOGICAL(2) invert /.TRUE./
REAL(8) upx /0.0/, upy /0.0/
REAL(8) downx /1000.0/, downy /1000.0/
status = SETWINDOW(invert, upx, upy, downx, downy)
```

See Also

GETWINDOWCOORD
SETCLIPRGN
SETVIEWORG
SETVIEWPORT
GRSTATUS
ARC_W
LINETO_W
MOVETO_W
PIE_W
POLYGON_W
RECTANGLE_W

SETWINDOWCONFIG (W*S)

QuickWin Function: *Sets the properties of a child window.*

Module

USE IFQWIN

Syntax

`result = SETWINDOWCONFIG (wc)`

wc

(Input) Derived type `windowconfig`. Contains window properties. The `windowconfig` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE windowconfig
  INTEGER(2) numxpixels         ! Number of pixels on x-axis.
  INTEGER(2) numypixels         ! Number of pixels on y-axis.
  INTEGER(2) numtextcols        ! Number of text columns
available.
  INTEGER(2) numtextrows        ! Number of text rows
available.
  INTEGER(2) numcolors          ! Number of color indexes.
  INTEGER(4) fontsize           ! Size of default font. Set to
                                !   QWIN$EXTENDFONT when
specifying
                                !   extended attributes, in
which
                                !   case extendfontsize sets
the
                                !   font size.
  CHARACTER(80) title               ! The window title.
  INTEGER(2) bitsperpixel       ! The number of bits per
pixel.
  INTEGER(2) numvideopages          ! Unused.
  INTEGER(2) mode               ! Controls scrolling mode.
  INTEGER(2) adapter                ! Unused.
  INTEGER(2) monitor                ! Unused.
  INTEGER(2) memory                 ! Unused.
  INTEGER(2) environment            ! Unused.
! The next three parameters provide extended font
! attributes.
  CHARACTER(32) extendfontname      ! The name of the desired
font.
  INTEGER(4) extendfontsize     ! Takes the same values as
fontsize,
                                ! when fontsize is set to
                                ! QWIN$EXTENDFONT.
  INTEGER(4) extendfontattributes   ! Font attributes such as
bold
                                    ! and italic.
END TYPE windowconfig
```

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The following value can be used to configure a QuickWin window so that it will show the last line written and the text cursor (if it is on):

```
wc%mode = QWIN$SCROLLDOWN
```

Note that if you scroll the window to another position, you will have to scroll back to the last line to see your input.

The following values can be used with SETWINDOWCONFIG extended fonts:

Style:	
QWIN$EXTENDFONT_NORMAL	No underline, no italic, and a font weight of 400 out of 1000.
QWIN$EXTENDFONT_UNDERLINE	Underlined characters.
QWIN$EXTENDFONT_BOLD	A font weight of 700 out of 1000.
QWIN$EXTENDFONT_ITALIC	Italic characters.
Pitch:	
QWIN$EXTENDFONT_FIXED_PITCH	QuickWin default. Equal character widths.
QWIN$EXTENDFONT_VARIABLE_PITCH	Variable character widths.
Font Families:	
QWIN$EXTENDFONT_FF_ROMAN	Variable stroke width, serifed. Times Roman, Century Schoolbook, etc.
QWIN$EXTENDFONT_FF_SWISS	Variable stroke width, sans-serifed. Helvetica, Swiss, etc.
QWIN$EXTENDFONT_FF_MODERN	QuickWin default. Constant stroke width, serifed or sans-serifed. Pica, Elite, Courier, etc.
QWIN$EXTENDFONT_FF_SCRIPT	Cursive, etc.
QWIN$EXTENDFONT_FF_DECORATIVE	Old English, etc.
Character Sets:	
QWIN$EXTENDFONT_ANSI_CHARSET	QuickWin default.
QWIN$EXTENDFONT_OEM_CHARSET	Use this to get Microsoft* LineDraw.

Using QWIN$EXTENDFONT_OEM_CHARSET with the font name 'MS LineDraw'C will get the old DOS-style character set with symbols that can be used to draw lines and boxes. The pitch and font family items can be specified to help guide the font matching algorithms used by CreateFontIndirect, the Windows* API used by SETWINDOWCONFIG.

If you use SETWINDOWCONFIG to set the variables in `windowconfig` to -1, the function sets the highest resolution possible for your system, given the other fields you specify, if any. You can set the actual size of the window by specifying parameters that influence the window size: the number of x and y pixels, the number of rows and columns, and the font size. If you do not call SETWINDOWCONFIG, the window defaults to the best possible resolution and a font size of 8x16. The number of colors available depends on the video driver used.

If you use SETWINDOWCONFIG, you should specify a value for each field (-1 or your own value for the numeric fields and a C string for the title, for example, "words of text"C). Using SETWINDOWCONFIG with only some fields specified can result in useless values for the unspecified fields.

If you request a configuration that cannot be set, SETWINDOWCONFIG returns .FALSE. and calculates parameter values that will work and are as close as possible to the requested configuration. A second call to SETWINDOWCONFIG establishes the adjusted values; for example:

```
status = SETWINDOWCONFIG(wc)
if (.NOT.status) status = SETWINDOWCONFIG(wc)
```

If you specify values for all four of the size parameters, *numxpixels*, *numypixel*, *numtextcols*, and *numtextrows*, the font size is calculated by dividing these values.The default font is Courier New and the default font size is 8x16. There is no restriction on font size, except that the window must be large enough to hold it.

Under Standard Graphics, the application attempts to start in Full Screen mode with no window decoration (window decoration includes scroll bars, menu bar, title bar, and message bar) so that the maximum resolution can be fully used. Otherwise, the application starts in a window. You can use ALT+ENTER at any time to toggle between the two modes.

If you are in Full Screen mode and the resolution of the window does not match the resolution of the video driver, graphics output will be slow compared to drawing in a window.

NOTE
You must call DISPLAYCURSOR($GCURSORON) to make the cursor visible after calling SETWINDOWCONFIG.

Example

```
USE IFQWIN
TYPE (windowconfig) wc
LOGICAL status /.FALSE./
! Set the x & y pixels to 800X600 and font size to 8x12
wc%numxpixels  = 800
wc%numypixels  = 600
wc%numtextcols = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%title= "This is a test"C
wc%fontsize = Z'0008000C'
status = SETWINDOWCONFIG(wc)  ! attempt to set configuration with above values
   ! if attempt fails, set with system estimated values
if (.NOT.status) status = SETWINDOWCONFIG(wc)
```

See Also
DISPLAYCURSOR
GETWINDOWCONFIG

SETWINDOWMENUQQ (W*S)

QuickWin Function: *Sets a top-level menu as the menu to which a list of current child window names is appended.*

Module

USE IFQWIN

Syntax

`result = SETWINDOWMENUQQ (menuID)`

menuID	(Input) INTEGER(4). Identifies the menu to hold the child window names, starting with 1 as the leftmost menu.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The list of current child window names can appear in only one menu at a time. If the list of windows is currently in a menu, it is removed from that menu. By default, the list of child windows appears at the end of the Window menu.

Example

```
USE IFQWIN
TYPE (windowconfig) wc
LOGICAL(4) result, status /.FALSE./
! Set title for child window
wc%numxpixels  = -1
wc%numypixels  = -1
wc%numtextcols = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%fontsize    = -1
wc%title= "I am child window name"C
if (.NOT.status) status = SETWINDOWCONFIG(wc)

! put child window list under menu 3 (View)
result = SETWINDOWMENUQQ(3)
END
```

See Also

APPENDMENUQQ

SETWRITEMODE (W*S)

Graphics Function: *Sets the current logical write mode, which is used when drawing lines with the LINETO, POLYGON, and RECTANGLE functions.*

Module

USE IFQWIN

Syntax

`result = SETWRITEMODE (wmode)`

wmode

(Input) INTEGER(2). Write mode to be set. One of the following symbolic constants (defined in `IFQWIN.F90`):

- $GPSET - Causes lines to be drawn in the current graphics color. (Default)
- $GAND - Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.
- $GOR - Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.
- $GPRESET - Causes lines to be drawn in the color that is the logical NOT of the current graphics color.
- $GXOR - Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.

In addition, one of the following binary raster operation constants can be used (described in the online documentation for the Windows* API SetROP2):

- $GR2_BLACK
- $GR2_NOTMERGEPEN
- $GR2_MASKNOTPEN
- $GR2_NOTCOPYPEN (same as $GPRESET)
- $GR2_MASKPENNOT
- $GR2_NOT
- $GR2_XORPEN (same as $GXOR)
- $GR2_NOTMASKPEN
- $GR2_MASKPEN (same as $GAND)
- $GR2_NOTXORPEN
- $GR2_NOP
- $GR2_MERGENOTPEN
- $GR2_COPYPEN (same as $GPSET)
- $GR2_MERGEPENNOT
- $GR2_MERGEPEN (same as $GOR)
- $GR2_WHITE

Results

The result type is INTEGER(2). The result is the previous write mode if successful; otherwise, -1.

The current graphics color is set with SETCOLORRGB (or SETCOLOR) and the current background color is set with SETBKCOLORRGB (or SETBKCOLOR). As an example, suppose you set the background color to yellow (Z'00FFFF') and the graphics color to purple (Z'FF00FF') with the following commands:

```
oldcolor = SETBKCOLORRGB(Z'00FFFF')
CALL CLEARSCREEN($GCLEARSCREEN)
oldcolor = SETCOLORRGB(Z'FF00FF')
```

If you then set the write mode with the $GAND option, lines are drawn in red (Z'0000FF'); with the $GOR option, lines are drawn in white (Z'FFFFFF'); with the $GXOR option, lines are drawn in turquoise (Z'FFFF00'); and with the $GPRESET option, lines are drawn in green (Z'00FF00'). Setting the write mode to $GPSET causes lines to be drawn in the graphics color.

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) result, oldmode
INTEGER(4) oldcolor
TYPE (xycoord) xy

oldcolor = SETBKCOLORRGB(Z'00FFFF')
CALL CLEARSCREEN ($GCLEARSCREEN)
oldcolor = SETCOLORRGB(Z'FF00FF')
CALL MOVETO(INT2(0), INT2(0), xy)
result = LINETO(INT2(200), INT2(200)) ! purple

oldmode = SETWRITEMODE( $GAND)
CALL MOVETO(INT2(50), INT2(0), xy)
result = LINETO(INT2(250), INT2(200)) ! red
END
```

See Also

GETWRITEMODE
GRSTATUS
LINETO
POLYGON

PUTIMAGE
RECTANGLE
SETCOLOR
SETLINESTYLE

SETWSIZEQQ (W*S)

QuickWin Function: *Sets the size and position of a window.*

Module

USE IFQWIN

Syntax

`result = SETWSIZEQQ (unit, winfo)`

unit

(Input) INTEGER(4). Specifies the window unit. Unit numbers 0, 5, and 6 refer to the default startup window only if the program does not explicitly open them with the OPEN statement. To set the size of the frame window (as opposed to a child window), set *unit* to the symbolic constant QWIN$FRAMEWINDOW (defined in `IFQWIN.F90`).

When called from INITIALSETTINGS, SETWSIZEQQ behaves slightly differently than when called from a user routine after initialization. See below under Results.

winfo

(Input) Derived type `qwinfo`. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame). The derived type `qwinfo`is defined in `IFQWIN.F90` as follows:

```
TYPE QWINFO
   INTEGER(2) TYPE  ! request type
   INTEGER(2) X     ! x coordinate for upper left
   INTEGER(2) Y     ! y coordinate for upper left
   INTEGER(2) H     ! window height
   INTEGER(2) W     ! window width
END TYPE QWINFO
```

This function's behavior depends on the value of `QWINFO%TYPE`, which can be any of the following:

- QWIN$MIN - Minimizes the window.
- QWIN$MAX - Maximizes the window.
- QWIN$RESTORE - Restores the minimized window to its previous size.
- QWIN$SET - Sets the window's position and size according to the other values in `qwinfo`.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero (unless called from INITIALSETTINGS). If called from INITIALSETTINGS, the following occurs:

- SETWSIZEQQ always returns -1.
- Only QWIN$SET will work.

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width specified for a frame window reflects the actual size in pixels of the frame window *including* any borders, menus, and status bar at the bottom.

Example

```
USE IFQWIN
INTEGER(4)     result
INTEGER(2)     numfonts, fontnum
TYPE (qwinfo)  winfo
TYPE (xycoord) pos
! Maximize frame window
winfo%TYPE = QWIN$MAX
result =     SETWSIZEQQ(QWIN$FRAMEWINDOW, winfo)
! Maximize child window
result =   SETWSIZEQQ(0, winfo)
numfonts = INITIALIZEFONTS( )
fontnum =  SETFONT ('t''Arial''h50w34i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT("BIG Window")
END
```

See Also

GETWSIZEQQ
INITIALSETTINGS

SHAPE

Inquiry Intrinsic Function (Generic): *Returns the shape of an array or scalar argument.*

Syntax

`result = SHAPE (source [, kind])`

source	(Input) Is a scalar or array. It may be of any data type. It must not be an assumed-size array, a disassociated pointer, or an allocatable array that is not allocated. It cannot be an assumed-size array.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is a rank-one integer array whose size is equal to the rank of *source*. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is the shape of *source*, unless *source* is assumed-rank and is associated with an assumed-size array. In that case, the last element of the value returned by SHAPE is -1.

The setting of compiler options specifying integer size can affect this function.

Example

SHAPE (2) has the value of a rank-one array of size zero.

If B is declared as B(2:4, -3:1), then SHAPE (B) has the value (3, 5).

The following shows another example:

```
INTEGER VEC(2)
REAL array(3:10, -1:3)
VEC = SHAPE(array)
WRITE(*,*) VEC ! prints   8       5
END
!
! Check if a mask is conformal with an array
REAL, ALLOCATABLE :: A(:,:,:)
LOGICAL, ALLOCATABLE :: MASK(:,:,:)
INTEGER B(3), C(3)
LOGICAL conform
ALLOCATE (A(5, 4, 3))
ALLOCATE (MASK(3, 4, 5))
! Check if MASK and A allocated. If they are, check
! that they have the same shape (conform).
IF(ALLOCATED(A) .AND. ALLOCATED(MASK)) THEN
  B = SHAPE(A); C = SHAPE(MASK)
  IF ((B(1) .EQ. C(1)) .AND. (B(2) .EQ. C(2))       &
       .AND. (B(3) .EQ. C(3))) THEN
     conform = .TRUE.
  ELSE
     conform = .FALSE.
  END IF
END IF
WRITE(*,*) conform  ! prints F
END
```

See Also

SIZE

SHARED Clause

Parallel Directive Clause: *Specifies variables that will be shared by all the threads in a team.*

Syntax

`SHARED (`*`list`*`)`

`list`	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

All threads within a team access the same storage area for SHARED data.

SHIFTA

Elemental Intrinsic Function (Specific): *Performs a right shift with fill. This function cannot be passed as an actual argument.*

Syntax

`result = SHIFTA (`*`i`*`,`*`shift`*`)`

i	(Input) Must be of type integer. This is the value to be shifted.
shift	(Input) Must be of type integer. It must be nonnegative and <= BIT_SIZE(*i*). This value is the number of positions to shift.

Results

The result type and kind are the same as *i*.

The result has the value obtained by shifting the bits of *i* to the right *shift* bits and replicating the leftmost bit of *i* in the left *shift* bits.

If *shift* is zero the result is *i*. Bits shifted off the right end are lost.

The model for the interpretation of an integer value as a sequence of bits is in Model for Bit Data.

Example

SHIFTA (IBSET (0, BIT_SIZE (0) - 1), 2) is equal to SHIFTL (7, BIT_SIZE (0) - 3).

SHIFTL

Elemental Intrinsic Function (Specific): *Logically shifts an integer left by a specified number of bits. This function cannot be passed as an actual argument.*

Syntax

```
result = SHIFTL (i,shift)
```

i	(Input) Must be of type integer. This is the value to be shifted.
shift	(Input) Must be of type integer. The value must be nonnegative and <= BIT_SIZE(*i*). This value is the number of positions to shift.

Results

The result type and kind are the same as *i*. The result is the value of *i* shifted left by *shift* bit positions. Bits shifted off the left end are lost; zeros are shifted in from the opposite end.

SHIFTL (i, j) is the same as ISHFT (i, j).

See Also

ISHFT

SHIFTR

Elemental Intrinsic Function (Specific): *Logically shifts an integer right by a specified number of bits. This function cannot be passed as an actual argument.*

Syntax

```
result = SHIFTR (i,shift)
```

i	(Input) Must be of type integer. This is the value to be shifted.
shift	(Input) Must be of type integer. The value must be nonnegative and <= BIT_SIZE(*i*). This value is the number of positions to shift.

Results

The result type and kind are the same as *i*. The result is the value of *i* shifted right by *shift* bit positions. Bits shifted off the right end are lost; zeros are shifted in from the opposite end.

SHIFTR (i, j) is the same as ISHFT (i, -j).

See Also

ISHFT

SHORT

Portability Function: *Converts an INTEGER(4) argument to INTEGER(2) type.*

Module

USE IFPORT

Syntax

`result = SHORT (int4)`

int4	(Input) INTEGER(4). Value to be converted.

Results

The result type is INTEGER(2). The result is equal to the lower 16 bits of *int4*. If the *int4* value is greater than 32,767, the converted INTEGER(2) value is not equal to the original.

Example

```
     USE IFPORT
     INTEGER(4) this_one
     INTEGER(2) that_one
     READ(*,*) this_one
     THAT_ONE = SHORT(THIS_ONE)
     WRITE(*,10) THIS_ONE, THAT_ONE
10   FORMAT (X," Long integer: ", I16, " Short integer: ", I16)
     END
```

See Also

INT

Type Declarations

SIGN

Elemental Intrinsic Function (Generic): *Returns the absolute value of the first argument times the sign of the second argument.*

Syntax

`result = SIGN (a, b)`

a	(Input) Must be of type integer or real.
b	(Input) Must have the same type as *a*.

Results

The result type and kind are the same as *a*. The value of the result is as follows:

- | *a* | if *b* > zero and -| *a* | if *b* < zero.
- | *a* | if *b* is of type integer and is zero.
- If *b* is of type real and zero and compiler option `assume minus0` is not specified, the value of the result is | *a* |.
- If *b* is of type real and zero and compiler option `assume minus0` is specified, the processor can distinguish between positive and negative real zero and the following occurs:
  - If *b* is positive real zero, the value of the result is | *a* |.
  - If *b* is negative real zero, the value of the result is -| *a* |.

The specific named versions of the function require that the arguments have the same kind parameters. The generic form of the function permits the kind types of the arguments to differ.

Specific Name	Argument Type	Result Type
BSIGN	INTEGER(1)	INTEGER(1)
IISIGN[1]	INTEGER(2)	INTEGER(2)
ISIGN [2]	INTEGER(4)	INTEGER(4)
KISIGN	INTEGER(8)	INTEGER(8)
SIGN [3]	REAL(4)	REAL(4)
DSIGN [3,4]	REAL(8)	REAL(8)
QSIGN	REAL(16)	REAL(16)

[1] Or HSIGN.

[2] Or JISIGN. For compatibility with older versions of Fortran, ISIGN is treated as a generic function.

[3] The setting of compiler options specifying real size can affect SIGN and DSIGN .

[4] The setting of compiler options specifying double size can affect DSIGN.

Example

SIGN (4.0, -6.0) has the value -4.0.

SIGN (-5.0, 2.0) has the value 5.0.

The following shows another example:

```
c = SIGN (5.2, -3.1)   ! returns -5.2
c = SIGN (-5.2, -3.1)  ! returns -5.2
c = SIGN (-5.2, 3.1)   ! returns  5.2
```

See Also

ABS

assume minus0 compiler option

SIGNAL

Portability Function: *Controls interrupt signal handling and changes the action for a specified signal.*

Module

USE IFPORT

Syntax

`result = SIGNAL (signum, proc, flag)`

signum	(Input) INTEGER(4). Number of the signal to change. The numbers and symbolic names are listed in a table below.
proc	(Input) Name of a signal-handler routine. It must be declared EXTERNAL and INTEGER(4). This routine is called only if *flag* is negative.
flag	(Input) INTEGER(4). If negative, the user's *proc* routine is called. If 0, the signal retains its default action; if 1, the signal should be ignored.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The result is the previous value of *proc* associated with the specified signal. For example, if the previous value of *proc* was SIG_IGN, the return value is also SIG_IGN. You can use this return value in subsequent calls to SIGNAL if the signal number supplied is invalid, if the flag value is greater than 1, or to restore a previous action definition.

A return value of SIG_ERR indicates an error, in which case a call to IERRNO returns EINVAL. If the signal number supplied is invalid, or if the flag value is greater than 1, SIGNAL returns -(EINVAL) and a call to IERRNO returns EINVAL.

An initial signal handler is in place at startup for SIGFPE (signal 8); its address is returned the first time SIGNAL is called for SIGFPE. No other signals have initial signal handlers.

Be careful when you use SIGNALQQ or the C signal function to set a handler, and then use the Portability SIGNAL function to retrieve its value. If SIGNAL returns an address that was not previously set by a call to SIGNAL, you cannot use that address with either SIGNALQQ or C's signal function, nor can you call it directly. You can, however, use the return value from SIGNAL in a subsequent call to SIGNAL. This allows you to restore a signal handler, no matter how the original signal handler was set.

The signal-handler argument *proc* accepts a single INTEGER(4) argument, which is the number of the signal to be handled. The function must use the compiler's default calling conventions. If you have used compiler options, such as option `iface`, to change the default conventions, add the following directive to the signal handler function to reset the calling conventions:

!DIR$ ATTRIBUTES DEFAULT :: *the-routine-specified-in-proc*

The signal-handler function returns an INTEGER(4) value. If the function has handled the signal, the value returned is an integer, but the value is not used.

Because signal-handler routines are usually called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. You cannot use the following kinds of signal-handler routines:

- Routines that perform low-level (such as FGETC) or high-level (such as READ) I/O.
- Heap routines or any routine that uses the heap routines (such as MALLOC and ALLOCATE).
- Functions that generate a system call (such as TIME).

The following table lists signals, their names and values:

Symbolic name	Number	Description
SIGABRT	6	Abnormal termination
SIGFPE	8	Floating-point error
SIGKILL[1]	9	Kill process
SIGILL	4	Illegal instruction
SIGINT	2	CTRL+C signal
SIGSEGV	11	Illegal storage access
SIGTERM	15	Termination request

[1]SIGKILL can be neither caught nor ignored.

The default action for all signals is to terminate the program with exit code.

ABORT does not assert the SIGABRT signal. The only way to assert SIGABRT or SIGTERM is to use KILL.

SIGNAL can be used to catch SIGFPE exceptions, but it cannot be used to access the error code that caused the SIGFPE. To do this, use SIGNALQQ instead.

Example

```
   USE IFPORT
   EXTERNAL h_abort
   INTEGER(4) :: h_abort
   INTEGER(4) iret1, iret2, procnum
   iret1 = SIGNAL(SIGABRT, h_abort, -1)
   WRITE(*,*) 'Set signal handler #1. Return = ', iret1

   procnum = getpid( )
   iret2 = KILL(procnum, SIGABRT)
   WRITE(*,*) 'Raised signal. Return = ', iret2
   END
!
!  Signal-handler routine
!
   INTEGER(4) FUNCTION h_abort (sig_num)
   INTEGER(4) sig_num
   !DIR$ ATTRIBUTES DEFAULT :: h_abort
   WRITE(*,*) 'In signal handler function h_abort for SIG$ABORT'
   WRITE(*,*) 'signum = ', sig_num
   h_abort = 1
   END
```

See Also

SIGNALQQ

SIGNALQQ

Portability Function: *Registers the function to be called if an interrupt signal occurs.*

Module

USE IFPORT

Syntax

```
result = SIGNALQQ (sig,func)
```

sig — (Input) INTEGER(2). Interrupt type. One of the following constants, defined in IFPORT.F90:

- SIG$ABORT - Abnormal termination
- SIG$FPE - Floating-point error
- SIG$ILL - Illegal instruction
- SIG$INT - CTRL+CSIGNAL
- SIG$SEGV - Illegal storage access
- SIG$TERM - Termination request

func — (Input) Function to be executed on interrupt. It must be declared EXTERNAL.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The result is a positive integer if successful; otherwise, -1 (SIG$ERR).

SIGNALQQ installs the function *func* as the handler for a signal of the type specified by *sig*. If you do not install a handler, the system by default terminates the program with exit code 3 when an interrupt signal occurs.

The argument *func* is the name of a function and must be declared with either the EXTERNAL or IMPLICIT statements, or have an explicit interface. A function described in an INTERFACE block is EXTERNAL by default, and does not need to be declared EXTERNAL.

NOTE
All signal-handler functions must be declared with the directive !DIR$ ATTRIBUTES C.

When an interrupt occurs, except a SIG$FPE interrupt, the *sig* argument SIG$INT is passed to *func*, and then *func* is executed.

When a SIG$FPE interrupt occurs, the function *func* is passed two arguments: SIG$FPE and the floating-point error code (for example, FPE$ZERODIVIDE or FPE$OVERFLOW) which identifies the type of floating-point exception that occurred. The floating-point error codes begin with the prefix FPE$ and are defined in IFPORT.F90.

If *func* returns, the calling process resumes execution immediately after the point at which it received the interrupt signal. This is true regardless of the type of interrupt or operating mode.

Because signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. Therefore, do not call heap routines or any routine that uses the heap routines (for example, I/O routines, ALLOCATE, and DEALLOCATE).

To test your signal handler routine you can generate interrupt signals by calling RAISEQQ, which causes your program either to branch to the signal handlers set with SIGNALQQ, or to perform the system default behavior if SIGNALQQ has set no signal handler.

The example below shows a signal handler for SIG$ABORT.

Example

```
!  This program shows a signal handler for
!  SIG$ABORT
```

```
USE IFPORT
INTERFACE
  FUNCTION h_abort (signum)
    !DIR$ ATTRIBUTES C :: h_abort
    INTEGER(4) h_abort
    INTEGER(2) signum
  END FUNCTION
END INTERFACE

INTEGER(2) i2ret
INTEGER(4) i4ret

i4ret = SIGNALQQ(SIG$ABORT, h_abort)
WRITE(*,*) 'Set signal handler. Return = ', i4ret

i2ret = RAISEQQ(SIG$ABORT)
WRITE(*,*) 'Raised signal. Return = ', i2ret
END
!
!      Signal handler routine
!
INTEGER(4) FUNCTION h_abort (signum)
  !DIR$ ATTRIBUTES C :: h_abort
  INTEGER(2) signum
  WRITE(*,*) 'In signal handler for SIG$ABORT'
  WRITE(*,*) 'signum = ', signum
  h_abort = 1
END
```

See Also

RAISEQQ
SIGNAL
KILL
GETEXCEPTIONPTRSQQ

SIMD Directive (OpenMP* API)

OpenMP* Fortran Compiler Directive: *Transforms a loop into a loop that will be executed concurrently using Single Instruction Multiple Data (SIMD) instructions.*

Syntax

```
!$OMP SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END SIMD]
```

clause — Is one of the following:

- ALIGNED (list [:n])
- [NO]ASSERT

Directs the compiler to assert (produce an error) or not to assert (produce a warning) when the vectorization fails. The default is NOASSERT. If this clause is specified more than once, a compile-time error occurs.

- COLLAPSE (n)
- EARLY_EXIT

  Allows vectorization of multiple exit loops. When this clause is specified the following occurs:

  - Each operation before the last lexical early exit of the loop may be executed as if the early exit were not triggered within the SIMD chunk.
  - After the last lexical early exit of the loop, all operations are executed as if the last iteration of the loop was found.
  - The last value for LINEARs and conditional LASTPRIVATEs are preserved with respect to scalar execution.
  - The last value for REDUCTIONs are computed as if the last iteration in the last SIMD chunk was executed upon exiting the loop.
  - The shared memory state may not be preserved with regard to scalar execution.
  - Exceptions are not allowed.

  When a SIMD loop is specified with the EARLY_EXIT clause, each list item specified in the LINEAR clause is computed based on the last iteration number upon exiting the loop.
- IF ([SIMD:] scalar-logical-expression)
- LASTPRIVATE ([CONDITIONAL:] list)
- LINEAR (var-list[: linear-step])
- NONTEMPORAL [(*var1* [, *var2*]...)])

  Directs the compiler to use non-temporal (that is, streaming) stores.

  By default, the compiler automatically determines whether a streaming store should be used for each variable.

  Streaming stores may cause significant performance improvements over non-streaming stores for large trip-count loops on certain processors. However, the misuse of streaming stores can significantly degrade performance.

  A variable cannot appear more than once in a NONTEMPORAL clause, and it cannot appear in more than one NONTEMPORAL clause.
- PRIVATE (list)
- REDUCTION (reduction-identifier : list)
- SAFELEN(*m*)

  Limits the number of iterations in a SIMD chunk (set of concurrent iterations).

  The *m* must be a constant positive integer expression; it indicates the number of iterations allowed in a SIMD chunk.

When this clause is used, no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than *m*.

The number of iterations that are executed concurrently at any given time is defined by the implementation. Each concurrent iteration is executed by a different SIMD vector lane.

At most one SAFELEN clause can appear in a SIMD directive.

- `SIMDLEN`(*n*)

  Specifies the preferred number of iterations to be executed concurrently. *n* must be a positive scalar integer constant. The number of iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane.

  If both SIMDLEN (*n*) and SAFELEN (*m*) are specified, the value of the *n* must be less than or equal to the value of *m*.

  At most one SIMDLEN clause can appear in a SIMD directive.

do-loop

Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

You cannot branch out of a DO loop associated with a SIMD directive.

A SIMD construct binds to the current task region. The binding thread set of the SIMD region is the current team.

If used, the END SIMD directive must appear immediately after the end of the loop. If you do not specify an END SIMD directive, an END SIMD directive is assumed at the end of *do-loop*.

The SIMD construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions. No other OpenMP* Fortran construct can appear in a SIMD directive.

A SIMD region binds to the current task region. The binding thread set of the SIMD region is the current team.

When any thread encounters a SIMD construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.

If an ORDERED directive with the SIMD clause is specified inside the SIMD region, the ordered regions encountered by any thread will use only a single SIMD lane to execute the ordered regions in the order of the loop iterations.

Example

The following is an example using the SIMD directive:

```
subroutine subr(a, b, c, n)
  implicit none
```

```
    real(kind=kind(0.0d0)),dimension(*) :: a, b, c
    integer                             :: n, i

    !$omp simd
      do i = 1, n
        a(i) = a(i) * b(i) + c(i)
      end do

   end subroutine
```

The following example demonstrates the EARLY_EXIT clause:

```
 !$omp simd early_exit
   do i = 1, n
      if (a(i) == i) exit
   enddo
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
qsimd-honor-fp-model, Qsimd-honor-fp-model compiler option
qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction compiler option
fp-model, fp compiler option
Nested DO Constructs
Parallel Processing Model for information about Binding Sets

SIMD Loop Directive

General Compiler Directive: *Requires and controls SIMD vectorization of loops.*

Syntax

`!DIR$ SIMD [`*clause*`[[,] `*clause*`]...]`

clause

Is an optional vectorization clause. It can be one or more of the following:

- `[NO]ASSERT`

  Directs the compiler to assert (produce an error) or not to assert (produce a warning) when the vectorization fails. The default is NOASSERT. If this clause is specified more than once, a compile-time error occurs.
- `FIRSTPRIVATE(`*list*`)`

  list

  Is a list of names of one or more variables or common blocks that are accessible to the scoping unit.

  Provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in a FIRSTPRIVATE list are subject to PRIVATE clause semantics. In addition, each variable has its initial value broadcast to all private instances for each iteration upon entering the SIMD loop.

A variable in a FIRSTPRIVATE clause can appear in a LASTPRIVATE clause.

A variable in a FIRSTPRIVATE clause cannot appear in a LINEAR, REDUCTION, or PRIVATE clause.

- `LASTPRIVATE` (*list*)

  list — Is a list of names of one or more variables or common blocks that are accessible to the scoping unit.

  Provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause semantics. In addition, when the SIMD loop is exited, each variable has the value that resulted from the sequentially last iteration of the SIMD loop (which may be undefined if the last iteration does not assign to the variable).

  A variable in a LASTPRIVATE clause can appear in a FIRSTPRIVATE clause.

  A variable in a LASTPRIVATE clause cannot appear in a LINEAR, REDUCTION, or PRIVATE clause.
- `LINEAR` (*var1*:*step1* [, *var2*:*step2*]...)

  var — Is a scalar variable.

  step — Is a positive, integer, scalar expression.

  For each iteration of a scalar loop, *var1* is incremented by *step1*, *var2* is incremented by *step2*, and so on. Therefore, every iteration of the vector loop increments the variables by VL (vector length)*step1, VL*step2, …, to VL*stepN, respectively. If more than one step is specified for a *var*, a compile-time error occurs. Multiple LINEAR clauses are merged as a union.

  A variable in a LINEAR clause cannot appear in a REDUCTION, PRIVATE, FIRSTPRIVATE, or LASTPRIVATE clause.
- `PRIVATE` (*list*)

  list — Is a list of names of one or more variables or common blocks that are accessible to the scoping unit. A variable that is part of another variable (for example, subobjects such as an array or structure element) cannot appear. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

  Causes each variable to be private to each iteration of a loop. Its initial and last values are undefined upon entering and exiting the SIMD loop. Multiple PRIVATE clauses are merged as a union.

  A variable in a PRIVATE clause cannot appear in a LINEAR, REDUCTION, FIRSTPRIVATE, or LASTPRIVATE clause.
- `REDUCTION` (*reduction-identifier* : *var1* [, *var2*]...)

reduction-identifier — Is a predefined reduction identifier (+, *, -, .AND., .OR., .EQV., or .NEQV.), or an accessible user-defined reduction identifier declared in a DECLARE REDUCTION directive.

var — Is a scalar variable.

Applies the vector reduction indicated by *reduction-identifier* to *var1*, *var2*, …, *varN*. A SIMD directive can have multiple REDUCTION clauses using the same or different operators. If more than one reduction operator is associated with a *var*, a compile-time error occurs.

A variable in a REDUCTION clause cannot appear in a LINEAR, PRIVATE, FIRSTPRIVATE, or LASTPRIVATE clause.

- [NO]VECREMAINDER
- VECTORLENGTH (*n1* [, *n2*]...)

n — Is a vector length (VL). It must be an integer that is a power of 2; the value must be 2, 4, 8, 16, 32 or 64. If you specify more than one *n*, the vectorizor will choose the VL from the values specified.

Causes each iteration in the vector loop to execute the computation equivalent to *n* iterations of scalar loop execution.

The VECTORLENGTH and VECTORLENGTHFOR clauses are mutually exclusive. You cannot use the VECTORLENGTH clause with the VECTORLENGTHFOR clause, and vice versa.

Multiple VECTORLENGTH clauses cause a syntax error.

- VECTORLENGTHFOR (*data-type*)

data-type — Is one of the following intrinsic data types:

Data Type	Fortran Intrinsic Type
INTEGER	Default INTEGER
INTEGER(1)	INTEGER (KIND=1)
INTEGER(2)	INTEGER (KIND=2)
INTEGER(4)	INTEGER (KIND=4)
INTEGER(8)	INTEGER (KIND=8)
REAL	Default REAL
REAL(4)	REAL (KIND=4)
REAL(8)	REAL (KIND=8)

Data Type	Fortran Intrinsic Type
COMPLEX	Default COMPLEX
COMPLEX(4)	COMPLEX (KIND=4)
COMPLEX(8)	COMPLEX (KIND=8)

Causes each iteration in the vector loop to execute the computation equivalent to *n* iterations of scalar loop execution where *n* is computed from `size_of_vector_register/ sizeof(data_type)`.

For example, VECTORLENGTHFOR (REAL (KIND=4)) results in *n*=4 for SSE2 to SSE4.2 targets (packed float operations available on 128-bit XMM registers) and *n*=8 for AVX target (packed float operations available on 256-bit YMM registers). VECTORLENGTHFOR(INTEGER (KIND=4)) results in *n*=4 for SSE2 to AVX targets.

The VECTORLENGTHFOR and VECTORLENGTH clauses are mutually exclusive. You cannot use the VECTORLENGTHFOR clause with the VECTORLENGTH clause, and vice versa.

Multiple VECTORLENGTHFOR clauses cause a syntax error.

Without explicit VECTORLENGTH and VECTORLENGTHFOR clauses, the compiler will choose a VECTORLENGTH using its own cost model. Misclassification of variables into PRIVATE, FIRSTPRIVATE, LASTPRIVATE, LINEAR, and REDUCTION, or the lack of appropriate classification of variables, may lead to unintended consequences such as runtime failures and/or incorrect results.

If you specify the SIMD directive with no clause, default rules are in effect for variable attributes, vector length, and so forth.

If you do not explicitly specify a VECTORLENGTH clause, the compiler will choose a VECTORLENGTH using its own cost model. Misclassification of variables into PRIVATE, FIRSTPRIVATE, LASTPRIVATE, LINEAR, and REDUCTION, or the lack of appropriate classification of variables, may lead to unintended consequences such as runtime failures and/or incorrect results.

You can only specify a particular variable in at most one instance of a PRIVATE, LINEAR, or REDUCTION clause.

If the compiler is unable to vectorize a loop, a warning occurs by default. However, if ASSERT is specified, an error occurs instead.

If the vectorizer has to stop vectorizing a loop for some reason, the fast floating-point model is used for the SIMD loop.

A SIMD loop may contain one or more nested loops or be contained in a loop nest. Only the loop preceded by the SIMD directive is processed for SIMD vectorization.

The vectorization performed on this loop by the SIMD directive overrides any setting you may specify for options `-fp-model` (Linux* and macOS*) and `/fp` (Windows*) for this loop.

Note that the SIMD directive may not affect all auto-vectorizable loops. Some of these loops do not have a way to describe the SIMD vector semantics.

The following restrictions apply to the SIMD directive:

- The countable loop for the SIMD directive has to conform to the DO-loop style of an OpenMP worksharing loop construct. Additionally, the loop control variable must be a signed integer type.
- The vector values must be signed 8-, 16-, 32-, or 64-bit integers, single or double-precision floating-point numbers, or single- or double-precision complex numbers.
- A SIMD directive loop performs memory references unconditionally. Therefore, all address computations must result in valid memory addresses, even though such locations may not be accessed if the loop is executed sequentially.

To disable the SIMD transformations for vectorization, specify option `-no-simd` (Linux* and macOS*) or `/Qsimd-` (Windows*).

To disable transformations that enable more vectorization, specify options `-no-vec-no-simd` (Linux and macOS*) or `/Qvec-/Qsimd-` (Windows).

Example

Consider the following:

```
  ...
subroutine add(A, N, X)
   integer N, X
   real    A(N)
cDIR$ SIMD
   DO I=X+1, N
      A(I) = A(I) + A(I-X)
   ENDDO
end
  ...
```

When the program containing this subroutine is compiled, the DO loop will be vectorized.

Because no optional clause was specified for the directive, default rules will apply for variable attributes, vector length, and so forth.

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
simd, Qsimd compiler option
qsimd-honor-fp-model, Qsimd-honor-fp-model compiler option
qsimd-serialize-fp-reduction, Qsimd-serialize-fp-reduction compiler option
fp-model, fp compiler option
vec, Qvec compiler option

SIN

Elemental Intrinsic Function (Generic): *Produces the sine of an argument.*

Syntax

`result = SIN (x)`

x	(Input) Must be of type real or complex. It must be in radians and is treated as modulo 2*pi.

Results

The result type and kind are the same as *x*.

If *x* is of type real, the result is a value in radians.

If *x* is of type complex, the real part of the result is a value in radians.

Specific Name	Argument Type	Result Type
SIN	REAL(4)	REAL(4)
DSIN	REAL(8)	REAL(8)
QSIN	REAL(16)	REAL(16)
CSIN [1]	COMPLEX(4)	COMPLEX(4)
CDSIN [2]	COMPLEX(8)	COMPLEX(8)
CQSIN	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CSIN.

[2]This function can also be specified as ZSIN.

Example

SIN (2.0) has the value 0.9092974.

SIN (0.8) has the value 0.7173561.

SIND

Elemental Intrinsic Function (Generic): *Produces the sine of an argument.*

Syntax

```
result = SIND (x)
```

x	(Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type and kind are the same as *x*.

Specific Name	Argument Type	Result Type
SIND	REAL(4)	REAL(4)
DSIND	REAL(8)	REAL(8)
QSIND	REAL(16)	REAL(16)

Example

SIND (2.0) has the value 3.4899496E-02.

SIND (0.8) has the value 1.3962180E-02.

SINGLE

OpenMP* Fortran Compiler Directive: *Specifies that a block of code is to be executed by only one thread in the team at a time.*

Syntax

```
!$OMP SINGLE [clause[[,] clause] ... ]
   block
!$OMP END SINGLE [modifier]
```

clause	Is one of the following: • FIRSTPRIVATE (list) • PRIVATE (list)
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.
modifier	Is one of the following: • COPYPRIVATE (list) • NOWAIT

The binding thread set for a SINGLE construct is the current team. A SINGLE region binds to the innermost enclosing parallel region.

Threads in the team that are not executing this directive wait at the END SINGLE directive unless NOWAIT is specified.

SINGLE directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

In the following example, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT:

```
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE
      CALL WORK(Y)
!$OMP END PARALLEL
```

You should not make assumptions as to which thread executes the SINGLE section. All other threads skip the SINGLE section and stop at the barrier at the END SINGLE construct. If other threads can proceed without waiting for the thread executing the SINGLE section, you can specify NOWAIT in the END SINGLE directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

SINH

Elemental Intrinsic Function (Generic): *Produces a hyperbolic sine.*

Syntax

```
result = SINH (x)
```

x	(Input) Must be of type real or complex.

Results

The result type and kind are the same as *x*.

If *x* is of type complex, the imaginary part of the result is in radians.

Specific Name	Argument Type	Result Type
SINH	REAL(4)	REAL(4)
DSINH	REAL(8)	REAL(8)
QSINH	REAL(16)	REAL(16)

Example

SINH (2.0) has the value 3.626860.

SINH (0.8) has the value 0.8881060.

SIZE Function

Inquiry Intrinsic Function (Generic): *Returns the total number of elements in an array, or the extent of an array along a specified dimension.*

Syntax

```
result = SIZE (array [, dim] [, kind])
```

array	(Input) Must be an array; it can be assumed-rank. It can be of any data type. It must not be a disassociated pointer or an allocatable array that is not allocated. It can be an assumed-size array if *dim* is present with a value less than the rank of *array*.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

In general, if *dim* is present, the result is the extent of dimension *dim* in *array*; otherwise, the result is the total number of elements in *array*. However, the following exceptions apply:

- If array is assumed-rank and associated with an assumed-size array and *dim* is present with a value equal to the rank of ARRAY, the result has a value of -1.
- If *dim* is absent and *array* is assumed-rank, the result has a value equal to PRODUCT(SHAPE(ARRAY, KIND)).

The setting of compiler options specifying integer size can affect this function.

Example

If B is declared as B(2:4, -3:1), then SIZE (B, DIM=2) has the value 5 and SIZE (B) has the value 15.

The following shows another example:

```
REAL(8) array (3:10, -1:3)
INTEGER i
i = SIZE(array, DIM = 2) ! returns 5
i = SIZE(array)          ! returns 40
```

See Also

SHAPE
Character Count Specifier

SIZEOF

Inquiry Intrinsic Function (Generic): *Returns the number of bytes of storage used by the argument. It cannot be passed as an actual argument.*

Syntax

`result = SIZEOF (x)`

x	(Input) Can be a scalar or array. It may be of any data type. It must *not* be an assumed-size array.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The result value is the number of bytes of storage used by *x*. If *x* is of derived type, the result includes storage used by padding and descriptors for pointer or allocatable components, if any, but not the storage used for the data of pointer or allocatable components.

Example

```
SIZEOF (3.44)              ! has the value 4

SIZEOF ('SIZE')            ! has the value 4
```

See Also

C_SIZEOF

SLEEP

Portability Subroutine: *Suspends the execution of a process for a specified interval.*

Module

USE IFPORT

Syntax

CALL SLEEP (*time*)

time	(Input) INTEGER(4). Length of time, in seconds, to suspend the calling process.

Example

```
USE IFPORT
integer(4) hold_time
hold_time = 1    ! lets the loop execute
DO WHILE (hold_time .NE. 0)
  write(*,'(A)') "Enter the number of seconds to suspend"
  read(*,*) hold_time
  CALL SLEEP (hold_time)
END DO
END
```

See Also

SLEEPQQ

SLEEPQQ

Portability Subroutine: *Delays execution of the program for a specified duration.*

Module

USE IFPORT

Syntax

CALL SLEEPQQ (*duration*)

duration	(Input) INTEGER(4). Number of milliseconds the program is to sleep (delay program execution).

Example

```
USE IFPORT
INTEGER(4) delay, freq, duration
delay    = 2000
freq     = 4000
duration = 1000
CALL SLEEPQQ(delay)
CALL BEEPQQ(freq, duration)
END
```

SNGL

Converts a REAL value to a default REAL result.

See Also

See REAL function.

SORTQQ

Portability Subroutine: *Sorts a one-dimensional array. The array elements cannot be derived types or record structures.*

Module

USE IFPORT

Syntax

CALL SORTQQ (*adrarray*,*count*,*size*)

adrarray

(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Address of the array (returned by LOC).

count

(Input; output) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. On input, number of elements in the array to be sorted. On output, number of elements actually sorted.

To be certain that SORTQQ is successful, compare the value returned in *count* to the value you provided. If they are the same, then SORTQQ sorted the correct number of elements.

size

(Input) INTEGER(4). Positive constant less than 32,767 that specifies the kind of array to be sorted. The following constants, defined in IFPORT.F90, specify type and kind for numeric arrays:

Constant	Type of array
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) or equivalent
SRT$INTEGER4	INTEGER(4) or equivalent
SRT$INTEGER8	INTEGER(8) or equivalent
SRT$REAL4	REAL(4) or equivalent
SRT$REAL8	REAL(8) or equivalent
SRT$REAL16	REAL(16) or equivalent

If the value provided in *size* is not a symbolic constant and is less than 32,767, the array is assumed to be a character array with *size* characters per element.

Caution

The location of the array must be passed by address using the LOC function. This defeats Fortran type-checking, so you must make certain that the *count* and *size* arguments are correct.

If you pass invalid arguments, SORTQQ attempts to sort random parts of memory. If the memory it attempts to sort is allocated to the current process, that memory is sorted; otherwise, the operating system intervenes, the program is halted, and you get a General Protection Violation message.

Example

```
!    Sort a 1-D array
! USE IFPORT
INTEGER(2) array(10)
INTEGER(2) i DATA ARRAY /143, 99, 612, 61, 712, 9112, 6, 555, 2223, 67/
!    Sort the array
Call SORTQQ (LOC(array), 10, SRT$INTEGER2)
!    Display the sorted array
DO i = 1, 10
  WRITE (*, 9000) i, array (i) 9000 FORMAT(1X, ' Array(',I2, '): ', I5)
END DO
END
```

See Also

BSEARCHQQ
LOC

SPACING

Elemental Intrinsic Function (Generic): *Returns the absolute spacing of model numbers near the argument value.*

Syntax

`result = SPACING (x)`

x	(Input) Must be of type real.

Results

The result type and kind are the same as *x*. The result has the value b^{e-p}. Parameters *b*, *e*, and *p* are defined in Model for Real Data. If the result value is outside of the real model range, the result is TINY(*x*).

Example

If 3.0 is a REAL(4) value, SPACING (3.0) has the value 2^{-22}.

The following shows another example:

```
REAL(4) res4
REAL(8) res8, r2
res4 = SPACING(3.0)   ! returns 2.384186E-07
res4 = SPACING(-3.0)  ! returns 2.384186E-07
r2   = 487923.3
res8 = SPACING(r2)    ! returns 5.820766091346741E-011
```

See Also

TINY
RRSPACING
Data Representation Models

SPLITPATHQQ

Portability Function: *Breaks a file path or directory path into its components.*

Module

USE IFPORT

Syntax

`result = SPLITPATHQQ (path,drive,dir,name,ext)`

path	(Input) Character*(*). Path to be broken into components. Forward slashes (/), backslashes (\), or both can be present in *path*.
drive	(Output) Character*(*). Drive letter followed by a colon.
dir	(Output) Character*(*). Path of directories, including the trailing slash.
name	(Output) Character*(*). Name of file or, if no file is specified in *path*, name of the lowest directory. A file name must not include an extension.
ext	(Output) Character*(*). File name extension, if any, including the leading period (.).

Results

The result type is INTEGER(4). The result is the length of *dir*.

The *path* parameter can be a complete or partial file specification.

$MAXPATH is a symbolic constant defined in module `IFPORT.F90` as 260.

Example

```
USE IFPORT
CHARACTER($MAXPATH) buf
CHARACTER(3)        drive
CHARACTER(256)      dir
CHARACTER(256)      name
CHARACTER(256)      ext
CHARACTER(256)      file

INTEGER(4)          length

buf = 'b:\fortran\test\runtime\tsplit.for'
length = SPLITPATHQQ(buf, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext
file = 'partial.f90'
length = SPLITPATHQQ(file, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext

END
```

See Also

FULLPATHQQ

SPORT_CANCEL_IO (W*S)

Serial Port I/O Function: *Cancels any I/O in progress to the specified port.*

Module

USE IFPORT

Syntax

```
result = SPORT_CANCEL_IO (port)
```

port (Input) Integer. The port number.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

> **NOTE**
> This call also kills the thread that keeps an outstanding read operation to the serial port. This call *must* be done before any of the port characteristics are modified.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult = SPORT_CANCEL_IO( 2 )
END
```

See Also

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_CONNECT (W*S)

Serial Port I/O Function: *Establishes the connection to a serial port and defines certain usage parameters.*

Module

USE IFPORT

Syntax

```
result = SPORT_CONNECT (port [,options])
```

port (Input) Integer. The port number of connection. The routine will open COM *n*, where *n* is the port number specified.

options (Input; optional) Integer. Defines the connection options. These options define how the *nnn*_LINE routines will work and also effect the data that is passed to the user. If more than one option is specified, the operator .OR. should be used between each option. Options are as follows:

Option	Description
DL_TOSS_CR	Removes carriage return (CR) characters on input.
DL_TOSS_LF	Removes linefeed (LF) characters on input.

Option	Description
DL_OUT_CR	Causes SPORT_WRITE_LINE to add a CR to each record written.
DL_OUT_LF	Causes SPORT_WRITE_LINE to add a LF to each record written.
DL_TERM_CR	Causes SPORT_READ_LINE to terminate READ when a CR is encountered.
DL_TERM_LF	Causes SPORT_READ_LINE to terminate READ when a LF is encountered.
DL_TERM_CRLF	Causes SPORT_READ_LINE to terminate READ when CR+LF is encountered.

If *options* is not specified, the following occurs by default:

```
(DL_OUT_CR .OR. DL_TERM_CR .OR. DL_TOSS_CR .OR. DL_TOSS_LF)
```

This specifies to remove carriage returns and linefeeds on input, to follow output lines with a carriage return, and to return input lines when a carriage return is encountered.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_CONNECT( 2 )
END
```

See Also

SPORT_RELEASE

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_CONNECT_EX (W*S)

Serial Port I/O Function: *Establishes the connection to a serial port, defines certain usage parameters, and defines the size of the internal buffer for data reception.*

Module

USE IFPORT

Syntax

```
result = SPORT_CONNECT_EX (port [,options] [,BufferSize])
```

port	(Input) Integer. The port number of connection. The routine will open COM *n*, where *n* is the port number specified.

options

(Input; optional) Integer. Defines the connection options. These options define how the *nnn*_LINE routines will work and also effect the data that is passed to the user. If more than one option is specified, the operator .OR. should be used between each option. Options are as follows:

Option	Description
DL_TOSS_CR	Removes carriage return (CR) characters on input.
DL_TOSS_LF	Removes linefeed (LF) characters on input.
DL_OUT_CR	Causes SPORT_WRITE_LINE to add a CR to each record written.
DL_OUT_LF	Causes SPORT_WRITE_LINE to add a LF to each record written.
DL_TERM_CR	Causes SPORT_READ_LINE to terminate READ when a CR is encountered.
DL_TERM_LF	Causes SPORT_READ_LINE to terminate READ when a LF is encountered.
DL_TERM_CRLF	Causes SPORT_READ_LINE to terminate READ when CR+LF is encountered.

If *options* is not specified, the following occurs by default:

```
(DL_OUT_CR .OR. DL_TERM_CR .OR. DL_TOSS_CR .OR. DL_TOSS_LF)
```

This specifies to remove carriage returns and linefeeds on input, to follow output lines with a carriage return, and to return input lines when a carriage return is encountered.

BufferSize

(Input; optional) Integer. Size of the internal buffer for data reception. If *BufferSize* is not specified, the size of the buffer is 16384 bytes (the default).

The size of the buffer must be 4096 bytes or larger. If you try to specify a size smaller than 4096 bytes, your specification will be ignored and the buffer size will be set to 4096 bytes.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_CONNECT_EX( 2, BufferSize = 8196 )
END
```

See Also

SPORT_CONNECT

SPORT_RELEASE

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_GET_HANDLE (W*S)

Serial Port I/O Function: *Returns the Windows* handle associated with the communications port. This is the handle that was returned by the Windows API CreateFile.*

Module

USE IFPORT

Syntax

`result = SPORT_GET_HANDLE (port,handle)`

port	(Input) Integer. The port number.
handle	(Output) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. This is the Windows handle that was returned from CreatFile( ) on the serial port.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows error value.

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER(KIND=INT_PTR_KIND( )) handle
iresult = SPORT_GET_HANDLE( 2, handle )
END
```

See Also

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_GET_STATE (W*S)

Serial Port I/O Function: *Returns the baud rate, parity, data bits setting, and stop bits setting of the communications port.*

Module

USE IFPORT

Syntax

`result = SPORT_GET_STATE (port [,baud] [,parity] [,dbits] [,sbits])`

port	(Input) Integer. The port number.
baud	(Output; optional) Integer. The baud rate of the port.
parity	(Output; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Output; optional) Integer. The data bits for the port.

sbits	(Output; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER    baud
INTEGER    parity
INTEGER    dbits
INTEGER    sbits

iresult =  SPORT_GET_STATE( 2, baud, parity, dbits, sbits )
END
```

See Also

SPORT_SET_STATE

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_GET_STATE_EX (W*S)

Serial Port I/O Function: *Returns the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.*

Module

USE IFPORT

Syntax

```
result = SPORT_GET_STATE_EX (port[,baud] [,parity] [,dbits] [,sbits] [,Binmode]
[,DTRcntrl]
   [,RTScntrl] [,OutCTSFlow] [,OutDSRFlow] [,DSRSense] [,OutXonOff] [,InXonOff] [,XonLim]
   [,XoffLim] [,TXContOnXoff] [,ErrAbort] [,ErrCharEnbl] [,NullStrip] [,XonChar]
[,XoffChar]
   [,ErrChar] [,EofChar] [,EvtChar])
```

port	(Input) Integer. The port number.
baud	(Input; optional) Integer. The baud rate of the port.
parity	(Output; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Output; optional) Integer. The data bits for the port.
sbits	(Output; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).
Binmode	(Output; optional) Integer. 1 if binary mode is enabled; otherwise, 0. Currently, the value of this parameter is always 1.

DTRcntrl	(Output; optional) Integer. 1 if DTR (data-terminal-ready) flow control is used; otherwise, 0.
RTScntrl	(Output; optional) Integer. 1 if RTS (request-to-send) flow control is used; otherwise, 0.
OutCTSFlow	(Output; optional) Integer. 1 if the CTS (clear-to-send) signal is monitored for output flow control; otherwise, 0.
OutDSRFlow	(Output; optional) Integer. 1 if the DSR (data-set-ready) signal is monitored for output flow control; otherwise, 0.
DSRSense	(Output; optional) Integer. 1 if the communications driver is sensitive to the state of the DSR signal; otherwise, 0.
OutXonOff	(Output; optional) Integer. 1 if XON/XOFF flow control is used during transmission; otherwise, 0.
InXonOff	(Output; optional) Integer. 1 if XON/XOFF flow control is used during reception; otherwise, 0.
XonLim	(Output; optional) Integer. The minimum number of bytes accepted in the input buffer before the XON character is set.
XoffLim	(Output; optional) Integer. The maximum number of bytes accepted in the input buffer before the XOFF character is set.
TXContOnXoff	(Output; optional) Integer. 1 if transmission stops when the input buffer is full and the driver has transmitted the *XoffChar* character; otherwise, 0.
ErrAbort	(Output; optional) Integer. 1 if read and write operations are terminated when an error occurs; otherwise, 0.
ErrCharEnbl	(Output; optional) Integer. 1 if bytes received with parity errors are replaced with the *ErrChar* character; otherwise, 0.
NullStrip	(Output; optional) Integer. 1 if null bytes are discarded; otherwise, 0.
XonChar	(Output; optional) Character. The value of the XON character that is used for both transmission and reception.
XoffChar	(Output; optional) Character. The value of the XOFF character that is used for both transmission and reception.
ErrChar	(Output; optional) Character. The value of the character that is used to replace bytes received with parity errors.
EofChar	(Output; optional) Character. The value of the character that is used to signal the end of data.
EvtChar	Output; optional) Character. The value of the character that is used to signal an event.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER(4) port, baud, parity, dbits, sbits
INTEGER(4) OutXonOff, InXonOff, OutDSRFlow
INTEGER(4) OutCTSFlow, DTRcntrl,  RTScntrl
INTEGER(4) DSRSense, XonLim, XoffLim
CHARACTER(1) XonChar, XoffChar
iresult = SPORT_GET_STATE_EX(port, baud, parity, dbits, sbits,               &
              OutXonOff=OutXonOff, InXonOff=InXonOff, OutDSRFlow=OutDSRFlow, &
              OutCTSFlow=OutCTSFlow, DTRcntrl=DTRcntrl, RTScntrl=RTScntrl,   &
              DSRSense = DSRSense, XonChar = XonChar, XoffChar = XoffChar,   &
              XonLim=XonLim, XoffLim=XoffLim)
END
```

See Also

SPORT_GET_STATE
SPORT_SET_STATE_EX

Communications, Communications Functions, and SetCommState in the Microsoft* Platform SDK

SPORT_GET_TIMEOUTS (W*S)

Serial Port I/O Function: *Returns the user selectable timeouts for the serial port.*

Module

USE IFPORT

Syntax

`result = SPORT_GET_TIMEOUTS (port [,rx_int] [,tx_tot_mult] [,tx_tot_const])`

port	(Input) Integer. The port number.
rx_int	(Output; optional) INTEGER(4). The receive interval timeout value.
tx_tot_mult	(Output; optional) INTEGER(4). The transmit multiplier part of the timeout value.
tx_tot_const	(Output; optional) INTEGER(4). The transmit constant part of the timeout value.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER*4  rx_int
INTEGER*4  tx_tot_mult
INTEGER*4  tx_tot_const

iresult =  SPORT_GET_TIMEOUTS( 2, rx_int, tx_tot_mult, tx_tot_const )
END
```

See Also

SPORT_SET_TIMEOUTS

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_PEEK_DATA (W*S)

Serial Port I/O Function: *Returns information about the availability of input data.*

Module

USE IFPORT

Syntax

```
result = SPORT_PEEK_DATA (port [,present] [,count])
```

port	(Input) Integer. The port number.
present	(Output; optional) Integer. 1 if data is present, 0 if no data has been read.
count	(Output; optional) Integer. The count of characters that will be returned by SPORT_READ_DATA.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

> **NOTE**
> CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER    present
INTEGER    count

iresult =  SPORT_PEEK_DATA( 2, present, count )
END
```

See Also

SPORT_CONNECT
SPORT_READ_DATA
SPORT_PEEK_LINE

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_PEEK_LINE (W*S)

Serial Port I/O Function: *Returns information about the availability of input records.*

Module

USE IFPORT

Syntax

```
result = SPORT_PEEK_LINE (port [,present] [,count])
```

port	(Input) Integer. The port number.
present	(Output; optional) Integer. 1 if data is present, 0 if no data has been read.
count	(Output; optional) Integer. The count of characters that will be returned by SPORT_READ_DATA.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

This routine will only return when a line terminator has been seen - as defined by the mode specified in the SPORT_CONNECT( ) call.

NOTE
CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER    present
INTEGER    count

iresult =  SPORT_PEEK_LINE( 2, present, count )
END
```

See Also

SPORT_CONNECT
SPORT_READ_DATA
SPORT_PEEK_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_PURGE (W*S)

Serial Port I/O Function: *Executes the Windows* API communications function PurgeComm on the specified port.*

Module

USE IFPORT

Syntax

```
result = SPORT_PURGE (port,function)
```

port	(Input) Integer. The port number.
function	(Input) INTEGER(4). The function for PurgeComm (see the Windows* documentation).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows error value.

Example

```
USE IFWINTY
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_PURGE( 2, (PURGE_TXABORT .or. PURGE_RXABORT) )
END
```

See Also

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_READ_DATA (W*S)

Serial Port I/O Function: *Reads available data from the specified port. This routine stalls until at least one character has been read.*

Module

USE IFPORT

Syntax

`result = SPORT_READ_DATA (port,buffer[,count])`

port	(Input) Integer. The port number.
buffer	(Output) Character*(*). The data that was read.
count	(Output; optional) Integer. The count of bytes read.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

> **NOTE**
> CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Example

```
USE IFPORT
INTEGER(4)      iresult
INTEGER         count
CHARACTER*1024  rbuff

iresult = SPORT_READ_DATA( 2, rbuff, count )
END
```

See Also

SPORT_CONNECT
SPORT_PEEK_DATA
SPORT_READ_LINE
SPORT_WRITE_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_READ_LINE (W*S)

Serial Port I/O Function: *Reads a record from the specified port. This routine stalls until at least one record has been read.*

Module

USE IFPORT

Syntax

```
result = SPORT_READ_LINE (port,buffer[, count])
```

port	(Input) Integer. The port number.
buffer	(Output) Character*(*). The data that was read.
count	(Output; optional) Integer. The count of bytes read.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

This routine will only return when a line terminator has been seen - as defined by the mode specified in the SPORT_CONNECT( ) call.

> **NOTE**
> CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Example

```
USE IFPORT
INTEGER(4)      iresult
INTEGER         count
CHARACTER*1024  rbuff

iresult = SPORT_READ_LINE( 2, rbuff, count )
END
```

See Also

SPORT_CONNECT
SPORT_PEEK_LINE
SPORT_READ_DATA
SPORT_WRITE_LINE

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_RELEASE (W*S)

Serial Port I/O Function: *Releases a serial port that was previously connected when SPORT_CONNECT was specified.*

Module

USE IFPORT

Syntax

```
result = SPORT_RELEASE (port)
```

port	(Input) Integer. The port number.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_RELEASE( 2 )
END
```

See Also

SPORT_CONNECT

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_SET_STATE (W*S)

Serial Port I/O Function: *Sets the baud rate, parity, data bits setting, and stop bits setting of the communications port.*

Module

USE IFPORT

Syntax

```
result = SPORT_SET_STATE (port [,baud] [,parity] [,dbits] [,sbits])
```

port	(Input) Integer. The port number.
baud	(Input; optional) Integer. The baud rate of the port.
parity	(Input; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Input; optional) Integer. The data bits for the port.
sbits	(Input; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

The following restrictions apply:

- The number of data bits must be 5 to 8 bits.
- The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

NOTE
This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_SET_STATE( 2, 9600, 0, 7, 0 )
END
```

See Also

SPORT_CANCEL_IO
SPORT_GET_STATE

Communications, Communications Functions, and SetCommState in the Microsoft* Platform SDK

SPORT_SET_STATE_EX (W*S)

Serial Port I/O Function: *Sets the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.*

Module

USE IFPORT

Syntax

```
result = SPORT_SET_STATE_EX (port [,baud] [,parity] [,dbits] [,sbits] [,Binmode]
[,DTRcntrl]
   [,RTScntrl] [,OutCTSFlow] [,OutDSRFlow] [,DSRSense] [,OutXonOff] [,InXonOff] [,XonLim]
   [,XoffLim] [,TXContOnXoff] [,ErrAbort] [,ErrCharEnbl] [,NullStrip] [,XonChar]
[,XoffChar]
   [,ErrChar] [,EofChar] [,EvtChar] [,fZeroDCB])
```

port	(Input) Integer. The port number.
baud	(Input; optional) Integer. The baud rate of the port.
parity	(Input; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Input; optional) Integer. The data bits for the port.
sbits	(Input; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).
Binmode	(Input; optional) Integer. 1 if binary mode should be enabled; otherwise, 0. Currently, if this parameter is used, the value must be 1.
DTRcntrl	(Input; optional) Integer. 1 if DTR (data-terminal-ready) flow control should be used; otherwise, 0.
RTScntrl	(Input; optional) Integer. 1 if RTS (request-to-send) flow control should be used; otherwise, 0.

OutCTSFlow	(Input; optional) Integer. 1 if the CTS (clear-to-send) signal should be monitored for output flow control; otherwise, 0.
OutDSRFlow	(Input; optional) Integer. 1 if the DSR (data-set-ready) signal should be monitored for output flow control; otherwise, 0.
DSRSense	(Input; optional) Integer. 1 if the communications driver should be sensitive to the state of the DSR signal; otherwise, 0.
OutXonOff	(Input; optional) Integer. 1 if XON/XOFF flow control should be used during transmission; otherwise, 0.
InXonOff	(Input; optional) Integer. 1 if XON/XOFF flow control should be used during reception; otherwise, 0.
XonLim	(Input; optional) Integer. The minimum number of bytes that should be accepted in the input buffer before the XON character is set.
XoffLim	(Input; optional) Integer. The maximum number of bytes that should be accepted in the input buffer before the XOFF character is set.
TXContOnXoff	(Input; optional) Integer. 1 if transmission should be stopped when the input buffer is full and the driver has transmitted the *XoffChar* character; otherwise, 0.
ErrAbort	(Input; optional) Integer. 1 if read and write operations should be terminated when an error occurs; otherwise, 0.
ErrCharEnbl	(Input; optional) Integer. 1 if bytes received with parity errors should be replaced with the *ErrChar* character; otherwise, 0.
NullStrip	(Input; optional) Integer. 1 if null bytes should be discarded; otherwise, 0.
XonChar	(Input; optional) Character. The value of the XON character that should be used for both transmission and reception.
XoffChar	(Input; optional) Character. The value of the XOFF character that should be used for both transmission and reception.
ErrChar	(Input; optional) Character. The value of the character that should be used to replace bytes received with parity errors.
EofChar	(Input; optional) Character. The value of the character that should be used to signal the end of data.
EvtChar	(Input; optional) Character. The value of the character that should be used to signal an event.
fZeroDCB	(Input; optional) Integer. 1 if all settings of the communications port should be set to zero before parameters are set; otherwise, 0.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

The following restrictions apply:

- The number of data bits must be 5 to 8 bits.
- The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

NOTE
This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_SET_STATE_EX( 2, 9600, 0, 7, 1, OutXonOff=1, InXonOff=1,    &
              XonLim=1024, XoffLim=512, XonChar=CHAR(17), XoffChar=CHAR(19), &
              fZeroDCB=1) )
END
```

See Also

SPORT_CANCEL_IO
SPORT_GET_STATE
SPORT_SET_STATE

Communications, Communications Functions, and SetCommState in the Microsoft* Platform SDK

SPORT_SET_TIMEOUTS (W*S)

Serial Port I/O Function: *Sets the user selectable timeouts for the serial port.*

Module

USE IFPORT

Syntax

`result = SPORT_SET_TIMEOUTS (port [,rx_int] [,tx_tot_mult] [,tx_tot_const])`

port	(Input) Integer. The port number.
rx_int	(Input; optional) INTEGER(4). The receive interval timeout value.
tx_tot_mult	(Input; optional) INTEGER(4). The transmit multiplier part of the timeout value.
tx_tot_const	(Input; optional) INTEGER(4). The transmit constant part of the timeout value.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE
This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_SET_TIMEOUTS( 2, 100, 0, 1000 )
END
```

See Also

SPORT_CANCEL_IO
SPORT_GET_TIMEOUTS

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_SHOW_STATE (W*S)

Serial Port I/O Function: *Displays the state of a port to standard output.*

Module

USE IFPORT

Syntax

`result = SPORT_SHOW_STATE (port,level)`

port (Input) Integer. The port number.

level (Input) Integer. Controls the level of detail displayed as follows:

0	Basic one line display
1	Basic information
2	Add modem signal control flow information
3	Add XON/XOFF information
4	Add event character information
11	Add timeout information
901	Add debug information

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE
This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_SHOW_STATE( 2, 0 )
END
```

See Also

SPORT_CANCEL_IO

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_SPECIAL_FUNC (W*S)

Serial Port I/O Function: *Executes the Windows* API communications function EscapeCommFunction on the specified port.*

Module

USE IFPORT

Syntax

`result = SPORT_SPECIAL_FUNC (port,function)`

port	(Input) Integer. The port number.
function	(Input) INTEGER(4). The function to perform.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, the result is a Windows* error value.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_SPECIAL_FUNC( 2, ? )
END
```

See Also

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_WRITE_DATA (W*S)

Serial Port I/O Function: *Outputs data to the specified port.*

Module

USE IFPORT

Syntax

`result = SPORT_WRITE_DATA (port,data[,count])`

port	(Input) Integer. The port number.
data	(Input) Character*(*). The data to be output.
count	(Input; optional) Integer. The count of bytes to write. If the value is zero, this number is computed by scanning the data backwards looking for a non-blank character.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE
When hardware (DTR, RTS, etc.) or software (XON/XOFF) flow controls are used, the functions SPORT_WRITE_DATA and SPORT_WRITE_LINE can write less bytes than required. When this occurs, the functions return the code ERROR_IO_INCOMPLETE, and the return value of parameter *count* contains the number of bytes that were really written.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_WRITE_DATA( 2, 'ATZ'//CHAR(13), 0 )
END
```

See Also

SPORT_WRITE_LINE
SPORT_READ_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_WRITE_LINE (W*S)

Serial Port I/O Function: *Outputs data, followed by a record terminator, to the specified port.*

Module

USE IFPORT

Syntax

`result = SPORT_WRITE_LINE (port,data[,count])`

port	(Input) Integer. The port number.
data	(Input) Character*(*). The data to be output.
count	(Input; optional) Integer. The count of bytes to write. If the value is zero, this number is computed by scanning the data backwards looking for a non-blank character.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

After the data is output, a line terminator character is added based on the mode used during the SPORT_CONNECT( ) call.

NOTE
When hardware (DTR, RTS, etc.) or software (XON/XOFF) flow controls are used, the functions SPORT_WRITE_DATA and SPORT_WRITE_LINE can write less bytes than required. When this occurs, the functions return the code ERROR_IO_INCOMPLETE, and the return value of parameter *count* contains the number of bytes that were really written.

Example

```
USE IFPORT
INTEGER(4) iresult
iresult = SPORT_WRITE_LINE( 2, 'ATZ', 0 )
END
```

See Also

SPORT_CONNECT
SPORT_WRITE_DATA
SPORT_READ_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

SPREAD

Transformational Intrinsic Function (Generic):
Creates a replicated array with an added dimension by making copies of existing elements along a specified dimension.

Syntax

`result = SPREAD (source,dim,ncopies)`

source	(Input) Must be a scalar or array. It may be of any data type. The rank must be less than 31.
dim	(Input) Must be scalar and of type integer. It must have a value in the range 1 to n + 1 (inclusive), where *n* is the rank of *source*.
ncopies	Must be scalar and of type integer. It becomes the extent of the additional dimension in the result.

Results

The result is an array of the same type as *source* and of rank that is one greater than *source*.

If *source* is an array, each array element in dimension *dim* of the result is equal to the corresponding array element in *source*.

If *source* is a scalar, the result is a rank-one array with *ncopies* elements, each with the value *source*.

If *ncopies* less than or equal to zero, the result is an array of size zero.

Example

SPREAD ("B", 1, 4) is the character array (/"B", "B", "B", "B"/).

B is the array [3, 4, 5] and NC has the value 4.

SPREAD (B, DIM=1, NCOPIES=NC) produces the array

```
 [ 3  4  5 ]
 [ 3  4  5 ]
 [ 3  4  5 ]
 [ 3  4  5 ].
```

SPREAD (B, DIM=2, NCOPIES=NC) produces the array

```
 [3  3  3  3 ]
 [4  4  4  4 ]
 [5  5  5  5 ].
```

The following shows another example:

```
INTEGER AR1(2, 3), AR2(3, 2)
AR1 = SPREAD((/1,2,3/),DIM= 1,NCOPIES= 2) ! returns
                                          ! 1 2 3
                                          ! 1 2 3

AR2 = SPREAD((/1,2,3/), 2, 2)   ! returns   1 1
                                !           2 2
                                !           3 3
```

See Also

PACK
RESHAPE

SQRT

Elemental Intrinsic Function (Generic): *Produces the square root of its argument.*

Syntax

`result = SQRT (x)`

x	(Input) must be of type real or complex. If *x* is type real, its value must be greater than or equal to zero.

Results

The result type and kind are the same as *x*. The result has a value equal to the square root of *x*.

A result of type complex is the principal value, with the real part greater than or equal to zero.

When the real part of the result is zero, the imaginary part of the result has the same sign as the imaginary part of *x* , even if the imaginary part of *x* is a negative real zero.

Specific Name	Argument Type	Result Type
SQRT	REAL(4)	REAL(4)
DSQRT	REAL(8)	REAL(8)
QSQRT	REAL(16)	REAL(16)
CSQRT [1]	COMPLEX(4)	COMPLEX(4)
CDSQRT[2]	COMPLEX(8)	COMPLEX(8)
CQSQRT	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CSQRT.

[2]This function can also be specified as ZSQRT.

Example

SQRT (16.0) has the value 4.0.

SQRT (3.0) has the value 1.732051.

The following shows another example:

```
! Calculate the hypotenuse of a right triangle
! from the lengths of the other two sides.
REAL sidea, sideb, hyp
sidea = 3.0
sideb = 4.0
hyp = SQRT (sidea**2 + sideb**2)
WRITE (*, 100) hyp
100 FORMAT (/ ' The hypotenuse is ', F10.3)
END
```

SRAND

Portability Subroutine: *Seeds the random number generator used with IRAND and RAND.*

Module

USE IFPORT

Syntax

`CALL SRAND (iseed)`

iseed (Input) INTEGER(4). Any value. The default value is 1.

SRAND seeds the random number generator used with IRAND and RAND. Calling SRAND is equivalent to calling IRAND or RAND with a new seed.

The same value for *iseed* generates the same sequence of random numbers. To vary the sequence, call SRAND with a different *iseed* value each time the program is executed.

Example

```
!  How many random numbers out of 100 will be between .5 and .6?
   USE IFPORT
   ICOUNT = 0
   CALL SRAND(123)
   DO I = 1, 100
     X = RAND(0)
     IF ((X>.5).AND.(x<.6)) ICOUNT = ICOUNT + 1
   END DO
   WRITE(*,*) ICOUNT, "numbers between .5 and .6!"
   END
```

See Also

RAND
IRAND
RANDOM_NUMBER
RANDOM_SEED

SSWRQQ

Portability Subroutine: *Returns the floating-point processor status word.*

Module

USE IFPORT

Syntax

```
CALL SSWRQQ (status)
```

status (Output) INTEGER(2). Floating-point processor status word.

SSWRQQ performs the same function as the run-time subroutine GETSTATUSFPQQ and is provided for compatibility.

Example

```
USE IFPORT
INTEGER(2) status
CALL SSWRQQ (status)
```

See Also

LCWRQQ
GETSTATUSFPQQ

STAT

Portability Function: *Returns detailed information about a file.*

Module

USE IFPORT

Syntax

```
result = STAT (name,statb)
```

name (Input) Character*(*). Name of the file to examine.

statb (Output) INTEGER(4) or INTEGER(8). One-dimensional array of size 12; where the system information is stored. The elements of *statb* contain the following values:

Element	Description	Values or Notes
statb(1)	Device the file resides on	W*S: Always 0 L*X, M*X: System dependent
statb(2)	File inode number	W*S: Always 0 L*X, M*X: System dependent
statb(3)	Access mode of the file	See the table in Results
statb(4)	Number of hard links to the file	W*S: Always 1 L*X, M*X: System dependent
statb(5)	User ID of owner	W*S: Always 1

Element	Description	Values or Notes
		L*X, M*X: System dependent
statb(6)	Group ID of owner	W*S: Always 1 L*X, M*X: System dependent
statb(7)	Raw device the file resides on	W*S: Always 0 L*X, M*X: System dependent
statb(8)	Size of the file	
statb(9)	Time when the file was last accessed[1]	W*S: Only available on non-FAT file systems; undefined on FAT systems L*X, M*X: System dependent
statb(10)	Time when the file was last modified[1]	
statb(11)	Time of last file status change[1]	W*S: Same as stat(10) L*X, M*X: System dependent
statb(12)	Blocksize for file system I/O operations	W*S: Always 1 L*X, M*X: System dependent

[1]Times are in the same format returned by the TIME function (number of seconds since 00:00:00 Greenwich mean time, January 1, 1970).

Results

The result type is INTEGER(4).

On Windows* systems, the result is zero if the inquiry was successful; otherwise, the error code ENOENT (the specified file could not be found). On Linux* and macOS* systems, the file inquired about must be currently connected to a logical unit and must already exist when STAT is called; if STAT fails, `errno`is set.

For a list of other error codes, see IERRNO.

The access mode (the third element of *statb*) is a bitmap consisting of an IOR of the following constants:

Symbolic name	Constant	Description	Notes
S_IFMT	O'0170000'	Type of file	
S_IFDIR	O'0040000'	Directory	

Symbolic name	Constant	Description	Notes
S_IFCHR	O'0020000'	Character special	Never set on Windows systems
S_IFBLK	O'0060000'	Block special	Never set on Windows systems
S_IFREG	O'0100000'	Regular	
S_IFLNK	O'0120000'	Symbolic link	Never set on Windows systems
S_IFSOCK	O'0140000'	Socket	Never set on Windows systems
S_ISUID	O'0004000'	Set user ID on execution	Never set on Windows systems
S_ISGID	O'0002000'	Set group ID on execution	Never set on Windows systems
S_ISVTX	O'0001000'	Save swapped text	Never set on Windows systems
S_IRWXU	O'0000700'	Owner's file permissions	
S_IRUSR, S_IREAD	O'0000400'	Owner's read permission	Always true on Windows systems
S_IWUSR, S_IWRITE	O'0000200'	Owner's write permission	
S_IXUSR, S_IEXEC	O'0000100'	Owner's execute permission	Based on file extension (.EXE, .COM, .CMD, or .BAT)
S_IRWXG	O'0000070'	Group's file permissions	Same as S_IRWXU on Windows systems
S_IRGRP	O'0000040'	Group's read permission	Same as S_IRUSR on Windows systems
S_IWGRP	O'0000020'	Group's write permission	Same as S_IWUSR on Windows systems
S_IXGRP	O'0000010'	Group's execute permission	Same as S_IXUSR on Windows systems
S_IRWXO	O'0000007'	Other's file permissions	Same as S_IRWXU on Windows systems
S_IROTH	O'0000004'	Other's read permission	Same as S_IRUSR on Windows systems
S_IWOTH	O'0000002'	Other's write permission	Same as S_IWUSR on Windows systems
S_IXOTH	O'0000001'	Other's execute permission	Same as S_IXUSR on Windows systems

STAT returns the same information as FSTAT, but accesses files by name instead of external unit number.

On Windows* systems, LSTAT returns exactly the same information as STAT. On Linux and macOS* systems, if the file denoted by *name* is a link, LSTAT provides information on the link, while STAT provides information on the file at the destination of the link.

You can also use the INQUIRE statement to get information about file properties.

Example

```
USE IFPORT
CHARACTER*12 file_name
INTEGER(4) info_array(12)
print *, 'Enter file to examine: '
read *, file_name
ISTATUS = STAT (file_name, info_array)
if (.not. istatus) then
  print *, info_array else
  print *, 'Error = ',istatus
end if
end
```

See Also

INQUIRE
GETFILEINFOQQ
FSTAT

Statement Function

Statement: *Defines a function in a single statement in the same program unit in which the procedure is referenced.*

Syntax

fun([*d-arg* [,*d-arg*]...]) = *expr*

fun
Is the name of the statement function.

d-arg
Is a dummy argument. A dummy argument can appear only once in any list of dummy arguments, and its scope is local to the statement function.

expr
Is a scalar expression defining the computation to be performed.

Named constants and variables used in the expression must have been declared previously in the specification part of the scoping unit or made accessible by use or host association.

If the expression contains a function or statement function reference, that function must have been defined previously as a function or statement function in the same program unit.

A statement function reference takes the following form:

fun([*a-arg*[, *a-arg*] ...])

fun
Is the name of the statement function.

a-arg
Is an actual argument.

Description

When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of a statement function can be explicitly defined in a type declaration statement. If no type is specified, the type is determined by implicit typing rules in effect for the program unit.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments.

Except for the data type, declarative information associated with an entity is not associated with dummy arguments in the statement function; for example, declaring an entity to be an array or to be in a common block does not affect a dummy argument with the same name.

The name of the statement function cannot be the same as the name of any other entity within the same program unit.

Any reference to a statement function must appear in the same program unit as the definition of that function.

A statement function reference must appear as (or be part of) an expression. The reference cannot appear on the left side of an assignment statement.

A statement function must not be provided as a procedure argument.

Example

The following are examples of statement functions:

```
REAL VOLUME, RADIUS
VOLUME(RADIUS) = 4.189*RADIUS**3

CHARACTER*10 CSF,A,B
CSF(A,B) = A(6:10)//B(1:5)
```

The following example shows a statement function and some references to it:

```
AVG(A,B,C) = (A+B+C)/3.
...
GRADE = AVG(TEST1,TEST2,XLAB)
IF (AVG(P,D,Q) .LT. AVG(X,Y,Z)) STOP
FINAL = AVG(TEST3,TEST4,LAB2)       ! Invalid reference; implicit
...                                 ! type of third argument does not
...                                 ! match implicit type of dummy argument
```

Implicit typing problems can be avoided if all arguments are explicitly typed.

The following statement function definition is invalid because it contains a constant, which cannot be used as a dummy argument:

```
REAL COMP, C, D, E
COMP(C,D,E,3.) = (C + D - E)/3.
```

The following shows another example:

```
Add (a, b) = a + b
REAL(4) y, x(6)
. . .
DO n = 2, 6
  x(n) = Add (y, x(n-1))
END DO
```

See Also

FUNCTION
Argument Association
Use and Host Association

STATIC

Statement and Attribute: *Controls the storage allocation of variables in subprograms (as does AUTOMATIC). Variables declared as STATIC and allocated in memory reside in the static storage area, rather than in the stack storage area.*

Syntax

The STATIC attribute can be specified in a type declaration statement or a STATIC statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] STATIC [, att-ls] :: v[, v] ...
```

Statement:

```
STATIC[::] v[, v] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
v	Is the name of a variable or an array specification. It can be of any type.

STATIC declarations only affect how data is allocated in storage.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

By default, the compiler allocates local scalar variables of non-recursive subprograms in the static storage area. Local arrays, except for allocatable arrays, are in the static storage area by default.

The compiler may choose to allocate a variable in temporary (stack or register) storage if it notices that the variable is always defined before use. Appropriate use of the SAVE attribute can prevent compiler warnings if a variable is used before it is defined.

To change the default for variables, specify them as AUTOMATIC or specify RECURSIVE in one of the following ways:

- As a keyword in a FUNCTION or SUBROUTINE statement
- As a compiler option
- As an option in an OPTIONS statement

To override any compiler option that may affect variables, explicitly specify the variables as STATIC.

> **NOTE**
> Variables that are data-initialized, and variables in COMMON and SAVE statements are always static. This is regardless of whether a compiler option specifies recursion.

A variable cannot be specified as STATIC more than once in the same scoping unit.

If the variable is a pointer, STATIC applies only to the pointer itself, not to any associated target.

Some variables cannot be specified as STATIC. The following table shows these restrictions:

Variable	STATIC
Dummy argument	No
Automatic object	No
Common block item	Yes
Use-associated item	No
Function result	No
Component of a derived type	No

A variable can be specified with both the STATIC and SAVE attributes.

If a variable is in a module's outer scope, it can be specified as STATIC.

Example

The following example shows a type declaration statement specifying the STATIC attribute:

```
INTEGER, STATIC :: ARRAY_A
```

The following example shows a STATIC statement:

```
...
CONTAINS
 INTEGER FUNCTION REDO_FUNC
   INTEGER I, J(10), K
   REAL C, D, E(30)
   AUTOMATIC I, J, K(20)
   STATIC C, D, E
   ...
 END FUNCTION
...
 INTEGER N1, N2
 N1 = -1
 DO WHILE (N1)
   N2 = N1*2
   call sub1(N1, N2)
   read *, N1
 END DO
 CONTAINS
 SUBROUTINE sub1 (iold, inew)
 INTEGER, intent(INOUT):: iold
 integer, STATIC ::N3
 integer, intent(IN) :: inew
 if (iold .eq. -1) then
   N3 = iold
 end if
 print *, 'New: ', inew, 'N3: ',N3
 END subroutine
 !
 END
```

See Also

AUTOMATIC
SAVE
Type Declarations

Compatible attributes
RECURSIVE
OPTIONS
POINTER
Modules and Module Procedures
recursive compiler option

STOP and ERROR STOP

Statements: *The STOP statement initiates normal termination of an image before the execution of an END statement of the main program. The ERROR STOP statement initiates error termination.*

Syntax

`STOP [stop-code]`

`ERROR STOP [stop-code]`

stop-code — (Optional) A message. It can be any of the following:

- A scalar character constant expression of type default character
- A non-negative integer constant less than or equal to 2147483647 (2**31-1); leading zeros are ignored
- A scalar integer constant expression

An ERROR STOP statement can appear in a PURE procedure, a STOP statement cannot appear.

If *stop-code* is specified, the STOP or ERROR STOP statement does the following:

- Writes the specified message to the standard error device.
- Writes one or more of the following messages to the standard error device indicating which IEEE floating-point exceptions are signaling if `assume fpe-summary` is specified:

```
IEEE_DIVIDE_BY_ZERO is signaling
IEEE_INVALID is signaling
IEEE_OVERFLOW is signaling
IEEE_UNDERFLOW is signaling
```

- STOP initiates normal termination on the image that executes it. ERROR STOP initiates error termination. If *stop-code* is a character constant, STOP returns a status of zero and ERROR STOP returns a status of non-zero. If *stop-code* is an integer, a status equal to *stop-code* is returned for both STOP and ERROR_STOP.

If *stop-code* is not specified on a STOP statement, the program is terminated, no message is printed, and a status of zero is returned. If *stop-code* is not specified on an ERROR STOP statement, the program is terminated, no message is printed, and a positive status value of 128 is returned.

When a program is running on multiple images, a STOP statement on one image does not affect the other images; they can continue to reference and define the coarrays located on the stopped image. An ERROR STOP initiates execution termination on all images, which may include flushing of I/O buffers, closing of I/O files, and reporting error status on one or more images.

Effect on Windows* Systems

In QuickWin programs, the following is displayed in a message box:

```
Program terminated with Exit Code stop-code
```

If the application has no console window, the stop-code is displayed in a message box.

Effect on Linux* and macOS* Systems

Operating system shells (such as bash, sh, csh, dash, etc.) work with a one byte exit status. So, when *stop-code* is an integer, only the lowest byte is significant. For example, consider the following statement:

```
  STOP 99999
```

In this case, the program returns a status equal to 159 because integer 99999 = Z'1869F', and the lowest byte is equal to Z'9F', which equals 159.

Example

The following examples show valid STOP statements:

```
STOP 98
STOP 'END OF RUN'

DO
  READ *, X, Y
  IF (X > Y) STOP 5555
END DO
```

The following shows another example:

```
      OPEN(1,FILE='file1.dat', status='OLD', ERR=100)
      . . .
 100  STOP 'ERROR DETECTED!'
      END
```

See Also

`assume` compiler option

EXIT

Program Termination

STOPPED_IMAGES

Transformational Intrinsic Function (Generic):
Returns an array of index images that have initiated normal termination.

Syntax

`result = STOPPED_IMAGES ([kind])`

kind	(Input; optional) Must be a scalar integer expression with a value that is a valid INTEGER kind type parameter.

Results

The result is a rank-one integer array with the same type kind parameters as *kind* if present; otherwise, default integer. The size of the array is equal to the number of images in the current (initial) team that have initiated normal termination.

The result array elements are the image index values of images on the current team that have initiated normal termination. The indices are arranged in increasing numeric order.

NOTE
STOPPED_IMAGES argument *team* has not yet been implemented. It will be added in a future release.

Example

If image 5 and 12 of the current team have initiated normal termination, the result of STOPPED_IMAGES ( ) is an array of default integer type with size 2 defined with the value [5, 12]. If no images in the current team are known to have failed, the result of STOPPED_IMAGES ( ) is a zero-sized array.

See Also

FAILED_IMAGES
IMAGE_STATUS

STORAGE_SIZE

Inquiry Intrinsic Function (Generic): *Returns the storage size in bits.*

Syntax

```
result = STORAGE_SIZE (a [, kind])
```

a	(Input) Must be a scalar or array. It can be of any type. If it has any deferred type parameters it must not be an unallocated allocatable variable, or a disassociated, or undefined pointer.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value is the size expressed in bits for an element of an array that has the dynamic type and kind parameters of *a*.

If the type and kind parameters are such that storage association applies, the result is consistent with the named constants defined in the intrinsic module ISO_FORTRAN_ENV.

An array element may take more bits to store than a scalar, since any hardware-imposed alignment requirements for array elements may not apply to a scalar variable.

NOTE
The result is intended to be the size in memory that an object takes when it is stored. This may differ from the size it takes during expression handling (which may be the native register size) or when stored in a file. If an object is never stored in memory but only in a register, this function nonetheless returns the size it would take if it were stored in memory.

Example

STORAGE_SIZE (1.0) has the same value as the named constant NUMERIC_STORAGE_SIZE in the intrinsic module ISO_FORTRAN_ENV.

See Also

Storage Association
ISO_FORTRAN_ENV Module

STRICT and NOSTRICT

General Compiler Directive: *STRICT disables language features not found in the language standard specified on the command line (Fortran 2008, Fortran 2003, Fortran 95, or Fortran 90). NOSTRICT (the default) enables these features.*

Syntax

```
!DIR$ STRICT
```

```
!DIR$ NOSTRICT
```

If STRICT is specified and no language standard is specified on the command line, the default is to disable features not found in Fortran 2008.

The STRICT and NOSTRICT directives can appear only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. STRICT and NOSTRICT cannot appear between program units, or at the beginning of internal subprograms. They do not affect any modules invoked with the USE statement in the program unit that contains them.

Example

```
! NOSTRICT by default
TYPE stuff
  INTEGER(4) k
  INTEGER(4) m
  CHARACTER(4) name
END TYPE stuff
TYPE (stuff) examp
DOUBLE COMPLEX cd  ! non-standard data type, no error
cd =(3.0D0, 4.0D0)
examp.k = 4        ! non-standard component designation,
                   ! no error
END
SUBROUTINE STRICTDEMO( )
  !DIR$ STRICT
   TYPE stuff
    INTEGER(4) k
    INTEGER(4) m
    CHARACTER(4) name
   END TYPE stuff
   TYPE (stuff) samp
   DOUBLE COMPLEX cd   ! ERROR
   cd =(3.0D0, 4.0D0)
   samp.k = 4          ! ERROR
END SUBROUTINE
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
stand compiler option
Equivalent Compiler Options

STRUCTURE and END STRUCTURE

Statement: *Defines the field names, types of data within fields, and order and alignment of fields within a record structure. Fields and structures can be initialized, but records cannot be initialized.*

Syntax

```
STRUCTURE [/structure-name/] [field-namelist]
   field-declaration
   [field-declaration]
   . . .
   [field-declaration]
END STRUCTURE
```

structure-name

Is the name used to identify a structure, enclosed by slashes.

Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names, but structures can share names with variables (scalar or array), record fields, PARAMETER constants, and common blocks.

Structure declarations can be nested (contain one or more other structure declarations). A structure name is required for the structured declaration at the outermost level of nesting, and is optional for the other declarations nested in it. However, if you wish to reference a nested structure in a RECORD statement in your program, it must have a name.

Structure, field, and record names are all local to the defining program unit. When records are passed as arguments, the fields in the defining structures within the calling and called subprograms must match in type, order, and dimension.

field-namelist

Is a list of fields having the structure of the associated structure declaration. A field namelist is allowed only in nested structure declarations.

field-declaration

Also called the declaration body. A *field-declaration* consists of any combination of the following:

- Type declarations

  These are ordinary Fortran data type declarations.
- Substructure declarations

  A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of both.
- Union declarations

  A union declaration is composed of one or more mapped field declarations.
- PARAMETER statements

  PARAMETER statements can appear in a structure declaration, but cannot be given a data type within the declaration block.

Type declarations for PARAMETER names must precede the PARAMETER statement and be outside of a STRUCTURE declaration, as follows:

```
  INTEGER*4 P
STRUCTURE /ABC/
  PARAMETER (P=4)
  REAL*4 F
END STRUCTURE
  REAL*4 A(P)
```

The Standard Fortran derived type replaces STRUCTURE and RECORD constructs, and should be used in writing new code. See Derived Data Types.

Unlike type declaration statements, structure declarations do not create variables. Structured variables (records) are created when you use a RECORD statement containing the name of a previously declared structure. The RECORD statement can be considered as a kind of type declaration statement. The difference is that aggregate items, not single items, are being defined.

Within a structure declaration, the ordering of both the statements and the field names within the statements is important, because this ordering determines the order of the fields in records.

In a structure declaration, each field offset is the sum of the lengths of the previous fields, so the length of the structure is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

By default, fields are aligned on natural boundaries; misaligned fields are padded as necessary. To avoid padding of records, you should lay out structures so that all fields are naturally aligned.

To pack fields on arbitrary byte boundaries, you must specify a compiler option. You can also specify alignment for fields by using the OPTIONS or PACK general directive.

A field name must not be the same as any intrinsic or user-defined operator (for example, EQ cannot be used as a field name).

Example

An item can be a RECORD statement that references a previously defined structure type:

```
STRUCTURE /full_address/
  RECORD  /full_name/ personsname
  RECORD  /address/   ship_to
  INTEGER*1           age
  INTEGER*4           phone
END STRUCTURE
```

You can specify a particular item by listing the sequence of items required to reach it, separated by a period (.). Suppose you declare a structure variable, `shippingaddress`, using the `full_address`structure defined in the previous example:

```
RECORD /full_address/ shippingaddress
```

In this case, the `age`item would then be specified by `shippingaddress.age`, the first name of the receiver by `shippingaddress.personsname.first_name`, and so on.

In the following example, the declaration defines a structure named `APPOINTMENT`. `APPOINTMENT` contains the structure `DATE`(field `APP_DATE`) as a substructure. It also contains a substructure named `TIME`(field `APP_TIME`, an array), a CHARACTER*20 array named `APP_MEMO`, and a LOGICAL*1 field named `APP_FLAG`.

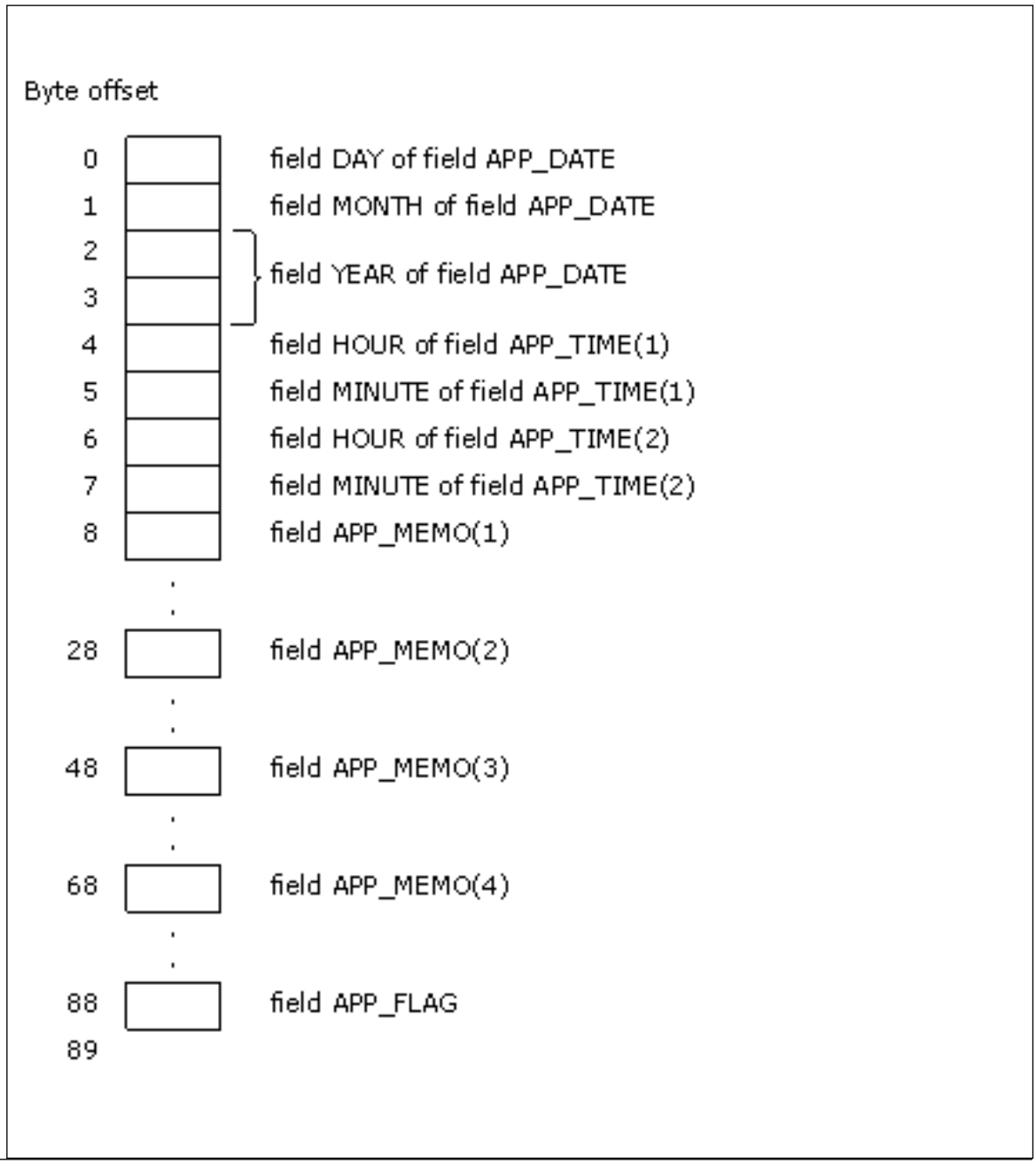
```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
```

```
END STRUCTURE

STRUCTURE /APPOINTMENT/
  RECORD /DATE/     APP_DATE
  STRUCTURE /TIME/  APP_TIME (2)
    INTEGER*1       HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20      APP_MEMO (4)
  LOGICAL*1         APP_FLAG
END STRUCTURE
```

The length of any instance of structure `APPOINTMENT` is 89 bytes.

The following figure shows the memory mapping of any record or record array element with the structure `APPOINTMENT`.

Memory Map of Structure APPOINTMENT



See Also

TYPE
MAP...END MAP
RECORD
UNION...END UNION
PACK Directive
OPTIONS Directive
Data Types, Constants, and Variables

Record Structures

SUBMODULE

Statement: *Marks the beginning of a submodule program unit, which contains specifications and definitions that can be used by one or more program units.*

Syntax

```
SUBMODULE (ancestor-module-name [:parent-submodule-name]) name
   [specification-part]
   [module-subprogram
   [module-subprogram]...]
END [SUBMODULE [name]]
```

ancestor-module-name	Must be the name of a nonintrinsic module. It is the name of the module at the root of the module or submodule tree.
parent-submodule-name	(Optional) Is the name of the parent submodule, if any. The parent submodule must be a descendant of *ancestor-module-name*.
name	Is the name of the submodule.
specification-part	Is one or more specification statements, except for the following: • ENTRY • FORMAT • AUTOMATIC (or its equivalent attribute) • INTENT (or its equivalent attribute) • OPTIONAL (or its equivalent attribute) • Statement functions An automatic object must not appear in a specification statement.
module-subprogram	Is a function or subroutine subprogram that defines the module procedure. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE. A module subprogram can contain internal procedures.

Description

If a name follows the END statement, it must be the same as the name specified in the SUBMODULE statement.

Each submodule has exactly one ancestor module and exactly one parent module or submodule. If the parent is a module then it must be the ancestor module. The relationship is tree-like, with the parent module or submodule at the root of the tree, and its descendant submodules as the branches of the tree.

A module or submodule may have one or more descendent submodules.

A submodule can access the entities from its parent module or submodule by host association. Unlike a module, a submodule cannot be referenced in other program units by use association. Entities declared in a submodule are only accessible from the submodule and its descendent submodules. Furthermore, a module procedure can have its interface declared in a module or submodule and its implementation contained in a descendent submodule in a separate file.

Submodules help the modularization of a large module in several ways:

- Internal data can be shared among submodules without being exposed to users of the module. Without the submodule feature, when a large module is split into smaller modules, internal data may be forced to become public in order to be shared among the modules.
- Even if a module procedure implementation in a submodule is changed, as long as the module procedure interface declared in the ancestor module remains the same, a recompilation would not be needed for the user of the module. This could reduce the rebuilding time for a complex system.
- Separate concepts with circular dependencies can now be implemented in different submodules. It cannot be done with just modules.

A submodule is uniquely identified by a *submodule identifier* which consists of its *ancestor-module-name* and the name of the submodule. The name of a submodule can therefore be the same as the name of another submodule so long as they do not have the same ancestor module.

The following rules also apply to submodules:

- The specification part of a submodule must not contain IMPORT, ENTRY, FORMAT, executable, or statement function statements.
- A variable, common block, or procedure pointer declared in a submodule implicitly has the SAVE attribute, which may be confirmed by explicit specification.
- If a specification or constant expression in the *specification-part* of a submodule includes a reference to a generic entity, there must be no specific procedures of the generic entity defined in the submodule subsequent to the specification or constant expression.

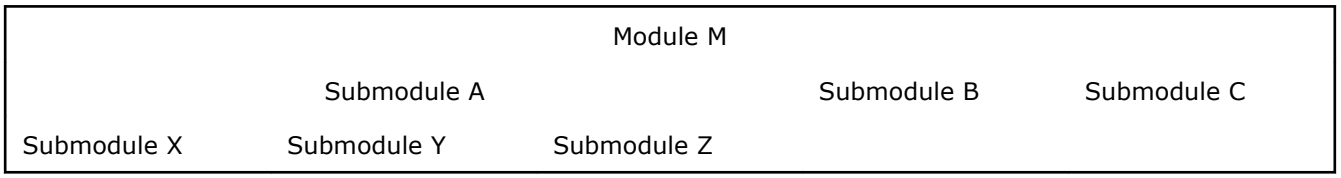
Unlike a module, the specification part of a submodule must not contain PUBLIC and PRIVATE specifications.

Any executable statements in a module or submodule can only be specified in a module or submodule subprogram.

A submodule can contain one or more procedure interface blocks, which let you specify an explicit interface for an external subprogram or dummy subprogram.

Example

Consider the following example of a multilevel submodule system:



In the above example, module M is extended by submodules up to two levels. Module M is the ancestor module of all the submodules. Entities declared in module M can be shared among all the submodules. A is the parent submodule of X, Y, and Z. Entities declared in submodule A can be shared among submodules X, Y, and Z, but not by B or C.

Only public entities declared in M can be available to other program units by use association.

The SUBMODULE declarations for A and X are:

```
SUBMODULE (M) A
   …
END SUBMODULE A

SUBMODULE (M : A) X
   …
END SUBMODULE X
```

The following example shows modules and submodules, separate module procedures, and how circular dependency can be avoided with submodules. There are five parts in the example:

1. module color_points

2. level-1 submodule color_points_a
3. level-2 submodule color_points_b
4. module palette_stuff
5. program main

```
! part 1
module color_points  ! This is the ancestor module
   type color_point
      private
      real :: x, y
      integer :: color
   end type color_point

   ! Below is the interface declaration of the separate module procedures.
   ! No IMPORT statement is used in the interface bodies.
   ! The separate module procedures are implemented in the two submodules.
   interface
      module subroutine color_point_del ( p )
      type(color_point), allocatable :: p
      end subroutine color_point_del

      real module function color_point_dist ( a, b )
      type(color_point), intent(in) :: a, b
      end function color_point_dist

      module subroutine color_point_draw ( p )
      type(color_point), intent(in) :: p
      end subroutine color_point_draw

      module subroutine color_point_new ( p )
      type(color_point), allocatable :: p
      end subroutine color_point_new
   end interface
end module color_points


! part 2
submodule ( color_points ) color_points_a  ! submodule of color_points
   integer :: instance_count = 0

   ! Below is the interface declaration of the separate module procedure
   !   inquire_palette, which is implemented in the submodule color_points_b.
   interface
      module subroutine inquire_palette ( pt, pal )
         use palette_stuff
         ! Later you will see that module palette_stuff uses color_points.
         ! This use of palette_stuff however does not cause a circular
         !   dependence because this use is not in the module.
         type(color_point), intent(in) :: pt
         type(palette), intent(out) :: pal
      end subroutine inquire_palette
   end interface

contains
   ! Here are the implementations of three of the four separate module
   !   procedures declared in module color_point.
   module subroutine color_point_del ( p )
     type(color_point), allocatable :: p
```

```
     instance_count = instance_count - 1
     deallocate ( p )
   end subroutine color_point_del

   real module function color_point_dist ( a, b ) result ( dist )
     type(color_point), intent(in) :: a, b
     dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
   end function color_point_dist

   module subroutine color_point_new ( p )
     type(color_point), allocatable :: p
     instance_count = instance_count + 1
     allocate ( p )
   end subroutine color_point_new

end submodule color_points_a


! part 3
submodule ( color_points:color_points_a ) color_points_b
! submodule of color_point_a
contains
   ! Implementation of a module procedure declared in the ancestor module
   module subroutine color_point_draw ( p )
     use palette_stuff, only: palette
     type(color_point), intent(in) :: p
     type(palette) :: MyPalette
     ...; call inquire_palette ( p, MyPalette ); ...
   end subroutine color_point_draw

   ! Implementation of a module procedure declared in the parent submodule
   module procedure inquire_palette
     ...
   end procedure inquire_palette

   ! A procedure only accessible from color_points_b and its submodules
   subroutine private_stuff
     ...
   end subroutine private_stuff

end submodule color_points_b


! part 4
module palette_stuff
   type :: palette ; ... ; end type palette
contains
   subroutine test_palette ( p )
     use color_points
     ! This does not cause a circular dependency because the
     !  "use palette_stuff" that is logically within color_points
     !  is in the color_points_a submodule.
     type(palette), intent(in) :: p
     ...
   end subroutine test_palette
end module palette_stuff
```

```
! part 5
program main
   use color_points
   ! Only public entities in color_points can be accessed here, not the
   !   entities from its submodules.

   ! The separate module procedure color_point_draw can be a specific
   !   procedure for a generic here. Recall that color_point_draw is
   !   implemented in a submodule, but its interface is public in a module.
   interface draw
     module procedure color_point_draw
   end interface

   type(color_point), allocatable :: c_1, c_2
   real :: rc
   ...
   call color_point_new (c_1)  ! body in color_points_a, interface in
                               !   color_points
   ...
   call draw (c_1)             ! body in color_points_b, specific interface
                               !   in color_points, generic interface here
   ...
   rc = color_point_dist (c_1, c_2)  ! body in color_points_a, interface in
                                     !   color_points
   ...
   call color_point_del (c_1)  ! body in color_points_a, interface in
                               !   color_points
   ...
end program main
```

See Also

MODULE
FUNCTION
SUBROUTINE
References to Generic Procedures

SUBROUTINE

Statement: *The initial statement of a subroutine subprogram. A subroutine subprogram is invoked in a CALL statement or by a defined assignment or operator, and does not return a particular value.*

Syntax

```
[prefix [prefix]] SUBROUTINE name [([d-arg-list]) [lang-binding]]
   [specification-part]
   [execution-part]
[CONTAINS
   [internal-subprogram-part] ]
END [SUBROUTINE [name]]
```

prefix (Optional) Is any of the following:

- ELEMENTAL

  Acts on one array element at a time. This is a restricted form of pure procedure.
- IMPURE

  Asserts that the procedure has side effects.
- MODULE

  Indicates a separate module procedure. See separate module procedures.
- NON_RECURSIVE

  Indicates a procedure is not recursive.
- PURE

  Asserts that the procedure has no side effects.
- RECURSIVE

  Permits direct or indirect recursion to occur.

At most one of each of the above can be specified. You cannot specify both NON_RECURSIVE and RECURSIVE. You cannot specify both PURE and IMPURE. You cannot specify ELEMENTAL if *lang-binding* is specified.

name — Is the name of the subroutine.

d-arg-list — (Optional) Is a list of one or more dummy arguments or alternate return specifiers (*) .

lang-binding — (Optional) Takes the following form:

BIND (C [, NAME=*ext-name*])

ext-name — Is a character scalar constant expression that can be used to construct the external name.

specification-part — Is one or more specification statements.

execution-part — Is one or more executable constructs or statements.

internal-subprogram-part — Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Description

A subroutine is invoked by a CALL statement or defined assignment. When a subroutine is invoked, dummy arguments (if present) become associated with the corresponding actual arguments specified in the call.

Execution begins with the first executable construct or statement following the SUBROUTINE statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

A subroutine subprogram *cannot* contain a BLOCK DATA statement, a PROGRAM statement, a MODULE statement, or a SUBMODULE statement. A subroutine can contain SUBROUTINE and FUNCTION statements to define internal procedures. ENTRY statements can be included to provide multiple entry points to the subprogram.

You need an interface block for a subroutine when:

- Calling arguments use argument keywords.

- Some arguments are optional.
- A dummy argument is an assumed-shape array, a pointer, or a target.
- The subroutine extends intrinsic assignment.
- The subroutine can be referenced by a generic name.
- The subroutine is in a dynamic-link library.

If the subroutine is in a DLL and is called from your program, use the option DLLEXPORT or DLLIMPORT, which you can specify with the ATTRIBUTES directive.

Note that if you specify *lang-binding*, you have to use the parentheses even if there are no arguments. For example, without *lang-binding* you can specify `SUBROUTINE F` but with *lang-binding* you have to specify `SUBROUTINE F( ) BIND (C)`.

Example

The following example shows a subroutine:

```
Main ProgramSubroutine
CALL HELLO_WORLD      SUBROUTINE HELLO_WORLD
...                   PRINT *, "Hello World"
END                   END SUBROUTINE
```

The following example uses alternate return specifiers to determine where control transfers on completion of the subroutine:

```
Main ProgramSubroutine
    CALL CHECK(A,B,*10,*20,C)             SUBROUTINE CHECK(X,Y,*,*,Q)
    TYPE *, 'VALUE LESS THAN ZERO'        ...
    GO TO 30                          50  IF (Z)  60,70,80
10  TYPE*, 'VALUE EQUALS ZERO'        60  RETURN
    GO TO 30                          70  RETURN 1
20  TYPE*, 'VALUE MORE THAN ZERO'     80  RETURN 2
30  CONTINUE                              END
    ...
```

The SUBROUTINE statement argument list contains two dummy alternate return arguments corresponding to the actual arguments *10 and *20 in the CALL statement argument list.

The value of Z determines the return, as follows:

- If Z < zero, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the main program.
- If Z = = zero, the return is to statement label 10 in the main program.
- If Z > zero, the return is to statement label 20 in the main program.

Note that an alternate return is an obsolescent feature in the Fortran Standard.

The following shows another example:

```
     SUBROUTINE GetNum (num, unit)
     INTEGER num, unit
 10  READ (unit, '(I10)', ERR = 10) num
     END
    ...
```

See Also

FUNCTION
INTERFACE
PURE
ELEMENTAL
CALL

RETURN
ENTRY
Argument Association
Program Units and Procedures
General Rules for Function and Subroutine Subprograms
Deleted and Obsolescent Language Features

SUM

Transformational Intrinsic Function (Generic):
Returns the sum of all the elements in an entire array or in a specified dimension of an array.

Syntax

```
result = SUM (array [,dim] [,mask])
```

array	(Input) Must be an array of type integer, real, or complex.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If SUM(*array*) is specified, the result is the sum of all elements of *array*. If *array* has size zero, the result is zero.
- If SUM(*array*, MASK= *mask*) is specified, the result is the sum of all elements of *array* corresponding to true elements of *mask*. If *array* has size zero, or every element of *mask* has the value .FALSE., the result is zero.

The following rules apply if *dim* is specified:

- If *array* has rank one, the value is the same as SUM(*array*[,MASK= *mask*]).
- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- The value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of SUM(*array*, *dim*[, *mask*]) is equal to SUM(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK = *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)].

Example

SUM ((/2, 3, 4/)) returns the value 9 (sum of 2 + 3 + 4). SUM ((/2, 3, 4/), DIM=1) returns the same result.

SUM (B, MASK=B .LT. 0.0) returns the arithmetic sum of the negative elements of B.

C is the array

```
  [ 1  2  3 ]
  [ 4  5  6 ].
```

SUM (C, DIM=1) returns the value (5, 7, 9), which is the sum of all elements in each column. 5 is the sum of 1 + 4 in column 1. 7 is the sum of 2 + 5 in column 2, and so forth.

SUM (C, DIM=2) returns the value (6, 15), which is the sum of all elements in each row. 6 is the sum of 1 + 2 + 3 in row 1. 15 is the sum of 4 + 5 + 6 in row 2.

The following shows another example:

```
INTEGER array (2, 3), i, j(3)
array = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  array is  1 3 5
!            2 4 6
i = SUM((/ 1, 2, 3 /))         ! returns 6
j = SUM(array, DIM = 1)        ! returns [3 7 11]
WRITE(*,*) i, j
END
```

See Also

PRODUCT

SYNC ALL

Statement: *Performs a synchronization of all images.*

Syntax

The SYNC ALL statement takes the following form:

```
SYNC ALL[([STAT=stat-var][, ERRMSG=err-var])]
```

stat-var	Is a scalar integer variable in which the status of the synchronization is stored.
err-var	Is a scalar default character variable in which an error condition is stored if such a condition occurs.

STAT= and ERRMSG= can appear in either order, but only once in a SYNC ALL statement.

Description

Execution on an image, for example, C, of the segment following the SYNC ALL statement is delayed until each other image has executed a SYNC ALL statement as many times as has image C.

The segments that executed before the SYNC ALL statement on an image, precede the segments that execute after the SYNC ALL statement on another image.

Example

In the following example, image 5 reads data and transmits the data to other images:

```
REAL :: W[*]
SYNC ALL
IF (THIS_IMAGE()==5) THEN
   READ (*,*) W
   DO I = 6, NUM_IMAGES()
      W[I] = W
   END DO
END IF
...
SYNC ALL
```

See Also

Image Control Statements
Coarrays
Using Coarrays

SYNC IMAGES

Statement: *Performs a synchronization of the image with each of the other images in the image set.*

Syntax

The SYNC IMAGES statement takes the following form:

```
SYNC IMAGES (image-set[, STAT=stat-var][, ERRMSG=err-var])
```

image-set	Is an integer expression or *. If it is an integer expression, it must be scalar or of rank one. If it is an array expression, the value of each element must be positive and not greater than the number of images; there must be no repeated values. If it is a scalar expression, the value must be positive and not greater than the number of images. If * is specified, it indicates all images.
stat-var	Is a scalar integer variable in which the status of the synchronization is stored.
err-var	Is a scalar default character variable in which an error condition is stored if such a condition occurs.

STAT= and ERRMSG= can appear in either order, but only once in a SYNC IMAGES statement.

Description

When SYNC IMAGES statements are executed on images C and G, they correspond if the number of times image C has executed a SYNC IMAGES statement with G in its image set is the same as the number of times image G has executed a SYNC IMAGES statement with C in its image set. The segments that executed before the SYNC IMAGES statement on either image, precede the segments that execute after the corresponding SYNC IMAGES statement on the other image.

In a program that uses SYNC ALL as its only synchronization mechanism, every SYNC ALL statement can be replaced by a SYNC IMAGES (*) statement. However, SYNC ALL may provide better performance. SYNC IMAGES statements are not required to specify the entire image set, or even the same image set, on all images that participate in the synchronization.

Example

In the following example, image 5 waits for each of the other images to complete using the data. The other images wait for image 5 to set up the data, but do not wait for any other image:

```
IF (THIS_IMAGE() == 5) then
   SYNC IMAGES(*)       ! Sets up coarray data for other images
ELSE
   SYNC IMAGES(5)       ! Other images use the data set up by image 5
END IF
```

In the following example, each image synchronizes with its neighbor:

```
INTEGER :: THIS_STEP, TOTAL_STEPS, TOTAL_IMAGES, MY_IMAGE
MY_IMAGE = THIS_IMAGE ()
TOTAL_IMAGES = NUM_IMAGES ()
   ! Set up calculation
SYNC ALL

DO THIS_STEP = 1, TOTAL_STEPS
```

```
   IF (MY_IMAGE > 1) SYNC IMAGES (MY_IMAGE - 1)
      ! Do the calculations
   IF (MY_IMAGE < TOTAL_IMAGES) SYNC IMAGES (MY_IMAGE + 1)
END DO
SYNC ALL
```

The calculation starts on image 1 since all the others will be waiting on SYNC IMAGES (TOTAL_IMAGES-1). When this is done, image 2 starts and image 1 performs its second calculation. This continues until they are all executing different steps at the same time. Eventually, image 1 finishes and then the others finish one by one.

See Also

Image Control Statements
Coarrays
Using Coarrays

SYNC MEMORY

Statement: *Ends one image segment and begins another. Each segment can then be ordered in some way with respect to segments on other images.*

Syntax

The SYNC MEMORY statement takes the following form:

```
SYNC MEMORY [([STAT=stat-var][, ERRMSG=err-var])]
```

stat-var	Is a scalar integer variable in which the status of the synchronization is stored.
err-var	Is a scalar default character variable in which an error condition is stored if such a condition occurs.

STAT= and ERRMSG= can appear in either order, but only once in a SYNC IMAGES statement.

Description

Unlike the other image control statements, this statement does not have any built-in synchronization effect.

The action regarding X on image Q precedes the action regarding Y on image Q if, by execution of statements on image P, the following is true:

- A variable X on image Q is defined, referenced, becomes undefined, or has its allocation status, pointer association status, array bounds, dynamic type, or type parameters changed or inquired about by execution of a statement
- That statement precedes a successful execution of a SYNC MEMORY statement
- A variable Y on image Q is defined, referenced, becomes undefined, or has its allocation status, pointer association status, array bounds, dynamic type, or type parameters changed or inquired about by execution of a statement that succeeds execution of that SYNC MEMORY statement,

User-defined ordering of segment Pi on image P to precede segment Qj on image Q occurs when the following happens:

- Image P executes an image control statement that ends segment Pi, and then executes statements that initiate a cooperative synchronization between images P and Q
- Image Q executes statements that complete the cooperative synchronization between images P and Q and then executes an image control statement that begins segment Qj

Execution of the cooperative synchronization between images P and Q must include a dependency that forces execution on image P of the statements that initiate the synchronization to precede the execution on image Q of the statements that complete the synchronization. The mechanisms available for creating such a dependency are processor dependent.

NOTE
SYNC MEMORY usually suppresses compiler optimizations that may reorder memory operations across the segment boundary defined by the SYNC MEMORY statement. It ensures that all memory operations initiated in the preceding segments in its image complete before any memory operations in the subsequent segment in its image are started.

Example

The following example should be run on two images:

```
use, intrinsic :: iso_fortran_env
logical (atomic_logical_kind), save :: locked[*] = .true.
logical val
integer :: iam

iam = this_image()

if (iam == 1) then
   sync memory
   call atomic_define(locked[2], .false.)
else if (iam == 2) then
   val = .true.
   do while (val)
      call atomic_ref(val, locked)
   end do
   sync memory
end if

print *, 'success'
end
```

See Also

Image Control Statements
Coarrays
Using Coarrays

SYSTEM

Portability Function: *Sends a command to the shell as if it had been typed at the command line.*

Module

USE IFPORT

Syntax

`result = SYSTEM (string)`

string (Input) Character*(*). Operating system command.

Results

The result type is INTEGER(4). The result is the exit status of the shell command. If -1, use IERRNO to retrieve the error. Errors can be one of the following:

- E2BIG: The argument list is too long.
- ENOENT: The command interpreter cannot be found.
- ENOEXEC: The command interpreter file has an invalid format and is not executable.
- ENOMEM: Not enough system resources are available to execute the command.

On Windows* systems, the calling process waits until the command terminates. To insure compatibility and consistent behavior, an image can be invoked directly by using the Windows API CreateProcess( ) in your Fortran code.

Commands run with the SYSTEM routine are run in a separate shell. Defaults set with the SYSTEM function, such as current working directory or environment variables, do not affect the environment the calling program runs in.

The command line character limit for the SYSTEM function is the same limit that your operating system command interpreter accepts.

Example

```
USE IFPORT
INTEGER(4) I, errnum
I = SYSTEM("dir > file.lst")
If (I .eq. -1) then
  errnum = ierrno( )
  print *, 'Error ', errnum
end if
END
```

See Also

SYSTEMQQ

SYSTEM_CLOCK

Intrinsic Subroutine: *Returns integer data from a real-time clock. SYSTEM_CLOCK returns the number of seconds from 00:00 Coordinated Universal Time (CUT) on 1 JAN 1970. The number is returned with no bias. To get the elapsed time, you must call SYSTEM_CLOCK twice, and subtract the starting time value from the ending time value.*

Syntax

`CALL SYSTEM_CLOCK ([count] [, count_rate] [, count_max])`

count	(Output; optional) Must be scalar and of type integer. It is set to a value based on the current value of the processor clock. The value is increased by one for each clock count until the value *count_max* is reached, and is reset to zero at the next count. (*count* lies in the range 0 to *count_max*.)
count_rate	(Output; optional) Must be scalar and of type integer or real. It is set to the number of processor clock counts per second.

If the type is INTEGER(2), *count_rate* is 1000. If the type is INTEGER(4), *count_rate* is 10000. If the type is INTEGER(8), *count_rate* is 1000000.

count_max (Output; optional) Must be scalar and of type integer. It is set to the maximum value that *count* can have, HUGE(0).

All integer arguments used must have the same integer kind parameter. If the type is INTEGER(1), *count*, *count_rate*, and *count_max* are all zero, indicating that there is no clock available to Intel® Fortran with an 8-bit range.

Example

Consider the following:

```
integer(2) :: ic2, crate2, cmax2
integer(4) :: ic4, crate4, cmax4
call system_clock(count=ic2, count_rate=crate2, count_max=cmax2)
call system_clock(count=ic4, count_rate=crate4, count_max=cmax4)
print *, ic2, crate2, cmax2
print *, ic4, crate4, cmax4
end
```

This program was run on Thursday Dec 11, 1997 at 14:23:55 EST and produced the following output:

```
13880   1000  32767
1129498807       10000  2147483647
```

See Also

DATE_AND_TIME
HUGE
GETTIM

SYSTEMQQ

Portability Function: *Executes a system command by passing a command string to the operating system's command interpreter.*

Module

USE IFPORT

Syntax

`result = SYSTEMQQ (commandline)`

commandline (Input) Character*(*). Command to be passed to the operating system.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The SYSTEMQQ function lets you pass operating-system commands as well as programs. SYSTEMQQ refers to the COMSPEC and PATH environment variables that locate the command interpreter file (usually named COMMAND.COM).

On Windows* systems, the calling process waits until the command terminates. To insure compatibility and consistent behavior, an image can be invoked directly by using the Windows API CreateProcess( ) in your Fortran code.

If the function fails, call GETLASTERRORQQ to determine the reason. One of the following errors can be returned:

- ERR$2BIG - The argument list exceeds 128 bytes, or the space required for the environment formation exceeds 32K.
- ERR$NOINT - The command interpreter cannot be found.
- ERR$NOEXEC - The command interpreter file has an invalid format and is not executable.
- ERR$NOMEM - Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

The command line character limit for the SYSTEMQQ function is the same limit that your operating system command interpreter accepts.

Example

```
USE IFPORT
LOGICAL(4) result
result = SYSTEMQQ('copy c:\bin\fmath.dat &
                        c:\dat\fmath2.dat')
```

See Also

SYSTEM

T to Z

T to Z

TAN

Elemental Intrinsic Function (Generic): *Produces the tangent of an argument.*

Syntax

`result = TAN (x)`

x	(Input) Must be of type real or complex. If *x* is of type real, it must be in radians and is treated as modulo.

Results

The result type and kind are the same as *x*.

If *x* is of type real, the result is a value in radians.

If *x* is of type complex, the real part of the result is a value in radians.

Specific Name	Argument Type	Result Type
TAN	REAL(4)	REAL(4)
DTAN	REAL(8)	REAL(8)
QTAN	REAL(16)	REAL(16)
CTAN [1]	COMPLEX(4)	COMPLEX(4)
CDTAN [2]	COMPLEX(8)	COMPLEX(8)

Specific Name	Argument Type	Result Type
CQTAN	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CTAN.

[2]This function can also be specified as ZTAN.

Example

TAN (2.0) has the value -2.185040.

TAN (0.8) has the value 1.029639.

TAND

Elemental Intrinsic Function (Generic): *Produces the tangent of an argument.*

Syntax

```
result = TAND (x)
```

x (Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type and kind are the same as *x*.

Specific Name	Argument Type	Result Type
TAND	REAL(4)	REAL(4)
DTAND	REAL(8)	REAL(8)
QTAND	REAL(16)	REAL(16)

Example

TAND (2.0) has the value 3.4920771E-02.

TAND (0.8) has the value 1.3963542E-02.

TANH

Elemental Intrinsic Function (Generic): *Produces a hyperbolic tangent.*

Syntax

```
result = TANH (x)
```

x (Input) Must be of type real or complex.

Results

The result type and kind are the same as *x*.

If *x* is of type complex, the imaginary part of the result is in radians.

Specific Name	Argument Type	Result Type
TANH	REAL(4)	REAL(4)
DTANH	REAL(8)	REAL(8)
QTANH	REAL(16)	REAL(16)

Example

TANH (2.0) has the value 0.9640276.

TANH (0.8) has the value 0.6640368.

TARGET DATA

OpenMP* Fortran Compiler Directive: *Maps variables to a device data environment for the extent of the region.*

Syntax

```
!$OMP TARGET DATA [clause[[,] clause]... ]
   block
!$OMP END TARGET DATA
```

clause

Is one or more of the following:

- DEVICE (integer-expression)
- IF ([TARGET DATA:] scalar-logical-expression)
- MAP ([[map-type-modifier[,]] map-type:] list)
- USE_DEVICE_PTR (*list*)

  Tells the construct to use a device pointer currently in the device data environment.

  The *list* items are converted into device pointers to the corresponding list item in the device data environment.

  References in the construct to a *list* item that appears in this clause must be to the address of the *list* item.

block

Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.

The binding task region for a TARGET DATA construct is the encountering task. The target region binds to the enclosing parallel or task region.

When a TARGET DATA construct is encountered, a new device data environment is created, and the encountering task executes the target data region.

A program must not depend on any ordering of the evaluations of the clauses of the TARGET DATA directive, or on any side effects of the evaluations of the clauses.

The same variable can be used in both the MAP clause and the USE_DEVICE_PTR clause:

```
!$OMP TARGET DATA MAP(X) USE_DEVICE_PTR(X)
    block
!$OMP END TARGET DATA
```

See Also

OpenMP Fortran Compiler Directives

Syntax Rules for Compiler Directives
TARGET Directive
TARGET UPDATE directive
Parallel Processing Model for information about Binding Sets

TARGET

OpenMP* Fortran Compiler Directive: *Creates a device data environment and executes the construct on that device.*

Syntax

```
!$OMP TARGET [clause[[,] clause]... ]
   block
!$OMP END TARGET
```

clause

Is one or more of the following:

- DEFAULTMAP (TOFROM:SCALAR)

  Causes a scalar variable to be treated as if it appeared in a MAP clause with a *map-type* of TOFROM.

  If this clause is not specified, a scalar variable is not mapped; instead it has an implicit attribute of FIRSTPRIVATE. At most one DEFAULTMAP clause can appear in the directive.
- DEPEND (dependence-type : list)
- DEVICE (integer-expression)
- IF ([TARGET:] scalar-logical-expression)
- IS_DEVICE_PTR (*list*)

  Indicates that a *list* item is a device pointer currently in the device data environment and that it should be used directly.

  The *list* is one or more dummy arguments.

  If a *list* item in a MAP clause is an array section, and the array section is derived from a variable with a POINTER or ALLOCATABLE attribute, then the behavior is unspecified if the corresponding list item's variable is modified in the region.
- MAP ([[map-type-modifier[,]] map-type:] list)
- NOWAIT

block

Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.

The binding task for a TARGET construct is the encountering task. The target region binds to the enclosing parallel or task region.

This construct provides a superset of the functionality provided by the TARGET DATA construct, except for the clause USE_DEVICE_PTR.

The TARGET construct also specifies that the region is executed by a device. The encountering task waits for the device to complete the target region at the end of the construct.

If a TARGET, TARGET DATA, or TARGET UPDATE construct appears within an OMP TARGET region, the construct is ignored.

If a variable appears in a REDUCTION or LASTPRIVATE clause on a combined construct for which the first construct is TARGET, it is treated as if it had appeared in a MAP clause with a *map-type* of TOFROM for REDUCTION and a *map-type* of FROM for LASTPRIVATE:

```
! SUM is treated as MAP (TOFROM)
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO REDUCTION(+:SUM)

! X is treated as MAP (FROM)
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO LASTPRIVATE(X)
```

A non-scalar variable referenced in a TARGET region and not explicitly mapped is implicitly treated as MAP (TOFROM):

```
INTEGER A(10)
!$OMP TARGET
  A(5) = 5          ! A is treated as MAP (TOFROM)
!$OMP  END TARGET
```

If only a subsection of a non-scalar variable has been mapped in an outer target region, and that variable is then mapped implicitly inside a nested target region, that variable should not access memory outside of the mapped subsection of the variable.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET DATA
TARGET UPDATE
DECLARE TARGET
Parallel Processing Model for information about Binding Sets

TARGET Statement

Statement and Attribute: *Specifies that an object can become the target of a pointer (it can be pointed to).*

Syntax

The TARGET attribute can be specified in a type declaration statement or a TARGET statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] TARGET [, *att-ls*] :: *object* [(*a-spec*)] [[*coarray-spec*]] [, *object*[(*a-spec*)] [[*coarray-spec*]]]...

Statement:

TARGET [::] *object* [(*a-spec*)] [[*coarray-spec*]] [, *object*[(*a-spec*)]] [[*coarray-spec*]]...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
object	Is the name of the object. The object must not be declared with the PARAMETER attribute.
a-spec	(Optional) Is an array specification or a coarray specification.
coarray-spec	(Optional) Is a deferred-coshape specification. The left bracket and right bracket are required.

Description

A pointer is associated with a target by pointer assignment or by an ALLOCATE statement.

If an object does not have the TARGET attribute or has not been allocated (using an ALLOCATE statement), no part of it can be accessed by a pointer.

Example

The following example shows type declaration statements specifying the TARGET attribute:

```
TYPE(SYSTEM), TARGET :: FIRST
REAL, DIMENSION(20, 20), TARGET :: C, D
```

The following is an example of a TARGET statement:

```
TARGET :: C(50, 50), D
```

The following fragment is from the program POINTER2.F90 in the `<install-dir>/samples` subdirectory:

```
! An example of pointer assignment.
    REAL, POINTER :: arrow1 (:)
    REAL, POINTER :: arrow2 (:)
    REAL, ALLOCATABLE, TARGET :: bullseye (:)

    ALLOCATE (bullseye (7))
    bullseye = 1.
    bullseye (1:7:2) = 10.
    WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye

    arrow1 => bullseye
    WRITE (*,'(/1x,a,7f8.0)') 'pointer',arrow1
    . . .
```

See Also

ALLOCATE
ASSOCIATED
POINTER
Pointer Assignments
Pointer Association
Type Declarations
Compatible attributes

TARGET ENTER DATA

OpenMP* Fortran Compiler Directive: *Specifies that variables are mapped to a device data environment.*

Syntax

```
!$OMP TARGET ENTER DATA [clause[[,] clause]... ]
```

clause

Is one or more of the following:

- DEPEND (dependence-type : list)
- DEVICE (integer-expression)
- IF ([TARGET ENTER DATA:] scalar-logical-expression)
- MAP ([map-type:] list)
- NOWAIT

The binding task for the TARGET ENTER DATA construct is the encountering task. TARGET ENTER DATA is a stand-alone directive.

When a TARGET ENTER DATA construct is encountered, the *list* items in the MAP clauses are mapped to the device data environment according to *map-type*. A *map-type* must be specified in all MAP clauses and must be either TO or ALLOC.

The TARGET ENTER DATA construct executes as if it was enclosed in a TASK construct.

When a DEPEND clause is present, it acts as if it appeared on the implicit TASK construct that encloses the TARGET ENTER DATA construct.

If there is no DEVICE clause, the default device is determined by the internal control variable (ICV) named `default-device-var`.

When an IF clause is present and the IF clause *scalar-logical-expression* evaluates to .FALSE., the device is the host.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET EXIT DATA
Parallel Processing Model for information about Binding Sets

TARGET EXIT DATA

OpenMP* Fortran Compiler Directive: *Specifies that variables are unmapped from a device data environment.*

Syntax

```
!$OMP TARGET EXIT DATA [clause[[,] clause]... ]
```

clause — Is one or more of the following:

- DEPEND (dependence-type : list)
- DEVICE (integer-expression)
- IF ([TARGET EXIT DATA:] scalar-logical-expression)
- MAP ([map-type:] list)
- NOWAIT

The binding task for the TARGET EXIT DATA construct is the encountering task. TARGET EXIT DATA is a stand-alone directive.

When a TARGET EXIT DATA construct is encountered, the *list* items in the MAP clauses are unmapped from the device data environment according to *map-type*. A *map-type* must be specified in all MAP clauses and must be FROM, RELEASE, or DELETE.

The TARGET EXIT DATA construct executes as if it was enclosed in a task construct.

When a DEPEND clause is present, it acts as if it appeared on the implicit task construct that encloses the TARGET EXIT DATA construct.

If there is no DEVICE clause, the default device is determined by the internal control variable (ICV) named `default-device-var`.

When an IF clause is present and the IF clause *scalar-logical-expression* evaluates to .FALSE., the device is the host.

See Also

OpenMP Fortran Compiler Directives

Syntax Rules for Compiler Directives
TARGET ENTER DATA
Parallel Processing Model for information about Binding Sets

TARGET PARALLEL

OpenMP* Fortran Compiler Directive: *Creates a device data environment in a parallel region and executes the construct on that device.*

Syntax

```
!$OMP TARGET PARALLEL [clause[[,] clause] ... ]
      block
[!$OMP END TARGET PARALLEL]
```

clause	Can be any of the clauses accepted by the TARGET or PARALLEL directives with identical meanings and restrictions.
block	Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.

This directive provides a shortcut for specifying a TARGET construct immediately followed by a PARALLEL construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately. The only exceptions are the following:

- If any IF clause in the directive includes a *directive-name-modifier* then all IF clauses in the directive must include a directive-name-modifier.
- At most one IF clause with no *directive-name-modifier* can appear on the directive.
- At most one IF clause with the PARALLEL *directive-name-modifier* can appear on the directive.
- At most one IF clause with the TARGET *directive-name-modifier* can appear on the directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
IF Clause
TARGET Directive
PARALLEL Directive

TARGET PARALLEL DO

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a TARGET directive containing a PARALLEL DO directive and no other statements.*

Syntax

```
!$OMP TARGET PARALLEL DO [clause[[,] clause] ... ]
      do-loop
[!$OMP END TARGET PARALLEL DO]
```

clause	Can be any of the clauses accepted by the TARGET or PARALLEL DO directives, except for the COPYIN clause.

do-loop — Is a DO iteration (a DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

You cannot branch out of a DO loop associated with a DO directive.

If the END TARGET PARALLEL DO directive is not specified, the TARGET PARALLEL DO is assumed at the end of *do-loop*. If used, the END TARGET PARALLEL DO directive must appear immediately after the end of *do-loop*.

The semantics are identical to explicitly specifying a TARGET directive immediately followed by a PARALLEL DO directive.

The restrictions for the TARGET and PARALLEL DO constructs apply to this directive except for the following:

- If any IF clause in the directive includes a *directive-name-modifier* then all IF clauses in the directive must include a directive-name-modifier.
- At most one IF clause with no *directive-name-modifier* can appear on the directive.
- At most one IF clause with the PARALLEL *directive-name-modifier* can appear on the directive.
- At most one IF clause with the TARGET *directive-name-modifier* can appear on the directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
IF Clause
TARGET Directive
PARALLEL DO

TARGET PARALLEL DO SIMD

OpenMP* Fortran Compiler Directive: *Specifies a TARGET construct that contains a PARALLEL DO SIMD construct and no other statement.*

Syntax

```
!$OMP TARGET PARALLEL DO SIMD [clause[[,] clause] ... ]
      do-loop
[!$OMP END TARGET PARALLEL DO SIMD]
```

clause — Can be any of the clauses accepted by the TARGET or PARALLEL DO SIMD directives, except for the COPYIN clause.

do-loop — Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

You cannot branch out of a DO loop associated with a DO SIMD directive.

If the END TARGET PARALLEL DO SIMD directive is not specified, an END TARGET PARALLEL DO SIMD directive is assumed at the end of do-loop.

The restrictions for the TARGET and PARALLEL DO SIMD constructs apply to this directive except for the following:

- If any IF clause in the directive includes a directive-name-modifier then all IF clauses in the directive must include a directive-name-modifier.
- At most one IF clause with no directive-name-modifier can appear on the directive.
- At most one IF clause with the PARALLEL directive-name-modifier can appear on the directive.
- At most one IF clause with the TARGET directive-name-modifier can appear on the directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
IF Clause
TARGET Directive
PARALLEL DO SIMD

TARGET SIMD

OpenMP* Fortran Compiler Directive: *Specifies a TARGET construct that contains a SIMD construct and no other statement.*

Syntax

```
!$OMP TARGET SIMD [clause[[,] clause] ... ]
      do-loop
[!$OMP END TARGET SIMD]
```

clause	Can be any of the clauses accepted by the TARGET or SIMD directive, with identical meanings and restrictions
do-loop	Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer. All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops. The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team. You cannot branch out of a DO loop associated with a DO SIMD directive.

If the END TARGET SIMD directive is not specified, an END TARGET SIMD directive is assumed at the end of do-loop.

The restrictions for the TARGET and SIMD constructs apply to this directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TARGET Directive
SIMD Directive (OpenMP* API)

TARGET TEAMS

OpenMP* Fortran Compiler Directive: *Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the master thread in each team executing the structured block.*

Syntax

```
!$OMP TARGET TEAMS [clause[[,] clause]... ]
   block
!$OMP END TARGET TEAMS
```

clause	Can be any of the clauses accepted by the TARGET or TEAMS directives with identical meanings and restrictions.
block	Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.

This directive provides a shortcut for specifying a TARGET construct immediately followed by a TEAMS construct.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET Directive
TEAMS

TARGET TEAMS DISTRIBUTE

OpenMP* Fortran Compiler Directive: *Creates a device data environment and executes the construct on the same device. It also specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a TEAMS construct.*

Syntax

```
!$OMP TARGET TEAMS DISTRIBUTE [clause[[,] clause]... ]
   do-loop
[!$OMP END TARGET TEAMS DISTRIBUTE]
```

clause	Can be any of the clauses accepted by the TARGET or TEAMS DISTRIBUTE directives with identical meanings and restrictions.
do-loop	Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer. All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TARGET construct followed immediately by a TEAMS DISTRIBUTE construct.

If the END TARGET TEAMS DISTRIBUTE directive is not specified, an END TARGET TEAMS DISTRIBUTE directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET Directive
TEAMS DISTRIBUTE

TARGET TEAMS DISTRIBUTE PARALLEL DO

OpenMP* Fortran Compiler Directive: *Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct.*

Syntax

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO [clause[[,] clause]... ]
   do-loop
[!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO]
```

clause
Can be any of the clauses accepted by the TARGET or TEAMS DISTRIBUTE PARALLEL DO directives with identical meanings and restrictions.

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TARGET construct followed immediately by a TEAMS DISTRIBUTE PARALLEL DO construct.

If the END TARGET TEAMS DISTRIBUTE PARALLEL DO directive is not specified, an END TARGET TEAMS DISTRIBUTE PARALLEL DO directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TARGET Directive
TEAMS DISTRIBUTE PARALLEL DO

TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD

OpenMP* Fortran Compiler Directive: *Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD]
```

clause
Can be any of the clauses accepted by the TARGET or TEAMS DISTRIBUTE PARALLEL DO SIMD directives with identical meanings and restrictions.

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TARGET construct followed immediately by a TEAMS DISTRIBUTE PARALLEL DO SIMD construct.

If the END TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD directive is not specified, an END TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET Directive
TEAMS DISTRIBUTE PARALLEL DO SIMD

TARGET TEAMS DISTRIBUTE SIMD

OpenMP* Fortran Compiler Directive: *Creates a device data environment and executes the construct on the same device. It also specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a TEAMS construct. It will be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP TARGET TEAMS DISTRIBUTE SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END TARGET TEAMS DISTRIBUTE SIMD]
```

clause
Can be any of the clauses accepted by the TARGET or TEAMS DISTRIBUTE SIMD directives with identical meanings and restrictions.

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TARGET construct containing a TEAMS DISTRIBUTE SIMD construct.

If the END TARGET TEAMS DISTRIBUTE SIMD directive is not specified, an END TARGET TEAMS DISTRIBUTE SIMD directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET Directive
TEAMS DISTRIBUTE SIMD

TARGET UPDATE

OpenMP* Fortran Compiler Directive: *Makes the list items in the device data environment consistent with their corresponding original list items.*

Syntax

```
!$OMP TARGET UPDATE motion-clause [, clause[[,] clause]... ]
```

motion-clause
Is one of the following:

- `FROM` (*list*)
- `TO` (*list*)

The *list* is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a common block name must appear between slashes (/ /).

For each *list* item in a TO or FROM clause, there is a corresponding list item and an original list item; for more information, see the MAP clause. Note that if the corresponding list item is not present in the device data environment, the behavior is unspecified.

	For each *list* item in a TO clause, the value of the original list item is assigned to the corresponding list item. For each *list* item in a FROM clause, the value of the corresponding list item is assigned to the original list item. A *list* item may only appear in a TO or FROM clause, but not both.
clause	Is one of the following: • DEPEND (dependence-type : list) • DEVICE (integer-expression) • IF ([TARGET UPDATE:] scalar-logical-expression) • NOWAIT

The binding task for a TARGET UPDATE construct is the encountering task. The TARGET UPDATE directive is a stand-alone directive.

A TARGET UPDATE construct must not appear inside of a target region.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TARGET Directive
TARGET DATA
Parallel Processing Model for information about Binding Sets

TASK

OpenMP* Fortran Compiler Directive: *Defines a task region.*

Syntax

```
!$OMP TASK [clause[[,] clause] ... ]
   block
!$OMP END TASK
```

clause	Is one of the following: • DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE) • DEPEND (dependence-type : list) • FINAL (scalar-logical-expression) • FIRSTPRIVATE (list) • IF ([TASK:] scalar-logical-expression) • IN_REDUCTION (operator | intrinsic: list) • MERGEABLE • PRIORITY (priority-value) • PRIVATE (list) • SHARED (list) • UNTIED
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

The binding thread set of a TASK construct is the current team. A task region binds to the innermost enclosing parallel region.

The TASK and END TASK directive pair must appear in the same routine in the executable section of the code.

The END TASK directive denotes the end of the task.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.

A thread that encounters a task scheduling point within the task region may temporarily suspend the task region. By default, a task is then tied and its suspended task region can only be resumed by the thread that started its execution. However, if the untied clause is specified in a TASK construct, any thread in the team can resume the task region after a suspension. The untied clause is ignored in these cases:

- If a final clause has been specified in the same TASK construct and the final clause expression evaluates to .TRUE..
- If a task is an included task.

A TASK construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

The TASK construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit task region includes a task scheduling point at its point of completion. An implementation may add task scheduling points anywhere in untied task regions.

Note that when storage is shared by an explicit task region, you must add proper synchronization to ensure that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

A program must not depend on any ordering of the evaluations of the clauses of the TASK directive and it must not depend on any side effects of the evaluations of the clauses. A program that branches into or out of a task region is non-conforming.

Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has unspecified behavior.

Example

The following example calculates a Fibonacci number. The Fibonacci sequence is 1,1,2,3,5,8,13, etc., where the current number is the sum of the previous two numbers. If a call to function fib is encountered by a single thread in a parallel region, a nested task region will be spawned to carry out the computation in parallel.

```
RECURSIVE INTEGER FUNCTION fib(n)
INTEGER n, i, j
IF ( n .LT. 2) THEN
  fib = n
ELSE
  !$OMP TASK SHARED(i)
     i = fib( n-1 )
  !$OMP END TASK
  !$OMP TASK SHARED(j)
     j = fib( n-2 )
  !$OMP END TASK
  !$OMP TASKWAIT      ! wait for the sub-tasks to
                            !   complete before summing
     fib = i+j
END IF
END FUNCTION
```

The following example generates a large number of tasks in one thread and executes them with the threads in the parallel team. While generating these tasks, if the implementation reaches the limit generating unassigned tasks, the generating loop may be suspended and the thread used to execute unassigned tasks. When the number of unassigned tasks is sufficiently low, the thread resumes execution of the task generating loop.

```
real*8 item(10000000)
integer i
!$omp parallel
!$omp single     ! loop iteration variable i is private
    do i=1,10000000
!$omp task
! i is firstprivate, item is shared
    call process(item(i))
!$omp end task
    end do
!$omp end single
!$omp end parallel
end
```

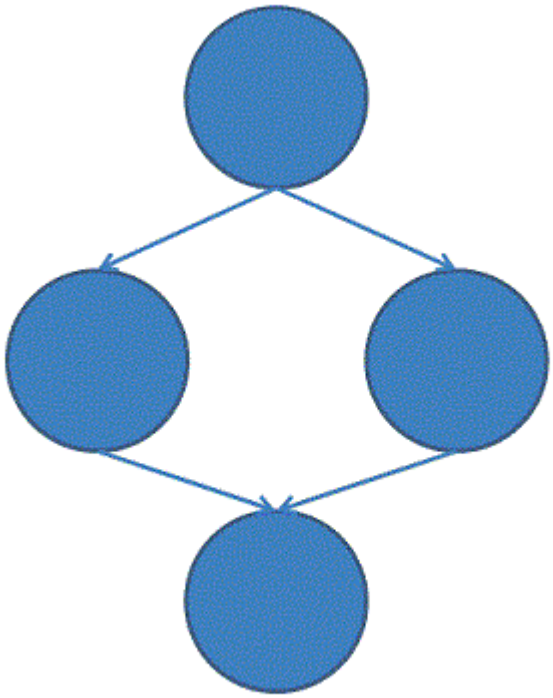
The following example modifies the previous one to use an untied task to generate the unassigned tasks. If the implementation reaches the limit generating unassigned tasks and the generating loop is suspended, any other thread that becomes available can resume the task generation loop.

```
real*8 item(10000000)
!$omp parallel
!$omp single!
$omp task untied
! loop iteration variable i is private
    do i=1,10000000
!$omp task     ! i is firstprivate, item is shared
    call process(item(i))
!$omp end task
    end do
!$omp end task
!$omp end single
!$omp end parallel
```

The following example demonstrates four tasks with dependences:

```
integer :: a

!$omp task depend(out:a)
!$omp end task

!$omp task depend(in:a)
!$omp end task

!$omp task depend(in:a)
!$omp end task

!$omp task depend(out:a)
!$omp end task
```

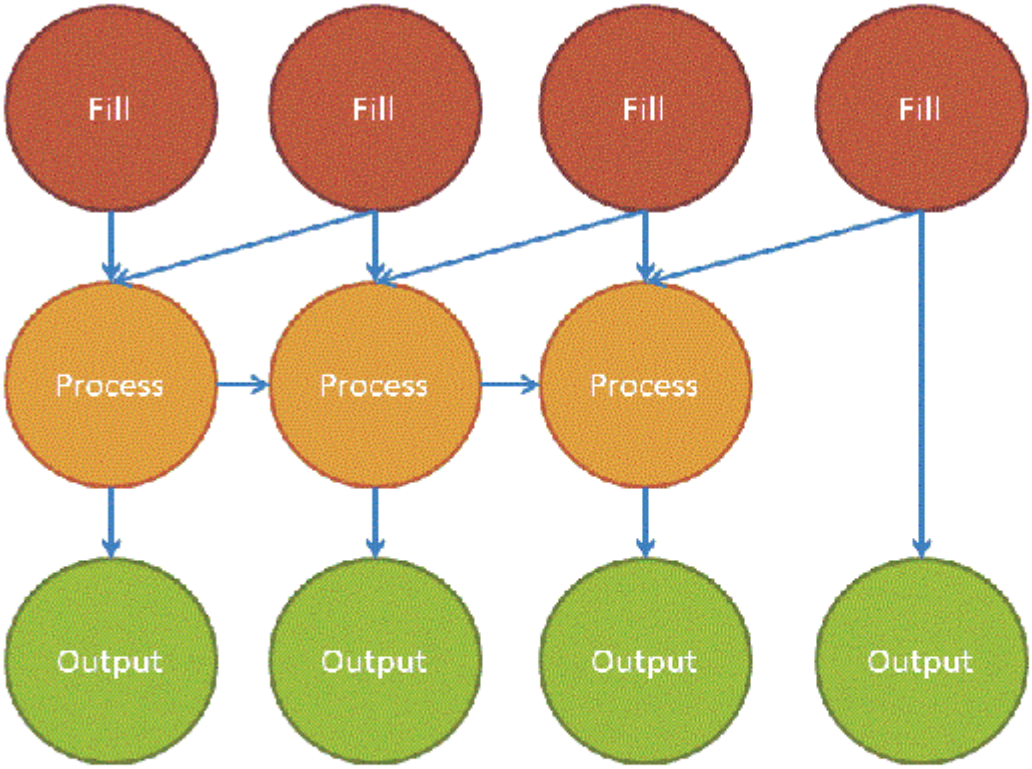
In the above example, the first task does not depend on any previous one. The second and third tasks depend on the first task but not on each other. The last task depends on the second and third tasks. The following shows the dependency graph:



The following example shows a set of sibling tasks that have dependences between them:

```
INTEGER :: A(0:N*B-1)

DO I=0,N-1
!$OMP TASK DEPEND(OUT:A(I*B:(I+1)*B-1))
      CALL FILL(A(I*B:(I+1)*B-1))
!$OMP END TASK
END DO

DO I=1,N-1
      IB = (I-1)*B+1
      BB = I*B+1
!$OMP TASK DEPEND(INOUT:A(I*B:(I+1)*B-1)) DEPEND(IN:A((I+1)*B:(I+2)*B-1))
      CALL PROCESS(A(I*B:(I+1)*B-1), A((I+1)*B:(I+2)*B-1))
!$OMP END TASK
END DO

DO I=1,N
      IB = (I-1)*B+1
!$OMP TASK DEPEND(IN:A(I*B:(I+1)*B-1))
      CALL OUTPUT(A(I*B:(I+1)*B-1))
!$OMP END TASK
END DO
```

In the above example, the tasks of the first loop will be independent of any other tasks since there is no previous task that expresses a dependence on the same list items. Tasks of the second loop will depend on two tasks from the first loop. Also, because dependences are constructed in a sequential order, the IN dependences force the tasks of the second loop to be dependent on the task from the previous iteration to be processed. Finally, tasks of the third loop can be executed when the corresponding Process task from the second loop has been executed. For example, if N was 4, the following shows the dependency graph:



See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
OpenMP* Run-time Library Routines for Fortran
PARALLEL DO
TASKWAIT
TASKYIELD
SHARED Clause
Parallel Processing Model for information about Binding Sets

TASK_REDUCTION

Specifies a reduction among tasks. The TASK_REDUCTION clause is a reduction scoping clause.

Syntax

```
TASK_REDUCTION (reduction-identifier : list)
```

Arguments *reduction-identifier* and *list* are defined in REDUCTION clause. All the common restrictions for the REDUCTION clause apply to this clause.

See Also

IN_REDUCTION
DECLARE REDUCTION
REDUCTION
TASKGROUP

TASKGROUP

OpenMP* Fortran Compiler Directive: *Specifies a wait for the completion of all child tasks of the current task and all of their descendant tasks.*

Syntax

```
!$OMP TASKGROUP [clause[[,] clause] ... ]
   block
!$OMP END TASKGROUP
```

clause	Is the following: • TASK_REDUCTION (reduction-identifier : list)
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

A TASKGROUP construct binds to the current task region. The binding thread set of the taskgroup region is the current team.

When a thread encounters a TASKGROUP construct, it starts executing the region. There is an implicit task scheduling point at the end of the TASKGROUP region. The current task is suspended at the task scheduling point until all child tasks that it generated in the TASKGROUP region and all of their descendant tasks complete execution.

Any number of reduction clauses can be specified in the TASKGROUP directive, but a list item can appear only once in reduction clauses for that directive.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TASK
TASKWAIT
TASKYIELD
Parallel Processing Model for information about Binding Sets

TASKLOOP

OpenMP* Fortran Compiler Directive: *Specifies that the iterations of one or more associated DO loops should be executed in parallel using OpenMP* tasks. The iterations are distributed across tasks that are created by the construct and scheduled to be executed.*

Syntax

```
!$OMP TASKLOOP [clause[[,] clause]... ]
   do-loop
[!$OMP END TASKLOOP]
```

clause

Is one of the following:

- COLLAPSE (n)
- DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
- FINAL (scalar-logical-expression)
- FIRSTPRIVATE (list)
- GRAINSIZE (*grain-size*)

  Specifies that the number of logical loop iterations assigned to each created task is greater than or equal to the minimum of the value of the *grain-size* positive integer expression and the number of logical loop iterations, but less than two times the value of the *grain-size* expression.

  At most one GRAINSIZE clause can appear in a TASKLOOP directive.
- IF ([TASKLOOP:] *scalar-logical-expression*)

  If the *scalar-logical-expression* evaluates to false, undeferred tasks are generated. If a variable appears in the IF clause expression, it causes an implicit reference to the variable in all enclosing constructs.

  At most one IF clause can appear in a TASKLOOP directive.
- IN_REDUCTION (reduction-identifier : list)
- LASTPRIVATE ([CONDITIONAL:] list)
- MERGEABLE
- NOGROUP

  Specifies that no implicit taskgroup region is created.
- NUM_TASKS (*num-tasks*)

  Causes the TASKLOOP construct to create as many tasks as the minimum of the *num-tasks* positive scalar integer expression and the number of logical loop iterations. Each task must have at least one logical loop iteration.
- PRIORITY (priority-value)
- PRIVATE (list)
- REDUCTION (reduction-identifier : list)

  If you specify this clause, you cannot specify the NOGROUP clause.
- SHARED (list)
- UNTIED

do-loop	Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer. If an END TASKLOOP directive follows a DO construct in which several loop statements share a DO termination statement, then the directive can only be specified for the outermost of these DO statements. The TASKLOOP construct inherits the restrictions of the loop construct. If any of the loop iteration variables would otherwise be shared, they are implicitly made private for the loop-iteration tasks created by the TASKLOOP construct. Unless the loop iteration variables are specified in a LASTPRIVATE clause on the TASKLOOP construct, their values after the loop are unspecified. You cannot branch out of a DO loop associated with a TASKLOOP directive.

If you do not specify GRAINSIZE or NUM_TASKS, the number of loop tasks created and the number of logical loop iterations assigned to these tasks is implementation defined. The GRAINSIZE clause and NUM_TASKS clauses are mutually exclusive; they *cannot* appear in the same TASKLOOP directive.

The binding thread set of the TASKLOOP region is the current team. A TASKLOOP region binds to the innermost enclosing parallel region.

When a thread encounters a TASKLOOP construct, the construct partitions the associated loops into tasks for parallel execution of the loop iterations. The data environment of the created tasks is created according to the clauses specified in the TASKLOOP construct, any data environment ICVs, and any defaults that apply. The order of the creation of the loop tasks is unspecified. Programs that rely on any execution order of the logical loop iterations are non-conforming.

If used, the END TASKLOOP directive must appear immediately after the end of the loop. If you do not specify an END TASKLOOP directive, an END TASKLOOP directive is assumed at the end of the *do-loop*.

By default, the TASKLOOP construct executes as if it was enclosed in a TASKGROUP construct with no statements or directives outside of the TASKLOOP construct. Therefore, the TASKLOOP construct creates an implicit TASKGROUP region.

When a REDUCTION clause appears in a TASKLOOP construct, it behaves as if a TASK_REDUCTION clause was applied to the implicit TASKGROUP construct enclosing the TASKLOOP construct. For each task that is generated by the TASKLOOP construct, an IN_REDUCTION clause with the same reduction operator and the same list items that appear in the REDUCTION clause is applied to the task.

When an IN_REDUCTION clause appears in a TASKLOOP construct, an IN_REDUCTION clause with the same reduction operation and the same list items of the IN_REDUCTION clause is applied to the generated tasks.

The following restrictions also apply:

- A program that branches into or out of a TASKLOOP region is non-conforming.
- All loops associated with the TASKLOOP construct must be perfectly nested. There must be no intervening code nor any other OpenMP* directive between any two loops.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TASK_REDUCTION

TASKLOOP SIMD

OpenMP* Fortran Compiler Directive: *Specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using OpenMP* tasks.*

Syntax

```
!$OMP TASKLOOP SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END TASKLOOP SIMD]
```

clause
Can be almost all of the clauses accepted by the TASKLOOP or SIMD directives with identical meanings and restrictions. The only exception is that you cannot specify a REDUCTION clause.

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team. You cannot branch out of a DO loop associated with a TASKLOOP directive.

If the END TASKLOOP SIMD directive is not specified, an END TASKLOOP SIMD directive is assumed at the end of do-loop.

The binding thread set of the TASKLOOP SIMD region is the current team. A TASKLOOP SIMD region binds to the innermost enclosing parallel region.

The TASKLOOP SIMD construct first distributes the iterations of the associated loops across tasks in a manner consistent with any clauses that apply to the TASKLOOP construct. The resulting tasks are then converted to SIMD loops in a manner consistent with any clauses that apply to the SIMD construct.

The effect of any clause that applies to both constructs is as if it were applied to both constructs separately, except for the COLLAPSE clause, which is applied once to the TASKLOOP construct.

This directive specifies a composite construct.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TASKWAIT

OpenMP* Fortran Compiler Directive: *Specifies a wait on the completion of child tasks generated since the beginning of the current task.*

Syntax

```
!$OMP TASKWAIT
```

A TASKWAIT construct binds to the current task region. The binding thread set of the taskwait region is the current team.

The TASKWAIT region includes an implicit task scheduling point in the current task region. The current task region is suspended at the task scheduling point until execution of all its child tasks generated before the TASKWAIT region are completed.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TASK directive
TASKYIELD directive
Parallel Processing Model for information about Binding Sets

TASKYIELD

OpenMP* Fortran Compiler Directive: *Specifies that the current task can be suspended at this point in favor of execution of a different task.*

Syntax

```
!$OMP TASKYIELD
```

A taskyield region binds to the current task region. The binding thread set of the taskyield region is the current team.

Because the TASKYIELD construct is a stand-alone directive, there are some restrictions on its placement within a program:

- The TASKYIELD directive can only be placed at a point where a Fortran executable statement is allowed.
- The TASKYIELD directive cannot be used as the action statement in an IF statement or as the executable statement following a label if the label is referenced in the program.

The TASKYIELD construct includes an explicit task scheduling point in the current task region.

Example

The following example shows use of the TASKYIELD directive. It is non-conforming, because the FLUSH, BARRIER, TASKWAIT, and TASKYIELD directives are stand-alone directives and cannot be the action statement of an IF statement or a labeled branch target.

```
SUBROUTINE NONCONFORMING_STANDALONE()
   INTEGER A
   A = 1
   ! the FLUSH directive must not be the action statement in an IF statement
   !
   IF (A .NE. 0) !$OMP FLUSH(A)

   ! the BARRIER directive must not be the action statement in an IF statement
   !
   IF (A .NE. 0) !$OMP BARRIER

   ! the TASKWAIT directive must not be the action statement in an IF statement
   !
   IF (A .NE. 0) !$OMP TASKWAIT

   ! the TASKYIELD directive must not be the action statement in an IF statement
   !
   IF (A .NE. 0) !$OMP TASKYIELD
```

```
   GOTO 100

   ! the FLUSH directive must not be a labeled branch target statement
   !
   100 !$OMP FLUSH(A)
   GOTO 200

   ! the BARRIER directive must not be a labeled branch target statement
   !
   200 !$OMP BARRIER
   GOTO 300

   ! the TASKWAIT directive must not be a labeled branch target statement
   !
   300 !$OMP TASKWAIT
   GOTO 400

   ! the TASKYIELD directive must not be a labeled branch target statement
   !
   400 !$OMP TASKYIELD
END SUBROUTINE
```

The following version of the above example is conforming because the FLUSH, BARRIER, TASKWAIT, and TASKYIELD directives are enclosed in a compound statement.

```
SUBROUTINE CONFORMING_STANDALONE ()
   INTEGER N
   N = 1
   IF (N .NE. 0) THEN
       !$OMP FLUSH(N)
   ENDIF
   IF (N .NE. 0) THEN
       !$OMP BARRIER
   ENDIF
   IF (N .NE. 0) THEN
       !$OMP TASKWAIT
   ENDIF
   IF (N .NE. 0) THEN
       !$OMP TASKYIELD
   ENDIF
   GOTO 100
   100 CONTINUE
   !$OMP FLUSH(N)
   GOTO 200
   200 CONTINUE
   !$OMP BARRIER
   GOTO 300
   300 CONTINUE
   !$OMP TASKWAIT
   GOTO 400
   400 CONTINUE
   !$OMP TASKYIELD
END SUBROUTINE
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
TASK directive

TASKWAIT directive
Parallel Processing Model for information about Binding Sets

TEAMS

OpenMP* Fortran Compiler Directive: *Creates a league of thread teams inside a target region to execute a structured block in the master thread of each team.*

Syntax

```
!$OMP TEAMS [clause[[,] clause]... ]
   block
!$OMP END TEAMS
```

clause

Is one of the following:

- DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
- FIRSTPRIVATE (list)
- NUM_TEAMS (*scalar-integer-expression*)

  Specifies the number of teams to be used in a parallel region. The *scalar-integer-expression* must evaluate to a positive scalar integer value.

  Only a single NUM_TEAMS clause can appear in the directive.
- PRIVATE (list)
- REDUCTION (reduction-identifier : list)
- SHARED (list)
- THREAD_LIMIT (*scalar-integer-expression*)

  Specifies the maximum number of threads participating in the contention group that each team initiates. The *scalar-integer-expression* must evaluate to a positive scalar integer value.

  At most one THREAD_LIMIT clause can appear in the directive.

block

Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.

The binding thread set for a TEAMS construct is the encountering thread.

The TEAMS construct must appear within a TARGET construct. The enclosing TARGET construct must contain no other statements or directives outside of the TEAMS construct. The following are the only OpenMP* constructs that can be nested in the team's region:

- DISTRIBUTE
- PARALLEL
- PARALLEL SECTIONS
- PARALLEL WORKSHARE
- PARALLEL DO
- PARALLEL DO SIMD

The thread that encounters this directive constructs a league of thread teams that execute the *block* in the master thread of each team. After the teams are created, the number of teams remains constant for the duration of the team's parallel region.

The region following the TEAMS construct is executed by the master thread of each team. Other threads do not begin execution until the master thread encounters a parallel region. After the teams complete execution of the TEAMS construct region, the encountering thread resumes execution of the enclosing target region.

If NUM_TEAMS is not specified, the default number of teams is one. If THREAD_LIMIT is not specified, the default number of threads is *the-number-of-available-hardware-threads* / NUM_TEAMS.

A program must not depend on any side effects or any ordering of the evaluation of clauses in the TEAMS directive.

Each team has a unique team number. You can use the OpenMP* run-time library routine OMP_GET_TEAM_NUM to get the team number of the calling thread.

Each thread within the team has a unique thread identifier returned by the OpenMP run-time library routine OMP_GET_THREAD_NUM. As in any thread team, the thread identifier starts at zero for the master thread up to THREAD_LIMIT - 1 for the remaining threads.

Immediately after this directive, only the master threads in each team are executing, the other team members will only start to execute at the next (nested) parallel region. Therefore, there are only NUM_TEAMS threads executing, and each of them has OMP_GET_THREAD_NUM() == 0.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
Parallel Processing Model for information about Binding Sets

TEAMS DISTRIBUTE

OpenMP* Fortran Compiler Directive: *Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a TEAMS construct.*

Syntax

```
!$OMP TEAMS DISTRIBUTE [clause[[,] clause]... ]
   do-loop
[!$OMP END TEAMS DISTRIBUTE]
```

clause
Can be any of the clauses accepted by the TEAMS or DISTRIBUTE directives with identical meanings and restrictions.

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TEAMS construct containing a DISTRIBUTE construct.

If the END TEAMS DISTRIBUTE directive is not specified, an END TEAMS DISTRIBUTE directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TEAMS DISTRIBUTE PARALLEL DO

OpenMP* Fortran Compiler Directive: *Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.*

Syntax

```
!$OMP TEAMS DISTRIBUTE PARALLEL DO [clause[[,] clause]... ]
   do-loop
[!$OMP END TEAMS DISTRIBUTE PARALLEL DO]
```

clause
Can be any of the clauses accepted by the TEAMS or DISTRIBUTE PARALLEL DO directives with identical meanings and restrictions.

do-loop
Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TEAMS construct immediately followed by a DISTRIBUTE PARALLEL DO construct. Some clauses are permitted on both constructs.

If the END TEAMS DISTRIBUTE PARALLEL DO directive is not specified, an END TEAMS DISTRIBUTE PARALLEL DO directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TEAMS DISTRIBUTE PARALLEL DO SIMD

OpenMP* Fortran Compiler Directive: *Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams. The loop will be distributed across the master threads of the teams region, which will be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP TEAMS DISTRIBUTE PARALLEL DO SIMD [clause[[,] clause]... ]
```

```
   do-loop
[!$OMP END TEAMS DISTRIBUTE PARALLEL DO SIMD]
```

clause — Can be any of the clauses accepted by the TEAMS or DISTRIBUTE PARALLEL DO SIMD directives with identical meanings and restrictions.

do-loop — Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TEAMS construct immediately followed by a DISTRIBUTE PARALLEL DO SIMD construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately.

If the END TEAMS DISTRIBUTE PARALLEL DO SIMD directive is not specified, an END TEAMS DISTRIBUTE PARALLEL DO SIMD directive is assumed at the end of do-loop.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TEAMS DISTRIBUTE SIMD

OpenMP* Fortran Compiler Directive: *Creates a league of thread teams to execute the structured block in the master thread of each team. It also specifies a loop that will be distributed across the master threads of the teams region. The loop will be executed concurrently using SIMD instructions.*

Syntax

```
!$OMP TEAMS DISTRIBUTE SIMD [clause[[,] clause]... ]
   do-loop
[!$OMP END TEAMS DISTRIBUTE SIMD]
```

clause — Can be any of the clauses accepted by the TEAMS or DISTRIBUTE SIMD directives with identical meanings and restrictions.

do-loop — Is one or more DO iterations (DO loops). The DO iteration cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.

All loops associated with the construct must be structured and perfectly nested; that is, there must be no intervening code and no other OpenMP* Fortran directives between any two loops.

The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.

This directive provides a shortcut for specifying a TEAMS construct immediately followed by a DISTRIBUTE SIMD construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately.

If the END TEAMS DISTRIBUTE SIMD directive is not specified, an END TEAMS DISTRIBUTE SIMD directive is assumed at the end of *do-loop*.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

THIS_IMAGE

Transformational Intrinsic Function (Generic):
Returns cosubscripts for the image.

Syntax

```
result = THIS_IMAGE ([coarray [, dim]])
```

coarray (Input; optional) Must be a coarray; it can be of any type. If it is allocatable, it must be allocated.

dim (Input; optional) Must be a scalar of type integer. Its value must be in the range 1 <= *dim* <= *n*, where *n* is the corank of *coarray*. The corresponding actual argument must not be an optional dummy argument.

Results

The result type is default integer. It is scalar if *coarray* does not appear or *dim* is present; otherwise, the result has rank one and its size is equal to the corank of *coarray*.

The result depends on which arguments are specified:

- If no argument is specified, as in THIS_IMAGE ( ), the result is a scalar with a value equal to the index of the invoking image.
- If only *coarray* is specified, the result is the sequence of cosubscript values for *coarray* that would specify the invoking image.
- If *coarray* and *dim* are specified, the result is the value of cosubscript *dim* in the sequence of cosubscript values for *coarray* that would specify the invoking image.

Example

Consider that coarray C is declared by the following statement:

```
REAL C(5, 10) [10, 0:9, 0:*]
```

On image 5, THIS_IMAGE ( ) has the value 5 and THIS IMAGE (C) has the value [5, 0, 0]. For the same coarray on image 213, THIS_IMAGE (C) has the value [3, 1, 2].

In the following example, image 1 is used to read data. The other images then copy the data:

```
IF (THIS_IMAGE()==1) READ (*,*) P
SYNC ALL
P = P[1]
```

THREADPRIVATE

OpenMP* Fortran Compiler Directive: *Specifies named common blocks and certain variables to be private (local) to each thread; they are global within the thread.*

Syntax

```
!$OMP THREADPRIVATE (list)
```

list

Is a comma-separated list of named common blocks, module variables, or variables that have the SAVE attribute. These objects are made private to a thread.

Note that common blocks must appear between slashes (/).

A blank common block cannot appear in a THREADPRIVATE directive.

A variable that appears in a THREADPRIVATE directive must be declared in the scope of a module or have the SAVE attribute, either explicitly or implicitly.

Each thread gets its own copy of the common block or variable, so data written to this object by one thread is not directly visible to other threads.

During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block or variable. On entry to the first parallel region, data in the THREADPRIVATE common blocks or variables should be assumed to be undefined unless a COPYIN clause is specified in the PARALLEL directive.

The THREADPRIVATE directive must appear in the section of a scoping unit in which the common block or variable is declared. Although variables in common blocks can be accessed by use association or host association, common block names cannot. This means that a common block name specified in a THREADPRIVATE directive must be declared to be a common block in the same scoping unit in which the THREADPRIVATE directive appears.

When a common block (which is initialized using DATA statements) appears in a THREADPRIVATE directive, each thread copy is initialized once prior to its first use. For subsequent parallel regions, data in THREADPRIVATE common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions.

A THREADPRIVATE common block or its constituent variables can appear only in a COPYIN clause. They are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

If a THREADPRIVATE directive specifying a common block name appears in one program unit, then the directive must also appear in every other program unit that contains a COMMON statement specifying the same common block name. It must appear after the last COMMON statement in the program unit.

If a THREADPRIVATE variable or a THREADPRIVATE common block is declared with the BIND attribute, the corresponding C entities must also be specified in a threadprivate directive in the C program.

The following are other restrictions for a THREADPRIVATE variable:

- The variable must not appear in any clause except the COPYIN, COPYPRIVATE, SCHEDULE, NUM_THREADS, THREAD_LIMIT, and IF clauses.
- It is affected by a COPYIN clause if the variable appears in the COPYIN clause or it is in a common block that appears in the COPYIN clause.
- It must not be an element of a common block or appear in an EQUIVALENCE statement.

- If it is part of another variable (as an array or structure element), it cannot appear in a THREADPRIVATE directive clause.

NOTE
On Windows* systems, if you specify option `/Qopenmp-threadprivate:compat`, the compiler does not generate threadsafe code for common blocks in an !$OMP THREADPRIVATE directive unless at least one element in the common block is explicitly initialized. For more information, see the article titled: */Qopenmp-threadprivate:compat doesn't work with uninitialized threadprivate common blocks*, which is located in http://intel.ly/1aHhsjc

Example

In the following example, the common blocks BLK1 and FIELDS are specified as thread private:

```
      COMMON /BLK/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK/,/FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/,ZFIELD)
```

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives

TIME Intrinsic Procedure

Intrinsic Subroutine (Generic): *Returns the current time as set within the system. TIME can be used as an intrinsic subroutine or as a portability routine. It is an intrinsic procedure unless you specify USE IFPORT. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

`CALL TIME (buf)`

buf	(Output) Is a variable, array, or array element of any data type, or a character substring. It must contain at least eight bytes of storage.

The date is returned as a 8-byte ASCII character string taking the form hh:mm:ss, where:

hh	is the 2-digit hour
mm	is the 2-digit minute
ss	is the 2-digit second

If *buf* is of numeric type and smaller than 8 bytes, data corruption can occur.

If *buf* is of character type, its associated length is passed to the subroutine. If *buf* is smaller than 8 bytes, the subroutine truncates the date to fit in the specified length. If an array of type character is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

Example

```
CHARACTER*1 HOUR(8)
...
CALL TIME (HOUR)
```

The length of the first array element in CHARACTER array HOUR is passed to the TIME subroutine. The subroutine then truncates the time to fit into the 1-character element, producing an incorrect result.

See Also
DATE_AND_TIME

TIME portability routine

TIME Portability Routine

Portability Function or Subroutine: *The function returns the system time, in seconds, since 00:00:00 Greenwich mean time, January 1, 1970. TIME can be used as a portability function or subroutine, or as an intrinsic procedure. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax
Function Syntax:

```
result = TIME( )
```

Subroutine Syntax:

```
CALL TIME (timestr)
```

timestr	(Output) Character*(*). Is the current time, based on a 24-hour clock, in the form hh:mm:ss, where hh, mm, and ss are two-digit representations of the current hour, minutes past the hour, and seconds past the minute, respectively.

Results

The result type is INTEGER(4). The result value is the number of seconds that have elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

The subroutine fills a parameter with the current time as a string in the format hh:mm:ss.

The value returned by this routine can be used as input to other portability date and time functions.

Example

```
USE IFPORT
INTEGER(4) int_time
character*8 char_time
int_time = TIME( )
call TIME(char_time)
print *, 'Integer: ', int_time, 'time: ', char_time
END
```

See Also
DATE_AND_TIME

TIME intrinsic procedure

TIMEF

Portability Function: *Returns the number of seconds since the first time it is called, or zero.*

Module

USE IFPORT

Syntax

```
result = TIMEF( )
```

Results

The result type is REAL(8). The result value is the number of seconds that have elapsed since the first time TIMEF was called.

The first time it is called, TIMEF returns 0.

Example

```
USE IFPORT
INTEGER i, j
REAL(8) elapsed_time
elapsed_time = TIMEF( )
DO i = 1, 100000
   j = j + 1
END DO
elapsed_time = TIMEF( )
PRINT *, elapsed_time
END
```

See Also

Date and Time Procedures

TINY

Inquiry Intrinsic Function (Generic): *Returns the smallest number in the model representing the same type and kind parameters as the argument.*

Syntax

```
result = TINY (x)
```

x	(Input) Must be of type real; it can be scalar or array valued.

Results

The result is a scalar with the same type and kind parameters as *x*. The result has the value b $^{e_{min}}$ - 1. Parameters *b* and e_{min} are defined in Model for Real Data.

Example

If X is of type REAL(4), TINY (X) has the value 2 $^{-126}$.

The following shows another example:

```
REAL(8) r, result
r = 487923.3D0
result = TINY(r)    ! returns 2.225073858507201E-308
```

See Also

HUGE
Data Representation Models

TRACEBACKQQ

Run-Time Subroutine: *Provides traceback information. Uses the Intel® Fortran run-time library traceback facility to generate a stack trace showing the program call stack as it appeared at the time of the call to TRACEBACKQQ( ).*

Module

USE IFCORE

Syntax

`CALL TRACEBACKQQ ([string] [,user_exit_code] [,status] [,eptr])`

string

(Input; optional) CHARACTER*(*). A message string to precede the traceback output. It is recommended that the string be no more than 80 characters (one line) since that length appears better on output. However, this limit is not a restriction and it is not enforced. The string is output exactly as specified; no formatting or interpretation is done.

If this argument is omitted, no header message string is produced.

user_exit_code

(Input; optional) INTEGER(4). An exit code. Two values are predefined:

- A value of -1 causes the run-time system to return execution to the caller after producing traceback.
- A value of zero (the default) causes the application to abort execution.

Any other specified value causes the application to abort execution and return the specified value to the operating system.

status

(Input; optional) INTEGER(4). A status value. If specified, the run-time system returns the status value to the caller indicating that the traceback process was successful. The default is not to return status.

Note that a returned status value is only an indication that the "attempt" to trace the call stack was completed successfully, not that it produced a useful result.

You can include the file `iosdef.for`in your program to obtain symbolic definitions for the possible return values. A return value of FOR$IOS_SUCCESS (0) indicates success.

eptr

(Input; optional) Integer pointer. It is required if calling from a user-specified exception filter. If omitted, the default in null.

To trace the stack after an exception has occurred, the runtime support needs access to the exception information supplied to the filter by the operating system.

The *eptr* argument is a pointer to a T_EXCEPTION_POINTERS structure, which is defined in `ifcore.f90`. This argument is optional and is usually omitted. On Windows* systems,

T_EXCEPTION_POINTERS is returned by the Windows* API GetExceptionInformation( ), which is usually passed to a C try/except filter function.

The TRACEBACKQQ routine provides a standard way for an application to initiate a stack trace. It can be used to report application detected errors, debugging, and so forth. It uses the stack trace support in the Intel Fortran run-time library, and produces the same output that the run-time library produces for unhandled errors and exceptions.

The error message string normally included by the run-time system is replaced with the user-supplied message text, or omitted if no string is specified. Traceback output is directed to the target destination appropriate for the application type, just as it is when traceback is initiated internally by the run-time system.

Example

In the most simple case, you can generate a stack trace by coding the call to TRACEBACKQQ with no arguments:

```
CALL TRACEBACKQQ()
```

This call causes the run-time library to generate a traceback report with no leading header message, from wherever the call site is, and terminate execution.

The following example generates a traceback report with no leading header message, from wherever the call site is, and aborts execution:

```
USE IFCORE
CALL TRACEBACKQQ( )
```

The following example generates a traceback report with the user-supplied string as the header, and aborts execution:

```
USE IFCORE
CALL TRACEBACKQQ("My application message string")
```

The following example generates a traceback report with the user-supplied string as the header, and aborts execution, returning a status code of 123 to the operating system:

```
USE IFCORE
CALL TRACEBACKQQ(STRING="Bad value for TEMP",USER_EXIT_CODE=123)
```

Consider the following:

```
...
USE IFCORE
INTEGER(4) RTN_STS
INCLUDE 'IOSDEF.FOR'
...
CALL TRACEBACKQQ(USER_EXIT_CODE=-1,STATUS=RTN_STS)
IF (RTN_STS .EQ. FOR$IOS_SUCCESS) THEN
  PRINT *,'TRACEBACK WAS SUCCESSFUL'
END IF
...
```

This example generates a traceback report with no header string, and returns to the caller to continue execution of the application. If the traceback process succeeds, a status will be returned in variable RTN_STS.

You can specify arguments that generate a stack trace with the user-supplied string as the header and instead of terminating execution, return control to the caller to continue execution of the application. For example:

```
CALL TRACEBACKQQ(STRING="Done with pass 1",USER_EXIT_CODE=-1)
```

By specifying a user exit code of -1, control returns to the calling program. Specifying a user exit code with a positive value requests that specified value be returned to the operating system. The default value is 0, which causes the application to abort execution.

Windows* Example:

The following example demonstrates the use of the EPTR argument when calling from a user-defined exception filter function. The premise of the example is a Fortran DLL containing entry points protected by a C `try/except` construct, such as:

```
__declspec(dllexport) void FPE_TEST_WRAPPER ()
{
__try {
     /*
     **  call the Fortran code...
     */
     FPE_TEST() ;
     }
__except ( CHECK_EXCEPTION_INFO ( GetExceptionInformation() ) )
     {
     /*
     **  noncontinuable exception handling here, if any.
     */
     }
```

The C function shown above is guarding against a floating-point divide-by-zero exception. This function calls the user-supplied FPE_TEST (not shown). The Fortran function CHECK_EXCEPTION_INFO shown below (called in the __except filter expression above) can use TRACEBACKQQ to get a stack trace as follows:

```
  INTEGER*4 FUNCTION CHECK_EXCEPTION_INFO ( ExceptionInfo )
  !DIR$ ATTRIBUTES DLLEXPORT::CHECK_EXCEPTION_INFO
  USE IFWINTY
  !DIR$ ATTRIBUTES REFERENCE :: ExceptionInfo
  TYPE(T_EXCEPTION_POINTERS) ExceptionInfo
  TYPE(T_EXCEPTION_RECORD) erecptr
  TYPE(T_CONTEXT) ctxptr
  POINTER(eptr,erecptr)
  POINTER(ctxp,ctxptr)
  INTEGER(4) EXIT_CODE,STS
  CHARACTER(LEN=70) MYSTR
! Init the arguments to TRACEBACKQQ appropriately for your needs...
  EXIT_CODE=-1
  eptr = ExceptionInfo.ExceptionRecord
  ctxp = ExceptionInfo.ContextRecord
  IF ( erecptr.ExceptionCode .EQ. STATUS_FLOAT_DIVIDE_BY_ZERO ) THEN
         PRINT *, 'Saw floating divide by zero.'
         PRINT '(1x,a,z8.8)', 'ContextRecord.FloatSave.StatusWord = ', &
                 ctxptr.FloatSave.StatusWord
         MYSTR = "FLTDIV EXCEPTION IN MY APPLICATION"
         CALL TRACEBACKQQ(MYSTR,EXIT_CODE,STS, %LOC(ExceptionInfo))
  END IF
  .
  .
```

```
.
CHECK_EXCEPTION_INFO = 1
END
```

To return a pointer to C run-time exception information pointers within a user-defined handler that was established with SIGNALQQ (or directly with the C signal function), you can call the GETEXCEPTIONPTRSQQ routine.

See Also

GETEXCEPTIONPTRSQQ

TRAILZ

Elemental Intrinsic Function (Specific): *Returns the number of trailing zero bits in an integer.*

Syntax

`result = TRAILZ (i)`

i	(Input) Must be of type integer or logical.

Results

The result type is default integer. The result value is the number of trailing zeros in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

Consider the following:

```
INTEGER*8 J, TWO
PARAMETER (TWO=2)
DO J= -1, 40
  TYPE *, TRAILZ(TWO**J)  ! Prints 64, then 0 up to
ENDDO                     !   40 (trailing zeros)
END
```

TRANSFER

Transformational Intrinsic Function (Generic): *Converts the bit pattern of the first argument according to the type and kind parameters of the second argument.*

Syntax

`result = TRANSFER (source,mold[,size])`

source	(Input) Must be a scalar or array (of any data type).
mold	(Input) Must be a scalar or array (of any data type). It provides the type characteristics (not a value) for the result.
size	(Input; optional) Must be scalar and of type integer. It provides the number of elements for the output result.

Results

The result has the same type and type parameters as *mold*.

If *mold* is a scalar and *size* is omitted, the result is a scalar.

If *mold* is an array and *size* is omitted, the result is a rank-one array. Its size is the smallest that is possible to hold all of *source*.

If *size* is present, the result is a rank-one array of size *size*.

When the size of *source* is greater than zero, *mold* must not be an array with elements of size zero.

If the internal representation of the result occupies `m` bits, and the internal representation of *source* occupies `n` bits, then if `m` > `n`, the right-most `src` bits of result contain the bit pattern contained in *source*, and the `m` minus `n` left-most bits of result are undefined. If `m` < `n`, then result contains the bit pattern of the right-most `m` bits of *source*, and the left-most `n` minus `m` bits of *source* are ignored. Otherwise, the result contains the bit pattern contained in *source*.

Example

TRANSFER (1082130432, 0.0) has the value 4.0 (on processors that represent the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000).

TRANSFER ((/2.2, 3.3, 4.4/), ((0.0, 0.0))) results in a scalar whose value is (2.2, 3.3).

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/)) results in a complex rank-one array of length 2. Its first element is (2.2,3.3) and its second element has a real part with the value 4.4 and an undefined imaginary part.

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/), 1) results in a complex rank-one array having one element with the value (2.2, 3.3).

The following shows another example:

```
COMPLEX CVECTOR(2), CX(1)
!  The next statement sets CVECTOR to
!  [ 1.1 + 2.2i, 3.3 + 0.0i ]
CVECTOR = TRANSFER((/1.1, 2.2, 3.3, 0.0/), &
                   (/(0.0, 0.0)/))
!  The next statement sets CX to [ 1.1 + 2.2i ]
CX = TRANSFER((/1.1, 2.2, 3.3/) , (/(0.0, 0.0)/), &
              SIZE= 1)
WRITE(*,*) CVECTOR
WRITE(*,*) CX
END
```

The following example shows an error because the *source* size is greater than zero but *mold* is an array whose elements have zero size:

```
CHARACTER(0),PARAMETER :: nothing1(100) = ''
PRINT *,SIZE(TRANSFER(111014,nothing1))         ! error
...
```

TRANSPOSE

Transformational Intrinsic Function (Generic):
Transposes an array of rank two.

Syntax

`result = TRANSPOSE (`*`matrix`*`)`

`matrix`	(Input) Must be a rank-two array. It may be of any data type.

Results

The result is a rank-two array with the same type and kind parameters as *matrix*. Its shape is (n, m), where (m, n) is the shape of *matrix*. For example, if the shape of *matrix* is (4,6), the shape of the result is (6,4).

Element (i, j) of the result has the value *matrix*(j, i), where *i* is in the range 1 to n, and *j* is in the range 1 to m.

Example

B is the array

```
 [ 2  3  4 ]
 [ 5  6  7 ]
 [ 8  9  1 ].
```

TRANSPOSE (B) has the value

```
 [ 2  5  8 ]
 [ 3  6  9 ]
 [ 4  7  1 ].
```

The following shows another example:

```
INTEGER array(2, 3), result(3, 2)
array = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
! array is  1  3  5
!           2  4  6
result = TRANSPOSE(array)
! result is 1  2
!           3  4
!           5  6
END
```

See Also

RESHAPE
PRODUCT

TRIM

Transformational Intrinsic Function (Generic):
Returns the argument with trailing blanks removed.

Syntax

`result = TRIM (`*`string`*`)`

string	(Input) Must be a scalar of type character.

Results

The result is of type character with the same kind parameter as *string*. Its length is the length of *string* minus the number of trailing blanks in *string*.

The value of the result is the same as *string*, except any trailing blanks are removed. If *string* contains only blank characters, the result has zero length.

Example

TRIM (' NAME ') has the value ' NAME'.

TRIM (' C D ') has the value ' C D'.

The following shows another example:

```
! next line prints 30
WRITE(*, *) LEN("I have blanks behind me       ")
! the next line prints 23
WRITE(*,*) LEN(TRIM("I have blanks behind me       "))
END
```

See Also
LEN_TRIM

TTYNAM
Portability Subroutine: *Specifies a terminal device name.*

Module
USE IFPORT

Syntax
CALL TTYNAM (*string*,*lunit*)

string	(Output) Character*(*). Name of the terminal device. If the Fortran logical unit is not connected to a terminal, it returns a string filled with blanks.
lunit	(Input) INTEGER(4). A Fortran logical unit number.

Type Declarations
Statement: *Explicitly specifies the properties of data objects or functions.*

Syntax
A type declaration has the general form:

type-spec[[, *att*] ... ::] *v*[/*c-list*/][, *v*[/*c-list*/]] ...

type-spec	Is one of the following: *intrin-type* TYPE (*intrin-type*) TYPE (derived-type-name) For information about parameterized derived types, see Parameterized Derived-Type Declarations. TYPE (*) This defines the entity as an assumed-type object. CLASS (derived-type-name) CLASS (*)
intrin-type	Is one of the following data type specifiers: BYTE

INTEGER[([KIND=]k)]

REAL[([KIND=]k)]

DOUBLE PRECISION

COMPLEX[([KIND=]k)]

DOUBLE COMPLEX

CHARACTER[([KIND=]k)]

LOGICAL[([KIND=]k)]

In the optional kind selector "([KIND=]k)", *k* is the kind parameter. It must be an acceptable kind parameter for that data type. If the kind selector is not present, entities declared are of default type.

When *type-spec* is *intrin-type*, kind parameters for intrinsic numeric and logical data types can also be specified using the *n format, where *n* is the length (in bytes) of the entity; for example, INTEGER*4. The *n format cannot be used when *type-spec* is TYPE (*intrin-type*).

See each data type for further information on that type.

att

Is one of the following attribute specifiers:

ALLOCATABLE	EXTERNAL	PROTECTED
ASYNCHRONOUS	INTENT	PUBLIC[1]
AUTOMATIC	INTRINSIC	SAVE
BIND	OPTIONAL	STATIC
CODIMENSION	PARAMETER	TARGET
CONTIGUOUS	POINTER	VALUE
DIMENSION	PRIVATE[1]	VOLATILE

[1]These are access specifiers.

You can also declare any attribute separately as a statement.

v

Is the name of a data object or function. It can optionally be followed by:

- An array specification, if the object is an array.

  In a function declaration, an array must be a deferred-shape array if it has the POINTER or ALLOCATABLE attribute; otherwise, it must be an explicit-shape array.
- A coarray specification, if the object is a coarray
- A character length, if the object is of type character.
- A constant expression preceded by = or by one of the following for pointer objects:
  - => NULL( )
  - => target (pointer initialization)
- A codimension

A function name must be the name of an intrinsic function, external function, function dummy procedure, or statement function.

c-list

Is a list of constants, as in a DATA statement. If *v* has the PARAMETER attribute, the *c-list* cannot be present.

The *c-list* cannot specify more than one value unless it initializes an array. When initializing an array, the *c-list* must contain a value for every element in the array.

Description

Type declaration statements must precede all executable statements.

In most cases, a type declaration statement overrides (or confirms) the implicit type of an entity. However, a variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The double colon separator (::) is required only if the declaration contains an attribute specifier or initialization; otherwise it is optional.

If *att* or a double colon (::) appears, *c-list* cannot be specified; for example:

```
INTEGER I /2/              ! Valid
INTEGER, SAVE :: I /2/     ! Invalid
INTEGER, SAVE :: I = 2     ! Valid
```

The same attribute must not appear more than once in a given type declaration statement, and an entity cannot be given the same attribute more than once in a scoping unit.

If CLASS is specified, the entity must be a dummy argument or have the ALLOCATABLE or POINTER attribute.

If the PARAMETER attribute is specified, the declaration must contain an constant expression. The PARAMETER attribute must not be specified for a dummy argument, a pointer, an allocatable entity, a function, or an object in a common block.

If => NULL( ) is specified for a pointer, its initial association status is disassociated.

If => target is specified for a pointer, the following rules apply:

- The pointer must be type compatible with the target.
- The pointer and target must have the same rank.
- All nondeferred type parameters of the pointer must have the same values as the corresponding type parameters of the target.
- If the pointer has the CONTIGUOUS attribute, the target must be contiguous.

A variable (or variable subobject) can only be initialized once in an executable program. If the variable is an array, it must have its shape specified in either the type declaration statement or a previous attribute specification statement in the same scoping unit.

The INTENT, VALUE, and OPTIONAL attributes can be specified only for dummy arguments.

The INTENT attribute must not be specified for a dummy procedure without the POINTER attribute.

If the VALUE attribute is specified, the length type parameter values must be omitted or specified by constant expressions. The VALUE attribute must not be specified for a dummy procedure.

An entity must not have both the EXTERNAL attribute and the INTRINSIC attribute. It can only have one of these attributes if it is a function.

The BIND attribute and the PROTECTED attribute can appear only in the specification part of a module.

A function result can be declared to have the POINTER or ALLOCATABLE attribute.

An automatic object cannot appear in a SAVE or DATA statement and it cannot be declared with a SAVE attribute nor be initially defined by an initialization.

The SAVE attribute must not be specified for:

- An object that is in a common block
- A procedure
- A dummy argument
- An automatic data object
- A function result
- An object with the PARAMETER attribute

The PROTECTED attribute is only allowed for a procedure pointer or named variable that is not in a common block. A pointer object that has the PROTECTED attribute and is accessed by use association must not appear as:

- A pointer object in a NULLIFY statement
- A data pointer object or procedure pointer object in a pointer assignment statement
- An allocate object in an ALLOCATE statement or DEALLOCATE statement
- An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

The following objects cannot be initialized in a type declaration:

- A dummy argument
- A function result
- An object in a named common block (unless the type declaration is in a block data program unit)
- An object in blank common
- An allocatable variable
- An external name
- An intrinsic name
- An automatic object
- An object that has the AUTOMATIC attribute

If a declaration contains a constant expression, but no PARAMETER attribute is specified, the object is a variable whose value is initially defined. The object becomes defined with the value determined from the constant expression according to the rules of intrinsic assignment.

When *type-spec* is *intrin-type*, the interpretation is exactly the same as it would have been without the keyword type and the parentheses. The following declarations for integers K and L and complexes A and B have identical results:

TYPE (INTEGER) :: K, L	INTEGER :: K, L
TYPE (COMPLEX (KIND (0.0D0))) :: A, B	COMPLEX (KIND (0.0D0)) :: A, B

A *derived-type-name* cannot be the same as the name of any standard intrinsic type. The type names BYTE and DOUBLECOMPLEX are not standard type names; they are Intel Fortran extensions. If BYTE or DOUBLECOMPLEX is declared to be a *derived-type-name*, it overrides the intrinsic name BYTE or DOUBLECOMPLEX. For example:

```
TYPE (DOUBLECOMPLEX) :: X     ! if DOUBLEXCOMPLEX is a defined-type
                              !    name, X is of that defined type
BYTE :: Y                     ! if BYTE is not a defined-type name,
                              !    Y is of intrinsic type BYTE, which
                              !    is the same as INTEGER(KIND=1)
```

The presence of initialization gives the object the SAVE attribute, except for objects in named common blocks or objects with the PARAMETER attribute.

When the entity is an assumed-type object, the following rules apply:

- The entity has no declared type and its dynamic type and type parameters are assumed from its effective argument. An assumed-type object is unlimited polymorphic.
- An assumed-type object must be a dummy variable that does not have the ALLOCATABLE, CODIMENSION, INTENT(OUT), POINTER, or VALUE attribute and is not an explicit-shape array.
- An assumed-type object that is not assumed-shape and not assumed-rank is intended to be passed as the C address of the object. A TYPE(*) explicit-shape array is not permitted because there is insufficient information passed for an assumed-type explicit-shape array that is an actual argument corresponding to an assumed-shape dummy argument to compute element offsets.
- An assumed-type variable name must not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-type, or as the first argument to any of the following intrinsic or intrinsic module functions: IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, UBOUND, and C_LOC.
- An assumed-type actual argument that corresponds to an assumed-rank dummy argument must be assumed-shape or assumed-rank.

An object can have more than one attribute. The following table lists the compatible attributes:

Compatible Attributes

Attribute	Compatible with:
ALLOCATABLE	AUTOMATIC, ASYNCHRONOUS, CODIMENSION, DIMENSION [1], PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
ASYNCHRONOUS	ALLOCATABLE, AUTOMATIC, BIND, DIMENSION, INTENT, OPTIONAL, POINTER, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VALUE, VOLATILE
AUTOMATIC	ALLOCATABLE, ASYNCHRONOUS, BIND, DIMENSION, POINTER, PROTECTED, TARGET, VOLATILE
BIND	ASYNCHRONOUS, AUTOMATIC, DIMENSION, EXTERNAL, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
CODIMENSION	ALLOCATABLE, DIMENSION, INTENT, OPTIONAL, PRIVATE, PROTECTED, PUBLIC, SAVE, TARGET
CONTIGUOUS	DIMENSION, INTENT, OPTIONAL, POINTER, PRIVATE, PROTECTED, PUBLIC, TARGET
DIMENSION	ALLOCATABLE, ASYNCHRONOUS, AUTOMATIC, BIND, CODIMENSION, CONTIGUOUS, INTENT, OPTIONAL, PARAMETER, POINTER, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
EXTERNAL	BIND, OPTIONAL, PRIVATE, PUBLIC
INTENT	ASYNCHRONOUS, CODIMENSION, CONTIGUOUS, DIMENSION, OPTIONAL, TARGET, VOLATILE
INTRINSIC	PRIVATE, PUBLIC
OPTIONAL	ASYNCHRONOUS, CODIMENSION, CONTIGUOUS, DIMENSION, EXTERNAL, INTENT, POINTER, TARGET, VALUE, VOLATILE

Attribute	Compatible with:
PARAMETER	DIMENSION, PRIVATE, PUBLIC
POINTER	ASYNCHRONOUS, AUTOMATIC, CONTIGUOUS, DIMENSION [1], OPTIONAL, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, VOLATILE
PRIVATE	ASYNCHRONOUS, ALLOCATABLE, BIND, CODIMENSION, CONTIGUOUS, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, PROTECTED, SAVE, STATIC, TARGET, VOLATILE
PROTECTED	ALLOCATABLE, ASYNCHRONOUS, BIND, CODIMENSION, CONTIGUOUS, DIMENSION, POINTER, PRIVATE, PUBLIC, SAVE, TARGET, VOLATILE
PUBLIC	ASYNCHRONOUS, ALLOCATABLE, BIND, CODIMENSION, CONTIGUOUS, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, PROTECTED, SAVE, STATIC, TARGET, VOLATILE
SAVE	ALLOCATABLE, ASYNCHRONOUS, BIND, CODIMENSION, DIMENSION, POINTER, PRIVATE, PROTECTED, PUBLIC, STATIC, TARGET, VOLATILE
STATIC	ALLOCATABLE, ASYNCHRONOUS, BIND, DIMENSION, POINTER, PRIVATE, PROTECTED, PUBLIC, SAVE, TARGET, VOLATILE
TARGET	ALLOCATABLE, ASYNCHRONOUS, AUTOMATIC, BIND, CODIMENSION, CONTIGUOUS, DIMENSION, INTENT, OPTIONAL, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC,VALUE, VOLATILE
VALUE	ASYNCHRONOUS, INTENT (IN only), OPTIONAL, TARGET
VOLATILE	ALLOCATABLE, ASYNCHRONOUS, AUTOMATIC, BIND, DIMENSION, INTENT, OPTIONAL, POINTER, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET

[1]With deferred shape

Example

The following show valid type declaration statements:

```
DOUBLE PRECISION B(6)
INTEGER(KIND=2) I
REAL(KIND=4) X, Y
REAL(4) X, Y
LOGICAL, DIMENSION(10,10) :: ARRAY_A, ARRAY_B
INTEGER, PARAMETER :: SMALLEST = SELECTED_REAL_KIND(6, 70)
REAL(KIND (0.0)) M
COMPLEX(KIND=8) :: D
TYPE(EMPLOYEE) :: MANAGER
```

```
REAL, INTRINSIC :: COS
CHARACTER(15) PROMPT
CHARACTER*12, SAVE :: HELLO_MSG
INTEGER COUNT, MATRIX(4,4), SUM
LOGICAL*2 SWITCH
REAL :: X = 2.0

TYPE (NUM), POINTER :: FIRST => NULL()
```

The following shows more examples:

```
REAL a (10)
LOGICAL, DIMENSION (5, 5) :: mask1, mask2
TYPE (COMPLEX) :: cube_root = (-0.5, 0.867)
INTEGER, PARAMETER :: short = SELECTED_INT_KIND (4)
REAL (KIND (0.0D0)) a1
TYPE (REAL (KIND = 4)) b
COMPLEX (KIND = KIND (0.0D0)) :: c
INTEGER (short) k ! Range at least -9999 to 9999
TYPE (member) :: george
```

The following shows an example of pointer initialization:

```
integer, target :: v (5) = [1, 2, 3, 4, 5]
type entry
  integer, pointer :: p (5) => v     ! pointer component default initialization
end type entry

type (entry), target :: bottom
type (entry), pointer :: top => bottom     ! pointer initialization
```

See Also

TYPE Declaration for Derived Types
CLASS Declaration
CHARACTER
COMPLEX
Default Initialization
DOUBLE COMPLEX
DOUBLE PRECISION
INTEGER
LOGICAL
REAL
IMPLICIT
RECORD
STRUCTURE

TYPE Statement (Derived Types)

Statement: *Declares a variable to be of derived type. It specifies the name of the user-defined type and the types of its components.*

Syntax

```
TYPE [[,attr-list] :: ] name [(type-param-name-list)]
   [type-param-def-stmts]
```

```
   [PRIVATE statement or SEQUENCE statement]. . .
   [component-definition]. . .
   [type-bound-procedure-part]
END TYPE [name]
```

attr-list

Is one of the following:

- *access-spec*

  Is the PUBLIC or PRIVATE attribute. The attribute can only be specified if the derived-type definition is in the specification part of a module.
- BIND(C)
- EXTENDS (*parent-type-name*)

  where *parent-type-name* is the name of a previously defined extensible type.
- ABSTRACT

The same *attr* must not appear more than once in any given derived-type statement.

If the type definition contains or inherits a deferred binding, ABSTRACT must appear.

The EXTENDS and ABSTRACT attributes apply to type extension and extended types. If ABSTRACT is specified, the type is an abstract type and it must be extensible.

A derived type that does not have the SEQUENCE or BIND(C) attribute is extensible. Conversely, a derived type that has the SEQUENCE or BIND(C) attribute is not extensible.

name

Is the name of the derived data type. It must not be the same as the name of any intrinsic type, or the same as the name of a derived type that can be accessed from a module.

type-param-name-list

Is a list of type parameter names separated by commas. For more information, see Parameterized Derived-Type Declarations.

type-param-def-stmts

Is one or more INTEGER declarations of the type parameters named in the *type-param-name-list*. For more information, see Parameterized Derived-Type Declarations.

component-definition

Is one or more type declaration statements or procedure pointer statements defining the component of derived type.

The first component definition can be preceded by an optional PRIVATE or SEQUENCE statement. (Only one PRIVATE and only one SEQUENCE statement can appear in a given derived-type definition.)

If SEQUENCE is present, all derived types specified in component definitions must be sequence types.

Procedure pointer component definitions are described in Procedure Pointers as Derived-Type Components.

The syntax for a type declaration component definition is described below.

type-bound-procedure-part — Is a CONTAINS statement, optionally followed by a PRIVATE statement, and one or more procedure binding statements (specific, generic, or final). For more information, see Type-Bound Procedures.

A type declaration component definition takes the following form:

type[[, *attr*] ::] *component*[(*a-spec*)] [[*coarray-spec*]] [**char-len*] [*init-ex*]

type — Is a type specifier. It can be an intrinsic type or a previously defined derived type.

If the POINTER or the ALLOCATABLE attribute follows this specifier, the type can also be the type being defined or any accessible derived type, whether previously defined or not.

attr — Is at most one of the following:

- An optional POINTER attribute for a pointer component
- An optional ALLOCATABLE attribute for a scalar component
- An optional DIMENSION or ALLOCATABLE attribute for an array component

  You cannot specify both the ALLOCATABLE and POINTER attribute.
- An optional access specifier, PUBLIC or PRIVATE
- An optional CODIMENSION [coarray-spec] to specify a coarray
- An optional CONTIGUOUS attribute if the object is contiguous

Each attribute can only appear once in a given component definition.

If neither the POINTER nor the ALLOCATABLE attribute is specified, *type* must specify an intrinsic type or a previously defined derived type.

If neither the POINTER nor the ALLOCATABLE attribute is specified, each *a-spec* must be an explicit-shape specification.

If the POINTER or ALLOCATABLE attribute is specified, each component array specification *a-spec* must be a deferred shape specification (d-spec).

If a *coarray-spec* appears, it must be a deferred specification list consisting of one or more colons (:), the component must have the ALLOCATABLE attribute, and the component must not be of type C_PTR or C_FUNPTR.

A data component whose type has a coarray ultimate component must be a nonpointer nonallocatable scalar and must not be a coarray.

If the CONTIGUOUS attribute is specified, the component must be an array with the POINTER attribute.

component — Is the name of the component being defined.

a-spec — Is an optional array specification, enclosed in parentheses. If POINTER or ALLOCATABLE is specified, the array is deferred shape; otherwise, it is explicit shape. In an explicit-shape specification, each bound must be a constant scalar integer expression.

If *component* is an array and the array bounds are not specified here, they must be specified following the DIMENSION attribute.

coarray-spec	Is a deferred-coshape specification. The left bracket and right bracket are required.
char-len	Is an optional scalar integer literal constant; it must be preceded by an asterisk (*). This parameter can only be specified if the component is of type CHARACTER.
init-ex	Is one of the following: • = constant expression for nonpointer components • => NULL( ) for pointer components If *init-ex* is specified, a double colon must appear in the component definition. The equals assignment symbol (=) can only be specified for nonpointer components. The constant expression is evaluated in the scoping unit of the type definition.

Description

If a name is specified following the END TYPE statement, it must be the same name that follows TYPE in the derived type statement.

A derived type can be defined only once in a scoping unit. If the same derived-type name appears in a derived-type definition in another scoping unit, it is treated independently.

A component name has the scope of the derived-type definition only. Therefore, the same name can be used in another derived-type definition in the same scoping unit.

The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement with no entity-list.

A PUBLIC or PRIVATE keyword can only appear on the TYPE statement of a derived-type definition, or on a component or type-bound procedure declaration of the type if the derived-type definition is in the specification part of a module.

The PRIVATE statement can only be specified in the *component-definition* or *type-bound-procedure-part* if the derived-type definition is in the specification part of a module. A PRIVATE statement inside a *component-definition* part specifies that the default accessibility of all components of the derived type are PRIVATE. Similarly, a PRIVATE statement inside a *type-bound-procedure-part* specifies that the default accessibility of the type bound procedures of the derived type are PRIVATE. If the default accessibility of components or type-bound procedures is PRIVATE, individual components or type-bound procedures can be declared PUBLIC, overriding the default accessibility attribute.

If a type definition is PRIVATE, the type name and the structure constructor for the type are accessible only within the module containing the definition and in the descendants of that module.

Two data entities have the same type if they are both declared to be of the same derived type; the derived-type definition can be accessed from a module or a host scoping unit.

If EXTENDS appears and the type being defined has a coarray ultimate component, its parent type must have a coarray ultimate component.

Data entities in different scoping units also have the same type if the following is true:

- They are declared with reference to different derived-type definitions that specify the same type name.
- All have the SEQUENCE property.
- All have the BIND attribute.
- They have no components with PRIVATE accessibility.
- They have type parameters and components that agree in order, name, and attributes.

Otherwise, they are of different derived types.

A data component is a coarray if the component declaration contains a coarray specification. If the component declaration contains a coarray specification, it specifies the corank.

If BIND (C) is specified, the following rules apply:

- The derived type cannot be a SEQUENCE type.
- The derived type must not have type parameters.
- The derived type must not have the EXTENDS attribute.
- The derived type must not have a *type-bound-procedure-part*.
- Each component of the derived type must be a nonpointer, nonallocatable data component with interoperable type and type parameters.

Example

```
!   DERIVED.F90
!   Define a derived-type structure,
!   type variables, and assign values

    TYPE member
      INTEGER age
      CHARACTER (LEN = 20) name
    END TYPE member

    TYPE (member) :: george
    TYPE (member) :: ernie

    george     = member( 33, 'George Brown' )
    ernie%age  = 56
    ernie%name = 'Ernie Brown'

    WRITE (*,*) george
    WRITE (*,*) ernie
    END
```

The following shows another example of a derived type:

```
TYPE mem_name
  SEQUENCE
  CHARACTER (LEN = 20) lastn
  CHARACTER (LEN = 20) firstn
  CHARACTER (len = 3) cos   ! this works because COS is a component name
END TYPE mem_name
TYPE member
    SEQUENCE
    TYPE (mem_name) :: name
    INTEGER age
    CHARACTER (LEN = 20) specialty
END TYPE member
```

In the following example, `a` and `b` are both variable arrays of derived type `pair`:

```
 USE, INTRINSIC :: ISO_C_BINDING
 TYPE, BIND(C) :: pair
   INTEGER(C_INT) :: i, j
 END TYPE
 TYPE (pair) :: a, b(3)
```

The following example shows how you can use derived-type objects as components of other derived-type objects:

```
TYPE employee_name
  CHARACTER(25) last_name
  CHARACTER(15) first_name
END TYPE
TYPE employee_addr
  CHARACTER(20) street_name
  INTEGER(2) street_number
  INTEGER(2) apt_number
  CHARACTER(20) city
  CHARACTER(2) state
  INTEGER(4) zip
END TYPE
```

Objects of these derived types can then be used within a third derived-type specification, such as:

```
TYPE employee_data
  TYPE (employee_name) :: name
  TYPE (employee_addr) :: addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END TYPE
```

The following program shows how you initialize components of a derived-type object:

```
TYPE list_node
   CHARACTER(20)                :: street_name = " "
   INTEGER(2)                   :: street_number = 0
   INTEGER(2)                   :: apt_number = -1
   CHARACTER(20)                :: city = " "
   CHARACTER(2)                 :: state = "NH"
   INTEGER(4)                   :: zip = 0
   TYPE (list_node), POINTER    :: next => NULL()
END TYPE

TYPE (list_node) :: x = list_node (zip = 03054)
PRINT *, x%state, x%zip
END
```

The above prints the following:

```
        NH 3054
```

Coarrays are a Fortran 2008 feature. The following example shows a derived-type definition with a coarray component:

```
TYPE NEW_TYPE
   REAL,ALLOCATABLE,CODIMENSION[:,:,:] :: NEW(:,:,:)
END TYPE NEW_TYPE
```

An object of type NEW_TYPE must be a scalar and it cannot be a pointer, allocatable, or a coarray.

The following example shows a derived-type definition containing a *coarray-spec*:

```
type t
   integer :: k
   real, allocatable :: x (:,:)[:,:]
end type t
```

In the following example, when the assignment to DEST occurs, DEST%TA is unallocated: it is first allocated with the size of SOURCE%TA and then the value of SOURCE%TA is assigned into DEST%TA:

```
TYPE T
  INTEGER I
END TYPE T
TYPE T2
  TYPE (T), ALLOCATABLE :: TA
END TYPE T2

TYPE (T2) SOURCE, DEST

ALLOCATE(SOURCE%TA)
DEST = SOURCE
```

The use of allocatable components of recursive type can involve direct recursion, in which the type has a component that is the same type as its parent. The following example creates a five-element linked list using allocatable components of recursive type:

```
TYPE T
INTEGER I
TYPE(T), ALLOCATABLE :: NEXT
END TYPE

TYPE (T), ALLOCATABLE, TARGET :: ORIG, COPY
TYPE (T), POINTER :: TA

ALLOCATE(ORIG)
TA => ORIG

DO I=1,5
  TA%I = I
  ALLOCATE(TA%NEXT)
  TA => TA%NEXT
END DO
```

By using allocatable components rather than pointer components, there is no need to explicitly nullify component NEXT - it is set to an unallocated state automatically by the compiler. Because the semantics of ALLOCATE(SOURCE=) require the allocation and copying of all allocated allocatable components from ORIG into COPY, we can create a complete, unique copy of the entire linked list in one easy statement:

```
ALLOCATE(COPY, SOURCE=ORIG)
```

In addition to direct recursion, indirect recursion or forward referencing can also be used. In the following example, component A of type T is of type T2, a forward reference to a type not yet defined, while component B of type T2 is of type T:

```
TYPE T
TYPE(T2), ALLOCATABLE :: A
END TYPE

TYPE T2
TYPE(T), ALLOCATABLE :: B
```

```
END TYPE

TYPE(T2) TY
```

See Also

Type Declarations
Procedure Pointers
Type-Bound Procedures
Passed-Object Dummy Arguments
Type Extension
Procedure Pointers as Derived-Type Components
DIMENSION
PRIVATE
PUBLIC
RECORD
SEQUENCE
STRUCTURE...END STRUCTURE
Derived Data Types
Default Initialization
Structure Components
Structure Constructors

UBOUND

Inquiry Intrinsic Function (Generic): *Returns the upper bounds for all dimensions of an array, or the upper bound for a specified dimension.*

Syntax

`result = UBOUND (array [, dim] [, kind])`

array	(Input) Must be an array; it can be assumed-rank. It may be of any data type. It must not be an allocatable array that is not allocated, or a disassociated pointer. It can be an assumed-size array if *dim* is present with a value less than the rank of *array*.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rule applies if *array* is not assumed-rank:

- If *dim* is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of *array*. Each element in the result corresponds to a dimension of *array*.

The following rules apply if *array* is assumed-rank:

- If *dim* is present and *array* is assumed-rank, the value of the result is as if *array* were a whole array, with the extent of the final dimension of *array* when *array* is associated with an assumed-size array being considered to be −1.
- If UBOUND is invoked for an assumed-rank object that is associated with a scalar and *dim* is absent, the result is a zero-sized array. UBOUND cannot be invoked for an assumed-rank object that is associated with a scalar if *dim* is present because the rank of a scalar is zero and dim must be >= 1.

If *array* is an array section or an array expression that is not a whole array or array structure component, UBOUND(*array*, *dim*) has a value equal to the number of elements in the given dimension.

If *array* is a whole array or array structure component, UBOUND(*array*, *dim*) has a value equal to the upper bound for subscript *dim* of *array* (if *dim* is nonzero). If *dim* has size zero, the corresponding element of the result has the value zero.

The setting of compiler options specifying integer size can affect this function.

Example

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)
REAL ARRAY_B (2:8, -3:20)
```

UBOUND (ARRAY_A) is (3, 8). UBOUND (ARRAY_A, DIM=2) is 8.

UBOUND (ARRAY_B) is (8, 20). UBOUND (ARRAY_B (5:8, :)) is (4,24) because the number of elements is significant for array section arguments.

The following shows another example:

```
REAL ar1(2:3, 4:5, -1:14), vec1(35)
INTEGER res1(3), res2, res3(1)
res1 = UBOUND (ar1)          ! returns [3, 5, 14]
res2 = UBOUND (ar1, DIM= 3)  ! returns 14
res3 = UBOUND (vec1)         ! returns [35]
END
```

See Also

LBOUND

UCOBOUND

Inquiry Intrinsic Function (Generic): *Returns the upper cobounds of a coarray.*

Syntax

`result = UCOBOUND (`*`coarray`* `[,`*`dim`* `[, `*`kind`*`])`

coarray	(Input) Must be a coarray; it can be of any type. It can be a scalar or an array. If it is allocatable, it must be allocated.
dim	(Input; optional) Must be an integer scalar with a value in the range 1 <= *dim* <= *n*, where *n* is the corank of *coarray*. The corresponding actual argument must not be an optional dummy argument.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter is that specified by kind; otherwise, the kind parameter is that of default integer type. The result is scalar if *dim* is present; otherwise, the result is an array of rank one and size *n*, where*n* is the corank of *coarray*.

The result depends on whether *dim* is specified:

- If *dim* is specified, UCOBOUND (COARRAY, DIM) has a value equal to the upper cobound for cosubscript *dim* of *coarray*.
- If *dim* is not specified, UCOBOUND (COARRAY) has a value whose ith element is equal to UCOBOUND (COARRAY, i), for i = 1, 2,. . . , *n*, where *n* is the corank of *coarray*.

Example

If A is allocated by the statement ALLOCATE (A [2:3, 7:*]), then LCOBOUND (A) is [2, 7] and LCOBOUND (A, DIM=2) is 7.

If NUM_IMAGES( ) has the value 30, and coarray B is allocated by the statement ALLOCATE (B[2:3, 0:7, *]), then UCOBOUND (B) is [3, 7, 2] and UCOBOUND (B, DIM=2) is 7. Note that the cosubscripts [3, 7, 2] do not correspond to an actual image.

UNDEFINE

Statement: *Removes a defined symbol.*

See Also

See DEFINE and UNDEFINE.

UNION and END UNION

Statements: *Define a data area that can be shared intermittently during program execution by one or more fields or groups of fields. A union declaration must be within a structure declaration.*

Syntax

Each unique field or group of fields is defined by a separate map declaration.

```
UNION
   map-declaration
   map-declaration
   [map-declaration]
   . . .
   [map-declaration]
END UNION
```

map-declaration Takes the following form:

```
MAP
   field-declaration
   [field-declaration]
   . . .
   [field-declaration]
END MAP
```

field-declaration	Is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a data field (having a data type) within a union. It can be of any intrinsic or derived type.

As with normal Fortran type declarations, data can be initialized in field declaration statements in union declarations. However, if fields within multiple map declarations in a single union are initialized, the data declarations are initialized in the order in which the statements appear. As a result, only the final initialization takes effect and all of the preceding initializations are overwritten.

The size of the shared area established for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the fields declared within it.

Manipulating data by using union declarations is similar to using EQUIVALENCE statements. The difference is that data entities specified within EQUIVALENCE statements are concurrently associated with a common storage location and the data residing there; with union declarations you can use one discrete storage location to alternately contain a variety of fields (arrays or variables).

With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration in the same union declaration is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

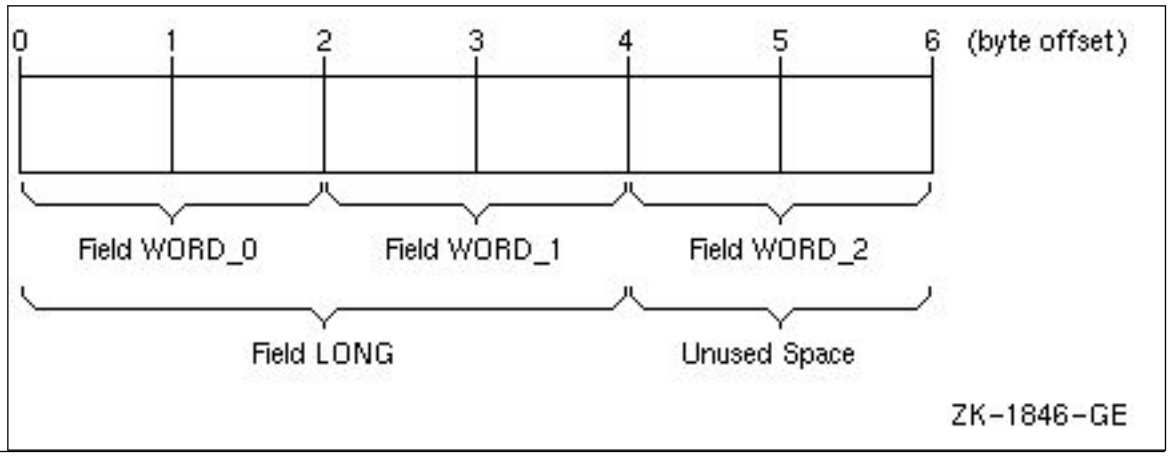
Example

In the following example, the structure WORDS_LONG is defined. This structure contains a union declaration defining two map fields. The first map field consists of three INTEGER*2 variables (WORD_0, WORD_1, and WORD_2), and the second, an INTEGER*4 variable, LONG:

```
STRUCTURE /WORDS_LONG/
  UNION
    MAP
      INTEGER*2  WORD_0, WORD_1, WORD_2
    END MAP
    MAP
      INTEGER*4  LONG
    END MAP
  END UNION
END STRUCTURE
```

The length of any record with the structure WORDS_LONG is 6 bytes. The following figure shows the memory mapping of any record with the structure WORDS_LONG:

Memory Map of Structure WORDS_LONG



In the following example, note how the first 40 characters in the string2 array are overlayed on 4-byte integers, while the remaining 20 are overlayed on 2-byte integers:

```
 UNION
    MAP
      CHARACTER*20 string1, CHARACTER*10 string2(6)
    END MAP
    MAP
      INTEGER*2 number(10), INTEGER*4 var(10), INTEGER*2
+      datum(10)
    END MAP
 END UNION
```

See Also

STRUCTURE and END STRUCTURE
Record Structures

UNLINK

Portability Function: *Deletes the file given by path.*

Module

USE IFPORT

Syntax

```
result = UNLINK (name)
```

name	(Input) Character*(*). Path of the file to delete. The path can use forward (/) or backward (\) slashes as path separators and can contain drive letters.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code. Errors can be one of the following:

- ENOENT: The specified file could not be found.

- EACCES: The specified file is read-only.

You must have adequate permission to delete the specified file.

On Windows systems, you will get the EACCES error if the file has been opened by any process.

Example

```
USE IFPORT
INTEGER(4) ISTATUS
CHARACTER*20 dirname
READ *, dirname
ISTATUS = UNLINK (dirname)
IF (ISTATUS) then
   print *, 'Error ', ISTATUS
END IF
END
```

See Also

SYSTEM
DELDIRQQ

UNPACK

Transformational Intrinsic Function (Generic):
Takes elements from a rank-one array and unpacks them into another (possibly larger) array under the control of a mask.

Syntax

`result = UNPACK (vector,mask,field)`

vector	(Input) Must be a rank-one array. It may be of any data type. Its size must be at least *t*, where *t* is the number of true elements in *mask*.
mask	(Input) Must be a logical array. It determines where elements of *vector* are placed when they are unpacked.
field	(Input) Must be of the same type and type parameters as *vector* and conformable with *mask*. Elements in *field* are inserted into the result array when the corresponding *mask* element has the value false.

Results

The result is an array with the same shape as *mask*, and the same type and type parameters as *vector*.

Elements in the result array are filled in array element order. If element i of *mask* is true, the corresponding element of the result is filled by the next element in *vector*. Otherwise, it is filled by *field* (if *field* is scalar) or the ith element of *field* (if *field* is an array).

Example

N is the array

```
 [ 0  0  1 ]
 [ 1  0  1 ]
 [ 1  0  0 ],
```

P is the array (2, 3, 4, 5), and Q is the array

```
 [ T  F  F ]
 [ F  T  F ]
 [ T  T  F ].
```

UNPACK (P, MASK=Q, FIELD=N) produces the result

```
 [ 2  0  1 ]
 [ 1  4  1 ]
 [ 3  5  0 ].
```

UNPACK (P, MASK=Q, FIELD=1) produces the result

```
 [ 2  1  1 ]
 [ 1  4  1 ]
 [ 3  5  1 ].
```

The following shows another example:

```
LOGICAL mask (2, 3)
INTEGER vector(3) /1, 2, 3/, AR1(2, 3)
mask = RESHAPE((/.TRUE.,.FALSE.,.FALSE.,.TRUE.,&
                .TRUE.,.FALSE./), (/2, 3/))
! vector = [1 2 3] and mask =  T F T
!                              F T F
AR1 = UNPACK(vector, mask, 8)  ! returns  1 8 3
                               !          8 2 8
END
```

See Also

PACK
RESHAPE
SHAPE

UNPACKTIMEQQ

Portability Subroutine: *Unpacks a packed time and date value into its component parts.*

Module

USE IFPORT

Syntax

CALL UNPACKTIMEQQ (*timedate*,*iyr*,*imon*,*iday*,*ihr*,*imin*,*isec*)

timedate	(Input) INTEGER(4). Packed time and date information.
iyr	(Output) INTEGER(2). Year (*xxxx*AD).
imon	(Output) INTEGER(2). Month (1 - 12).
iday	(Output) INTEGER(2). Day (1 - 31).
ihr	(Output) INTEGER(2). Hour (0 - 23).
imin	(Output) INTEGER(2). Minute (0 - 59).
isec	(Output) INTEGER(2). Second (0 - 59).

GETFILEINFOQQ returns time and date in a packed format. You can use UNPACKTIMEQQ to unpack these values.

The output values reflect the time zone set on the local computer and the daylight savings rules for that time zone.

Use PACKTIMEQQ to repack times for passing to SETFILETIMEQQ. Packed times can be compared using relational operators.

Example

```
USE IFPORT
CHARACTER(80)    file
TYPE (FILE$INFO) info
INTEGER(4) handle, result
INTEGER(2) iyr, imon, iday, ihr, imin, isec

file = 'd:\f90ps\bin\t???.*'
handle = FILE$FIRST
result = GETFILEINFOQQ(file, info, handle)
CALL UNPACKTIMEQQ(info%lastwrite, iyr, imon,&
                  iday, ihr, imin, isec)
WRITE(*,*) iyr, imon, iday
WRITE(*,*) ihr, imin, isec
END
```

See Also

PACKTIMEQQ
GETFILEINFOQQ

UNREGISTERMOUSEEVENT (W*S)

QuickWin Function: *Removes the callback routine registered for a specified window by an earlier call to REGISTERMOUSEEVENT.*

Module

USE IFQWIN

Syntax

`result = UNREGISTERMOUSEEVENT (`*`unit`*`,`*`mouseevents`*`)`

unit

(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. Unit number of the window whose callback routine on mouse events is to be unregistered.

mouseevents

(Input) INTEGER(4). One or more mouse events handled by the callback routine to be unregistered. Symbolic constants (defined in `IFQWIN.F90`) for the possible mouse events are:

- MOUSE$LBUTTONDOWN - Left mouse button down
- MOUSE$LBUTTONUP - Left mouse button up
- MOUSE$LBUTTONDBLCLK - Left mouse button double-click
- MOUSE$RBUTTONDOWN - Right mouse button down
- MOUSE$RBUTTONUP - Right mouse button up
- MOUSE$RBUTTONDBLCLK - Right mouse button double-click
- MOUSE$MOVE - Mouse moved

Results

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer that can be one of the following:

- MOUSE$BADUNIT - The unit specified is not open, or is not associated with a QuickWin window.
- MOUSE$BADEVENT - The event specified is not supported.

Once you call UNREGISTERMOUSEEVENT, QuickWin no longer calls the callback routine specified earlier for the window when mouse events occur. Calling UNREGISTERMOUSEEVENT when no callback routine is registered for the window has no effect.

See Also

REGISTERMOUSEEVENT
WAITONMOUSEEVENT

UNROLL and NOUNROLL

General Compiler Directive: *Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop. These directives can only be applied to iterative DO loops.*

Syntax

```
!DIR$ UNROLL [(n)] -or- !DIR$ UNROLL [=n]

!DIR$ NOUNROLL
```

n Is an integer constant. The range of *n* is 0 through 255.

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

To use these directives, compiler option O2 or O3 must be set.

Example

```
!DIR$ UNROLL
  do i =1, m
    b(i) = a(i) + 1
    d(i) = c(i) + 1
  enddo
```

The following shows another example:

```
!DIR$ UNROLL= 4
  do i =1, m
    b(i) = a(c(i)) + 1
  enddo
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements
O compiler option

UNROLL_AND_JAM and NOUNROLL_AND_JAM

General Compiler Directive: *Hints to the compiler to enable or disable loop unrolling and jamming. These directives can only be applied to iterative DO loops.*

Syntax

```
!DIR$ UNROLL_AND_JAM [(n)] -or- !DIR$ UNROLL_AND_JAM [=n]
!DIR$ NOUNROLL_AND_JAM
```

n Is an integer constant. The range of *n* is 0 through 255.

The UNROLL_AND_JAM directive partially unrolls one or more loops higher in the nest than the innermost loop and fuses (jams) the resulting loops back together. This transformation allows more reuses in the loop.

This directive is not effective on innermost loops. You must ensure that the immediately following loop is not the innermost loop after compiler-initiated interchanges are completed.

Specifying this directive is a hint to the compiler that the unroll and jam sequence is legal and profitable. The compiler will enable this transformation whenever possible.

The UNROLL_AND_JAM directive must precede the DO statement for each DO loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer chooses the number of times to unroll the loop. The compiler generates correct code by comparing *n* and the loop count.

This directive is supported only when compiler option `O3` is set. The UNROLL_AND_JAM directive overrides any setting of loop unrolling from the command line.

When unrolling a loop increases register pressure and code size, it may be necessary to prevent unrolling of a nested or imperfect nested loop. In such cases, use the NOUNROLL_AND_JAM directive, which hints to the compiler not to unroll a specified loop.

Example

```
integer a(10,10), b(10,10), c(10,10), d(10,10)
integer i, j, k

!dir$ unroll_and_jam = 6

do i=1,10
  !dir$ unroll_and_jam (6)
  do j=1,10
    do k=1,10
      a (j, i) = a (j, i) + b (k, i) *c (k, j)
    end do              ! k
  end do                ! j
end do                  ! i

end
```

See Also

General Compiler Directives
Syntax Rules for Compiler Directives
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements
O compiler option

UNTIED Clause

Parallel Directive Clause: *Specifies that the task is never tied to the thread that started its execution.*

Syntax

```
UNTIED
```

Any thread in the team can resume the task region after a suspension. For example, during runtime, the execution of a given task can start on thread A, break execution, and later resume on thread B.

USE

Statement: *Gives a program unit accessibility to public entities in a module.*

Syntax

```
USE [[, mod-nature] ::] name [, rename-list]
USE [[, mod-nature] ::] name, ONLY : [ only-list]
```

mod-nature

Is INTRINSIC or NON_INTRINSIC. If INTRINSIC is used, *name* must be the name of an intrinsic module. If NON_INTRINSIC is used, *name* must be the name of an nonintrinsic module. If *mod-nature* is not specified, *name* must be the name of an intrinsic or nonintrinsic module. If both are provided, the nonintrinsic module is used. It is an error to specify a user module and an intrinsic module of the same name in the same program unit (see Examples).

name

Is the name of the module.

rename-list

Is one or more items, separated by commas, having the following form:

local-name=> mod-name

local-name

Is the name of the entity in the program unit using the module or is "OPERATOR (*op-name*)", where *op-name* is the name of a defined operator in the program unit using the module.

mod-name

Is the name of a public entity in the module or is "OPERATOR (*op-name*)", where *op-name* is the name of a public entity in the module.

only-list

Is one or more items, separated by commas, where each item is the name of a public entity in the module or a generic identifier (a generic name, a defined operator specified as "OPERATOR (*op-name*)", or defined assignment).

An entity in the *only-list* can also take the form:

[*local-name* =>] *mod-name*

Description

If the USE statement is specified without the ONLY option, the program unit has access to all public entities in the named module.

If the USE statement is specified with the ONLY option, the program unit has access to only those entities following the option.

If more than one USE statement for a given module appears in a scoping unit, the following rules apply:

- If one USE statement does not have the ONLY option, all public entities in the module are accessible, and any *rename-list*s and *only-list*s are interpreted as a single, concatenated *rename-list*.
- If all the USE statements have ONLY options, all the *only-list*s are interpreted as a single, concatenated *only-list*. Only those entities named in one or more of the *only-list*s are accessible.

If two or more generic interfaces that are accessible in a scoping unit have the same name, the same operator, or are both assignments, they are interpreted as a single generic interface. Otherwise, multiple accessible entities can have the same name only if no reference to the name is made in the scoping unit.

The local names of entities made accessible by a USE statement must not be declared locally other than in a PUBLIC, PRIVATE, VOLATILE, or ASYNCHRONOUS statement. Within a module, if a use-associated entity is declared VOLATILE or ASYNCHRONOUS, it has the default accessibility of a locally declared identifier.

The local names of use-associated entities can appear in namelist group lists, but not in a COMMON or EQUIVALENCE statement.

If the name of every module from which a use-associated entity is accessed appears in an accessibility statement, the default accessibility of the entity is PRIVATE if every such accessibility statement is PRIVATE, and PUBLIC if any such accessibility statement is PUBLIC.

The accessibility of a use associated entity within a module can be determined by applying the following rules in the specified order:

1. It is PUBLIC if the entity is specified in the entity-list of a PUBLIC statement. It is PRIVATE if the entity is specified in the entity-list of a PRIVATE statement.
2. If the entity is declared locally in an ASYNCHRONOUS or VOLATILE statement, go to 5.
3. It is PUBLIC if any module through which the entity is accessible appears in a PUBLIC statement.
4. It is PRIVATE if every module thru which the entity is accessible appears in a PRIVATE statement.
5. It is PUBLIC if there is a PUBLIC statement with no entity-list specified in the module. It is PRIVATE if there is a PRIVATE statement with no entity-list specified in the module.
6. It is PUBLIC.

Examples

The following shows examples of the USE statement:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5), D(100)
END MODULE MOD_A
...
SUBROUTINE SUB_Y
  USE MOD_A, DX => D, EX => E   ! Array D has been renamed DX and array E
  ...                           ! has been renamed EX. Scalar variables B
END SUBROUTINE SUB_Y            ! and C are also available to this
...                             ! subroutine (using their module names).
SUBROUTINE SUB_Z
  USE MOD_A, ONLY: B, C         ! Only scalar variables B and C are
  ...                           !   available to this subroutine
END SUBROUTINE SUB_Z
...
```

You must not specify a user module and an intrinsic module of the same name in the same program unit. For example, if you specify a user module named ISO_FORTRAN_ENV, then it is illegal to specify the following in the same program unit:

```
USE :: ISO_FORTRAN_ENV
USE, INTRINSIC :: ISO_FORTRAN_ENV
```

The following example shows a module containing common blocks:

```
MODULE COLORS
  COMMON /BLOCKA/ C, D(15)
  COMMON /BLOCKB/ E, F
  ...
END MODULE COLORS
...
FUNCTION HUE(A, B)
  USE COLORS
  ...
END FUNCTION HUE
```

The USE statement makes all of the variables in the common blocks in module COLORS available to the function HUE.

To provide data abstraction, a user-defined data type and operations to be performed on values of this type can be packaged together in a module. The following example shows such a module:

```
MODULE CALCULATION
  TYPE ITEM
    REAL :: X, Y
  END TYPE ITEM

  INTERFACE OPERATOR (+)
    MODULE PROCEDURE ITEM_CALC
  END INTERFACE

CONTAINS
  FUNCTION ITEM_CALC (A1, A2)
    TYPE(ITEM) A1, A2, ITEM_CALC
    ...
  END FUNCTION ITEM_CALC
  ...
END MODULE CALCULATION

PROGRAM TOTALS
USE CALCULATION
TYPE(ITEM) X, Y, Z
  ...
  X = Y + Z
  ...
END
```

The USE statement allows program TOTALS access to both the type ITEM and the extended intrinsic operator + to perform calculations.

The following shows another example:

```
 ! Module containing original type declarations
 MODULE geometry
 type square
    real side
    integer border
 end type
```

```
type circle
   real radius
   integer border
end type
END MODULE

! Program renames module types for local use.
PROGRAM test
USE GEOMETRY,LSQUARE=>SQUARE,LCIRCLE=>CIRCLE
! Now use these types in declarations
type (LSQUARE) s1,s2
type (LCIRCLE) c1,c2,c3
```

The following shows a defined operator in a USE statement:

```
 USE mymod, OPERATOR(.localop.) => OPERATOR(.moduleop.)
```

Entities in modules can be accessed either through their given name, or through aliases declared in the USE statement of the main program unit. For example:

```
 USE MODULE_LIB, XTABS => CROSSTABS
```

This statement accesses the routine called `CROSSTABS` in `MODULE_LIB` by the name `XTABS`. This way, if two modules have routines called `CROSSTABS`, one program can use them both simultaneously by assigning a local name in its USE statement.

When a program or subprogram renames a module entity, the local name (`XTABS`, in the preceding example) is accessible throughout the scope of the program unit that names it.

The ONLY option also allows public variables to be renamed. Consider the following:

```
 USE MODULE_A, ONLY: VARIABLE_A => VAR_A
```

In this case, the host program accesses only `VAR_A` from module `A`, and refers to it by the name `VARIABLE_A`.

Consider the following example:

```
 MODULE FOO
    integer foos_integer
  PRIVATE
    integer foos_my_integer
 END MODULE FOO
```

PRIVATE, in this case, makes the PRIVATE attribute the default for the entire module `FOO`. To make `foos_integer` accessible to other program units, add the line:

```
 PUBLIC :: foos_integer
```

Alternatively, to make only `foos_my_integer` inaccessible outside the module, rewrite the module as follows:

```
 MODULE FOO
   integer foos_integer
   integer, private::foos_my_integer
 END MODULE FOO
```

Given the following module declarations:

```
 MODULE mod1
   INTEGER x1
 END MODULE
```

```
MODULE mod2
  USE mod1
  INTEGER x2
END MODULE
```

Then in module mod3 below, x1 has default accessibility of PRIVATE:

```
MODULE mod3
  USE mod2
  PUBLIC x2     ! x2 is PUBLIC
  PRIVATE       ! x1 is PRIVATE
END MODULE
```

In mod4 below, even though x1 is declared in mod1, and mod1 is declared PRIVATE in mod4, x1 is PUBLIC since it is declared VOLATILE and it assumes the default accessibility of PUBLIC of a locally declared identifier in mod4:

```
MODULE mod4
  USE mod1
  PRIVATE mod1
  VOLATILE x1    ! x1 is PUBLIC
END MODULE
```

In mod5 below, x1 and x2 are both PRIVATE since all modules from which they are accessed appear in PRVATE statements:

```
MODULE mod5
  USE mod1
  USE mod2
  PRIVATE mod1
  PRIVATE mod2   ! x1 and x2 are PRIVATE
END MODULE
```

However, in mod6 below, both x1 and x2 are PUBLIC since they are accessible thru the use of mod2 which is declared PUBLIC:

```
MODULE mod6
  USE mod1
  USE mod2
  PRIVATE mod1
  PUBLIC mod2    !  x1 and x2 are PUBLIC
END MODULE
```

See Also

Program Units and Procedures

%VAL

Built-in Function: *Changes the form of an actual argument. Passes the argument as an immediate value.*

Syntax

```
%VAL (a)
```

a	(Input) An expression, record name, procedure name, array, character array section, or array element.

Description

The argument is passed as follows:

- On IA-32 architecture, as a 32-bit immediate value. If the argument is integer (or logical) and shorter than 32 bits, it is sign-extended to a 32-bit value. For complex data types, %VAL passes two 32-bit arguments.
- On Intel® 64 architecture, as a 64-bit immediate value. If the argument is integer (or logical) and shorter than 64 bits, it is sign-extended to a 64-bit value. For complex data types, %VAL passes two 64-bit arguments.

You must specify %VAL in the actual argument list of a CALL statement or function reference. You cannot use it in any other context.

The following tables list the Intel® Fortran defaults for argument passing, and the allowed uses of %VAL:

Expressions

Actual Argument Data Type	Default	%VAL
Logical	REF	Yes[1]
Integer	REF	Yes[1]
REAL(4)	REF	Yes
REAL(8)	REF	No
REAL(16)	REF	No
COMPLEX(4)	REF	Yes
COMPLEX(8)	REF	Yes
COMPLEX(16)	REF	No
Character	See table note [2]	No
Hollerith	REF	No
Aggregate[3]	REF	No
Derived	REF	No

Array Name

Actual Argument Data Type	Default	%VAL
Numeric	REF	No
Character	See table note [2]	No
Aggregate[3]	REF	No
Derived	REF	No

Procedure Name

Actual Argument Data Type	Default	%VAL
Numeric	REF	No
Character	See table note [2]	No

[1] If a logical or integer value occupies less than 64 bits of storage on Intel® 64 architecture, or 32 bits of storage on IA-32 architecture, it is converted to the correct size by sign extension. Use the ZEXT intrinsic function if zero extension is desired.

[2] A character argument is passed by address and hidden length.

[3] In Intel® Fortran record structures

The %VAL and %REF functions override related !DIR$ ATTRIBUTE settings.

Example

```
CALL SUB(2, %VAL(2))
```

Constant 2 is passed by reference. The second constant 2 is passed by immediate value.

See Also

CALL
%REF
%LOC

VALUE

Statement and Attribute: *Specifies a type of argument association for a dummy argument.*

Syntax

The VALUE attribute can be specified in a type declaration statement or a VALUE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] VALUE [, *att-ls*] :: *arg* [, *arg*] ...

Statement:

VALUE [::] *arg* [, *arg*]...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
arg	Is the name of a dummy argument.

Description

The VALUE attribute can be used in INTERFACE body or in a procedure. It can only be specified for dummy arguments. It cannot be specified for a dummy procedure.

An entity with the VALUE attribute must be a dummy data object that is not an assumed-size array or a coarray, and does not have a coarray ultimate component.

When this attribute is specified for a present dummy argument, the dummy argument becomes associated with a temporary, definable data object whose initial value is that of the corresponding actual argument. The actual mechanism by which this happens is determined by the compiler. If the procedure also has the BIND(C) attribute, the dummy argument is interoperable with the corresponding formal parameter of the C language prototype, causing the argument to be passed or received by value if C would do so.

When the VALUE attribute is used in a type declaration statement, any length type parameter values must be omitted or they must be specified by initialization expressions.

If the VALUE attribute is specified, you cannot specify a PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, VOLATILE, or INTENT (INOUT or OUT) attribute in the same scoping unit.

You can use option assume nostd_value to tell the compiler to use non-standard semantics for VALUE so that the value of the actual argument is passed to the called procedure, not the address of the actual argument nor the address of a copy of the actual argument.

Example

The following example shows how the VALUE attribute can be applied in a type declaration statement.

```
j = 3
call sub (j)
write (*,*) j ! Writes 3
contains
subroutine sub (i)
integer, value :: I
i = 4
write (*,*) i ! Writes 4
end subroutine sub
end
```

See Also

Type Declarations

Compatible attributes

assume compiler option

VECREMAINDER Clause

General Directive Clause: *Allows the compiler to vectorize (or not to vectorize) any peel or remainder loops when the original loop is vectorized.*

Syntax

```
[NO]VECREMAINDER
```

If !DIR$ VECTOR ALWAYS is specified, the following occurs:

- If VECREMAINDER is specified, the compiler vectorizes peel and remainder loops when the original main loop is vectorized.
- If NOVECREMAINDER is specified, the compiler does not vectorize peel or remainder loops when the original main loop is vectorized.
- If neither the VECREMAINDER or NOVECREMAINDER clause is specified, the compiler overrides efficiency heuristics of the vectorizer and it determines whether the loop can be vectorized.

See Also

VECTOR and NOVECTOR

SIMD Loop Directive

VECTOR and NOVECTOR

General Compiler Directive: *Overrides default heuristics for vectorization of DO loops. It can also affect certain optimizations.*

Syntax

```
!DIR$ VECTOR [clause[[,] clause]...]
```

```
!DIR$ NOVECTOR
```

clause	Is an optional vectorization or optimizer clause. It can be one or more of the following:

- ALIGNED | UNALIGNED

  Specifies that all data is aligned or no data is aligned in a DO loop. These clauses override efficiency heuristics in the optimizer.

  The ALIGNED clause instructs the compiler to assume all array references are aligned. As a result, misalignment properties analyzed from the program context, if any, will be discarded. The UNALIGNED clause is a hint to the compiler to avoid alignment optimizations; the compiler can still use available alignment information. These clauses disable advanced alignment optimizations of the compiler, such as dynamic or static loop peeling to make references aligned.

  Be careful when using the ALIGNED clause. The compiler may choose to use aligned data movement instructions, instructions with memory operands that require alignment, and/or nontemporal stores. Such instructions will cause a runtime exception if some of the access patterns are actually unaligned.
- ALWAYS [ASSERT]

  Enables or disables vectorization of a DO loop.

  The ALWAYS clause overrides efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized. If the ASSERT keyword is added, the compiler will generate an error-level assertion message saying that the compiler efficiency heuristics indicate that the loop cannot be vectorized. You should use the IVDEP directive to ignore assumed dependences.
- DYNAMIC_ALIGN [(*var*)] | NODYNAMIC_ALIGN]

  Enables or disables dynamic alignment optimization. Dynamic alignment is an optimization the compiler attempts to perform by default. It involves peeling iterations from the vector loop into a scalar loop before the vector loop so that the vector loop aligns with a particular memory reference.

  If you specify (*var*) for DYNAMIC_ALIGN, you can indicate a scalar or array variable name on which to align. Specifying DYNAMIC_ALIGN, with or without (*var*) does not guarantee the optimization is performed; the compiler still uses heuristics to determine feasibility of the optimization. The NODYNAMIC_ALIGN clause disables the optimization for the loop.
- G2S or NOG2S

  Enables or disables the use of gather/scatter instructions in the lexically next loop.

  The G2S clause tells the optimizer to disable the generation of gather/scatter and to transform gather/scatter into unit-strided loads/stores plus a set of shuffles wherever possible. The NOG2S clause tells the optimizer to enable the generation of gather/scatter instructions and not to transform gather/scatter into unit-strided loads/stores.

  These clauses do not affect loops nested in the specified loop.
- MASK_READWRITE | NOMASK READWRITE

Enables or disables the generation of masked load and store operations within conditional statements.

The MASK_READWRITE clause directs the compiler to disable memory speculation, causing the generation of masked load and store operations within conditional statements. The NOMASK_READWRITE clause directs the compiler to enable memory speculation, causing the generation of unmasked loads and stores within conditional statements.

- `TEMPORAL | NONTEMPORAL [(var1 [, var2]...)]`

  var — Is an optional memory reference in the form of a variable name.

  Controls how the "stores" of register contents to memory are performed (streaming versus non-streaming).

  The TEMPORAL clause directs the compiler to use temporal (that is, non-streaming) stores. The NONTEMPORAL clause directs the compiler to use non-temporal (that is, streaming) stores.

  By default, the compiler automatically determines whether a streaming store should be used for each variable.

  Streaming stores may cause significant performance improvements over non-streaming stores for large numbers on certain processors. However, the misuse of streaming stores can significantly degrade performance.
- [NO]VECREMAINDER
- `VECTORLENGTH (n1 [, n2]…)`

  Tells the vectorizor which vector length/factor to use when generating the main vector loop. *n* is an integer power of 2; the value must be 2, 4, 8, 16, 32, or 64. If more than one value is specified, the vectorizor will choose one of the specified vector lengths based on a cost model decision.

The VECTOR and NOVECTOR directives control vectorization of the DO loop that directly follows the directive.

If the `MASK_READWRITE` clause is specified, the compiler generates masked loads and stores within all conditional branches in the loop. If the `NOMASK_READWRITE` clause is specified, the compiler generates unmasked loads and stores for increased performance.

Caution
The VECTOR directive should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.

Example

The compiler normally does not vectorize DO loops that have a large number of non-unit stride references (compared to the number of unit stride references).

In the following example, vectorization would be disabled by default, but the directive overrides this behavior:

```
!DIR$ VECTOR ALWAYS
  do i = 1, 100, 2
    ! two references with stride 2 follow
    a(i) = b(i)
  enddo
```

There may be cases where you want to explicitly avoid vectorization of a loop; for example, if vectorization would result in a performance regression rather than an improvement. In these cases, you can use the NOVECTOR directive to disable vectorization of the loop.

In the following example, vectorization would be performed by default, but the directive overrides this behavior:

```
!DIR$ NOVECTOR
  do i = 1, 100
    a(i) = b(i) + c(i)
  enddo
```

See Also

General Compiler Directives

Syntax Rules for Compiler Directives
IVDEP directive
Rules for General Directives that Affect DO Loops
Rules for Loop Directives that Affect Array Assignment Statements

VERIFY

Elemental Intrinsic Function (Generic): *Verifies that a set of characters contains all the characters in a string by identifying the first character in the string that is not in the set.*

Syntax

`result = VERIFY (`*`string`*`, `*`set`*` [, `*`back`*`] [, `*`kind`*`])`

string	(Input) Must be of type character.
set	(Input) Must be of type character with the same kind parameter as *string*.
back	(Input; optional) Must be of type logical.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* is omitted (or is present with the value false) and *string* has at least one character that is not in *set*, the value of the result is the position of the leftmost character of *string* that is not in *set*.

If *back* is present with the value true and *string* has at least one character that is not in *set*, the value of the result is the position of the rightmost character of *string* that is not in *set*.

If each character of *string* is in *set* or the length of *string* is zero, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Example

VERIFY ('CDDDC', 'C') has the value 2.

VERIFY ('CDDDC', 'C', BACK=.TRUE.) has the value 4.

VERIFY ('CDDDC', 'CD') has the value zero.

The following shows another example:

```
INTEGER(4) position

position = VERIFY ('banana', 'nbc')  ! returns 2
position = VERIFY ('banana', 'nbc', BACK=.TRUE.)
                                     ! returns 6
position = VERIFY ('banana', 'nbca') ! returns 0
```

See Also

SCAN

VIRTUAL

Statement: *Has the same form and effect as the DIMENSION statement. It is included for compatibility with PDP-11 FORTRAN.*

See Also

DIMENSION

VOLATILE

Statement and Attribute: *Specifies that the value of an object is entirely unpredictable, based on information local to the current program unit. It prevents objects from being optimized during compilation.*

Syntax

The VOLATILE attribute can be specified in a type declaration statement or a VOLATILE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] VOLATILE [, *att-ls*] :: *object*[, *object*] ...

Statement:

VOLATILE [::] *object*[, *object*] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
object	Is the name of an object, or the name of a common block enclosed in slashes.

A variable or COMMON block must be declared VOLATILE if it can be read or written in a way that is not visible to the compiler. For example:

- If an operating system feature is used to place a variable in shared memory (so that it can be accessed by other programs), the variable must be declared VOLATILE.

- If a variable is accessed or modified by a routine called by the operating system when an asynchronous event occurs, the variable must be declared VOLATILE.

If an array is declared VOLATILE, each element in the array becomes volatile. If a common block is declared VOLATILE, each variable in the common block becomes volatile.

If an object of derived type is declared VOLATILE, its components become volatile.

If a pointer is declared VOLATILE, the pointer itself becomes volatile.

A VOLATILE statement cannot specify the following:

- A procedure
- A namelist group

The VOLATILE attribute must not be specified for a coarray that is accessed by use or host association. A noncoarray object that has the VOLATILE attribute can be associated with an object that does not have the VOLATILE attribute, including by use or host association.

Example

The following example shows a type declaration statement specifying the VOLATILE attribute:

```
INTEGER, VOLATILE :: D, E
```

The following example shows a VOLATILE statement:

```
PROGRAM TEST
LOGICAL(KIND=1) IPI(4)
INTEGER(KIND=4) A, B, C, D, E, ILOOK
INTEGER(KIND=4) P1, P2, P3, P4
COMMON /BLK1/A, B, C
VOLATILE /BLK1/, D, E
EQUIVALENCE(ILOOK, IPI)
EQUIVALENCE(A, P1)
EQUIVALENCE(P1, P4)
```

The named common block, BLK1, and the variables D and E are volatile. Variables P1 and P4 become volatile because of the direct equivalence of P1 and the indirect equivalence of P4.

See Also

Type Declarations
Compatible attributes

WAIT

Statement: *Performs a wait operation for a specified pending asynchronous data transfer operation. It takes one of the following forms:*

Syntax

`WAIT([UNIT=]`*`io-unit`* `[, END=`*`label`*`] [, EOR=`*`label`*`] [, ERR=`*`label`*`] [, ID=`*`id-var`*`] [, IOMSG=`*`msg-var`*`] [, IOSTAT=`*`i-var`*`])`

`WAIT` *`io-unit`*

`io-unit`	(Input) Is an external unit specifier.
`label`	(Input) Is the label of the branch target statement that receives control if an error occurs.

id-var	(Input) Is a scalar integer variable that is the identifier of a pending data transfer operation for the specified unit. If it is specified, a wait operation is performed for that pending operation. If it is omitted, wait operations are performed for all pending data transfers for the specified unit.
msg-var	(Output) Is a scalar default character variable that is assigned an explanatory message if an I/O error occurs.
i-var	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

A wait operation completes the processing of a pending data transfer operation. Each wait operation completes only a single data transfer operation, although a single statement may perform multiple wait operations.

The WAIT statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT= keyword is optional if the unit specifier is the first item in the I/O control list.

The EOR= specifier only has effect if the pending data transfer operation is a nonadvancing read. The END= specifier only has effect if the pending data transfer operation is a READ.

Example

The following example shows how the WAIT statement can be applied.

```
program test
integer, asynchronous, dimension(100) :: array
open (unit=1,file='asynch.dat',asynchronous='YES',&
  form='unformatted')
write (1) (i,i=1,100)
rewind (1)
read (1,asynchronous='YES') array
wait(1)
write (*,*) array(1:10)
end
```

WAITONMOUSEEVENT (W*S)

QuickWin Function: *Waits for the specified mouse input from the user.*

Module

USE IFQWIN

Syntax

`result = WAITONMOUSEEVENT (`*`mouseevents`*`,`*`keystate`*`,`*`x`*`,`*`y`*`)`

mouseevents	(Input) INTEGER(4). One or more mouse events that must occur before the function returns. Symbolic constants for the possible mouse events are: • MOUSE$LBUTTONDOWN - Left mouse button down • MOUSE$LBUTTONUP - Left mouse button up • MOUSE$LBUTTONDBLCLK - Left mouse button double-click • MOUSE$RBUTTONDOWN - Right mouse button down • MOUSE$RBUTTONUP - Right mouse button up • MOUSE$RBUTTONDBLCLK - Right mouse button double-click

- MOUSE$MOVE - Mouse moved

keystate

(Output) INTEGER(4). Bitwise inclusive OR of the state of the mouse during the event. The value returned in *keystate* can be any or all of the following symbolic constants:

- MOUSE$KS_LBUTTON - Left mouse button down during event
- MOUSE$KS_RBUTTON - Right mouse button down during event
- MOUSE$KS_SHIFT - SHIFTkey held down during event
- MOUSE$KS_CONTROL - CTRLkey held down during event

x

(Output) INTEGER(4). X position of the mouse when the event occurred.

y

(Output) INTEGER(4). Y position of the mouse when the event occurred.

Results

The result type is INTEGER(4). The result is the symbolic constant associated with the mouse event that occurred if successful. If the function fails, it returns the constant MOUSE$BADEVENT, meaning the event specified is not supported.

WAITONMOUSEEVENT does not return until the specified mouse input is received from the user. While waiting for a mouse event to occur, the status bar changes to read "Mouse input pending in XXX", where XXX is the name of the window. When a mouse event occurs, the status bar returns to its previous value.

A mouse event must happen in the window that had focus when WAITONMOUSEEVENT was initially called. Mouse events in other windows will not end the wait. Mouse events in other windows cause callbacks to be called for the other windows, if callbacks were previously registered for those windows.

For every BUTTONDOWN or BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double click interval, set in the system's CONTROL PANEL/MOUSE.

Example

```
  USE IFQWIN
  INTEGER(4) mouseevent, keystate, x, y, result
  ...
  mouseevent = IOR(MOUSE$RBUTTONDOWN,MOUSE$LBUTTONDOWN)
  result = WAITONMOUSEEVENT (mouseevent, keystate, x , y)
!
! Wait until right or left mouse button is clicked, then check the keystate
! with the following:
!
  if (IAND(MOUSE$KS_SHIFT,keystate) == MOUSE$KS_SHIFT)       &
 & write (*,*) 'Shift key was down'
  if (IAND(MOUSE$KS_CONTROL,keystate) == MOUSE$KS_CONTROL)   &
 & write (*,*) 'Ctrl key was down'
```

WHERE

Statement and Construct: *Lets you use masked array assignment, which performs an array operation on selected elements. This kind of assignment applies a logical test to an array on an element-by-element basis.*

Syntax

Statement:

```
WHERE (mask-expr1) assign-stmt
```

Construct:

```
[name:]WHERE (mask-expr1)
    [where-body-stmt] ...
[ELSE WHERE(mask-expr2) [name]
    [where-body-stmt] ...]
[ELSE WHERE[name]
    [where-body-stmt] ...]
END WHERE [name]
```

mask-expr1, mask-expr2	Are logical array expressions (called mask expressions).
assign-stmt	Is an assignment statement of the form: array variable = array expression.
name	Is the name of the WHERE construct.
where-body-stmt	Is one of the following: • An *assign-stmt* The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental. • A WHERE statement or construct

Description

If a construct *name* is specified in a WHERE statement, the same name must appear in the corresponding END WHERE statement. The same construct name can optionally appear in any ELSE WHERE statement in the construct. (ELSE WHERE cannot specify a different name.)

In each assignment statement, the mask expression, the variable being assigned to, and the expression on the right side, must all be conformable. Also, the assignment statement cannot be a defined assignment.

Only the WHERE statement (or the first line of the WHERE construct) can be labeled as a branch target statement.

The following shows an example using a WHERE statement:

```
INTEGER A, B, C
DIMENSION A(5), B(5), C(5)
DATA A /0,1,1,1,0/
DATA B /10,11,12,13,14/
C = -1

WHERE(A .NE. 0) C = B / A
```

The resulting array C contains: -1,11,12,13, and -1.

The assignment statement is only executed for those elements where the mask is true. Think of the mask expression as being evaluated first into a logical array that has the value true for those elements where A is positive. This array of trues and falses is applied to the arrays A, B and C in the assignment statement. The right side is only evaluated for elements for which the mask is true; assignment on the left side is only performed for those elements for which the mask is true. The elements for which the mask is false do not get assigned a value.

In a WHERE construct, the mask expression is evaluated first and only once. Every assignment statement following the WHERE is executed as if it were a WHERE statement with " *mask-expr1*" and every assignment statement following the ELSE WHERE is executed as if it were a WHERE statement with ".NOT. *mask-expr1*". If ELSE WHERE specifies "*mask-expr2*", it is executed as "(.NOT. *mask-expr1*) .AND. *mask-expr2*" during the processing of the ELSE WHERE statement.

You should be careful if the statements have side effects, or modify each other or the mask expression.

The following is an example of the WHERE construct:

```
DIMENSION PRESSURE(1000), TEMP(1000), PRECIPITATION(1000)
WHERE(PRESSURE .GE. 1.0)
  PRESSURE = PRESSURE + 1.0
  TEMP = TEMP - 10.0
ELSEWHERE
  PRECIPITATION = .TRUE.
ENDWHERE
```

The mask is applied to the arguments of functions on the right side of the assignment if they are considered to be elemental functions. Only elemental intrinsics are considered elemental functions. Transformational intrinsics, inquiry intrinsics, and functions or operations defined in the subprogram are considered to be nonelemental functions.

Consider the following example using LOG, an elemental function:

```
WHERE(A .GT. 0)  B = LOG(A)
```

The mask is applied to A, and LOG is executed only for the positive values of A. The result of the LOG is assigned to those elements of B where the mask is true.

Consider the following example using SUM, a nonelemental function:

```
REAL A, B
DIMENSION A(10,10), B(10)
WHERE(B .GT. 0.0)  B = SUM(A, DIM=1)
```

Since SUM is nonelemental, it is evaluated fully for all of A. Then, the assignment only happens for those elements for which the mask evaluated to true.

Consider the following example:

```
REAL A, B, C
DIMENSION A(10,10), B(10), C(10)
WHERE(C .GT. 0.0)  B = SUM(LOG(A), DIM=1)/C
```

Because SUM is nonelemental, all of its arguments are evaluated fully regardless of whether they are elemental or not. In this example, LOG(A) is fully evaluated for all elements in A even though LOG is elemental. Notice that the mask is applied to the result of the SUM and to C to determine the right side. One way of thinking about this is that everything inside the argument list of a nonelemental function does not use the mask, everything outside does.

Example

```
REAL(4) a(20)
. . .
WHERE (a > 0.0)
  a = LOG (a)
     !LOG is invoked only for positive elements
END WHERE
```

See Also

FORALL
Arrays

WORKSHARE

OpenMP* Fortran Compiler Directive: *Divides the work of executing a block of statements or constructs into separate units. It also distributes the work of executing the units to threads of the team so each unit is only executed once.*

Syntax

```
!$OMP WORKSHARE
   block
!$OMP END WORKSHARE [NOWAIT]
```

block

Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.

The *block* is executed so that each statement is completed before the next statement is started and the evaluation of the right hand side of an assignment is completed before the effects of assigning to the left hand side occur.

The following are additional rules for *block*:

- It may contain statements which bind to lexically enclosed PARALLEL constructs. Statements in these PARALLEL constructs are not restricted.
- It may contain ATOMIC directives and CRITICAL constructs.
- It must only contain array assignment statements, scalar assignment statements, FORALL statements, FORALL constructs, WHERE statements, or WHERE constructs.
- It must not contain any user defined function calls unless the function is ELEMENTAL.

The binding thread set for a WORKSHARE construct is the current team. A workshare region binds to the innermost enclosing parallel region.

If you do not specify the NOWAIT keyword, synchronization is implied following the code.

Multithreaded code is not always generated for the statements inside the block of an OMP WORKSHARE construct. Some statements parallelize; others do not parallelize and instead execute sequentially inside an OMP SINGLE construct to preserve the correct semantics of WORKSHARE. Some specific details follow:

- Simple array assignments such as A = B + C parallelize.
- Simple array assignments with overlap such as A = A + B + C parallelize.
- Array assignments with user-defined function calls parallelize such as A = A + F (B). F must be ELEMENTAL.
- Array assignments with array slices on the right hand side of the assignment such as A = A + B(1:4) + C(1:4) parallelize. If the lower bound of the left hand side or the array slice lower bound or the array slice stride on the right hand side is not 1, then the statement does not parallelize.
- Assigning into array slices does not parallelize.
- Scalar assignments do not parallelize – there is no work that needs to be done in parallel.
- FORALL and WHERE constructs do not parallelize.

See Also

OpenMP Fortran Compiler Directives
Syntax Rules for Compiler Directives
PARALLEL WORKSHARE

Parallel Processing Model for information about Binding Sets

WRAPON (W*S)

Graphics Function: *Controls whether text output is wrapped.*

Module

USE IFQWIN

Syntax

`result = WRAPON (option)`

option	(Input) INTEGER(2). Wrap mode. One of the following symbolic constants: • $GWRAPOFF - Truncates lines at right edge of window border. • $GWRAPON - Wraps lines at window border, scrolling if necessary.

Results

The result type is INTEGER(2). The result is the previous value of *option*.

WRAPON controls whether text output with the OUTTEXT function wraps to a new line or is truncated when the text output reaches the edge of the defined text window.

WRAPON does not affect font routines such as OUTGTEXT.

Example

```
USE IFQWIN
INTEGER(2) row, status2
INTEGER(4) status4
TYPE ( rccoord ) curpos
TYPE ( windowconfig ) wc
LOGICAL status

status = GETWINDOWCONFIG( wc )
wc%numtextcols = 80
wc%numxpixels  = -1
wc%numypixels  = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%fontsize    = -1
wc%title = "This is a test"C
wc%bitsperpixel = -1
status = SETWINDOWCONFIG( wc )
status4= SETBKCOLORRGB(#FF0000 )
CALL CLEARSCREEN( $GCLEARSCREEN )

!  Display wrapped and unwrapped text in text windows.
CALL SETTEXTWINDOW( INT2(1),INT2(1),INT2(5),INT2(25))
CALL SETTEXTPOSITION(INT2(1),INT2(1), curpos )
status2 = WRAPON( $GWRAPON )
status4 = SETTEXTCOLORRGB(#00FF00)
DO i = 1, 5
   CALL OUTTEXT( 'Here text does wrap. ')
END DO
CALL SETTEXTWINDOW(INT2(7),INT2(10),INT2(11),INT2(40))
```

```
CALL SETTEXTPOSITION(INT2(1),INT2(1),curpos)
status2 = WRAPON( $GWRAPOFF )
status4 = SETTEXTCOLORRGB(#008080)
DO row = 1, 5
   CALL SETTEXTPOSITION(INT2(row), INT2(1), curpos )
   CALL OUTTEXT('Here text does not wrap. ')
   CALL OUTTEXT('Here text does not wrap.')
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

See Also

OUTTEXT
SCROLLTEXTWINDOW
SETTEXTPOSITION
SETTEXTWINDOW

WRITE Statement

Statement: *Transfers output data to external sequential, direct-access, or internal records.*

Syntax

Sequential

Formatted:

WRITE (*eunit*, *format* [, *advance*] [, *asynchronous*] [, *decimal*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, [*iomsg*])[*io-list*]

Formatted - List-Directed:

WRITE (*eunit*, * [, *asynchronous*] [, *decimal*] [, *delim*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*])[*io-list*]

Formatted - Namelist:

WRITE (*eunit*, *nml-group* [, *asynchronous*] [, *decimal*] [, *delim*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*])

Unformatted:

WRITE (*eunit* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *iomsg*])[*io-list*]

Direct-Access

Formatted:

WRITE (*eunit*, *format*, *rec* [, *asynchronous*] [, *decimal*] [, *delim*] [, *id*] [, *pos*] [, *round*] [, *sign*] [, *iostat*] [, *err*] [, *iomsg*])[*io-list*]

Unformatted:

WRITE (*eunit*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *iomsg*])[*io-list*]

Internal

WRITE (*iunit*, *format* [, *nml-group*] [, *iostat*] [, *err*] [, *iomsg*])[*io-list*]

Internal Namelist

WRITE (*iunit*, *nml-group* [, *iostat*] [, *err*] [, *iomsg*])[*io-list*]

eunit	Is an external unit specifier, optionally prefaced by UNIT=. UNIT= is required if *eunit* is not the first specifier in the list.
format	Is a format specifier. It is optionally prefaced by FMT= if *format* is the second specifier in the list and the first specifier indicates a logical or internal unit specifier *without* the optional keyword UNIT=.
advance	Is an advance specifier (ADVANCE=*c-expr*). If the value of *c-expr* is 'YES', the statement uses advancing input; if the value is 'NO', the statement uses nonadvancing input. The default value is 'YES'.
asynchronous	Is an asynchronous specifier (ASYNCHRONOUS=*i-expr*). If the value of *i-expr* is 'YES', the statement uses asynchronous input; if the value is 'NO', the statement uses synchronous input. The default value is 'NO'.
decimal	Is a decimal mode specifier (DECIMAL=*dmode*) that evaluates to 'COMMA' or 'POINT'. The default value is 'POINT'.
delim	Is a delimiter specifier (DELIM=*del*). If the value of *del* is 'APOSTROPHE', apostrophes delimit character constants. If the value is 'QUOTE', quotes delimit character constants. If the value of del is 'NONE', character constants have no delimiters.
id	Is an id specifier (ID=*id-var*). If ASYNCHRONOUS='YES' is specified and the operation completes successfully, the id specifier becomes defined with an implementation-dependent value that can be specified in a future WAIT or INQUIRE statement to identify the particular data transfer operation. If an error occurs, the id specifier variable becomes undefined.
pos	Is a pos specifier (POS=*p*) that indicates a file position in file storage units in a stream file (ACCESS='STREAM'). It can only be specified on a file opened for stream access. If omitted, the stream I/O occurs starting at the next file position after the current file position.
round	Is a rounding specifier (ROUND=*rmode*) that determines the I/O rounding mode for this WRITE statement. If omitted, the rounding mode is unchanged. Possible values are UP, DOWN, ZERO, NEAREST, COMPATIBLE or PROCESSOR_DEFINED.
sign	Is a plus sign specifier (SIGN=*sn*). This controls whether optional plus characters appear in formatted numeric output.
iostat	Is the name of a variable to contain the completion status of the I/O operation. Optionally prefaced by IOSTAT=.
err	Are branch specifiers if an error (ERR=*label*) condition occurs.
iomsg	Is an I/O message specifier (IOMSG=*msg-var*).
io-list	Is an I/O list: the names of the variables, arrays, array elements, or character substrings from which or to which data will be transferred. Optionally an implied-DO list.
form	Is the nonkeyword form of a format specifier (no FMT=).
*	Is the format specifier indicating list-directed formatting. (It can also be specified as FMT= *.)

nml-group	Is the namelist group specification for namelist I/O. Optionally prefaced by NML=. NML= is required if *nml-group* is not the second I/O specifier. For more information, see Namelist Specifier.
rec	Is the cell number of a record to be accessed directly. It must be prefaced by REC=.
iunit	Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if *iunit* is not the first specifier in the list. It must be a character variable. It must not be an array section with a vector subscript. If an item in *io-list* is an expression that calls a function, that function must not execute an I/O statement or the EOF intrinsic function on the same external unit as *eunit*.

If you specify DECIMAL=, ROUND=, or SIGN= you must also specify FMT= or NML=.

If you specify ID=, you must also specify ASYNCHRONOUS='YES'.

Example

```
! write to file
open(1,FILE='test.dat')
write (1, '(A20)') namedef
! write with FORMAT statement
WRITE (*, 10) (n, SQRT(FLOAT(n)), FLOAT(n)**(1.0/3.0), n = 1, 100)
10 FORMAT (I5, F8.4, F8.5)
```

The following shows another example:

```
WRITE(6,'("Expected ",F12.6)') 2.0
```

See Also

I/O Lists
I/O Control List
Forms for Sequential WRITE Statements
Forms for Direct-Access WRITE Statements
Forms and Rules for Internal WRITE Statements
READ
PRINT
OPEN
I/O Formatting

XOR

Elemental Intrinsic Function (Generic): *An alternative name for intrinsic function IEOR.*

See Also

See IEOR.

ZEXT

Elemental Intrinsic Function (Generic): *Extends an argument with zeros. This function is used primarily for bit-oriented operations. It cannot be passed as an actual argument.*

Syntax

```
result = ZEXT (x [,kind])
```

x	(Input) Must be of type logical or integer.
kind	(Input; optional) Must be a scalar integer constant expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value is *x* extended with zeros and treated as an unsigned value.

The storage requirements for integer constants are never less than two bytes. Integer constants within the range of constants that can be represented by a single byte still require two bytes of storage.

The setting of compiler options specifying integer size can affect this function.

Specific Name [1]	Argument Type	Result Type
IZEXT	LOGICAL(1)	INTEGER(2)
	LOGICAL(2)	INTEGER(2)
	INTEGER(1)	INTEGER(2)
	INTEGER(2)	INTEGER(2)
JZEXT	LOGICAL(1)	INTEGER(4)
	LOGICAL(2)	INTEGER(4)
	LOGICAL(4)	INTEGER(4)
	INTEGER(1)	INTEGER(4)
	INTEGER(2)	INTEGER(4)
	INTEGER(4)	INTEGER(4)
KZEXT	LOGICAL(1)	INTEGER(8)
	LOGICAL(2)	INTEGER(8)
	LOGICAL(4)	INTEGER(8)
	LOGICAL(8)	INTEGER(8)
	INTEGER(1)	INTEGER(8)
	INTEGER(2)	INTEGER(8)
	INTEGER(4)	INTEGER(8)

Specific Name [1]	Argument Type	Result Type
	INTEGER(8)	INTEGER(8)

[1]These specific functions cannot be passed as actual arguments.

Example

Consider the following example:

```
INTEGER(2) W_VAR  /'FFFF'X/
INTEGER(4) L_VAR
L_VAR = ZEXT( W_VAR )
```

This example stores an INTEGER(2) quantity in the low-order 16 bits of an INTEGER(4) quantity, with the resulting value of L_VAR being '0000FFFF'X. If the ZEXT function had not been used, the resulting value would have been 'FFFFFFFF'X, because W_VAR would have been converted to the left-hand operand's data type by sign extension.

Glossary

This section contains abbreviated definitions of some commonly used terms in this manual.

Glossary A

absolute pathname	A directory path specified in fixed relationship to the root directory. On Windows* systems, the first character is a backslash (\). On Linux* and macOS* systems, the first character is a slash (/).
abstract interface	A named set of procedure characteristics that can be referenced in PROCEDURE declarations.
abstract type	A derived type, declared with the ABSTRACT keyword, which can be extended to declare an object. An abstract type cannot be used directly to declare an object.
active image	An image that is neither a stopped image nor a failed image.
active screen buffer	The screen buffer that is currently displayed in a console's window.
active window	A top-level window of the application with which the user is working. The Windows system identifies the active window by highlighting its title bar and border.
actual argument	A value (a variable, expression, or procedure) passed from a calling program unit to a subprogram (function or subroutine). See also dummy argument.
adjustable array	An explicit-shape array that is a dummy argument to a subprogram. The term is from FORTRAN 77. See also explicit-shape array.
aggregate reference	A reference to a record structure field.
allocatable array	A named array that has the ALLOCATABLE attribute. The array's rank is specified at compile time, but its bounds are determined at run time. Once space has been allocated for this type of array, the array has a shape and can be defined (and redefined) or referenced. It is an error to allocate an allocatable array that is currently allocated.

allocation status	Indicates whether an allocatable array or pointer is allocated. An allocation status is one of: allocated, deallocated, or undefined. An undefined allocation status means an array can no longer be referenced, defined, allocated, or deallocated. See also *association status*.
alphanumeric	Pertaining to letters and digits.
alternate return	A subroutine argument that permits control to branch immediately to some position other than the statement following the call. The actual argument in an alternate return is the statement label to which control should be transferred. (An alternate return is an obsolescent feature in Standard Fortran.)
ANSI	The American National Standards Institute. An organization through which accredited organizations create and maintain voluntary industry standards.
argument	Can be either of the following: • An actual argument--A variable, expression, or procedure passed from a calling program unit to a subprogram. See also *actual argument*. • A dummy argument--A variable whose name appears in the parenthesized list followuing the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement. See also dummy argument.
argument association	The relationship (or "matching up") between an actual argument and dummy argument during the execution of a procedure reference.
argument keyword	The name of a dummy (formal) argument. The name is used in a subprogram definition. Argument keywords can be used when the subprogram is invoked to associate dummy arguments with actual arguments, so that the subprogram arguments can appear in any order. Argument keywords are supplied for many of the intrinsic procedures.
array	A set of scalar data that all have the same type and kind type parameters. An array can be referenced by element (using a subscript), by section (using a section subscript list), or as a whole. An array has a rank (up to 31), bounds, size, and a shape. An individual array element is a scalar object. An array section, which is itself an array, is a subset of the entire array. Contrast with scalar. See also bounds, conformable, shape, size, whole array, and zero-sized array.
array constructor	A mechanism used to specify a sequence of scalar values that produce a rank-one array. To construct an array of rank greater than one, you must apply the RESHAPE intrinsic function to the array constructor.
array element	A scalar (individual) item in an array. An array element is identified by the array name followed by one or more subscripts in parentheses, indicating the element's position in the array. For example, B(3) or A(2,5).
array pointer	A pointer to an array. See also *array* and pointer.

array section	A subobject (or portion) of an array. It consists of the set of array elements or substrings of this set. The set (or section subscript list) is specified by subscripts, subscript triplets, or vector subscripts. If the set does not contain at least one subscript triplet or vector subscript, the reference indicates an array element, not an array.
array specification	A program statement specifying an array name and the number of dimensions the array contains (its rank). An array specification can appear in a DIMENSION or COMMON statement, or in a type declaration statement.
ASCII	The American Standard Code for Information Interchange. A 7-bit character encoding scheme associating an integer from 0 through 127 with 128 characters.
assignment statement	Usually, a statement that assigns (stores) the value of an expression on the right of an equal sign to the storage location of the variable to the left of the equal sign. In the case of Fortran pointers, the storage location is assigned, not the pointer itself.
associate name	The name of a construct entity associated with a selector of an ASSOCIATE or SELECT TYPE construct.
association	The relationship that allows an entity to be referenced by different names in one scoping unit or by the same or different names in more than one scoping unit. The principal kinds of association are argument, host, pointer, storage, and use association. See also *argument association*, host association, pointer association, storage association, and use association.
association status	Indicates whether or not a pointer is associated with a target. An association status is one of: undefined, associated, or disassociated. An undefined association status means a pointer can no longer be referenced, defined, or deallocated. An undefined pointer can, however, be allocated, nullified, or pointer assigned to a new target. See also *allocation status*.
assumed-length character argument	A dummy argument that assumes the length attribute of the corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.
assumed-shape array	A dummy argument array that assumes the shape of its associated actual argument array. The rank of the array is the number of colons (:) specified in parentheses.
assumed-size array	A dummy array whose size (only) is assumed from its associated actual argument. The upper bound of its last dimension is specified by an asterisk (*). All other extents (if any) must be specified.
atomic subroutine	An intrinsic subroutine that performs an action on a variable (its ATOM argument) indivisibly.
attribute	A property of a data object that can be specified in a type declaration statement. These properties determine how the data object can be used in a program. Most attributes can be alternatively specified in statements. For example, the DIMENSION statement has the same meaning as the DIMENSION attribute appearing in a type declaration statement.

automatic array	An explicit-shape array that is a local variable in a subprogram. It is not a dummy argument, and has bounds that are nonconstant specification expressions. The bounds (and shape) are determined at entry to the procedure by evaluating the bounds expressions. *See also automatic object*.
automatic object	A local data object that is created upon entry to a subprogram and disappears when the execution of the subprogram is completed. There are two kinds of automatic objects: arrays (of any data type) and objects of type CHARACTER. Automatic objects cannot be saved or initialized. An automatic object is not a dummy argument, but is declared with a specification expression that is not a constant expression. The specification expression can be the bounds of the array or the length of the character object.

Glossary B

background process	On Linux* systems, a process for which the command interpreter is not waiting. Its process group differs from that of its controlling terminal, so it is blocked from most terminal access. Contrast with foreground process.
background window	Any window created by a thread other than the foreground thread.
big endian	A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the highest addressed byte. Contrast with little endian.
binary constant	A constant that is a string of binary (base 2) digits (0 or 1) enclosed by apostrophes or quotation marks and preceded by the letter B.
binary operator	An operator that acts on a pair of operands. The exponentiation, multiplication, division, and concatenation operators are binary operators.
bit constant	A constant that is a binary, octal, or hexadecimal number.
bit field	A contiguous group of bits within a binary pattern; they are specified by a starting bit position and length. Some intrinsic functions (for example, IBSET and BTEST) and the intrinsic subroutine MVBITS operate on bit fields.
bitmap	An array of bits that contains data that describes the colors found in a rectangular region on the screen (or the rectangular region found on a page of printer paper).
blank common	A common block (one or more contiguous areas of storage) without a name. Common blocks are defined by a COMMON statement.
block	In general, a group of related items treated as a physical unit. For example, a block can be a group of constructs or statements that perform a task; the task can be executed once, repeatedly, or not at all.
block data program unit	A program unit, containing a BLOCK DATA statement and its associated specification statements, that establishes common blocks and assigns initial values to the variables in named common blocks. In FORTRAN 77, this was called a block data subprogram.

bounds	The range of subscript values for elements of an array. The lower bound is the smallest subscript value in a dimension, and the upper bound is the largest subscript value in that dimension. Array bounds can be positive, zero, or negative. These bounds are specified in an array specification. See also array specification.
breakpoint	A critical point in a program, at which execution is stopped so that you can see if the program variables contain the correct values. Breakpoints are often used to debug programs.
brush	A bitmap that is used to fill the interior of closed shapes, polygons, ellipses, and paths.
brush origin	A coordinate that specifies the location of one of the pixels in a brush's bitmap. The Windows system maps this pixel to the upper left corner of the window that contains the object to be painted. See also *bitmap*.
built-in procedure	See intrinsic procedure.
byte	A group of 8 contiguous bits (binary digits) starting on an addressable boundary.
byte-order mark	A special Unicode character (0xFEFF) that is placed at the beginning of Unicode text files to indicate that the text is in Unicode format.

Glossary C

carriage-control character	A character in the first position of a printed record that determines the vertical spacing of the output line.
character constant	A constant that is a string of printable ASCII characters enclosed by apostrophes (') or quotation marks (").
character expression	A character constant, variable, function value, or another constant expression, separated by a concatenation operator (//); for example, DAY// ' FIRST'.
character storage unit	The unit of storage for holding a scalar value of default character type (and character length one) that is not a pointer. One character storage unit corresponds to one byte of memory.
character string	A sequence of contiguous characters; a character data value. See also *character constant*.
character substring	One or more contiguous characters in a character string.
child process	A process initiated by another process (the parent). The child process can operate independently from the parent process. Also, the parent process can suspend or terminate without affecting the child process. See also parent process.
child window	A window that has the WS_CHILD style. A child window always appears within the client area of its parent window. See also parent window.
column-major order	See order of subscript progression.

coarray	A data entity that has nonzero corank. A Fortran program containing coarrays is interpreted as if it were replicated a fixed number of times and all copies were executed asynchronously. Its corank, cobounds, and coextents are given by the data in square brackets in its declaration or allocation.
cobound	A bound (limit) of a codimension. The values of each lower cobound and upper cobound determine the cobounds of the coarray along a particular codimension.
codimension	A dimension of the pattern formed by a set of corresponding coarrays.
comment	Text that documents or explains a program. In free source form, a comment begins with an exclamation point (!), unless it appears in a Hollerith or character constant. In fixed and tab source form, a comment begins with a letter C or an asterisk (*) in column 1. A comment can also begin with an exclamation point anywhere in a source line (except in a Hollerith or character constant) or in column 6 of a fixed-format line. The comment extends from the exclamation point to the end of the line. The compiler does not process comments, but shows them in program listings. See also compiler directive.
common block	A physical storage area shared by one or more program units. This storage area is defined by a COMMON statement. If the common block is given a name, it is a named common block; if it is not given a name, it is a blank common. See also blank common and named common block.
compilation unit	The source or files that are compiled together to form a single object file, possibly using interprocedural optimization across source files.
compiler directive	A structured comment that tells the compiler to perform certain tasks when it compiles a source program unit. Compiler directives are usually compiler-specific. (Some Fortran compilers call these directives "metacommands".)
compiler option	An option (or flag) that can be used on the compiler command line to override the default behavior of the Intel® Fortran compiler.
complex constant	A constant that is a pair of real or integer constants representing a complex number; the pair is separated by a comma and enclosed in parentheses. The first constant represents the real part of the number; the second constant represents the imaginary part. The following types of complex constants are available on all systems: COMPLEX(KIND=4), COMPLEX(KIND=8), and COMPLEX(KIND=16).
complex type	A data type that represents the values of complex numbers. The value is expressed as a complex constant. See also data type.
component	Part of a derived-type definition. There must be at least one component (intrinsic or derived type) in every derived-type definition.
component order	The ordering of the components of a derived type that is used for intrinsic formatted input/output and for structure constructors.
concatenate	The combination of two items into one by placing one of the items after the other. In Standard Fortran, the concatenation operator (//) is used to combine character items. See also *character expression*.
conformable	Pertains to dimensionality. Two arrays are conformable if they have the same shape. A scalar is conformable with any array.

conformance	See shape conformance.
conservative automatic inlining	The inline expansion of small procedures, with conservative heuristics to limit extra code.
console	An interface that provides input and output to character-mode applications.
constant	A data object whose value does not change during the execution of a program; the value is defined at the time of compilation. A constant can be named (using the PARAMETER attribute or statement) or unnamed. An unnamed constant is called a literal constant. The value of a constant can be numeric or logical, or it can be a character string. Contrast with variable.
constant expression	An expression that you can use as a kind type parameter, a named constant, or to specify an initial value for an entity. It is evaluated when a program is compiled.
construct	A series of statements starting with a statement denoting the kind of construct, such as DO, SELECT CASE, IF, FORALL, or WHERE, and ending with the appropriate termination statement.
construct association	The association between a selector and an associate name in an ASSOCIATE or SELECT TYPE construct.
contiguous	Pertaining to entities that are adjacent (next to one another) without intervening blanks (spaces); for example, contiguous characters or contiguous areas of storage.
control edit descriptor	A format descriptor that directly displays text or affects the conversions performed by subsequent data edit descriptors. Except for the slash descriptor, control edit descriptors are nonrepeatable.
control statement	A statement that alters the normal order of execution by transferring control to another part of a program unit or a subprogram. A control statement can be conditional (such as the IF construct or computed GO TO statement) or unconditional (such as the STOP or GO TO statement).
corank	The number of codimensions of a coarray.
critical section	An object used to synchronize the threads of a single process. Only one thread at a time can own a critical-section object.
current team	The initial ordered set of all images.

Glossary D

data abstraction	A style of programming in which you define types to represent objects in your program, define a set of operations for objects of each type, and restrict the operations to only this set, making the types abstract. The Standard Fortran modules, derived types, and defined operators, support this programming paradigm.
data edit descriptor	A repeatable format descriptor that causes the transfer or conversion of data to or from its internal representation. In FORTRAN 77, this term was called a field descriptor.
data entity	A data object that has a data type. It is the result of the evaluation of an expression, or the result of the execution of a function reference (the function result).

data item	A unit of data (or value) to be processed. Includes constants, variables, arrays, character substrings, or records.
data object	A constant, variable, or subobject (part) of a constant or variable. Its type may be specified implicitly or explicitly.
data type	The properties and internal representation that characterize data and functions. Each intrinsic and user-defined data type has a name, a set of operators, a set of values, and a way to show these values in a program. The basic intrinsic data types are integer, real, complex, logical, and character. The data value of an intrinsic data type depends on the value of the type parameter. See also type parameter.
data type declaration	See type declaration statement.
data type length specifier	The form *n appended to Intel® Fortran-specific data type names. For example, in REAL*4, the *4 is the data type length specifier.
deadlock	A bug where the execution of thread A is blocked indefinitely waiting for thread B to perform some action, while thread B is blocked waiting for thread A. For example, two threads on opposite ends of a named pipe can become deadlocked if each thread waits to read data written by the other thread. A single thread can also deadlock itself. See also thread.
declaration	See specification statement.
declared type	The type that a data entity is declared to have. For polymorphic data entities, it may differ from the type during execution (the dynamic type) .
decorated name	An internal representation of a procedure name or variable name that contains information about where it is declared; for procedures, the information includes how it is called. Decorated names are mainly of interest in mixed-language programming, when calling Fortran routines from other languages.
default character	The kind for character constants if no kind type parameter is specified. Currently, the only kind type parameter for character constants is CHARACTER(1), the default character kind.
default complex	The kind for complex constants if no kind type parameter is specified. The default complex kind is affected by compiler options specifying double size. If no compiler option is specified, default complex is COMPLEX(4) (COMPLEX*8). See also *default real*.
default integer	The kind for integer constants if no kind type parameter is specified. The default integer kind is affected by the INTEGER directive, the OPTIONS statement, and by compiler options specifying integer size. If none of these are specified, default integer is INTEGER(4) (INTEGER*4). If a command line option affecting integer size has been specified, the integer has the kind specified, unless it is outside the range of the kind specified by the option. In this case, the kind type of the integer is the smallest integer kind which can hold the integer.
default logical	The kind for logical constants if no kind type parameter is specified. The default logical kind is affected by the INTEGER directive, the OPTIONS statement, and by compiler options specifying integer size. If none of these are specified, default logical is LOGICAL(4) (LOGICAL*4). See also *default integer*.

default real	The kind for real constants if no kind type parameter is specified. The default real kind is affected by compiler options specifying real size and by the REAL directive. If neither of these is specified, default real is REAL(4) (REAL*4). If a real constant is encountered that is outside the range for the default, an error occurs.
deferred-shape array	An array pointer (an array with the POINTER attribute) or an allocatable array (an array with the ALLOCATABLE attribute). The size in each dimension is determined by pointer assignment or when the array is allocated. The array specification contains a colon (:) for each dimension of the array. No bounds are specified.
definable	A property of variables. A variable is definable if its value can be changed by the appearance of its name or designator on the left of an assignment statement. An example of a variable that is not definable is an allocatable array that has not been allocated.
defined	For a data object, the property of having or being given a valid value.
defined assignment	An assignment statement that is not intrinsic, but is defined by a subroutine and an ASSIGNMENT(=) interface block. See also *derived type* and interface block.
defined operation	An operation that is not intrinsic, but is defined by a function subprogram containing a generic interface block with the specifier OPERATOR. See also *derived type* and interface block.
denormalized number	A computational floating-point result smaller than the lowest value in the normal range of a data type (the smallest representable normalized number). You cannot write a constant for a subnormal number. Older floating-point standard documents used the term denormal; newer standards call such numbers subnormal.
derived type	A data type that is user-defined and not intrinsic. It requires a type definition to name the type and specify its components (which can be intrinsic or user-defined types). A structure constructor can be used to specify a value of derived type. A component of a structure is referenced using a percent sign (%). Operations on objects of derived types (structures) must be defined by a function with an OPERATOR interface. Assignment for derived types can be defined intrinsically, or be redefined by a subroutine with an ASSIGNMENT(=) interface. Structures can be used as procedure arguments and function results, and can appear in input and output lists. Also called a user-defined type. See also record, the first definition.
designator	A name that references a subobject (part of a data object) that can be defined and referenced separately from other parts of the data object. A designator is the name of the object followed by a selector that selects the subobject. For example, B(3) is a designator for an array element. Also called a subobject designator. See also selector and subobject.
dimension	A range of values for one subscript or index of an array. An array can have from 1 to 7 dimensions. The number of dimensions is the rank of the array.
dimension bounds	See bounds.

direct access	A method for retrieving or storing data in which the data (record) is identified by the record number, or the position of the record in the file. The record is accessed directly (nonsequentially); therefore, all information is equally accessible. Also called random access. Contrast with sequential access.
DLL	See *Dynamic Link Library*.
double-byte character set (DBCS)	A mapping of characters to their identifying numeric values, in which each value is 2 bytes wide. Double-byte character sets are sometimes used for languages that have more than 256 characters.
double-precision constant	A processor approximation to the value of a real number that occupies 8 bytes of memory and can assume a positive, negative, or zero value. The precision is greater than a constant of real (single-precision) type. For the precise ranges of the double-precision constants, see Data Representation Overview in the *Compiler Reference*. See also *denormalized number*.
driver program	A program that is the user interface to the language compiler. It accepts command line options and file names and causes one or more language utilities or system programs to process each file.
dummy aliasing	The sharing of memory locations between dummy (formal) arguments and other dummy arguments or COMMON variables that are assigned.
dummy argument	A variable whose name appears in the parenthesized list following the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement. A dummy argument takes the value of the corresponding actual argument in the calling program unit (through argument association). Also called a formal argument.
dummy array	A dummy argument that is an array.
dummy pointer	A dummy argument that is a pointer.
dummy procedure	A dummy argument that is specified as a procedure or appears in a procedure reference. The corresponding actual argument must be a procedure.
Dynamic Link Library (DLL)	A separate source module compiled and linked independently of the applications that use it. Applications access the DLL through procedure calls. The code for a DLL is not included in the user's executable image, but the compiler automatically modifies the executable image to point to DLL procedures at run time.
dynamic type	The type of a data entity during execution of a program. The dynamic type of a data entity that is not polymorphic is the same as its declared type.

Glossary E

edit descriptor	A descriptor in a format specification. It can be a data edit descriptor, control edit descriptor, or string edit descriptor. See also control edit descriptor, data edit descriptor, and string edit descriptor.
element	See array element.
elemental	Pertains to an intrinsic operation, intrinsic procedure, or assignment statement that is independently applied to either of the following:

	• The elements of an array • Corresponding elements of a set of conformable arrays and scalars
end-of-file	The condition that exists when all records in a file open for sequential access have been read.
entity	A general term referring to any Standard Fortran concept; for example, a constant, a variable, a program unit, a statement label, a common block, a construct, an I/O unit and so forth.
environment variable	A symbolic variable that represents some element of the operating system, such as a path, a filename, or other literal data.
error number	An integer value denoting an I/O error condition, obtained by using the IOSTAT keyword in an I/O statement.
error termination	When error termination is initiated on an image, all images terminate execution. If an error condition occurs, error termination is initiated unless the error occurs in one of the following ways: • During the execution of a statement that specifies a STAT= specifier • During the execution of an I/O statement that specifies a STAT= or ERR= specifier • During the execution of an intrinsic procedure with a present STAT argument. In these cases, the STAT argument of the intrinsic procedure, or the *stat-variable* of the STAT= specifier becomes defined with a positive integer value and the program execution continues. See also normal termination.
exceptional values	For floating-point numbers, values outside the range of normalized numbers, including subnormal numbers, infinity, Not-a-Number (NaN) values, zero, and other architecture-defined numbers.
executable construct	A CASE, DO, IF, WHERE, or FORALL construct.
executable program	A set of program units that include only one main program.
executable statement	A statement that specifies an action to be performed or controls one or more computational instructions.
explicit interface	A procedure interface whose properties are known within the scope of the calling program, and do not have to be assumed. These properties are the names of the procedure and its dummy arguments, the attributes of a procedure (if it is a function), and the attributes and order of the dummy arguments. The following have explicit interfaces: • Internal and module procedures (explicit by definition) • Intrinsic procedures • External procedures that have an interface block • External procedures that are defined by the scoping unit and are recursive • Dummy procedures that have an interface block
explicit-shape array	An array whose rank and bounds are specified when the array is declared.
expression	A data reference or a computation formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.

extended type	An extensible type that is an extension of another type. A type that is declared with the EXTENDS attribute.
extensible type	A type from which new types may be derived using the EXTENDS attribute. A nonsequence type that does not have the BIND attribute.
extent	The size of (number of elements in) one dimension of an array.
external file	A sequence of records that exists in a medium external to the executing program.
external procedure	A procedure that is contained in an external subprogram. External procedures can be used to share information (such as source files, common blocks, and public data in modules) and can be used independently of other procedures and program units. Also called an external routine.
external subprogram	A subroutine or function that is not contained in a main program, module, or another subprogram. A module is not a subprogram.

Glossary F

failed image	An image that has not initiated normal or error termination but has stopped participating in program execution.
field	Can be either of the following: • A set of contiguous characters, considered as a single item, in a record or line. • A substructure of a STRUCTURE declaration.
field descriptor	See data edit descriptor.
field separator	The comma (,) or slash (/) that separates edit descriptors in a format specification.
field width	The total number of characters in the field. See also *field*, the first definition.
file	A collection of logically related records. If the file is in internal storage, it is an internal file; if the file is on an input/output device, it is an external file.
file access	The way records are accessed (and stored) in a file. The standard Fortran file access modes are sequential and direct.
file handle	A unique identifier that the system assigns to a file when the file is opened or created. A file handle is valid until the file is closed.
file organization	The way records in a file are physically arranged on a storage device. Standard Fortran files can have sequential or relative organization.
final subroutine	A subroutine whose name appears in a FINAL statement in a type definition. It can be automatically invoked by the processor when an object of that type is finalized.
final task	A task that forces all of its descendant tasks to become included tasks.
fixed-length record type	A file format in which all the records are the same length.

floating-point environment	A collection of registers that control the behavior of floating-point (FP) machine instructions and indicate the current FP status. The floating-point environment may include rounding mode controls, exception masks, flush-to-zero controls, exception status flags, and other floating-point related features.
focus window	The window to which keyboard input is directed.
foreground process	On Linux* systems, a process for which the command interpreter is waiting. Its process group is the same as that of its controlling terminal, so the process is allowed to read from or write to the terminal. Contrast with background process.
foreground window	The window the user is currently working with. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.
foreign file	An unformatted file that contains data from a foreign platform, such as data from a CRAY*, IBM*, or big endian IEEE* machine.
format	A specific arrangement of data. A FORMAT statement specifies how data is to be read or written.
format specification	The part of a FORMAT statement that specifies explicit data arrangement. It is a list within parentheses that can include edit descriptors and field separators. A character expression can also specify format; the expression must evaluate to a valid format specification.
formatted data	Data written to a file by using formatted I/O statements. Such data contains ASCII representations of binary values.
formatted I/O statement	An I/O statement specifying a format for data transfer. The format specified can be explicit (specified in a format specification) or implicit (specified using list-directed or namelist formatting). Contrast with unformatted I/O statement. See also list-directed I/O statement and namelist I/O statement.
frame window	The outermost parent window in QuickWin.
function	A series of statements that perform some operation and return a single value (through the function or result name) to the calling program unit. A function is invoked by a function reference in a main program unit or a subprogram unit. In Standard Fortran, a function can be used to define a new operator or extend the meaning of an intrinsic operator symbol. The function is invoked by the appearance of the new or extended operator in the expression (along with the appropriate operands). For example, the symbol * can be defined for logical operands, extending its intrinsic definition for numeric operands. See also *function subprogram*, statement function, and subroutine.
function reference	Used in an expression to invoke a function, it consists of the function name and its actual arguments. A function reference returns a value (through the function or result name) that is used to evaluate the calling expression.
function result	The result value associated with a particular execution or call to a function. This result can be of any data type (including derived type) and can be array-valued. In a FUNCTION statement, the RESULT option can be used to give the result a name different from the function name. This option is required for a recursive function that directly calls itself.

function subprogram	A sequence of statements beginning with a FUNCTION (or optional OPTIONS) statement that is not in an interface block and ending with the corresponding END statement. See also *function*.

Glossary G

generic identifier	A generic name, operator, or assignment specified in an INTERFACE statement that is associated with all of the procedures within the interface block. Also called a generic specification.
global entity	An entity (a program unit, common block, or external procedure) that can be used with the same meaning throughout the executable program. A global entity has global scope; it is accessible throughout an executable program. See also local entity.
global section	A data structure (for example, global COMMON) or shareable image section potentially available to all processes in the system.

Glossary H

handle	A value (often, but not always, a 32-bit integer) that identifies some operating system resource, for example, a window or a process. The handle value is returned from an operating system call when the resource is created; your program then passes that value as an argument to subsequent operating system routines to identify which resource is being accessed. Your program should consider the handle value a "private" type and not try to interpret it as having any specific meaning (for example, an address).
hexadecimal constant	A constant that is a string of hexadecimal (base 16) digits (range 0 to 9, or an uppercase or lowercase letter in the range A to F) enclosed by apostrophes or quotation marks and preceded by the letter Z.
Hollerith constant	A constant that is a string of printable ASCII characters preceded by *n*H, where *n* is the number of characters in the string (including blanks and tabs).
host	Either the main program or subprogram that contains an internal procedure, or the module that contains a module procedure. The data environment of the host is available to the (internal or module) procedure.
host association	The process by which a module procedure, internal procedure, or derived-type definition accesses the entities of its host.

Glossary I

image	An instance of a Fortran program.
image index	An integer value that identifies an image.
image control statement	A statement that affects the execution ordering between images.

implicit interface	A procedure interface whose properties (the collection of names, attributes, and arguments of the procedure) are not known within the scope of the calling program, and have to be assumed. The information is assumed by the calling program from the properties of the procedure name and actual arguments in the procedure call.
implicit typing	The mechanism by which the data type for a variable is determined by the beginning letter of the variable name.
import library	A .LIB file that contains information about one or more dynamic-link libraries (DLLs), but does not contain the DLL's executable code. To provide the information needed to resolve the external references to DLL functions, the linker uses an import library when building an executable module of a process.
included task	A task for which execution is sequentially included in the generating task region; that is, the task is undeferred and executed immediately by the encountering thread.
index	Can be either of the following: • The variable used as a loop counter in a DO statement. • An intrinsic function specifying the starting position of a substring inside a string.
inheritance association	The relationship between the inherited components and the parent component in an extended type.
initial team	The team existing when the program begins execution, consisting of all images.
initialize	The assignment of an initial value to a variable.
inlining	An optimization that replaces a subprogram reference (CALL statement or function invocation) with the replicated code of the subprogram.
input/output (I/O)	The data that a program reads or writes. Also, devices to read and write data.
inquiry function	An intrinsic function whose result depends on properties of the principal argument, not the value of the argument.
integer constant	A constant that is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.
intent	An attribute of a dummy argument that is not a procedure or a pointer. It indicates whether the argument is used to transfer data into the procedure, out of the procedure, or both.
interactive process	A process that must periodically get user input to do its work. Contrast with background process.
interface	See procedure interface.
interface block	The sequence of statements starting with an INTERFACE statement and ending with the corresponding END INTERFACE statement.
interface body	The sequence of statements in an interface block starting with a FUNCTION or SUBROUTINE statement and ending with the corresponding END statement. Also called a procedure interface body.
internal file	The designated internal storage space (or variable buffer) that is manipulated during input and output. An internal file can be a character variable, character array, character array element, or

	character substring. In general, an internal file contains one record. However, an internal file that is a character array has one record for each array element.
internal procedure	A procedure (other than a statement function) that is contained within an internal subprogram. The program unit containing an internal procedure is called the host of the internal procedure. The internal procedure (which appears between a CONTAINS and END statement) is local to its host and inherits the host's environment through host association.
internal subprogram	A subprogram contained in a main program or another subprogram.
intrinsic	Describes entities defined by the Fortran language (such as data types and procedures). Intrinsic entities can be used freely in any scoping unit.
intrinsic procedure	A subprogram supplied as part of the Fortran library that performs array, mathematical, numeric, character, bit manipulation, and other miscellaneous functions. Intrinsic procedures are automatically available to any Fortran program unit (unless specifically overridden by an EXTERNAL statement or a procedure interface block). Also called a built-in or library procedure.
invoke	To call upon; used especially with reference to subprograms. For example, to invoke a function is to execute the function.
iteration count	The number of executions of the DO range, which is determined as follows: [(terminal value - initial value + increment value) / increment value]

Glossary K

keyword	See argument keyword and statement keyword.
kind type parameter	Indicates the range of an intrinsic data type; for example: INTEGER(KIND=2). For real and complex types, it also indicates precision. If a specific kind parameter is not specified, the kind is the default for that type (for example, default integer). See also default character, default complex, default integer, default logical, and default real.

Glossary L

label	An integer, from 1 to 5 digits long, that precedes a statement and identifies it. For example, labels can be used to refer to a FORMAT statement or branch target statement.
language extension	An Intel® Fortran language element or interpretation that is not part of the Fortran 2008 standard.
lexical token	A sequence of one or more characters that have an indivisible interpretation. A lexical token is the smallest meaningful unit (a basic language element) of a Fortran statement; for example, constants, and statement keywords.
library routines	Files that contain functions, subroutines, and data that can be used by Fortran programs.

	For example: one library contains routines that handle the various differences between Fortran and C in argument passing and data types; another contains run-time functions and subroutines for Windows* graphics and QuickWin* applications. Some library routines are intrinsic (automatically available) to Fortran; others may require a specific USE statement to access the module defining the routines. See also intrinsic procedure.
line	A source form record consisting of 0 or more characters. A standard Fortran line is limited to a maximum of 132 characters.
linker	A system program that creates an executable program from one or more object files produced by a language compiler or assembler. The linker resolves external references, acquires referenced library routines, and performs other processing required to create Linux* and Windows* executable files.
list-directed I/O statement	An implicit, formatted I/O statement that uses an asterisk (*) specifier rather than an explicit format specification. See also formatted I/O statement and namelist I/O statement.
listing	A printed copy of a program.
literal constant	A constant without a name; its value is directly specified in a program. See also named constant.
little endian	A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the lowest addressed byte. This is the method used on Intel® systems. Contrast with big endian.
local entity	An entity that can be used only within the context of a subprogram (its scoping unit); for example, a statement label. A local entity has local scope. See also global entity.
local optimization	A level of optimization enabling optimizations within the source program unit and recognition of common expressions. See also optimization.
local symbol	A name defined in a program unit that is not accessible outside of that program unit.
logical constant	A constant that specifies the value .TRUE. or .FALSE..
logical expression	An integer or logical constant, variable, function value, or another constant expression, joined by a relational or logical operator. The logical expression is evaluated to a value of either true or false. For example, .NOT. 6.5 + (B .GT. D).
logical operator	A symbol that represents an operation on logical expressions. The logical operators are .AND., .OR., .NEQV., .XOR., .EQV., and .NOT..
logical unit	A channel in memory through which data transfer occurs between the program and the device or file. See also unit identifier.
longword	Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered 0 to 31. The address of the longword is the address of the byte containing bit 0. When the longword is interpreted as a signed integer, bit 31 is the sign bit. The value of signed integers is in the range -2**31 to 2**31-1. The value of unsigned integers is in the range 0 to 2**32-1.

loop	A group of statements that are executed repeatedly until an ending condition is reached.
lower bounds	See bounds.

Glossary M

main program	The first program unit to receive control when a program is run; it exercises control over subprograms. The main program usually contains a PROGRAM statement (or does not contain a SUBROUTINE, FUNCTION, or BLOCK DATA statement). Contrast with subprogram.
makefile	On Linux* and macOS* systems, an argument to the make command containing a sequence of entries that specify dependencies. On Windows* systems, a file passed to the NMAKE utility containing a sequence of entries that specify dependencies. The contents of a makefile override the system built-in rules for maintaining, updating, and regenerating groups of programs. For more information on makefiles on Linux* and macOS* systems, see make(1). For more information on using makefiles, see Using Makefiles to Compile Your Application in *Compiler Setup*.
many-one array section	An array section with a vector subscript having two or more elements with the same value.
master thread	In an OpenMP* Fortran program, the thread that creates a team of threads when a parallel region (PARALLEL directive construct) is encountered. The statements in the parallel region are then executed in parallel by each thread in the team. At the end of the parallel region, the team threads synchronize and only the master thread continues execution. See also thread.
merged task	A task whose data environment, inclusive of internal control variables, is the same as that of its generating task region. Internal control variables (ICVs) are discussed in the latest OpenMP* specifications.
message file	A Linux* and macOS* catalog that contains the diagnostic message text of errors that can occur during program execution (run time).
metacommand	See compiler directive.
misaligned data	Data not aligned on a natural boundary. See also natural boundary.
module	A program unit that contains specifications and definitions that other program units can access (unless the module entities are declared PRIVATE). Modules are referenced in USE statements.
module procedure	A subroutine or function that is not an internal procedure and is contained in a module. The module procedure appears between a CONTAINS and END statement in its host module, and inherits the host module's environment through host association. A module procedure can be declared PRIVATE to the module; it is public by default.
multibyte character set	A character set in which each character is identified by using more than one byte. Although Unicode characters are 2 bytes wide, the Unicode character set is not referred to by this term.
multitasking	The ability of an operating system to execute several programs (tasks) at once.

multithreading	The ability of an operating system to execute different parts of a program, called threads, simultaneously. If the system supports parallel processing, multiple processors may be used to execute the threads.

Glossary N

name	Identifies an entity within a Fortran program unit (such as a variable, function result, common block, named constant, procedure, program unit, namelist group, or dummy argument). A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign. In earlier versions of Fortran, this term was called a symbolic name.
name association	Pertains to argument, host, or use association. See also argument association, host association, and use association.
named common block	A common block (one or more contiguous areas of storage) with a name. Common blocks are defined by a COMMON statement.
named constant	A constant that has a name. In earlier versions of Fortran, this term was called a symbolic constant.
namelist I/O statement	An implicit, formatted I/O statement that uses a namelist group specifier rather than an explicit format specifier. See also formatted I/O statement and list-directed I/O statement.
NaN	Not-a-Number. The condition that results from a floating-point operation that has no mathematical meaning; for example, zero divided by zero.
natural boundary	The virtual address of a data item that is the multiple of the size of its data type. For example, a REAL(8) (REAL*8) data item aligned on natural boundaries has an address that is a multiple of eight.
naturally aligned record	A record that is aligned on a hardware-specific natural boundary; each field is naturally aligned. Contrast with packed record.
nesting	The placing of one entity (such as a construct, subprogram, format specification, or loop) inside another entity of the same kind. For example, nesting a loop within another loop (a nested loop), or nesting a subroutine within another subroutine (a nested subroutine).
nonexecutable statement	A Fortran statement that describes program attributes, but does not cause any action to be taken when the program is executed.
nonsignaled	The state of an object used for synchronization in one of the wait functions is either signaled or nonsignaled. A nonsignaled state can prevent the wait function from returning. See also wait function.
normal termination	Normal termination of an image is initiated when the image executes a STOP statement or an END [PROGRAM] statement. The image becomes a stopped image. Its coarrays remain accessible to other active images and may be defined or referenced by them. When the last active image initiates normal termination, all images terminate execution. See also error termination.

numeric expression	A numeric constant, variable, or function value, or combination of these, joined by numeric operators and parentheses, so that the entire expression can be evaluated to produce a single numeric value. For example, -L or X+(Y-4.5*Z).
numeric operator	A symbol designating an arithmetic operation. In Standard Fortran, the symbols +, -, *, /, and ** are used to designate addition, subtraction, multiplication, division, and exponentiation, respectively.
numeric storage unit	The unit of storage for holding a non-pointer scalar value of type default real, default integer, or default logical. One numeric storage unit corresponds to 4 bytes of memory.
numeric type	Integer, real, or complex type.

Glossary O

object	See data object.
object file	The binary output of a language processor (such as an assembler or compiler), which can either be executed or used as input to the linker.
obsolescent feature	A feature of earlier versions of Fortran that is considered to be redundant in Fortran 2008. These features are still in frequent use.
octal constant	A constant that is a string of octal (base 8) digits (range of 0 to 7) enclosed by apostrophes or quotation marks and preceded by the letter O.
operand	The passive element in an expression on which an operation is performed. Every expression must have at least one operand. For example, in I .NE. J, I and J are operands. Contrast with *operator*.
operation	A computation involving one or two operands.
operator	The active element in an expression that performs an operation. An expression can have zero or more operators. Intrinsic operators are arithmetic (+, -, *, /, and **) or logical (.AND., .NOT., and so on). For example, in I .NE. J, .NE. is the operator. Executable programs can define operators which are not intrinsic.
optimization	The process of producing efficient object or executing code that takes advantage of the hardware architecture to produce more efficient execution.
optional argument	A dummy argument that has the OPTIONAL attribute (or is included in an OPTIONAL statement in the procedure definition). This kind of argument does not have to be associated with an actual argument when its procedure is invoked.
order of subscript progression	A characteristic of a multidimensional array in which the leftmost subscripts vary most rapidly. Also called column-major order.
overflow	An error condition occurring when an arithmetic operation yields a result that is larger than the maximum value in the range of a data type.

Glossary P

packed record	A record that starts on an arbitrary byte boundary; each field starts in the next unused byte. Contrast with naturally aligned record.
pad	The filling of unused positions in a field or character string with dummy data (such as zeros or blanks).
parallel processing	The simultaneous use of more than one processor (CPU) to execute a program.
parameter	Can be either of the following: • In general, any quantity of interest in a given situation; often used in place of the term "argument". • A Fortran named constant.
parent component	The component of an entity of extended type that corresponds to its inherited portion.
parent process	A process that initiates and controls another process (child). The parent process defines the environment for the child process. Also, the parent process can suspend or terminate without affecting the child process. See also child process.
parent window	A window that has one or more child windows. See also child window.
parent type	The extensible type from which an extended type is derived.
passed-object dummy argument	The dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the object through which the procedure was invoked.
pathname	The path from the root directory to a subdirectory or file. See also root.
pipe	A connection that allows one program to get its input directly from the output of another program.
platform	A combination of operating system and hardware that provides a distinct environment in which to use a software product (for example, Microsoft* Windows* on processors using IA-32 architecture).
pointer	A pointer is either a data pointer or a procedure pointer. A data pointer is a data entity that has the POINTER attribute. A procedure pointer is a procedure entity that has the POINTER attribute. A pointer is associated with a target by pointer assignment. A data pointer can also be associated with a target by allocation. A pointer that is not associated must not be referenced or defined. A disassociated pointer is not associated with a target. A pointer is disassociated following one of these events: • Execution of a NULLIFY statement • Pointer assignment with a disassociated pointer • Default initialization • Explicit initialization A data pointer can also be disassociated by execution of a DEALLOCATE statement. If a data pointer is an array, the rank is declared, but the extents are determined when the pointer is associated with a target. A data pointer is one of the following:

	• A Fortran pointer A data object that has the POINTER attribute. To be referenced or defined, it must be "pointer-associated" with a target (have storage space associated with it). If the pointer is an array, it must be pointer-associated to have a shape. See also *pointer association*. • An integer pointer A data object that contains the address of its paired variable. This is also called a Cray* pointer.
pointer assignment	The association of a pointer with a target by the execution of a pointer assignment statement or the execution of an assignment statement for a data object of derived type having the pointer as a subobject.
pointer association	The association of storage space to a Fortran pointer by means of a target. A pointer is associated with a target after pointer assignment or the valid execution of an ALLOCATE statement.
polymorphic object	An object that can have different types during program execution. An object declared with the CLASS keyword is polymorphic.
potential subobject component	For a derived type or a structure, a nonpointer component or a potential subobject component of a nonpointer component.
precision	The number of significant digits in a real number. See also double-precision constant, kind type parameter, and single-precision constant.
primary	The simplest form of an expression. A primary can be any of the following data objects: • A constant • A constant subobject (parent is a constant) • A variable (scalar, structure, array, or pointer; an array cannot be assumed size) • An array constructor • A structure constructor • A function reference • An expression in parentheses
primary thread	The initial thread of a process. Also called the main thread or thread 1. See also thread.
procedure	A computation that can be invoked during program execution. It can be a subroutine or function, an internal, external, dummy or module procedure, or a statement function. A subprogram can define more than one procedure if it contains an ENTRY statement. See also subprogram.
procedure interface	The statements that specify the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units. If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration).

process object	A virtual address space, security profile, a set of threads that execute in the address space of the process, and a set of resources visible to all threads executing in the process. Several thread objects can be associated with a single process.
program	A set of instructions that can be compiled and executed by itself. Program blocks contain a declaration and an executable section.
program section	A particular common block or local data area for a particular routine containing equivalence groups.
program unit	The fundamental component of an executable program. A sequence of statements and comment lines. It can be a main program, a module, an external subprogram, or a block data program unit.

Glossary Q

quadword	Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered 0 to 63. (Bit 63 is used as the sign bit.) A quadword is identified by the address of the word containing the low-order bit (bit 0). The value of a signed quadword integer is in the range -2**63 to 2**63-1.

Glossary R

random access	See direct access.
rank	The number of dimensions of an array. A scalar has a rank of zero.
rank-one object	A data structure comprising scalar elements with the same data type and organized as a simple linear sequence. See also scalar.
real constant	A constant that is a number written with a decimal point, exponent, or both. It can have single precision (REAL(KIND=4)), double precision (REAL(KIND=8)), or quad precision (REAL(KIND=16)).
record	Can be either of the following: • A set of logically related data items (in a file) that is treated as a unit; such a record contains one or more fields. This definition applies to I/O records and items that are declared in a record structure. • One or more data items that are grouped in a structure declaration and specified in a RECORD statement.
record access	The method used to store and retrieve records in a file.
record structure declaration	A block of statements that define the fields in a record. The block begins with a STRUCTURE statement and ends with END STRUCTURE. The name of the structure must be specified in a RECORD statement.
record type	The property that determines whether records in a file are all the same length, of varying length, or use other conventions to define where one record ends and another begins.
recursion	Pertains to a subroutine or function that directly or indirectly references itself.

reference	Can be any of the following: • For a data object, the appearance of its name, designator, or associated pointer where the value of the object is required. When an object is referenced, it must be defined. • For a procedure, the appearance of its name, operator symbol, or assignment symbol that causes the procedure to be executed. Procedure reference is also called "calling" or "invoking" a procedure. • For a module, the appearance of its name in a USE statement.
relational expression	An expression containing one relational operator and two operands of numeric or character type. The result is a value that is true or false. For example, A-C .GE. B+2 or DAY .EQ. 'MONDAY'.
relational operator	The symbols used to express a relational condition or expression. The relational operators are (.EQ., .NE., .LT., .LE., .GT., and .GE.).
relative file organization	A file organization that consists of a series of component positions, called cells, numbered consecutively from 1 to n. Intel Fortran uses these numbered, fixed-length cells to calculate the component's physical position in the file.
relative pathname	A directory path expressed in relation to any directory other than the root directory. Contrast with absolute pathname.
root	On Windows* systems, the top-level directory on a disk drive; it is represented by a backslash (\). For example, C:\ is the root directory for drive C. On Linux* systems, the top-level directory in the file system; it is represented by a slash (/).
routine	A subprogram; a function or procedure. See also function, subroutine, and procedure.
run time	The time during which a computer executes the statements of a program.

Glossary S

saved object	A variable that retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement in the scoping unit containing the declaration.
scalar	Pertaining to data items with a rank of zero. A single data object of any intrinsic or derived data type. Contrast with array. See also rank-one object.
scalar memory reference	A reference to a scalar variable, scalar record field, or array element that resolves into a single data item (having a data type) and can be assigned a value with an assignment statement. It is similar to a scalar reference, but it excludes constants, character substrings, and expressions.
scalar reference	A reference to a scalar variable, scalar record field, derived-type component, array element, constant, character substring, or expression that resolves into a single data item having a data type.
scalar variable	A variable name specifying one storage location.

scale factor	A number indicating the location of the decimal point in a real number and, if there is no exponent, the size of the number on input.
scope	The portion of a program in which a declaration or a particular name has meaning. Scope can be global (throughout an executable program), scoping unit (local to the scoping unit), or statement (within a statement, or part of a statement).
scoping unit	The part of the program in which a name has meaning. It is one of the following: • A program unit or subprogram • A derived-type definition • A procedure interface body Scoping units cannot overlap, though one scoping unit can contain another scoping unit. The outer scoping unit is called the host scoping unit.
screen coordinates	Coordinates relative to the upper left corner of the screen.
section subscript	A subscript list (enclosed in parentheses and appended to the array name) indicating a portion (section) of an array. At least one of the subscripts in the list must be a subscript triplet or vector subscript. The number of section subscripts is the rank of the array. See also array section, subscript, *subscript triplet*, and vector subscript.
seed	A value (which can be assigned to a variable) that is required in order to properly determine the result of a calculation; for example, the argument k in the random number generator (RAN) function syntax: y = RAN (k)
selector	A mechanism for designating the following: • Part of a data object (an array element or section, a substring, a derived type, or a structure component) • The set of values for which a CASE block is executed
sequence	A set ordered by a one-to-one correspondence with the numbers 1 through n, where *n* is the total number of elements in the sequence. A sequence can be empty (contain no elements).
sequential access	A method for retrieving or storing data in which the data (record) is read from, written to, or removed from a file based on the logical order (sequence) of the record in the file. (The record cannot be accessed directly.) Contrast with direct access.
sequential file organization	A file organization in which records are stored one after the other, in the order in which they were written to the file.
shape	The rank and extents of an array. Shape can be represented by a rank-one array (vector) whose elements are the extents in each dimension.
shape conformance	Pertains to the rule concerning operands of binary intrinsic operations in expressions: to be in shape conformance, the two operands must both be arrays of the same shape, or one or both of the operands must be scalars.
short field termination	The use of a comma (,) to terminate the field of a numeric data edit descriptor. This technique overrides the field width (w) specification in the data edit descriptor and therefore avoids padding of the input field. The comma can only terminate fields less than w characters long. See also data edit descriptor.

signal	The software mechanism used to indicate that an exception condition (abnormal event) has been detected. For example, a signal can be generated by a program or hardware error, or by request of another program.
single-precision constant	A processor approximation of the value of a real number that occupies 4 bytes of memory and can assume a positive, negative, or zero value. The precision is less than a constant of double-precision type. For the precise ranges of the single-precision constants, see Data Representation Overview in the *Compiler Reference*. See also subnormal number.
size	The total number of elements in an array (the product of the extents).
source file	A program or portion of a program library, such as an object file, or image file.
specification expression	A restricted expression that is of type integer and has a scalar value. This type of expression appears only in the declaration of array bounds and character lengths.
specification statement	A nonexecutable statement that provides information about the data used in the source program. Such a statement can be used to allocate and initialize variables, arrays, records, and structures, and define other characteristics of names used in a program.
statement	An instruction in a programming language that represents a step in a sequence of actions or a set of declarations. In Standard Fortran, an ampersand can be used to continue a statement from one line to another, and a semicolon can be used to separate several statements on one line. There are two main classes of statements: executable and nonexecutable.
statement function	A computing procedure defined by a single statement in the same program unit in which the procedure is referenced.
statement function definition	A statement that defines a statement function. Its form is the statement function name (followed by its optional dummy arguments in parentheses), followed by an equal sign (=), followed by a numeric, logical, or character expression. A statement function definition must precede all executable statements and follow all specification statements.
statement keyword	A word that begins the syntax of a statement. All program statements (except assignment statements and statement function definitions) begin with a statement keyword. Examples are INTEGER, DO, IF, and WRITE.
statement label	See label.
static variable	A variable whose storage is allocated for the entire execution of a program.
stopped image	An image that has begun normal termination, either by execution of a STOP statement, or by execution of an END statement or END PROGRAM statement terminating the main program.
storage association	The relationship between two storage sequences when the storage unit of one is the same as the storage unit of the other. Storage association is provided by the COMMON and EQUIVALENCE

	statements. For modules, pointers, allocatable arrays, and automatic data objects, the SEQUENCE statement defines a storage order for structures.
storage location	An addressable unit of main memory.
storage sequence	A sequence of any number of consecutive storage units. The size of a storage sequence is the number of storage units in the storage sequence. A sequence of storage sequences forms a composite storage sequence. See also *storage association* and *storage unit*.
storage unit	In a storage sequence, the number of storage units needed to represent one real, integer, logical, or character value. See also character storage unit, numeric storage unit, and *storage sequence*.
stride	The increment between subscript values that can optionally be specified in a subscript triplet. If it is omitted, it is assumed to be one.
string edit descriptor	A format descriptor that transfers characters to an output record.
structure	Can be either of the following: • A scalar data object of derived (user-defined) type. • An aggregate entity containing one or more fields or components.
structure component	Can be either of the following: • One of the components of a structure. • An array whose elements are components of the elements of an array of derived type.
structure constructor	A mechanism that is used to specify a scalar value of a derived type. A structure constructor is the name of the type followed by a parenthesized list of values for the components of the type.
subnormal number	A computational floating-point result smaller than the lowest value in the normal range of a data type (the smallest representable normalized number). You cannot write a constant for a subnormal number. Older floating-point standard documents used the term denormal; newer standards call such numbers subnormal.
subobject	Part of a data object (parent object) that can be referenced and defined separately from other parts of the data object. A subobject can be an array element, an array section, a substring, a derived type, or a structure component. Subobjects are referenced by designators and can be considered to be data objects themselves. See also designator.
subobject designator	See designator.
subprogram	A function or subroutine subprogram that can be invoked from another program unit to perform a specific task. A subprogram can define more than one procedure if it contains an ENTRY statement. Contrast with main program. See also procedure.
subroutine	A procedure that can return many values, a single value, or no value to the calling program unit (through arguments). A subroutine is invoked by a CALL statement in another program unit. In recent versions of Fortran, a subroutine can also be used to specify a defined assignment. Such assignments are invoked with an ASSIGNMENT(=) interface block rather than the CALL statement. See also function, *statement function*, and *subroutine subprogram*.

subroutine subprogram	A sequence of statements starting with a SUBROUTINE (or optional OPTIONS) statement and ending with the corresponding END statement. See also *subroutine*.
subscript	A scalar integer expression (enclosed in parentheses and appended to the array name) indicating the position of an array element. The number of subscripts is the rank of the array. See also array element.
subscript triplet	An item in a section subscript list specifying a range of values for the array section. A subscript triplet contains at least one colon and has three optional parts: a lower bound, an upper bound, and a stride. Contrast with vector subscript. See also array section and *section subscript*.
substring	A contiguous portion of a scalar character string. Do not confuse this with the substring selector in an array section, where the result is another array section, not a substring.
symbolic name	See name.
syntax	The formal structure of a statement or command string.

Glossary T

target	The named data object associated with a pointer (in the form pointer-object => target). A target is declared in a type declaration statement that contains the TARGET attribute. See also pointer and pointer association.
team	The initial ordered set of all images.
thread	Part of a program that can run at the same time as other parts, usually with some form of communication and/or synchronization among the threads. See also multithreading.
transformational function	An intrinsic function that is not an elemental or inquiry function. A transformational function usually changes an array actual argument into a scalar result or another array, rather than applying the argument element by element.
truncation	Can be either of the following: • A technique that approximates a numeric value by dropping its fractional value and using only the integer portion. • The process of removing one or more characters from the left or right of a number or string.
type declaration statement	A nonexecutable statement specifying the data type of one or more variables: an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, CHARACTER, LOGICAL, or TYPE statement. A type declaration statement may also specify attributes for the variables. Also called a type declaration or type specification.
type parameter	Defines an intrinsic data type. The type parameters are kind and length. The kind type parameter (KIND=) specifies the range for the integer data type, the precision and range for real and complex data types, and the machine representation method for the character and logical data types. The length type parameter (LEN=) specifies the length of a character string. See also kind type parameter.
type-bound procedure	A procedure that is bound to a derived type and referenced by means of an object of that type.

Glossary U

ultimate component	For a derived type or a structure, a component that is of intrinsic type or has the ALLOCATABLE or POINTER attribute, or an ultimate component of a component that is a derived type and does not have the ALLOCATABLE or POINTER attribute.
unary operator	An operator that operates on one operand. For example, the minus sign in -A and the .NOT. operator in .NOT. (J .GT. K).
undeferred task	A task for which execution is not deferred with respect to its generating task region; that is, its generating task region is suspended until execution of the undeferred task has finished.
undefined	For a data object, the property of not having a determinate value.
underflow	An error condition occurring when the result of an arithmetic operation yields a result that is smaller than the minimum value in the range of a data type. For example, in unsigned arithmetic, underflow occurs when a result is negative. See also subnormal number.
unformatted data	Data written to a file by using unformatted I/O statements; for example, binary numbers.
unformatted I/O statement	An I/O statement that does not contain format specifiers and therefore does not translate the data being transferred. Contrast with formatted I/O statement.
unformatted record	A record that is transmitted in internal format between internal and external storage.
unit identifier	The identifier that specifies an external unit or internal file. The identifier can be any one of the following: • An integer expression whose value must be zero or positive • An asterisk (*) that corresponds to the default (or implicit) I/O unit • The name of a character scalar memory reference or character array name reference for an internal file Also called a device code, or logical unit number.
unspecified storage unit	A unit of storage for holding a pointer or a scalar that is not a pointer and is of type other than default integer, default character, or default real.
upper bounds	See bounds.
use association	The process by which the entities in a module are made accessible to other scoping units (through a USE statement in the scoping unit).
user-defined assignment	See defined assignment.
user-defined operator	See defined operation.
user-defined type	See derived type.

Glossary V

variable	A data object (stored in a memory location) whose value can change during program execution. A variable can be a named data object, an array element, an array section, a structure component, or a substring. In FORTRAN 77, a variable was always scalar and named. Contrast with constant.
variable format expression	A numeric expression enclosed in angle brackets (<>) that can be used in a FORMAT statement. If necessary, it is converted to integer type before use.
variable-length record type	A file format in which records may be of different lengths.
vector subscript	A rank-one array of integer values used as a section subscript to select elements from a parent array. Unlike a subscript triplet, a vector subscript specifies values (within the declared bounds for the dimension) in an arbitrary order. Contrast with subscript triplet. See also array section and section subscript.

Glossary W

wait function	A function that blocks the execution of a calling thread until a specified set of conditions has been satisfied.
whole array	An array reference (for example, in a type declaration statement) that consists of the array name alone, without subscript notation. Whole array operations affect every element in the array. See also array.

Glossary Z

zero-sized array	An array with at least one dimension that has at least one extent of zero. A zero-sized array has a size of zero and contains no elements. See also array.

Part V

Compilation

Supported Environment Variables

You can customize your system environment by specifying paths where the compiler searches for certain files such as libraries, include files, configuration files, and certain settings.

Compiler Compile-Time Environment Variables

The following table shows the compile-time environment variables that affect the compiler:

Compile-Time Environment Variable	Description
IFORTCFG	Specifies a configuration file that the compiler should use instead of the default configuration file. By default, the compiler uses the default configuration file (ifort.cfg) from the same directory where the compiler executable resides. **NOTE** On Windows*, this environment variable cannot be set from Visual Studio.
INTEL_LICENSE_FILE	Specifies the location for the Intel license file. **NOTE** On Windows*, this environment variable cannot be set from Visual Studio.
__INTEL_PRE_FFLAGS __INTEL_POST_FFLAGS	Specifies a set of compiler options to add to the compile line. This is an extension to the facility already provided in the compiler configuration file ifort.cfg. **NOTE** By default, a configuration file named (Windows*), icc.cfg (Linux*, macOS*), or icpc.cfg (Linux*, macOS*) ifort.cfg is used. This file is in the same directory as the compiler executable. To use another configuration file in another location, you can use the IFORTCFG environment variable to assign the directory and file name for the configuration file.

Compile-Time Environment Variable	Description
	You can insert command line options in the prefix position using `__INTEL_PRE_FFLAGS`, or in the suffix position using `__INTEL_POST_FFLAGS`. The command line is built as follows: **Syntax:** `ifort` *`<PRE flags> <flags from configuration file> <flags from the compiler invocation> <POST flags>`* **NOTE** The driver issues a warning that the compiler is overriding an option because of an environment variable, but only when you include the option `/W5` (Windows*) or `-w3` (Linux* and macOS*).
`INTEL_TARGET_ARCH_IA32` (Linux* and Windows*)	Set this environment variable to target 32-bit compilations for all associated tools (this includes the compiler and Intel-specific linker tools). Without this environment variable, you will be required to use the explicit command line options, `/Qm32` on Windows* and `-m32` on Linux*, for each compiler invocation. **NOTE** IA-32 architecture is no longer supported on macOS*.
`PATH`	Specifies the directories the system searches for binary executable files. **NOTE** On Windows*, this also affects the search for Dynamic Link Libraries (DLLs).

`TMP`
`TMPDIR`
`TEMP`
	Specifies the location for temporary files. If none of these are specified, or writeable, or found, the compiler stores temporary files in `/tmp` (Linux, macOS*) or the current directory (Windows).
	The compiler searches for these variables in the following order: `TMP`, `TMPDIR`, and `TEMP`.
	NOTE
	On Windows*, these environment variables cannot be set from Visual Studio.

`LD_LIBRARY_PATH` (Linux*)	Specifies the location for shared objects (.so files).
`DYLD_LIBRARY_PATH` (macOS*)	Specifies the path for dynamic libraries.
`INCLUDE` (Windows*)	Specifies the directory path for the include files (files included by `INCLUDE` statements, `#include` files, `RC INCLUDE` files[1], and module files referenced by `USE` statements). [1] Files from the Resource Compiler, used to create dialog boxes and other Windows-GUI interfaces.

Compile-Time Environment Variable	Description
LIB (Windows*)	Specifies the directories for all libraries used by the compiler and linker.
GNU Environment Variables and Extensions	
CPATH (Linux* and macOS*)	Specifies the path for include and module files.
GCCROOT (Linux*)	Specifies the location of the gcc* binaries. Set this variable only when the compiler cannot locate the gcc binaries when using the -gcc-name option.
GXX_INCLUDE (Linux*)	Specifies the location of the gcc headers. Set this variable to specify the locations of the gcc installed files when the compiler does not find the needed values as specified by the use of -gcc-name=*directory-name*/gcc or -gxx-name=*directory-name*/g++.
GXX_ROOT (Linux*)	Specifies the location of the gcc binaries. Set this variable to specify the locations of the gcc installed files when the compiler does not find the needed values as specified by the use of -gcc-name=*directory-name*/gcc or -gxx-name=*directory-name*/g++.
LIBRARY_PATH (Linux* and macOS*)	Specifies the path for libraries to be used during the link phase.

NOTE
INTEL_ROOT is an environment variable that is reserved for the Intel compiler. It's use is not supported.

Compiler Run-Time Environment Variables

The Intel® Fortran Compiler run-time system recognizes a number of environment variables. These variables can be used to customize run-time diagnostic error reporting, allow program continuation under certain conditions, disable the display of certain dialog boxes under certain conditions, and allow just-in-time debugging.

Environment variables relating to file I/O are interpreted when the file is opened. Other variables are read when the program starts.

The order of precedence for run-time environment variables is:

1. OPEN keyword
2. Environment variable(s)
3. Command line option

For Fortran runtime environment variables that are boolean (either enabled or disabled), the following tables describe values you can use to enable or disable them. These rules do not apply to environment variables for OpenMP* (OMP_) and their extensions (KMP_), or PGO.

Setting a Run-Time Environment Variable to ON or OFF	Examples
An integer composed entirely of digits other than 0 enables the environment variable.	1 4938493848
A string starting with **t** or **T**, **y** or **Y**, enables the environment variable. Because you cannot include quotes, or spaces preceding the string, **" T"** and **' YES'** do not work, because both spaces and quotation marks are considered letters.	T t TRUE TFALSE Y y yes yeti
Anything that doesn't enable the environment variable disables it.	0 -1 +1 <no value set>

The following table summarizes compiler environment variables that are recognized at run time.

Run-Time Environment Variable	Description
`F_UFMTENDIAN`	This variable specifies the numbers of the units to be used for little-endian-to-big-endian conversion purposes. See Environment Variable F_UFMTENDIAN Method. The variable is retrieved once when the first unit is opened, and then checked for every open.
`FOR_COARRAY_CONFIG_FILE` (Windows* and Linux*)	This variable specifies the coarray configuration file and path to be used at execution time. The variable overrides the value specified by the `[Q]coarray-config-file=`*value* qualifier at runtime.
`FOR_COARRAY_DEBUG_STARTUP` (Windows* and Linux*)	Boolean. When set to TRUE, this variable tells the Fortran run-time library to display the Message Passing Interface (MPI) launcher command that begins the underlying parallel support for coarrays.
`FOR_COARRAY_MPI_VERBOSE` (Windows* and Linux*)	Boolean. When set to TRUE, this variable tells the Fortran run-time library to pass the '-verbose' qualifier to the MPI support underlying the coarray implementation so that MPI will issue status and activity messages.

Run-Time Environment Variable	Description
FOR_COARRAY_NUM_IMAGES (Windows* and Linux*)	This variable specifies the number of images created for coarrays. The value overrides the value specified by the [Q]coarray-num-images=*value* qualifier at runtime. If neither is specified, the number of logical processors is used.
FOR_DUMP_CORE_FILE	Boolean. This variable must be lowercase for Linux. When set to TRUE, a core dump will be taken when any severe Intel® Fortran run-time error occurs. **Default:** FALSE decfort_dump_flag is an alternate spelling for FOR_DUMP_CORE_FILE
FOR_FASTMEM_NORETRY	Boolean. When set to TRUE, this variable specifies that if an allocation from FASTMEM fails because either the libmemkind library is not linked into the executable or HBW memory is not available on the node then the run-time will report the failure to the user via STAT= or ERRMSG= or aborting the program with the appropriate error message. The variable is retrieved once at program initialization, and checked for each FASTMEM memory allocation. **Default:** TRUE
FOR_FASTMEM_RETRY	Boolean. When set to TRUE, this variable specifies that if an allocation from FASTMEM fails because either the libmemkind library is not linked into the executable or HBW memory is not available on the node then memory will be allocated from the default memory allocator for that platform. The variable is retrieved once at program initialization, and checked for each FASTMEM memory allocation. **Default:** FALSE
FOR_FASTMEM_RETRY_WARN	Boolean. When set to TRUE, this variable specifies that if an allocation from FASTMEM fails because either the libmemkind library is not linked into the executable or HBW memory is not available on the node then a warning message will be issued to stdout and memory will be allocated from the default memory allocator for that platform.

Run-Time Environment Variable	Description
	The variable is retrieved once at program initialization, and checked for each `FASTMEM` memory allocation. **Default:** FALSE
`FOR_FMT_TERMINATOR`	This variable specifies the numbers of the units to have a specific record terminator. See Record Types. The variable is retrieved once when the first unit is opened, and then checked for every open.
`FORT_FMT_NO_WRAP_MARGIN`	Boolean. When set to TRUE, disables column wrapping in Fortran list-directed output when the record being written is longer than 80 characters. **NOTE** There is no environment variable to set a value for the right margin, this only disables wrapping. The RECL= open specifier can be used to set the right margin when the unit is opened. **Default:** FALSE (Wrap margin)
`FOR_ACCEPT`	The `ACCEPT` statement does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the `FOR_ACCEPT` environment variable. If `FOR_ACCEPT` is *not* defined, the code `ACCEPT f,iolist` reads from standard input. If `FOR_ACCEPT` is defined (as a file name optionally containing a path), the specified file would be read.
`FOR_DEBUGGER_IS_PRESENT`	Boolean. When set to TRUE, this variable tells the Fortran run-time library that your program is executing under a debugger, and generates debug exceptions whenever severe or continuable errors are detected. Under normal conditions, this variable can be extracted from the operating system for Windows*.

Run-Time Environment Variable	Description
	User-convenience feature If the user sets this environment variable to a TRUE value (such as 1) on Windows and the user's program executes an ERROR STOP statement, the Windows system function `__debugbreak` will be called and will bring up the Windows debugger (if it is really present). The user can use the debugger to look at the state of the application, and any step or continue command will continue the ERROR STOP process and terminate the application. On Linux* and macOS*, this variable must be set for debug exceptions. Setting this variable to TRUE when a program is *not* executing under a debugger causes unpredictable behavior. **Default:** FALSE
`FOR_DEFAULT_PRINT_DEVICE` (Windows*)	This variable lets you specify the print device other than the default print device PRN (LPT1) for files closed (`CLOSE` statement) with the `DISPOSE='PRINT'` specifier. To specify a different print device for the file associated with the `CLOSE` statement `DISPOSE='PRINT'` specifier, set `FOR_DEFAULT_PRINT_DEVICE` to any legal spooled print device before executing the program.
`FOR_DIAGNOSTIC_LOG_FILE`	If this variable is set to the name of a file, diagnostic output is written to the specified file. The Fortran run-time system attempts to open that file (append output) and write the error information (ASCII text) to the file. Because the setting of `FOR_DIAGNOSTIC_LOG_FILE` is independent of `FOR_DISABLE_DIAGNOSTIC_DISPLAY`, you can disable the screen display of information but still capture the error information in a file. Because the text string you assign for the file name is used literally, you must specify the full name. If the file open fails, no error is reported and the run-time system continues diagnostic processing. See also Locating Run-Time Errors and Traceback.
`FOR_DISABLE_DIAGNOSTIC_DISPLAY`	Boolean. When set to TRUE, this variable disables the display of all error information. This variable is helpful if you just want to test the error status of your

Run-Time Environment Variable	Description
	program and do not want the Fortran run-time system to display any information about an abnormal program termination. See also Traceback.
FOR_DISABLE_KMP_MALLOC	Boolean. When set to TRUE, this variable forces Fortran allocate statements to resolve to glibc malloc. **Default:** FALSE
FOR_FORCE_STACK_TRACE	Boolean. When set to TRUE, this variable forces a traceback to follow any run-time diagnostic message. If FOR_DISABLE_STACK_TRACE is also set, FOR_FORCE_STACK_TRACE takes precedence over FOR_DISABLE_STACK_TRACE . **Default:** FALSE
FOR_DISABLE_STACK_TRACE	Boolean. When set to TRUE, this variable disables the call stack trace information that typically follows the displayed severe error message text. The Fortran run-time error message is displayed regardless of whether FOR_DISABLE_STACK_TRACE is set to TRUE. If the program is executing under a debugger, the automatic output of the stack trace information by the Fortran library will be disabled to reduce noise. Use the debugger's stack trace facility to view the stack trace. **Default:** FALSE See also Locating Run-Time Errors and Traceback.
FOR_IGNORE_EXCEPTIONS	Boolean. When set to TRUE, this variable disables default run-time exception handling to allow, for example, just-in-time debugging. The run-time system exception handler returns EXCEPTION_CONTINUE_SEARCH to the operating system, which looks for other handlers to service the exception. **Default:** FALSE
FOR_NOERROR_DIALOGS	Boolean. When set to TRUE, this variable disables the display of dialog boxes when certain exceptions or errors occur. This is useful when running many test programs in batch mode to prevent a failure from stopping execution of the entire test stream.

Run-Time Environment Variable	Description
	Default: FALSE
FOR_PRINT	Neither the PRINT statement nor a WRITE statement with an asterisk (*) in place of a unit number includes an explicit logical unit number. Instead, both use an implicit internal logical unit number and the FOR_PRINT environment variable. If FOR_PRINT is *not* defined, the code PRINT f,iolist or WRITE (*,f) iolist writes to standard output. If FOR_PRINT is defined (as a file name optionally containing a path), the specified file would be written to.
FOR_READ	A READ statement that uses an asterisk (*) in place of a unit number does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the FOR_READ environment variable. If FOR_READ is *not* defined, the code READ (*,f) iolist or READ f,iolist reads from standard input. If FOR_READ is defined (as a file name optionally containing a path), the specified file would be read.
FOR_TYPE	The TYPE statement does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the FOR_TYPE environment variable. If FOR_TYPE is *not* defined, the code TYPE f,iolist writes to standard output. If FOR_TYPE is defined (as a file name optionally containing a path), the specified file would be written to.
FORT_BLOCKSIZE	Specifies the default BLOCKSIZE value to be used when BLOCKSIZE= is omitted on the OPEN statement. Valid values are 0 to 2147467264. Sizes are rounded up to the next 512-byte boundary. This variable applies to all Fortran I/O units except stderr, which is never buffered. The variable is retrieved once at program initialization, and checked for each unit open.
FORT_BUFFERCOUNT	Specifies the default BUFFERCOUNT value to be used when BUFFERCOUNT= is omitted on the OPEN statement. Valid values are 0 to 127. If set to 0, the default value of 1 is used. This variable applies to all Fortran I/O units except stdout (units ***** and **6**) and stderr. The variable is retrieved once at program initialization, and checked for each unit open.
FORT_BUFFERED	Boolean.

Run-Time Environment Variable	Description
	When set to TRUE, this variable specifies that buffered I/O should be used at run time for input and output on all Fortran I/O units, except stdout (units ***** and **6**). Output to stderr is never buffered. **Default:** FALSE
FORT_BUFFERING_THRESHOLD=n	Specifies dynamic buffering for unformatted sequential READ operations: • I/O list items with a size <=*n* are buffered and are moved one at a time from the buffer to the I/O list item. • I/O list items with a size >*n* are not buffered and are moved one at a time from the file to the I/O list item.
FORT_CONVERT*n*	Specifies the data format for an unformatted file associated with a particular unit number (*n*), as described in Methods of Specifying the Data Format.
FORT_CONVERT.*ext* and FORT_CONVERT_*ext*	Specifies the data format for unformatted files with a particular file extension suffix (.*ext*), as described in Methods of Specifying the Data Format.
FORT_FMT_RECL	Specifies the default record length (normally 132 bytes) for formatted files. The variable is retrieved once at program initialization, and checked for each unit open.
FORT_UFMT_RECL	Specifies the default record length (normally 2040 bytes) for unformatted files. The variable is retrieved once at program initialization, and checked for each unit open.
FORT*n*	Specifies the file name for a particular unit number *n*, when a file name is not specified in the OPEN statement or an implicit OPEN is used, and the compiler option fpscomp with option keyword filesfromcmd was not specified. Preconnected files attached to units 0, 5, and 6 are associated with system standard I/O files by default.
INTEL_CHKP_REPORT_MODE (Linux*)	Changes the pointer checker reporting mode at runtime.
INTEL_ISA_DISABLE	Causes named features (in a comma-separated list) not to be visible on the host even if the CPUID reports that it has them onboard.
NLSPATH (Linux* and macOS*)	Specifies the path for the Intel® Fortran run-time error message catalog.

Run-Time Environment Variable	Description
TBK_ENABLE_VERBOSE_STACK_TRACE	Boolean. When set to TRUE, traceback output displays more detailed call stack information in the event of an error. The default brief output is usually sufficient to determine where an error occurred. Brief output includes up to twenty stack frames, reported one line per stack frame. For each frame, the image name containing the PC, routine name, line number, and source file are given. The verbose output, if selected, will provide the exception context record (in addition to the information in brief output) if the error was a machine exception (machine register dump), and for each frame, the return address, frame pointer and stack pointer and possible parameters to the routine. This output can be quite long (limited to 16K bytes) and use of the environment variable FOR_DIAGNOSTIC_LOG_FILE is recommended if you want to capture the output accurately. Most situations should not require the use of verbose output. **Default:** FALSE See also Locating Run-Time Errors and Traceback.
TBK_FULL_SRC_FILE_SPEC	Boolean. When set to TRUE, traceback output displays complete file name information including the path. By default, the traceback output displays only the file name and extension in the source file field. **Default:** FALSE See also Locating Run-Time Errors and Traceback.
FORT_TMPDIR TMP TMPDIR TEMP	Specifies an alternate working directory where scratch files are created.
GNU extensions (recognized by the Intel OpenMP* compatibility library)	
GOMP_CPU_AFFINITY (Linux*)	GNU extension recognized by the intel OpenMP* compatibility library. Specifies a list of OS processor IDs. You must set this environment variable before the first parallel region or before certain API calls including omp_get_max_threads(),

Run-Time Environment Variable	Description
	`omp_get_num_procs()` and any affinity API calls. For detailed information on this environment variable, see *Thread Affinity Interface*. **Default:** Affinity is disabled
`GOMP_STACKSIZE` (Linux*)	GNU extension recognized by the Intel OpenMP compatibility library. Same as `OMP_STACKSIZE`.`KMP_STACKSIZE` overrides `GOMP_STACKSIZE`, which overrides `OMP_STACKSIZE`. **Default:** See the description for `OMP_STACKSIZE`.
OpenMP* Environment Variables (OMP_) and Extensions (KMP_)	
`OMP_CANCELLATION`	Activates cancellation of the innermost enclosing region of the type specified. If set to `TRUE`, the effects of the cancel construct and of cancellation points are enabled and cancellation is activated. If set to `FALSE`, cancellation is disabled and the cancel construct and cancellation points are effectively ignored. **NOTE** Internal barrier code will work differently depending on whether the cancellation is enabled. Barrier code should repeatedly check the global flag to figure out if the cancellation had been triggered. If a thread observes the cancellation it should leave the barrier prematurely with the return value 1 (may wake up other threads). Otherwise, it should leave the barrier with the return value 0. Enables (`TRUE`) or disables (`FALSE`) cancellation of the innermost enclosing region of the type specified. **Default:** FALSE **Example:** `OMP_CANCELLATION=TRUE`
`OMP_DISPLAY_ENV`	Enables (`TRUE`) or disables (`FALSE`) the printing to stderr of the OpenMP version number and the values associated with the OpenMP environment variable. Possible values are: TRUE, FALSE, or VERBOSE. **Default:** FALSE **Example:** `OMP_DISPLAY_ENV=TRUE`

Run-Time Environment Variable	Description
OMP_DEFAULT_DEVICE	Sets the device that will be used in a target region. The OpenMP routine omp_set_default_device or a device clause in a parallel directive can override this variable. If no device with the specified device number exists, the code is executed on the host. If this environment variable is not set, device number 0 is used.
OMP_DYNAMIC	Enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads. **Default:** FALSE **Example:** OMP_DYNAMIC=TRUE
OMP_MAX_ACTIVE_LEVELS	The maximum number of levels of parallel nesting for the program. **Default:** 1 **Syntax:** OMP_MAX_ACTIVE_LEVELS=TRUE
OMP_NESTED	Enables (TRUE) or disables (FALSE) nested parallelism. **Default:** FALSE **Example:** OMP_NESTED=TRUE
OMP_NUM_THREADS	Sets the maximum number of threads to use for OpenMP* parallel regions if no other value is specified in the application. The value can be a single integer, in which case it specifies the number of threads for all parallel regions. The value can also be a comma-separated list of integers, in which case each integer specifies the number of threads for a parallel region at a nesting level. The first position in the list represents the outer-most parallel nesting level, the second position represents the next-inner parallel nesting level, and so on. At any level, the integer can be left out of the list. If the first integer in a list is left out, it implies the normal default value for threads is used at the outer-most level. If the integer is left out of any other level, the number of threads for that level is inherited from the previous level. This environment variable applies to the options Qopenmp (Windows) or qopenmp (Linux and macOS*), and Qparallel (Windows) or parallel (Linux and macOS*) . **Default:** The number of processors visible to the operating system

Run-Time Environment Variable	Description
	Syntax: `OMP_NUM_THREADS=value[,value]*`

`OMP_PLACES`	Specifies an explicit ordered list of places, either as an abstract name describing a set of places or as an explicit list of places described by nonnegative numbers. An exclusion operator "!" can also be used to exclude the number or place immediately following the operator.
	
	For **explicit lists**, the meaning of the numbers and how the numbering is done for a list of nonnegative numbers are implementation defined. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread.
	
	Intervals can be specified using the `<lower-bound>` : `<length>` : `<stride>` notation to represent the following list of numbers:
	
	`"<lower-bound>, <lower-bound> + <stride>, ..., <lower-bound> +(<length>-1)*<stride>."`
	
	When `<stride>` is omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences of places.
	
	`# EXPLICIT LIST EXAMPLE`
	`setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"`
	`setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"`
	`setenv OMP_PLACES "{0:4}:4:4"`
	
	The **abstract names** listed below should be understood by the execution and runtime environment:
	
	• `threads`: Each place corresponds to a single hardware thread on the target machine.
	• `cores`: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
	• `sockets`: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.
	
	When requesting fewer places or more resources than available on the system, the determination of which resources of type `abstract_name` are to be included in the place list is implementation-defined. The precise definitions of the abstract names are implementation defined. An implementation may also add abstract names as appropriate for the target platform. The abstract name may be

Run-Time Environment Variable	Description

	appended by a positive number in parentheses to denote the length of the place list to be created, that is `abstract_name(num-places)`.
	`# ABSTRACT NAMES EXAMPLE`
	`setenv OMP_PLACES threads`
	`setenv OMP_PLACES threads(4)`
	NOTE
	If any numerical values cannot be mapped to a processor on the target platform the behavior is implementation-defined. The behavior is also implementation-defined when the `OMP_PLACES` environment variable is defined using an abstract name.

`OMP_PROC_BIND` (Windows, Linux)	Sets the thread affinity policy to be used for parallel regions at the corresponding nested level. Enables (`TRUE`) or disables (`FALSE`) the binding of threads to processor contexts. If enabled, this is the same as specifying `KMP_AFFINITY=scatter`. If disabled, this is the same as specifying `KMP_AFFINITY=none`. **Acceptable values:** `TRUE`, `FALSE`, or a comma separated list, each element of which is one of the following values: `MASTER`, `CLOSE`, `SPREAD`. **Default:** `FALSE` If set to `FALSE`, the execution environment may move OpenMP* threads between OpenMP* places, thread affinity is disabled, and proc_bind clauses on parallel constructs are ignored. Otherwise, the execution environment should not move OpenMP* threads between OpenMP* places, thread affinity is enabled, and the initial thread is bound to the first place in the OpenMP* place list. If set to `MASTER`, all threads are bound to the same place as the master thread. If set to `CLOSE`, threads are bound to successive places, close to where the master thread is bound. If set to `SPREAD`, the master thread's partition is subdivided and threads are bound to single place successive sub-partitions. **NOTE** `KMP_AFFINITY` takes precedence over `GOMP_CPU_AFFINITY` and `OMP_PROC_BIND`. `GOMP_CPU_AFFINITY` takes precedence over `OMP_PROC_BIND`.

Run-Time Environment Variable	Description
OMP_SCHEDULE	Sets the run-time schedule type and an optional chunk size. **Default:** STATIC, no chunk size specified **Example syntax:** OMP_SCHEDULE="kind[,chunk_size]" **NOTE** Some environment variables are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.
OMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP* thread to use as the private stack for the thread. Recommended size is 16M. Use the optional suffixes to specify byte units: B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be K (Kilobytes). This variable does not affect the native operating system threads created by the user program, or the thread executing the sequential part of an OpenMP* program or parallel programs created using the option Qparallel (Windows) or parallel (Linux and macOS*) . The kmp_{set,get}_stacksize_s() routines set/retrieve the value. The kmp_set_stacksize_s() routine must be called from sequential part, before first parallel region is created. Otherwise, calling kmp_set_stacksize_s() has no effect. **Default (IA-32 architecture):** 2M **Default (Intel® 64 architecture):** 4M **NOTE** IA-32 architecture is no longer supported on macOS*. **Related environment variables:** KMP_STACKSIZE (overrides OMP_STACKSIZE). **Syntax:** OMP_STACKSIZE=value
OMP_THREAD_LIMIT	Limits the number of simultaneously-executing threads in an OpenMP* program.

Run-Time Environment Variable	Description
	If this limit is reached and another native operating system thread encounters OpenMP* API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP* parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following options: `Qopenmp` (Windows) or `qopenmp` (Linux and macOS*), or `Qparallel` (Windows) or `parallel` (Linux and macOS*) . The `omp_get_thread_limit()` routine returns the value of the limit. **Default:** No enforced limit **Related environment variable:** `KMP_ALL_THREADS` (overrides `OMP_THREAD_LIMIT`). **Example syntax:** `OMP_THREAD_LIMIT=value`
`OMP_WAIT_POLICY`	Decides whether threads spin (active) or yield (passive) while they are waiting. `OMP_WAIT_POLICY=ACTIVE` is an alias for `KMP_LIBRARY=turnaround`, and `OMP_WAIT_POLICY=PASSIVE` is an alias for `KMP_LIBRARY=throughput`. **Default:** Passive **Syntax:** `OMP_WAIT_POLICY=value`
`KMP_AFFINITY` (Windows, Linux)	Enables run-time library to bind threads to physical processing units. You must set this environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls. For detailed information on this environment variable, see *Thread Affinity Interface*. **Default:** noverbose,warnings,respect,granularity=core,none Default (Windows* with multiple processor groups): noverbose,warnings,norespect,granularity=group,compact,0,0

Run-Time Environment Variable	Description
	NOTE On Windows* with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows*). Otherwise, the respect affinity modifier is used.
KMP_ALL_THREADS	Limits the number of simultaneously-executing threads in an OpenMP* program. If this limit is reached and another native operating system thread encounters OpenMP* API calls or constructs, then the program may abort with an error message. If this limit is reached at the time an OpenMP* parallel region begins, a one-time warning message may be generated indicating that the number of threads in the team was reduced, but the program will continue execution. This environment variable is only used for programs compiled with the Qopenmp(Windows) or qopenmp (Linux and macOS*) option. **Default:** No enforced limit.
KMP_BLOCKTIME	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. Use the optional character suffixes: s (seconds), m (minutes), h (hours), or d (days) to specify the units. Specify infinite for an unlimited wait time. **Default:** 200 milliseconds **Related Environment Variable:** KMP_LIBRARY environment variable.
KMP_CPUINFO_FILE	Specifies an alternate file name for a file containing the machine topology description. The file must be in the same format as /proc/cpuinfo. **Default:** None
KMP_DETERMINISTIC_REDUCTION	Enables (TRUE) or disables (FALSE) the use of a specific ordering of the reduction operations for implementing the reduction clause for an OpenMP* parallel region. This has the effect that, for a given number of threads, in a given parallel region, for a given data set and reduction operation, a floating point reduction done for an OpenMP* reduction clause has a consistent floating point result from run to run, since round-off errors are identical. **Default:** FALSE

Run-Time Environment Variable	Description
KMP_DYNAMIC_MODE	Selects the method used to determine the number of threads to use for a parallel region when OMP_DYNAMIC=TRUE. Possible values: (asat | load_balance | thread_limit), where, • asat: estimates number of threads based on parallel start time; **NOTE** Support for asat (automatic self-allocating threads) is now deprecated and will be removed in a future release. • load_balance: tries to avoid using more threads than available execution units on the machine; • thread_limit: tries to avoid using more threads than total execution units on the machine. **Default (IA-32 architecture):** load_balance (on all supported OSes) **Default (Intel® 64 architecture):** load_balance (on all supported OSes) **NOTE** IA-32 architecture is no longer supported on macOS*.
KMP_HOT_TEAMS_MAX_LEVEL	Sets the maximum nested level to which teams of threads will be hot. **NOTE** A *hot* team is a team of threads optimized for faster reuse by subsequent parallel regions. In a hot team, threads are kept ready for execution of the next parallel region, in contrast to the cold team, which is freed after each parallel region, with its threads going into a common pool of threads. For values of 2 and above, nested parallelism should be enabled. **Default:** 1
KMP_HOT_TEAMS_MODE	Specifies the run-time behavior when the number of threads in a hot team is reduced. Possible values: • 0: Extra threads are freed and put into a common pool of threads.

Run-Time Environment Variable	Description
	• 1: Extra threads are kept in the team in reserve, for faster reuse in subsequent parallel regions. **Default:** 0
KMP_HW_SUBSET	Specifies the number of sockets, cores per socket, and the number of threads per core, to use with an OpenMP* application, as an alternative to writing explicit affinity settings or a process affinity mask. You can also specify an offset value to set which resources to use. An extended syntax is available when KMP_TOPOLOGY_METHOD=hwloc. Depending on what resources are detected, you may be able to specify additional resources, such as NUMA nodes and groups of hardware resources that share certain cache levels. **Basic syntax:** *sockets*S[@*offset*],*cores*C[@*offset*],*threads*T S, C and T are not case-sensitive. *sockets* — The number of sockets to use. *cores* — The number of cores to use per socket. *threads* — The number of threads to use per core. *offset* — (Optional) The number of sockets or cores to skip. **Extended syntax when KMP_TOPOLOGY_METHOD=hwloc:** *sockets*S[@*offset*],*numas*N[@*offset*],*tiles*L2[@*offset*],*cores*C[@*offset*],*threads*T S, N, L2, C and T are not case-sensitive. Some designators are aliases on some machines. Specifying duplicate or multiple alias designators for the same resource type is not allowed. *sockets* — The number of sockets to use. *numas* — If detectable, the number of NUMA nodes to use per socket,where available. *tiles* — If detectable, the number of tiles to use per NUMA node, where available, otherwise per socket. *cores* — The number of cores to use per socket, where available, otherwise per NUMA node, or per socket.

Run-Time Environment Variable	Description
	threads The number of threads to use per core. *offset* (Optional) The number of sockets or cores to skip. **NOTE** If you don't specify one or more types of resource, sockets, cores or threads, all available resources of that type are used. **NOTE** If a particular type of resource is specified, but detection of that resource is not supported by the chosen topology detection method, the setting of `KMP_HW_SUBSET` is ignored. **NOTE** This variable does not work if the OpenMP* affinity is set to `disabled`. **Default:** If omitted, the default value is to use all the available hardware resources. **Examples:** • `2s,4c,2t`: Use the first 2 sockets (s0 and s1), the first 4 cores on each socket (c0 - c3), and 2 threads per core. • `2s@2,4c@8,2t`: Skip the first 2 sockets (s0 and s1) and use 2 sockets (s2-s3), skip the first 8 cores (c0-c7) and use 4 cores on each socket (c8-c11), and use 2 threads per core. • `5C@1,3T`: Use all available sockets, skip the first core and use 5 cores, and use 3 threads per core. • `2T`: Use all cores on all sockets, 2 threads per core. • `4C@12`: Use 4 cores with offset 12, all available threads per core.
`KMP_INHERIT_FP_CONTROL`	Enables (`TRUE`) or disables (`FALSE`) the copying of the floating-point control settings of the master thread to the floating-point control settings of the OpenMP* worker threads at the start of each parallel region. **Default:** `TRUE`

Run-Time Environment Variable	Description
KMP_LIBRARY	Selects the OpenMP* run-time library execution mode. The values for this variable are `serial`, `turnaround`, or `throughput`. **Default:** `throughput`
KMP_PLACE_THREADS	Deprecated; use KMP_HW_SUBSET instead.
KMP_SETTINGS	Enables (`TRUE`) or disables (`FALSE`) the printing of OpenMP* run-time library environment variables during program execution. Two lists of variables are printed: user-defined environment variables settings and effective values of variables used by OpenMP* run-time library. **Default:** `FALSE`
KMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP* thread to use as its private stack. Recommended size is 16m. Use the optional suffixes to specify byte units: `B` (bytes), `K` (Kilobytes), `M` (Megabytes), `G` (Gigabytes), or `T` (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be `K` (Kilobytes). This variable does not affect the native operating system threads created by the user program nor the thread executing the sequential part of an OpenMP* program or parallel programs created using the option `Qparallel` (Windows) or `parallel` (Linux and macOS*) . `KMP_STACKSIZE` overrides `GOMP_STACKSIZE`, which overrides `OMP_STACKSIZE`. **Default (IA-32 architecture):** 2m **Default (Intel® 64 architecture):** 4m **NOTE** IA-32 architecture is no longer supported on macOS*.
KMP_TOPOLOGY_METHOD	Forces OpenMP* to use a particular machine topology modeling method. Possible values are: • `cpuid_leaf11` Decodes the APIC identifiers as specfied by leaf 11 of the *cpuid* instruction. • `cpuid_leaf4`

Run-Time Environment Variable	Description
	Decodes the APIC identifiers as specified in leaf 4 of the *cpuid* instruction. • `cpuinfo` If `KMP_CPUINFO_FILE` is not specified, forces OpenMP* to parse `/proc/cpuinfo` to determine the topology (Linux* only). If `KMP_CPUINFO_FILE` is specified as described above, uses it (Windows* or Linux*). • `group` Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows* 64-bit only) . • `flat` Models the machine as a flat (linear) list of processors. • `hwloc` Models the machine as the Portable Hardware Locality* (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa nodes, packages, cores, hardware threads, caches, and Windows* processor groups.
`KMP_VERSION`	Enables (`TRUE`) or disables (`FALSE`) the printing of OpenMP* run-time library version information during program execution. **Default:** `FALSE`
`KMP_WARNINGS`	Enables (`TRUE`) or disables (`FALSE`) displaying warnings from the OpenMP* run-time library during program execution. **Default:** `TRUE`
Profile Guided Optimization (PGO_) Environment Variables	
`INTEL_PROF_DUMP_CUMULATIVE`	When using interval profile dumping (initiated by `INTEL_PROF_DUMP_INTERVAL` or the function `_PGOPTI_Set_Interval_Prof_Dump`) during the execution of an instrumented user application, allows creation of a single `.dyn` file to contain profiling information instead of multiple `.dyn` files. If not set, executing an instrumented user application creates a new `.dyn` file for each interval.

Run-Time Environment Variable	Description
	Setting this environment variable is useful for applications that do not terminate or those that terminate abnormally (bypass the normal exit code).
INTEL_PROF_DUMP_INTERVAL	Initiates interval profile dumping in an instrumented user application. This environment variable may be used to initiate Interval Profile Dumping in an instrumented application.
INTEL_PROF_DYN_PREFIX	Specifies the prefix to be used for the .dyn filename to distinguish it from the other .dyn files dumped by other PGO runs. Executing the instrumented application generates a .dyn filename as follows: <prefix>_<timestamp>_<pid>.dyn, where <prefix> is the identifier that you have specified. **NOTE** The value specified in this environment variable must not contain < > : " / \ | ? * characters. The default naming scheme is used if an invalid prefix is specified.
PROF_DIR	Specifies the directory where profiling files (files with extensions .dyn, .dpi, .spi and so on) are stored. The default is to store the .dyn files in the source directory of the file containing the first executed instrumented routine in the binary compiled with [Q]prof-gen option. This variable applies to all three phases of the profiling process: • Instrumentation compilation and linking • Instrumented execution • Feedback compilation
PROF_DPI	Name for the .dpi file. **Default:** pgopti.dpi
PROF_DUMP_INTERVAL	Deprecated; use INTEL_PROF_DUMP_INTERVAL instead.
PROF_NO_CLOBBER	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges data from all dynamic information files and creates a new pgopti.dpi file if the .dyn files are newer than an existing pgopti.dpi file.

Run-Time Environment Variable	Description
	When this variable is set, the compiler does not overwrite the existing `pgopti.dpi` file. Instead, the compiler issues a warning. You must remove the `pgopti.dpi` file if you want to use additional dynamic information files.

See Also

Qopenmp compiler option
parallel, Qparallel compiler option
prof-gen, Qprof-gen compiler option
Thread Affinity Interface

See Also

Temporary Files Created by the Compiler or Linker
Traceback
Methods of Specifying the Data Format
Locating Run-Time Errors
Understanding Run-Time Errors
Environment Variable F_UFMTENDIAN Method
Record Types

Using Other Methods to Set Environment Variables

All Operating Systems

From within a program: Call the `SETENVQQ` routine to set environment variables. For example:

```
USE IFPORT
LOGICAL success
success = SETENVQQ("FORT9=C:\mydir\test.dat")
```

Depending on your operating system, there are additional ways to set environment variables.

Linux* and macOS*

Within the Bourne*/Bourne* Again shell (`sh`/`bash`), or the Korn shell (`ksh`), use the `export` command and assignment command to set an environment variable:

```
export FORT9
FORT9=/usr/users/smith/test.dat
```

To remove the association of an environment variable and its value within the Bourne*, Korn shell, or bash shells, use the `unset` command:

```
unset FORT9
```

Within the C shell (`csh`), use the `setenv` command to set an environment variable:

```
setenv FORT9 /usr/users/smith/test.dat
```

To remove the association of an environment variable and its value within the C shell, use the `unsetenv` command.

```
unsetenv FORT9
```

Windows*

Environment variables in a command prompt session can be set using the SET command. Changes here will affect programs run from that command prompt only. For example:

```
set FORT9=C:\mydir\test.dat
```

To remove an environment variable, omit the value. For example:

```
set FORT9=
```

To display the current value, use the ECHO command and enclose the environment variable name in the percent signs as shown below:

```
echo %FORT9%
```

To display the values of all variables whose names begin with a string, use SET and omit the equal sign as shown below:

```
set FORT
```

In Microsoft Visual Studio you can specify environment variables that will be set when a program is run. In the project properties, edit the Configuration Properties > Debugging > Environment property and set the field to a series of name=value pairs separated by semicolons. For example:

```
FORT9=C:\mydir\test.dat;FOR_FORCE_STACK_TRACE=TRUE
```

A default set of environment variables is established per user and at the system level. These can be specified through Control Panel > System > Advanced System Settings > Environment Variables.

> **NOTE**
> Changing system-wide environment variables affects command line builds (those done without IDE involvement), but not builds done through the IDE. IDE builds are managed in the IDE using **Tools** > **Options**. An exception to this is an IDE build (`devenv`) done from the command line that specifies the `useenv` option. In this case, the IDE uses the `PATH`, `INCLUDE`, and `LIB` environment variables defined for that command line.

See Also

Specifying Path Library and Include Directories
Supported Environment Variables

Understanding Files Associated with Intel® Fortran Applications (Windows)*

There are a number of files associated with Intel® Fortran Compiler applications. The following table shows common files used by the Intel® Fortran Compiler and lists their contents.

File Extension	Type	Contents
.asm	Intermediate	Assembly file, passed to the assembler
.exe	Output	Executable, dynamic- link library, or library files
.dll		
.lib		
.fi	Source	Header files
.fd		
.for	Source	Fortran source files (fixed format)
.f		
.fpp		
.f90	Source	Fortran source file (free format)
.def	Source	Linker
.idl	Source	Microsoft IDL (non-Fortran)
.ilk	Intermediate	Incremental link file
.map	Output	Map file; output from the linker
.mod	Intermediate	Module file; created if a source file defines a Fortran module
.obj	Intermediate	Object file; passed to the linker
.pdb	Output (Debug)	Program debug database file
.tbl	Output (MIDL)	Type library; passed to Resource
.rc	Resource	Resource file (non-Fortran)
.res	Intermediate	Resource file; passed to the linker
.sln	Solution	Visual Studio* solution file and solution options file
.suo		
.vfproj	Project	Intel® Fortran, Intel® C++, and Microsoft Visual C++* project files
.icproj		
.vcproj		

You can specify additional Fortran file extensions to be recognized by Visual Studio*. For more information, see Specifying Fortran File Extensions.

See Also

Specifying Fortran File Extensions file extensions recognized by Microsoft Visual Studio*

Understanding Solutions Projects and Configurations working with Visual Studio* solutions and projects

Compiling and Linking Multithreaded Programs

When building a multithreaded application, be sure to link against the thread-safe version of the Fortran run-time libraries and to enable thread safety. This is specified by the `threads` and `reentrancy:threaded` compiler options.The Visual Studio* integrated development environment (IDE), described later in this topic, can also be used on Windows*.

You must also link with the correct library files. Some libraries, such as those for OpenMP* or coarrays, are always linked dynamically.

Linux* and macOS*

To create statically-linked, multithreaded programs, link with the static library named `libifcoremt.a`.

To use shared libraries, link your application with `libifcoremt.so` (Linux*) or `libifcoremt.dylib` (macOS*).

To compile and link your multithreaded program from the command line:

1. Make sure the *IA32ROOT* environment variable points to the directory containing your library files.
2. Compile and link the program with the `-threads` option. For example:

```
ifort -threads mythread.f90
```

> **NOTE**
> To ensure that a threadsafe and/or reentrant run-time library is linked and correctly initialized, option threads should also be used for the link step and for the compilation of the main routine.

Windows*

To create statically linked multithreaded programs, link with the re-entrant support library `LIBIFCOREMT.LIB`.

To use shared libraries, use the shared `LIBIFCOREMD.DLL` library, which is also re-entrant, and is referenced by linking your application with the `LIBIFCOREMD.LIB` import library.

Programs built with `LIBIFCOREMT.LIB` do not share Fortran run-time library code or data with any dynamic-link libraries they call. You must link with `LIBIFCOREMD.LIB` if you plan to call a DLL.

Additional Notes:

- The `/threads` option is automatically set when a multithreaded application is specified in the IDE.
- Specify the compiler options `/libs=dll` and `/threads` if you are using both multithreaded code and DLLs.
- You can use the `/libs=dll` and `/threads` options only with Fortran Console projects, not QuickWin applications.

To compile and link your multithreaded program using Visual Studio*

1. Create a new project by clicking **File** > **New** > **Project**.
2. Select the Intel® Visual Fortran project type from **Intel® Visual Fortran Projects** in the left pane.
3. Add the file containing the source code to the project.

   From the **Project** menu, select **Properties**. The **Property Pages** dialog box appears.

4. Choose the **Fortran** folder, **Libraries** category, and set the Runtime Library to **Multithreaded** or **Multithread DLL** (or their debug equivalents).
5. Create the executable file by choosing **Build Solution** from the **Build** menu.

To compile and link your multithreaded program from the command line:

1. Make sure your *LIB* environment variable points to the directory containing your library files.
2. Compile and link the program with the `/threads` option. For example:

```
ifort /threads mythread.f90
```

NOTE
To ensure that a threadsafe and/or reentrant run-time library is linked and correctly initialized, option threads should also be used for the link step and for the compilation of the main routine.

Linking Tools and Options

This topic describes how to use the Intel® linking tools, `xild` (Linux* and macOS*) and `xilink` (Windows*).

The Intel® linking tools behave differently on different platforms. The following sections summarize the primary differences between linking behavior.

Linux* and macOS* Linking Behavior Summary

The linking tool invokes the Intel® Fortran Compiler to perform IPO if objects containing `IR` (intermediate representation) are found. These are mock objects. The tool invokes GNU `ld` to link the application.

The command-line syntax for `xild` is the same as that of the GNU linker: where:

- `[<options>]`: One or more options supported only by `xild` (optional).
- `<normal command-line>`: Linker command line containing a set of valid arguments for `ld`.

To create the file `app` using IPO, use the option `o`*[filename]* as shown in the following example:

The linking tool calls the compiler to perform IPO for objects containing IR and creates a new list of object(s) to be linked. The linker then calls `ld` to link the object files that are specified in the new list and produce the application with the name specified by the `o` option. The linker supports the `ipo`[*n*] option and `ipo-separate` option.

To display a list of the supported link options from `xild`, use the following command:

Windows* Linking Behavior Summary

The linking tool invokes the Intel® Fortran Compiler to perform multi-file IPO if objects containing `IR` (intermediate representation) is found. These are mock objects. It invokes the Microsoft linker `link.exe` to link the application.

The command-line syntax for the Intel® linker is the same as that of the Microsoft linker: where:

- `[<options>]`: One or more options supported only by `xilink` (optional).
- `<normal command-line>`: Linker command line containing a set of valid arguments for the Microsoft linker.

Windows* Linking Behavior Summary
To place the multifile IPO executable in `ipo_file.exe`, use the linker option `out`*:[filename]* , for example: The linker calls the compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. The linker calls Microsoft `link.exe` to link the object files that are specified in the new list and produce the application with the name specified by the `out`*:[filename]* linker option. To display a list of support link options from `xilink` , use the following command: `xilink.exe` accepts all the options of `link.exe` and will pass them on to `link.exe` at the final linking stage.

Using the Linking Tools

You must use the Intel® linking tools to link your application if the following conditions apply:

- Your source files were compiled with multi-file IPO enabled. Multi-file IPO is enabled by specifying compiler option `[Q]ipo`.
- You would normally invoke the GNU linker (`ld`) to link your application.
- You would normally invoke the Microsoft linker (`link.exe`) to link your application.

Linker Options

The following table provides information on linking options.

Linking Tools Option	Description
`qdiag-`[*type*]=[*diag-list*]	Controls the display of diagnostic information. The *type* is an action to perform on diagnostics. Possible values are: • **Enable:** Enables a diagnostic message or a group of messages. • **Disable:** Disables a diagnostic message or a group of messages. The *diag-list* is a diagnostic group or ID value. Possible values are: • **thread:** Specifies diagnostic messages that help in thread-enabling a program. • **vec:** Specifies diagnostic messages issued by the vectorizer. • **par:** Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer). • **openmp:** Specifies diagnostic messages issued by the OpenMP* parallelizer. • **warn:** Specifies diagnostic messages that have a "warning" severity level. • **error:** Specifies diagnostic messages that have an "error" severity level. • **remark:** Specifies diagnostic messages that are remarks or comments. • **cpu-dispatch:** Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default. • **id[,id,...]:** Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each "id". • **tag[,tag,...]:** Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each "tag".

Linking Tools Option	Description
	NOTE Diagnostic messages generated by this option can be affected by other options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.
`m32` (Linux* only), `m64` (Linux* and macOS*) `Qm32`, `Qm64` (Windows*)	`[Q]m32` generates code for IA-32 architecture. Option `-m32` is only available on Linux* systems. `[Q]m64` generates code for Intel® 64 architecture. For example, when your compilation environment is configured for Intel® 64 architecture, and you use [Q]m32 with the compiler, you also need to use `qm32` on the linker command line to make sure the proper compilation target is set up for any IPO compilations or the final link.

See Also

Using IPO from the command line

Using Configuration Files

You can decrease the time you spend entering command-line options by using the configuration file to automate command-line entries. Configuration files are automatically processed every time you run the Intel® Fortran Compiler. You can insert any valid command-line options into the configuration file. The compiler processes options in the configuration file in the order in which they appear, followed by the specified command-line options when the compiler is invoked.

NOTE
Options in the configuration file are executed every time you run the compiler. If you have varying option requirements for different projects, use Using Response Files.

Sample Configuration Files

The default configuration file `ifort.cfg` is located in the same directory as the compiler executable file. If you want to use a different configuration file than the default, you can use the `IFORTCFG` environment variable to specify the location of another configuration file.

NOTE
Anytime you instruct the compiler to use a different configuration file, the default configuration file(s) are ignored.

The following examples illustrate basic configuration files. The pound (#) character indicates that the rest of the line is a comment.

In the Windows* examples, the compiler reads the configuration file and invokes the `I` option every time you run the compiler, along with any options specified on the command line.

Example

```
## Sample ifort.cfg file
## Define preprocessor macro MY_PROJECT.
  -DMY_PROJECT

## Set extended-length source lines.
  -extend_source

## Set maximum floating-point significand precision.
  -pc80
```

```
## Sample ifort.cfg file
## Define preprocessor macro MY_PROJECT
  /DMY_PROJECT

## Set extended-length source lines.
  /extend_source

## Additional directories to be searched for include
## files, before the default.
  /Ic:\project\include

## Use the static, multithreaded run-time library.
  /MT
```

See Also

Supported Environment Variables
Using Response Files

Using Response Files

You can use response files to:

- Specify options used during particular compilations or projects.
- Save this information in individual files.

Response files are invoked as options on the command line. Options in response files are inserted in the command line at the point where the response file is invoked. Unlike configuration files, which are automatically processed every time you run the compiler, response files must be invoked as an option on the command line. If you create a response file without specifying it on the command line, it will not be invoked.

Use response files to decrease the time spent entering command-line options and to ensure consistency by automating command-line entries. Use individual response files to maintain options for specific projects.

Any number of options or file names can be placed on a line in a response file. Several response files can be referenced in the same command line. The following example shows how to specify a response file on the command line:

```
ifort @ [@responsefile2 ... ]
```

NOTE
An "at" sign (@) must precede the name of the response file on the command line.

See Also
Using Configuration Files

Creating Fortran Executables

The simplest way to build an application is to compile all of your Fortran source files, then link the resulting object files into a single executable. You can build single-file executables using the `ifort` command. You can also use the Visual Studio* IDE on Windows*.

The executable file you build with this method contains all of the code needed to execute the program, including the run-time library. Because the program resides in a single file, it is easy to copy or install. The project contains all of the source and object files for the routines used to build the application. To use these routines in other projects, all source and object files must be relinked.

Exceptions to this are as follows:

- If you are using shared libraries, all code will not be contained in the executable file.
- On macOS*, the object files contain debug information and would need to be copied along with the executable.

Linking Debug Information

Windows*

Use option `Z7` at compile time or option `debug` at link time to tell the compiler to generate symbolic debugging information in the object file. Alternately, use option `Zi` at link time to generate executables with debug information in the `.pdb` file.

Linux*

Use option `g` at compile time to tell the compiler to generate symbolic debugging information in the object file.

Use option `gsplit-dwarf` to create a separate object file containing DWARF debug information. Because the DWARF object file is not used by the linker, this reduces the amount of debug information the linker must process and it results in a smaller executable file. See `gsplit-dwarf` for detailed information.

macOS*

You can link the DWARF debug information from the object files for an executable using `dsymutil`, a utility included with Xcode*. By linking the debug information in an executable, you eliminate the need to retain object files specifically for debugging purposes.

The utility runs automatically in the following cases:

- When you use the Intel® Fortran Compiler to compile directly from source to executable using the command line with option `g`. For example:

  Example

  ```
  ifort -g myprogram.f90
  ```

- When you compile using Xcode*.

In other cases, you must explicitly run `dsymutil`, such as when you compile using a make file that builds `.o` files and subsequently links the program.

Debugging

Depending on your operating system and architecture platform, several debuggers may be available to you.

You can use the debugger provided by your operating system:

- On Linux* and macOS*, this debugger is `gdb`.
- On Windows*, the debugger is the Microsoft Visual Studio* debugger.

On Windows* systems, you can use your local (host) system to debug an application running on a remote system. For more information, see Using Remote Debugging.

See Also

Using Remote Debugging

Preparing Your Program for Debugging

This section describes preparing your program for debugging.

Preparing for Debugging using the Command Line

To prepare your program for debugging when using the command line (ifort command):

1. Correct any compilation and linker errors.
2. In a command window, such as the Fortran command window available from the Intel® Fortran program folder, compile and link the program with full debug information and no optimization:

```
//  (Linux* OS and macOS*)
ifort -g file.f90
```

```
//  (Windows* OS)
    ifort /debug:full file.f90
```

On Linux* OS and on macOS*, specify the `g` compiler option to create unoptimized code and provide the symbol table and traceback information needed for symbolic debugging. The `notraceback` option cancels the traceback information.

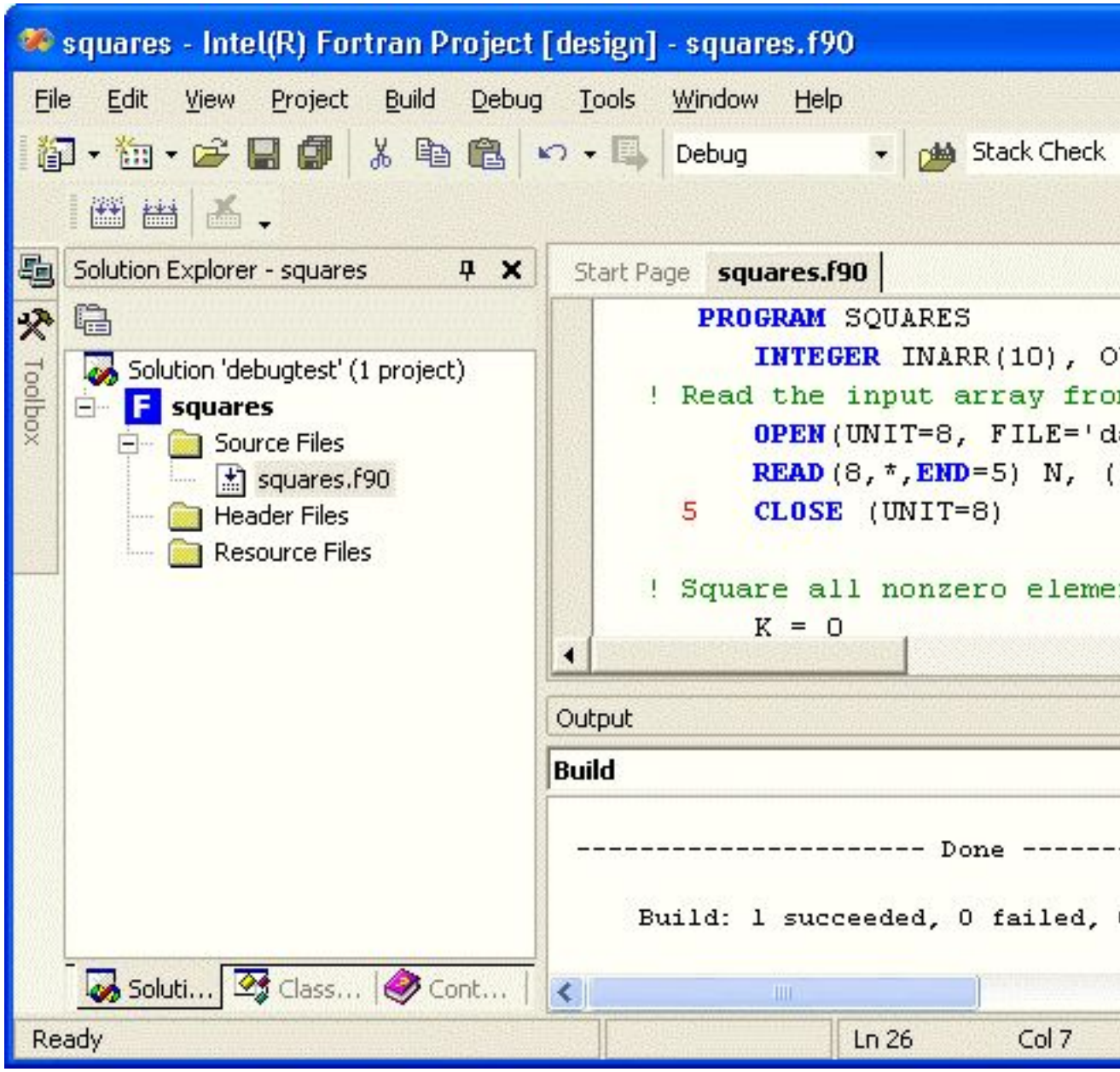
On Windows* OS, specify the `debug` compiler option with keyword `full` to produce full debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking.

Preparing for Debugging using Visual Studio*

The following applies to Windows* operating systems only.

To prepare your program for debugging when using the integrated development environment (IDE):

1. Start the IDE (select the appropriate version of Visual Studio* in the program folder).
2. Open the appropriate solution (using the Solution menu, either **Open Solution** or **Recent Projects**).
3. Open the **Solution Explorer** View.
4. To view the source file to be debugged, double-click on the file name. The screen resembles the following:



5. In the **Build** menu, select **Configuration Manager** and select the **Debug** configuration.
6. To check your project settings for compiling and linking, select the project name in **Solution Explorer** . Now, in the **Project** menu, select **Properties** , then click the Fortran folder in the left pane. Similarly, to check the debug options set for your project (such as command arguments or working directory), click the Debugging folder in the **Property Pages** dialog box.
7. To build your application, select **Build** > **Build Solution**.
8. Eliminate any compiler diagnostic messages using the text editor to resolve problems detected in the source code and recompile if needed.
9. Set breakpoints in the source file and debug the program. For more information, see Debugging the Squares Example Program.

NOTE You must set a breakpoint before starting the program, otherwise the program will run to completion, and exit.

See Also

Debugging the Squares Example Program

Using Breakpoints in the Debugger

This topic applies to Fortran applications for Windows* only.

This topic describes how to use the Microsoft Visual Studio* debugger to set file and data breakpoints. Information is organized as follows:

- Viewing All Current Breakpoints
- Working with File Breakpoints
- Adding Conditions to File Breakpoints
- Working with Data Breakpoints

In addition to file and data breakpoints, the Visual Studio* debugger supports the following:

- Address breakpoints
- Function breakpoints
- Tracepoints

For information on these additional features, see the Microsoft documentation for your version of Visual Studio*.

Viewing All Current Breakpoints

You can view all currently set breakpoints in the **Breakpoints** window. To view all set breakpoints using the Breakpoints window:

1. In the **Debug** menu, select **Windows** > **Breakpoints**.
2. Scroll up or down in the Breakpoints list to view the breakpoints. Enabled breakpoints have a check mark in the check box and disabled breakpoints have an empty check box. The window displays all types of breakpoints.

Working with File Breakpoints

Use a file breakpoint to interrupt program execution when the program reaches a specified location within a file. Several symbols are used to show the status of file breakpoints; these appear in the left margin of the **Source** window:

Enabled file breakpoints are identified as a red circle.

Disabled file breakpoints are identified as a hollow circle in the left margin.

During debugging, a **currently active** breakpoint appears as a red circle containing a yellow arrow in the left margin of the **Source** window.

For a walk-through of a sample debugging session using file breakpoints, see Debugging the Squares Example Program.

To set (enable) a file breakpoint:

1. Open the desired source file in the solution.
2. In the **Source** window, click the line at which you want to enable a file breakpoint.
3. Do one of the following:
   - Click in the left margin of the line or press the F9 key. When you set a breakpoint, it is enabled by default.

- Right click on the desired line and select **Insert Breakpoint** from the pop-up menu.

To disable a file breakpoint:

1. In the **Source** window, click the line containing the file breakpoint you want to disable.
2. Do one of the following:
   - Right click on that line, and select **Disable Breakpoint** from the pop-up menu.
   - In the **Debug** menu, select **Windows** > **Breakpoints**, which opens the **Breakpoints** window. Select the check box for that breakpoint so it is unchecked (disabled).

To remove a file breakpoint:

1. In the **Source** window, click the line containing the file breakpoint you want to remove.
2. Do one of the following:
   - Click in the left margin of the line or press the F9 key. If the breakpoint was originally disabled, press F9 again to remove it.
   - Right click on that line, and select **Delete Breakpoint** from the pop-up menu.
   - In the **Debug** menu, select **Windows** > **Breakpoints**. In the **Breakpoints** window, select the breakpoint in the Name column and click the **Delete** button (which looks like an "**X**").

To view the source code where a file breakpoint is set:

1. In the **Debug** menu, select **Windows** > **Breakpoints**.
2. In the Breakpoints window, click a file breakpoint.
3. Click the **Go To Source Code** button.

   This action takes you to the source code for a breakpoint set at a line number or function (or subroutine) name. In the case of function (or subroutine) names, the debugger must be running for this to work.

To remove all breakpoints (including data breakpoints):

In the **Debug** menu, select **Delete All Breakpoints**.

Adding Conditions to File Breakpoints

When you associate a condition with a file breakpoint, the program executes until that location is reached *and* the specified condition is met, such as when the value of an array element is greater than 1.

To add a condition to a breakpoint:

1. Set a file breakpoint.
2. Right-click in the associated line and select **Breakpoint** > **Condition...**.

   The **Breakpoint Condition** dialog box appears.

The condition that you specify in the **Breakpoint Condition** dialog box must relate to a local variable within the breakpoint's scoping unit, or to a global variable. You must apply proper Fortran syntax in the **Condition** field in order for the specified condition to be properly detected. Consider the following examples:

To break when the local variable `RecNum` reaches a value of 7520, enter the following condition:

```
RecNum == 7520
```

You can also specify joint conditions. To break when `RecNum` reaches at least 7520 and the variable `Var2` is equal to 90, enter the following condition:

```
RecNum >= 7520 .and. Var2 == 90
```

You can also specify an `or` condition. For example:

```
Var1 == 3.14 .or. Var2 < 500.0
```

If the condition that you specify occurs at the chosen location when the program is run, a message box appears. If the breakpoint is set in part of a loop, continuing execution proceeds until the debugger detects another specified condition. If no such condition is detected, the debugger continues to the next breakpoint or, if no more breakpoints are set, until the end of the loop.

To disable, enable, or remove a file breakpoint with a defined condition, follow the general procedures for a file breakpoint. Use the checkbox for the **Condition** field to enable or disable the condition setting.

Working with Data Breakpoints

Use a data breakpoint to interrupt program execution when the value of a certain variable changes. A data breakpoint displays a message box when the value of a variable changes, or, if a condition has been defined, when a condition is met. Unlike a file breakpoint, data breakpoints are not associated with a specific source location.

Using Visual C++ Data Breakpoint Support

The Visual Studio* IDE does not provide data breakpoint support specifically for Fortran, but you can still use data breakpoints in a Fortran program; they are handled using the existing mechanism provided by Microsoft Visual C++*. This means that you must use Visual C++* syntax; you cannot enter the variable name or condition using Fortran syntax. The following guidelines apply:

- For scalar variables, enter the variable name in upper case. This allows Visual C++* to find the variable in the debug information and apply the correct breakpoint.
- For arrays and types, use the Fortran LOC intrinsic function in a **Watch** window or the **Immediate** window to obtain the address of the variable. Use this address in the variable name edit box.

For example, to prepare to set a data breakpoint at an array element a(5), do the following:

1. Select **Debug** > **Windows** > **Immediate**.
2. Enter `loc(a(5))` in the immediate window and press Enter.

   The result displayed is the address of the array element. The address may be displayed as a decimal number or a hexadecimal number, depending upon your current display mode. For a decimal number, you will enter it in the **Address:** field. For a hexadecimal number, you will use the Visual C++* hexadecimal syntax (`0xnnnnnnnn`) rather than the Fortran syntax (`#nnnnnnnn`) in the fields.

The following procedure applies to debugging the current routine (current scope).

To set a data breakpoint:

1. Start debugging.
2. In the **Debug** menu, select **New Breakpoint** > **New Data Breakpoint..**.
3. Enter the desired address. Also enter the Byte Count and Language (specify C++).

If you want to associate a condition with this breakpoint, right-click on the breakpoint in the **Breakpoints** window and choose **Condition..**.

To disable, enable, or remove a data breakpoint:

1. In the **Debug** menu, select **Windows** > **Breakpoints**.
2. To disable or enable the data breakpoint, use the check box to the left of the data breakpoint (check indicates enabled).
3. To remove a data breakpoint, select the data breakpoint and click the **Delete** button.

Under certain conditions, the debugger may disable a data breakpoint. In this case, you should either try to enable it or remove and set it again.

To remove all breakpoints (including file breakpoints):

In the **Debug** menu, select **Delete All Breakpoints**.

See Also
Debugging the Squares Example Program

Debugging the Squares Example Program

This topic applies to Fortran applications for Windows* only.

The example below shows a program called `SQUARES` that requires debugging. The program was compiled and linked without diagnostic messages from either the compiler or the linker, however, this program contains a logic error in an arithmetic expression.

```
   PROGRAM SQUARES
      INTEGER INARR(10), OUTARR(10), I, K
! Read the input array from the data file.
      OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
      READ(8,*,END=5) N, (INARR(I), I=1,N)
  5   CLOSE (UNIT=8)

! Square all nonzero elements and store in OUTARR.
      K  = 0
      DO I = 1, N
        IF (INARR(I) .NE. 0) THEN
        OUTARR(K) = INARR(I)**2
        ENDIF
      END DO

! Print the squared output values. Then stop.
      PRINT 20, N
  20  FORMAT (' Total number of elements read is',I4)
      PRINT 30, K
  30  FORMAT (' Number of nonzero elements is',I4)
      DO I=1,K
        PRINT 40, I, OUTARR(K)
  40    FORMAT(' Element', I4, 'Has value',I6)
      END DO
  END PROGRAM SQUARES
```

The program `SQUARES` performs the following functions:

- Reads a sequence of integer numbers from a data file and saves these numbers in the array `INARR`. The file `datafile.dat` contains one record with the integer values 4, 3, 2, 5, and 2. The first number indicates the number of data items that follow.
- Enters a loop in which it copies the square of each nonzero integer into another array `OUTARR`.
- Prints the number of nonzero elements in the original sequence and the square of each such element.

In the program, the logic error occurs because variable `K`, which keeps track of the current index into `OUTARR`, is not incremented in the loop on lines 9 through 13. The statement `K = K + 1` should be inserted just before line 11.

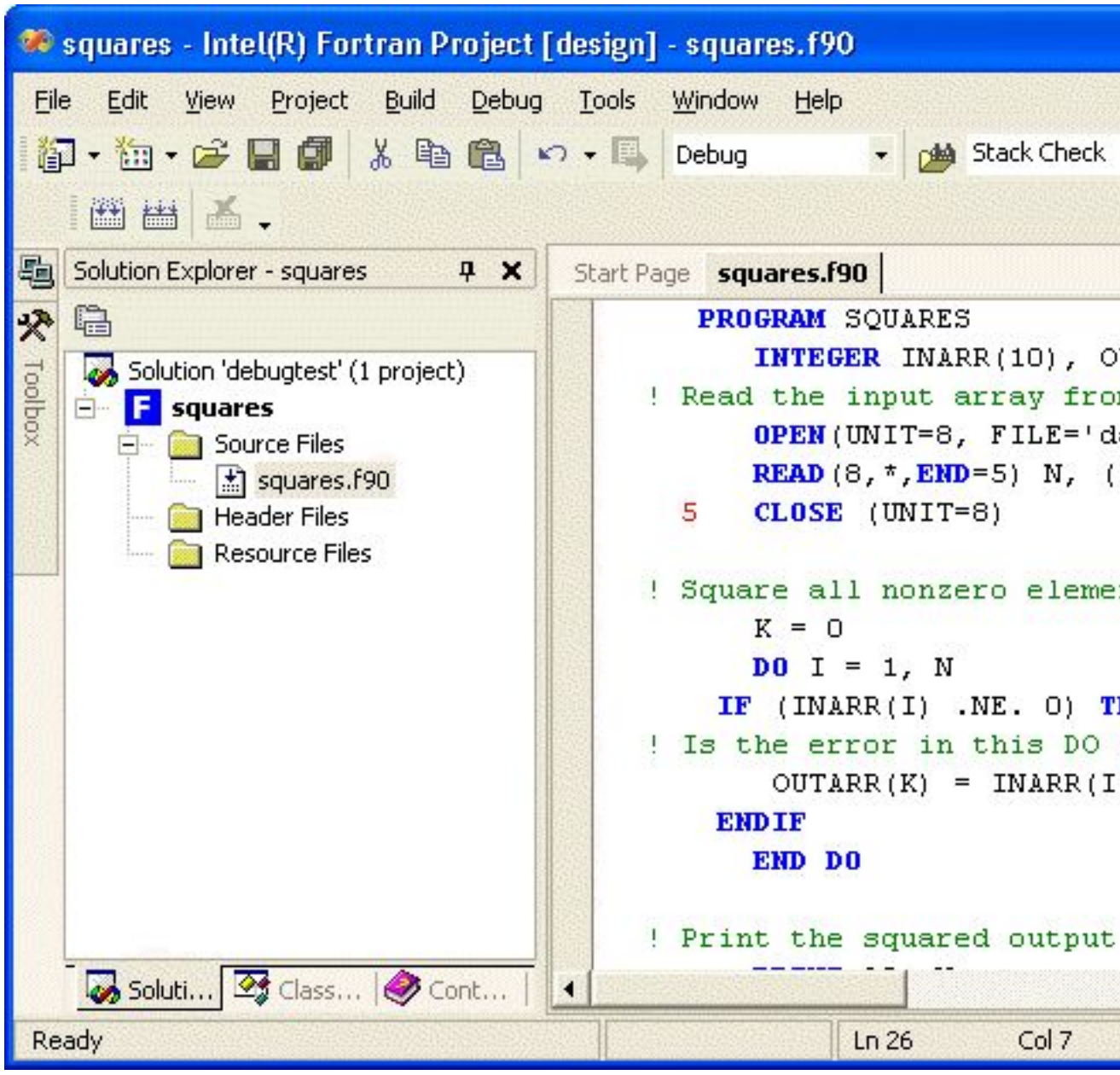
This example assumes that the program was executed without array bounds checking (set by specifying `nobounds` for the `check` command-line option). When executed with array bounds checking, a run-time error message appears.

Debugging Example: The following shows how to debug a Windows* OS based program using the Microsoft integrated debugger within the integrated development environment (see Preparing Your Program for Debugging and Using Breakpoints in the Debugger). This example assumes a solution already exists.

To debug this program:

1. Start Visual Studio*.
2. In the **File** menu, select **Open Solution**. Open the solution containing the file.

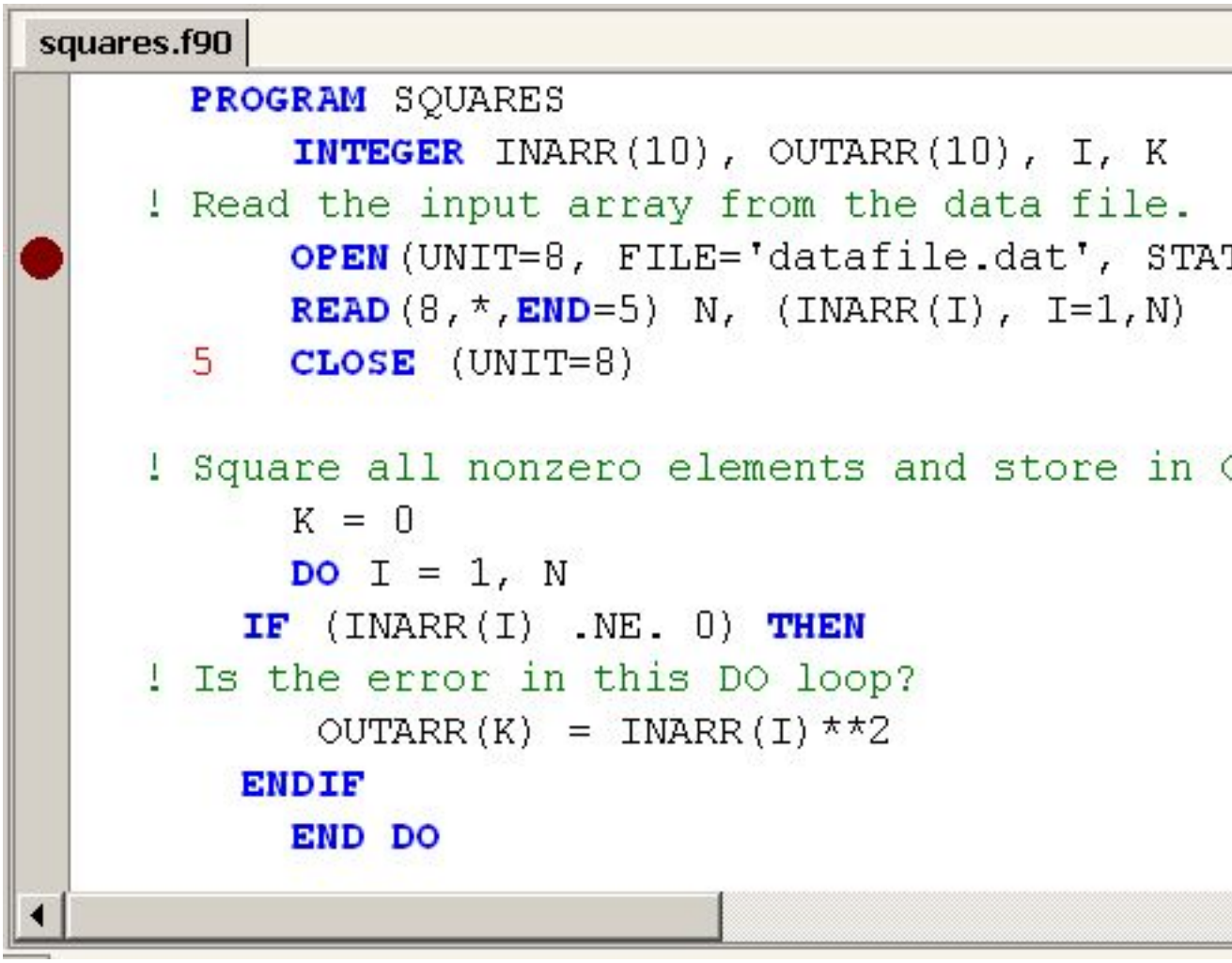
3. Edit the file `squares.f90`: double-click the file name in the **Solution Explorer** View. The screen appears as follows:



   The following toolbars are shown: **Build** toolbar, **Standard** toolbar.
4. To change the displayed toolbars, select **View** > **Toolbars**. To display the debug toolbar, select **Debug**.
5. Click the first executable line to set the cursor position. In this case, click the beginning of the `OPEN` statement line:

```
OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
```

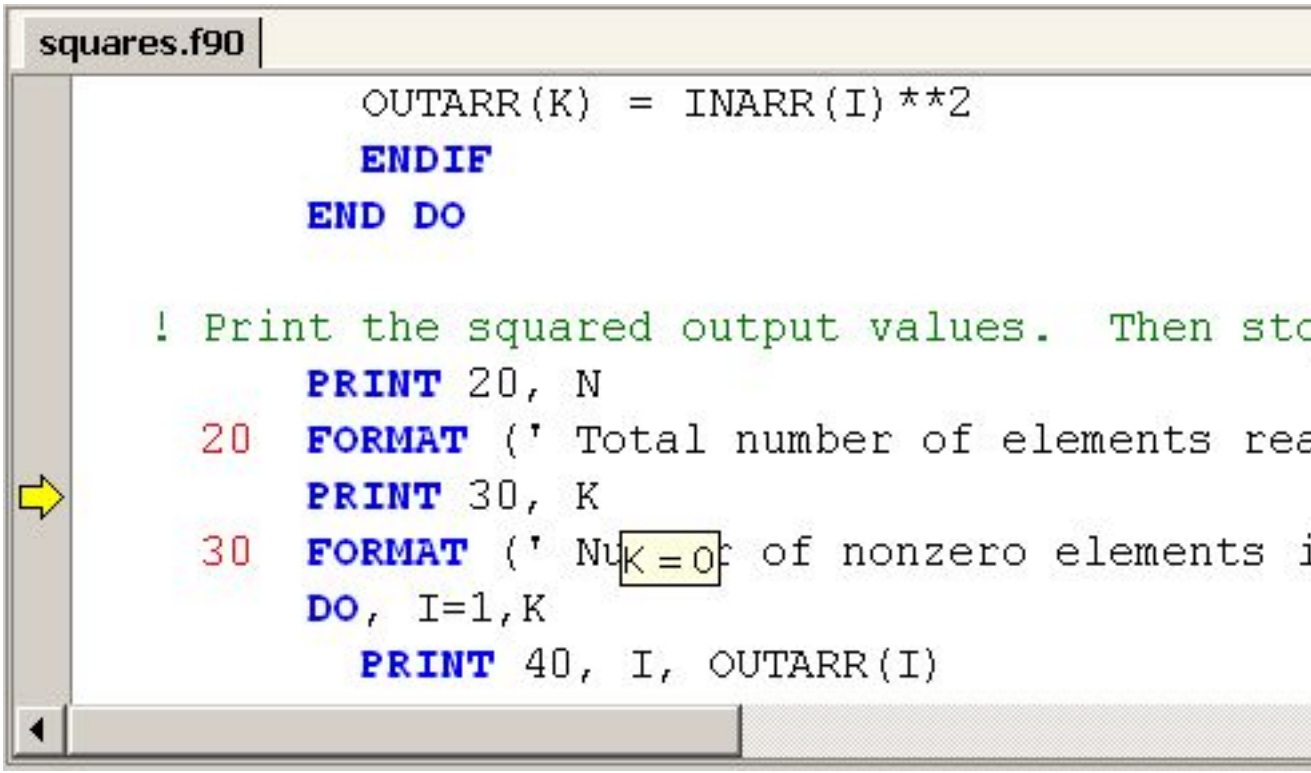
6. Click in the left margin of the first executable line to set a breakpoint. The red circle in the left margin of the text editor/debugger window shows where a breakpoint is set:



7. This example assumes you have previously built your application (see Preparing Your Program for Debugging).

   In the **Debug** menu, select **Start Debugging**.

   The debugger is now active. The current position is marked by a yellow arrow at the first executable line (the initial breakpoint).
8. If needed, you can set another breakpoint. Position the cursor at the line where you want to add a breakpoint, right click, and select **Breakpoint** > **Insert Breakpoint.**
9. Step through the lines of source code. You can do this with **Debug** > **Step Over** or with the **Step Over** button on the **Debug** toolbar.
10. Repeat the **Step Over** action and follow program execution into the `DO` loop and unto the end of the program. Position the cursor over the variable `K` to view its value in a **Data Tip** box:



The error seems related to the value of variable `K`.

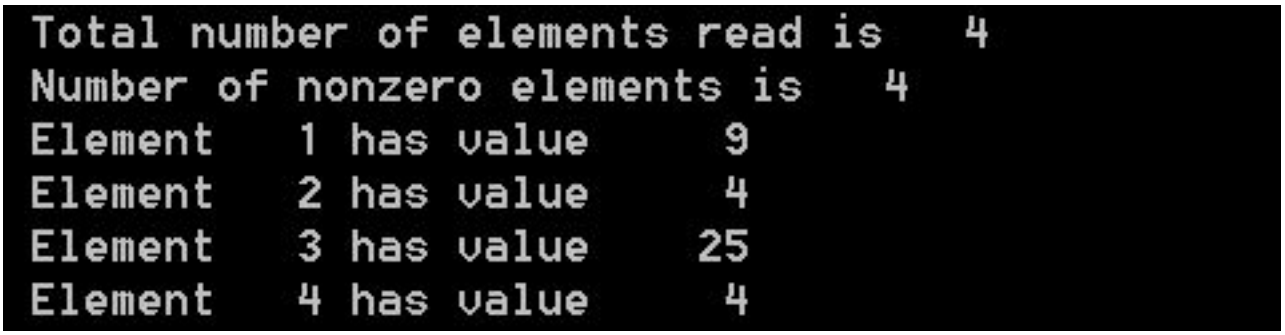
11. In the **Text Editor** , add the line `K = K + 1` as follows:

```
! Square all nonzero elements and store in OUTARR.
K = 0
DO I = 1, N
IF (INARR(I) .NE. 0) THEN
K = K + 1       ! add this line
OUTARR(K) = INARR(I)**2
ENDIF
END DO
```

12. Since you have modified the source, you need to rebuild the application:

- In the **Debug** menu, select **Stop Debugging**
- In the **Build** menu, select **Build debugtest**.
- In the **Debug** menu, select **Start Debugging**.

The output screen appears as follows:



13. The program now generates better results. You can examine the values of both the input array `INARR` (read from the file) and the output array `OUTARR` that the program calculates. In the **Text Editor** window, the previously set breakpoint remains set.

In the **Debug** menu, select **Start Debugging**.

14. To view the values of certain variables as the program executes, display the Locals window. In the **Debug** menu, select **Windows** > **Locals**.
15. You can view the values of the local variables in the Locals window. Click the plus sign to view array values.

    The **Locals** window does not let you display module variables or other non-local variables. To display non-local variables, display one of the **Watch** windows.
16. Although this example does not use module variables or non-local variables, you can drag a variable name into the **Watch** window so the variable can be displayed. The **Watch** window allows you to display expressions.

    In the **Text Editor** window, select the variable name `INARR` (without its subscript syntax), drag it, and drop it into the **Watch** window, Name column:
17. Also drag the `OUTARR` array name to the **Watch** window. Click the Plus sign (**+**) to the left of the `OUTARR` variable's name to display the values of its array elements.
18. Execute lines of the program by using the **Step Over** button on the **Debug** toolbar. As the program executes, you can view the values of scalar variables with the data tips feature and view the values of arrays (or other variables) in the **Watch** window.

If a **Disassembly** window (shows disassembled code with source-code symbols) unintentionally appears, click the **Step Out** button on the debugger toolbar (or select **Step Out** in the **Debug** menu) as needed to dismiss the **Disassembly** window.

Preparing Your Program for Debugging
Using Breakpoints in the Debugger

Viewing Fortran Data Types in the Microsoft Debugger

This topic applies to Fortran applications for Windows* only.

The following general suggestions apply to different types of Fortran data:

- For scalar (nonarray) data, use the data tips (leave pointer on a variable name) feature or use the **Locals** window or a **Watch** window. Intel® Fortran does not support the **Autos** window.
- For single-dimension array data, derived-type data, record structure data, and COMPLEX data, use the **Locals** window or a **Watch** window.
- For common blocks exported from a DLL, enter the name of the common block in a **Watch** window. You will be able to view the common block variables like any other structure.
- By default, values of named constants (parameters) are not visible in the debugger. To make these visible, add the `/debug-parameteres:used` option or `/debug-parameters:all` option when compiling the sources.

For information on using Data Tips, the **Locals** window, or a **Watch** window, see Debugging the Squares Example Program.

The following sections apply to using a **Watch** window:

- Specifying Array Sections
- Specifying Module Variables
- Specifying Format Specifiers

To display a **Watch** window:

1. In the **Debug** menu, select **Windows** > **Watch**.
2. In the submenu, click Watch 1, 2, 3, or 4.

Specifying Array Sections

You can specify array sections in a **Watch** window. For example, consider an array declared as:

```
integer foo(10)
```

You can specify the following statement in a **Watch** window to see the 2nd, 5th, and 8th elements:

```
foo(2:10:3)
```

When working with character arrays, this syntax may be combined with a substring specification. Consider the following array declaration:

```
character*8 chr_arr(10)
```

You can specify the following statement in a **Watch** window to display the substring made up of character 3 through 8 of array elements 2 through 5:

```
chr_arr(2:5)(3:8)
```

This support is available for arrays of any type, including array pointers, assumed-shape, allocatable, and assumed-size arrays.

Any valid integer expression can be used when specifying lower bound, upper bound, or stride. If the lower bound is omitted, the array lower bound is used. If the upper bound is omitted, the array upper bound is used. For example, consider the following declaration:

```
integer foo(10)
```

To display:

- Elements 1 through 8, specify `foo(:8)`
- Elements 5 through 10, specify `foo(5:)`
- All 10 elements, specify `foo(:)`

Specifying Module Variables

To view a module variable in a Watch window, specify the module name, followed by "::", followed by the variable name.

For example, to watch variable "bar" of module "foo", specify the following expression:

```
foo::bar
```

Specifying Format Specifiers

You can use format specifiers in **Watch** windows to display variables in different data formats.

For example, given a REAL variable 'foo' in a program, it is now possible to see 'foo' in different floating point notation (by typing "foo,f" "foo,g" or "foo,e" in a **Watch** window) or as an integer ("foo,i" or "foo,d"), a hexadecimal value ("foo,x"), an an octal value ("foo,o"), and so on.

You can change the display format of variables in a **Watch** window using the formatting symbols in the following table:

Symbol	Format	Value	Displays
d,i	*signed* decimal integer	0xF000F065	-268373915
o	*unsigned* octal integer	0xF065	0170145
x,X	Hexadecimal integer	61541 (decimal)	#0000F065
f	*signed* floating-point	3./2.	1.5000000

Symbol	Format	Value	Displays
e	*signed* scientific notation	3./2.	0.1500000E +01
g	*signed* floating-point or *signed* scientific notation, whichever is shorter	3./2.	1.500000
c	Single character	0x0065	'e'
s	String	0x0012fde8	"Hello world"

To use a formatting symbol, type the variable name, followed by a comma and the appropriate symbol. For example, if `var` has a value of `0x0065`, and you want to see the value in character form, type `var,c` in the **Name** column on the tab of the **Watch** window. When you press ENTER, the character-format value appears:

```
var,c = 'e'
```

You can use the formatting symbols shown in the following table to format the contents of memory locations:

Symbol	Format	Displays
ma	64 ASCII characters	0x0012ffac .4...0...".0W&.....1W&.0.:W..1...".. 1.JO&.1.2.."..1...0y....1
m	16 bytes in hexadecimal, followed by 16 ASCII characters	0x0012ffac B3 34 CB 00 84 30 94 80 FF 22 8A 30 57 26 00 00 .4...0...".0W&..
mb	16 bytes in hexadecimal, followed by 16 ASCII characters	0x0012ffac B3 34 CB 00 84 30 94 80 FF 22 8A 30 57 26 00 00 .4...0...".0W&..
mw	8 words	0x0012ffac 34B3 00CB 3084 8094 22FF 308A 2657 0000
md	4 doublewords	0x0012ffac 00CB34B3 80943084 308A22FF 00002657

With the memory location formatting symbols, you can type any value or expression that evaluates to a location.

A formatting character can also follow an expression:

```
rep+1,x
xloc,g
count,d
```

NOTE
You can apply formatting symbols to structures, arrays, pointers, and objects as unexpanded variables only. If you expand the variable, the specified formatting affects all members. You cannot apply formatting symbols to individual members.

See Also

Debugging the Squares Example Program

Viewing the Call Stack in the Microsoft Debugger

In most cases, your program will automatically stop at the point where the exception occurs, allowing you to view the source code and values of variables. If the error is related to an I/O statement, you may want to view the call stack.

If you want to view where your program is currently executing in the hierarchy of routines , you can display the **Call Stack** window in the debugger. For example, you may want to display this window to determine where in your program an exception occurs.

To view the call stack:

1. Start the debugger and stop at a breakpoint.
2. Select **Debug** > **Windows** > **Call Stack**.

A severe unhandled I/O programming error (such as an End-of-File condition) can occur while the program is executing in the debugger. When this occurs, the Fortran run-time system will raise a debug event automatically to stop program execution, allowing display of the **Call Stack**.

When the severe unhandled I/O error occurs in the debugger:

- An information box is displayed that contains:

```
User breakpoint called from code at 0xnnnnnnn
```

- A window appears with your cursor in NTDLL disassembly code

Click **OK** to dismiss the information box.

Scanning down the **Call Stack** display, there will be a few frames from NTDLL and the Fortran run-time system displayed, then the actual Fortran routine with the I/O statement that caused the error. In the call stack, select the Fortran routine to display the Fortran code and the variables using the **Locals** window. The green arrow points to the I/O statement that caused the error.

This action all occurs after the error message and traceback information has been displayed. The error message and traceback information is available in the program output window. To view the program output window, either iconize (minimize) the integrated development environment (IDE) or click the icon for the output window in the task bar. You should not need the stack dump because you have the **Call Stack** window in the IDE, but the error message with the file name might be useful to see.

See Also

Locating Run-Time Errors

Traceback

Locating Unaligned Data

Unaligned data can slow program execution. You should determine the cause of the unaligned data, fix the source code (if necessary), and recompile and relink the program.

If your program encounters unaligned data at run time, to make it easier to debug the program, you should recompile and relink with the `-g` (Linux* OS and macOS*) or `/debug:full` (Windows* OS) option to generate sufficient table information and debug unoptimized code.

Debugging a Program that Encounters a Signal or Exception

If your program encounters a signal (exception) at run time, you may want to recompile and relink with certain command-line options before debugging the cause. The following will make it easier to debug the program:

- Use the `fpe`*`[:]n`* option to control the handling of floating point exceptions.

- To generate sufficient symbol table information and debug unoptimized code, as with other debugging tasks, use the `-g` (Linux* and macOS*) or `/debug:full` (Windows*) compiler option.

Debugging an Exception in the Microsoft Debugger (Windows*)

You can request that the program always stop when a certain type of exception occurs. Because certain exceptions are caught by default by the Intel® Fortran run-time library, your program stops in the run-time library code. In most cases, you want the program to stop in your program's source code instead.

To change how an exception is handled in the Microsoft debugger:

1. In the **Debug** menu, select **Exceptions**.
2. View the displayed exceptions.
3. Select **Windows Exceptions**. Select each type of exception to be changed and change its handling using the radio buttons.
4. Start program execution using **Start** in the **Debug** menu.
5. When the exception occurs, you can now view the source line being executed, examine current variable values, execute the next instruction, and so on, to help you better understand that part of your program.
6. After you locate the error and correct the program, consider whether you want to reset the appropriate type of exception to "**Use Parent Setting**" before you debug the program again.

For machine exceptions, you can use the just-in-time debugging feature to debug your programs as they run outside of the visual development environment. To do this, set the following items:

- In **Tools** > **Options**, select **Native** in the **Debugging Just-In Time** category.
- Set the `FOR_IGNORE_EXCEPTIONS`, as listed in Supported Environment Variables, to `TRUE`.

See Also

Supported Environment Variables

Debugging and Optimizations

This topic describes the relationship between various command-line options that control debugging and optimizing.

Whenever you enable debugging with the `-g` option (Linux* and macOS*) or `/debug:full` option (Windows*) , you disable optimizations. You can override this behavior by explicitly specifying compiler options for optimizations on the command line.

The following summarizes commonly used options for debugging and for optimization.

`-O0` (Linux* and macOS*) `/Od` (Windows*)	Disables optimizations so you can debug your program before any optimization is attempted. This is the default behavior when debugging. **NOTE** On Linux* and macOS*, `-fno-omit-frame-pointer` is set if either option `-O0` or `-g` is specified. For more information, see the following topics: • `-O0` (Linux* and macOS*) compiler option • `/Od` (Windows*) compiler option
`O1`, `O2`, or `O3` (Linux* and macOS*) `O` (Windows*)	Specifies the code optimization level for applications. If you use any of these options, it is recommended that you use `-debug extended` when debugging. For more information, see the following topics:

	• `-O1`, `-O2`, `-O3` (Linux* and macOS*) compiler options • `/O` (Windows*) compiler option
`-g` (Linux* and macOS*) `/debug:full` (Windows*)	Generates symbolic debugging information and line numbers in the object code for use by the source-level debuggers. Turns off `O2` and makes `-O0` (Linux* and macOS*) or `/Od` (Windows*) the default. The exception to this is if options `O1`, `O2`, or `O3` are explicitly specified in the command line. For more information, see the following topics: • `-g` (Linux* and macOS*) • `/debug:full` (Windows*) compiler option
`-debug extended` (Linux* and macOS*)	Specifies settings that enhance debugging. For more information, see the following topic: • `-debug extended` (Linux* and macOS*)
`-fp` (Linux* OS and macOS*) `/Oy` (Windows*) (IA-32 architecture only)	Disables the ebp register in optimizations and sets the ebp register to be used as the frame pointer. For more information, see the following compiler option topics: • `-fp` (Linux* and macOS*) • `/Oy` (Windows*)
`traceback`	Causes the compiler to generate extra information in the object file, which allows a symbolic stack traceback. For more information, see the following topic: • `traceback` compiler option

Combining Optimization and Debugging

The compiler lets you generate code to support symbolic debugging when `O1`, `O2`, or `O3` optimization options are specified on the command line along with `-g` (Linux* and macOS*) or `/debug:full` (Windows*); this produces symbolic debug information in the object file.

NOTE
Note that if you specify an `O1`, `O2`, or `O3` option with the `-g` (Linux* and macOS*) or `/debug:full` (Windows*) option, some of the debugging information returned may be inaccurate as a side-effect of optimization.

To counter this on Linux* OS and macOS*, you should also specify the `-debug extended` option.

It is best to make your optimization and/or debugging choices explicit:

- If you need to debug your program excluding any optimization effect, use the `-O0` (Linux* and macOS*) or `/Od` (Windows*) option, which turns off all the optimizations.
- If you need to debug your program with optimizations enabled, then you can specify the `O1`, `O2`, or `O3` option on the command line along with `debug extended`.

NOTE
When no optimization level is specified, the `-g` or `/debug:full` option slows program execution; this is because this option turns on `-O0` or `/Od`, which causes the slowdown. However, if both `O2` and `-g` (Linux* and macOS*) or `/debug:full` (Windows*) are specified, for example, the code should not experience much of a slowdown.

Refer to the table below for the summary of the effects of using the `-g` or `/debug:full` option with the optimization options.

These options	Produce these results
`-g` (Linux* and macOS*) `/debug:full` (Windows*)	Debugging information produced, `-O0` (Linux* and macOS*) or `/Od` (Windows*) enabled (meaning optimizations are disabled). For Linux* and macOS*, `-fp` is also enabled for compilations targeted for IA-32 architecture.
`-g-O1` (Linux* and macOS*) `/debug:full /O1` (Windows*)	Debugging information produced, `O1` optimizations enabled.
`-g-O2` (Linux* and macOS*) `/debug:full /O2` (Windows*)	Debugging information produced, `O2` optimizations enabled.
`-g-O2` (Linux* and macOS*) `/debug:full /O2 /Oy-` (Windows*)	Debugging information produced, `O2` optimizations enabled. For Windows* using IA-32 architecture, `/Oy` disabled.
`-g -O3 -fp` (Linux* and macOS*) `/debug:full /O3` (Windows*)	Debugging information produced, `O3` optimizations enabled. For Linux*, `-fp` enabled for compilations targeted for IA-32 architecture.

NOTE
Even the use of option `debug extended` with optimized programs may not allow you to examine all variables or to set breaks at all lines, due to code movement or removal during the optimization process.

Debugging Mixed-Language Programs

The Visual Studio* debugger let you debug mixed-language programs. Program flow of control across subprograms written in different languages is transparent.

The debugger uses debug information associated with the program to automatically identify the language of the current subprogram or code segment.

For example, if program execution is suspended in a subprogram in Fortran, the current language is seen as Fortran. If the debugger stops the program in a C function, the current language becomes C.

Debugging Multithreaded Programs

The debugging of multithreaded programs discussed in this topic applies to both the OpenMP* Fortran API and the Intel® Fortran parallel compiler directives. When a program uses parallel decomposition directives, you must take into consideration that the bug might be caused either by an incorrect program statement or it might be caused by an incorrect parallel decomposition directive. In either case, the program to be debugged can be executed by multiple threads simultaneously.

To determine the correctness of and debug multithreaded programs, you can use the following:

- Linux* and macOS* systems: `gdb`.
- Windows* operating systems: Microsoft Visual Studio* Debugger.
- Intel® Fortran Compiler debugging options and methods; in particular, `debug` and `traceback` options.

Using Remote Debugging

Using Remote Debugging

The following applies to Microsoft* Visual Studio* 2013 and 2015.

Remote debugging lets you use your local (host) system to debug an application running on a remote system.

Remote Debugging using Microsoft* Visual Studio*

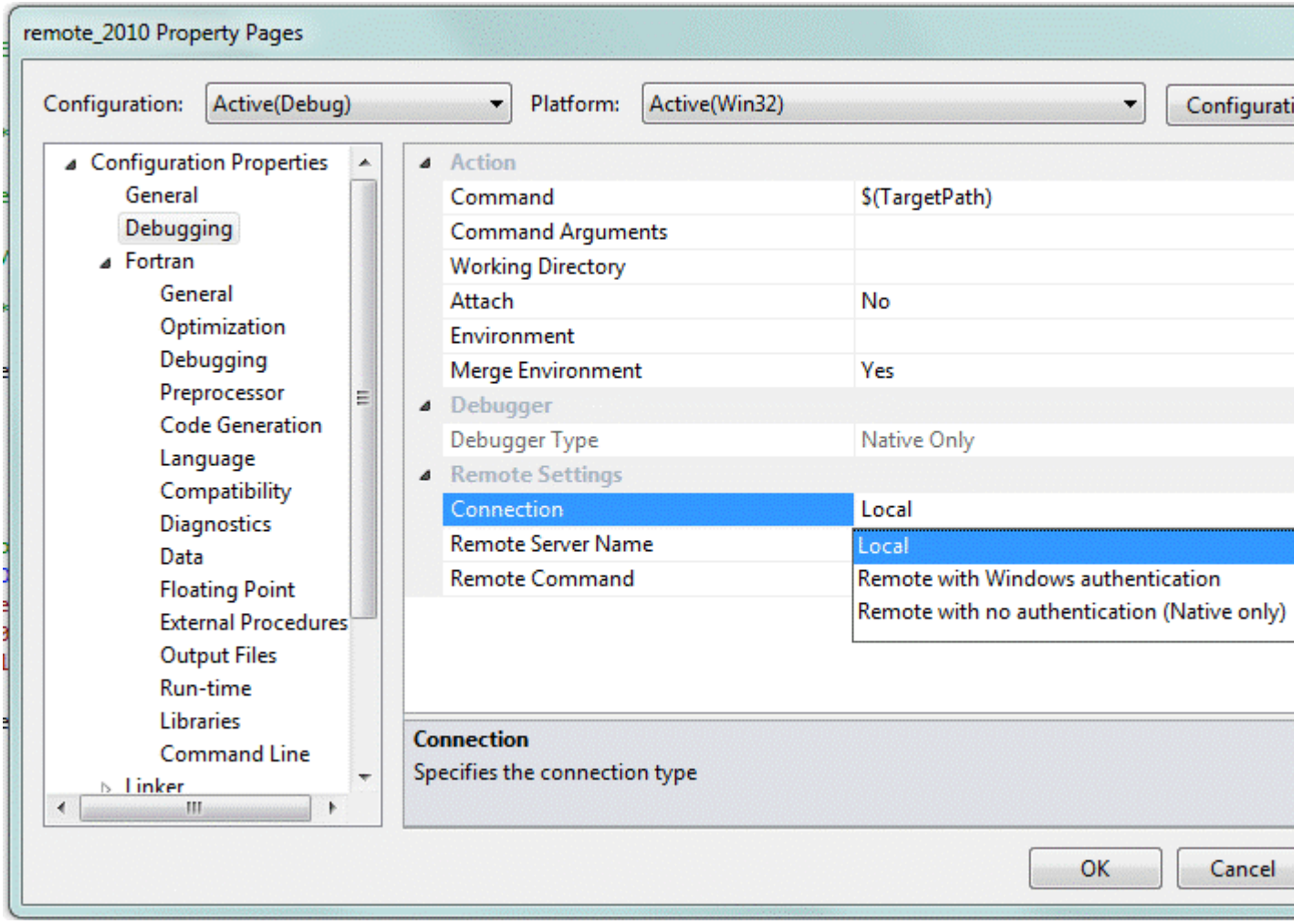
There are two remote connection types available: DCOM ("Remote with Windows authentication") and TCP/IP ("Remote with no authentication"). For more information on initial setup for remote debugging, refer to the Microsoft* Visual Studio* remote debugging setup topics in the Microsoft* MSDN documentation.

Remote debugging typically requires use of the debugging monitor (`msvsmon.exe`). Run the monitor in the correct mode for the remote system's architecture. To do this, on the remote system, select the program group **Microsoft Visual Studio 2013** (or applicable version year), then choose **Visual Studio Tools** and select the correct **Remote Debugger Folder**.

Remote debugging also requires proper set up of the IDE Debugging property page on the host machine.

To set up remote debugging using the IDE property pages:

1. On the host machine, choose **Project** > **Properties** > **Configuration Properties** > **Debugging**. Your screen should resemble this image:



2. Specify the following:

   Connection: Use the drop-down box to select the type of connection.

   Remote Server Name: Use the drop-down box to select the name of the remote system.

   Remote Command: Type the command you want to issue on the remote machine. Typically, this is a pathname to an executable file (`.exe`) on the remote machine; for example: `d:\remote\myapp.exe`
3. Optionally, set **Attach** to **Yes** to attach to an application that is already running on the remote machine.

To start remote debugging:

1. Make sure you have the necessary remote debugging permissions on the remote machine. For example, you need Administrator privileges if you want to debug a process running on a remote machine under another account name.
2. Run the Remote Debugging Monitor on the remote machine.
3. Launch Microsoft* Visual Studio* on the host machine and use it to attach to a program you want to debug on the remote machine, or launch a program you want to debug on the remote machine.

Remote Debugging Scenario

This topic applies to Fortran applications for Windows* only.

The remote debugging scenario described here includes the cross-platform and cross-compilation remote debugging features. The following parameters apply:

Host platform:	IA-32
Remote platform:	Intel® 64
Name of host machine:	*HOST_MACHINE* (domain name of host machine)
Name of remote machine:	*REMOTE_SERVER* (domain name of remote machine)
Host OS:	Windows* 10 - 32 Bit
Remote OS:	Windows* 10 - 64 Bit
Executable for debugging:	Win64 (x64)
Type of executable:	Console application
Visual Studio*:	Microsoft Visual Studio* 2017
Connection types:	Remote with/without Windows authentication
Sessions:	Remote debugging, remote execution, attach to process on a remote machine

1. *HOST_MACHINE*

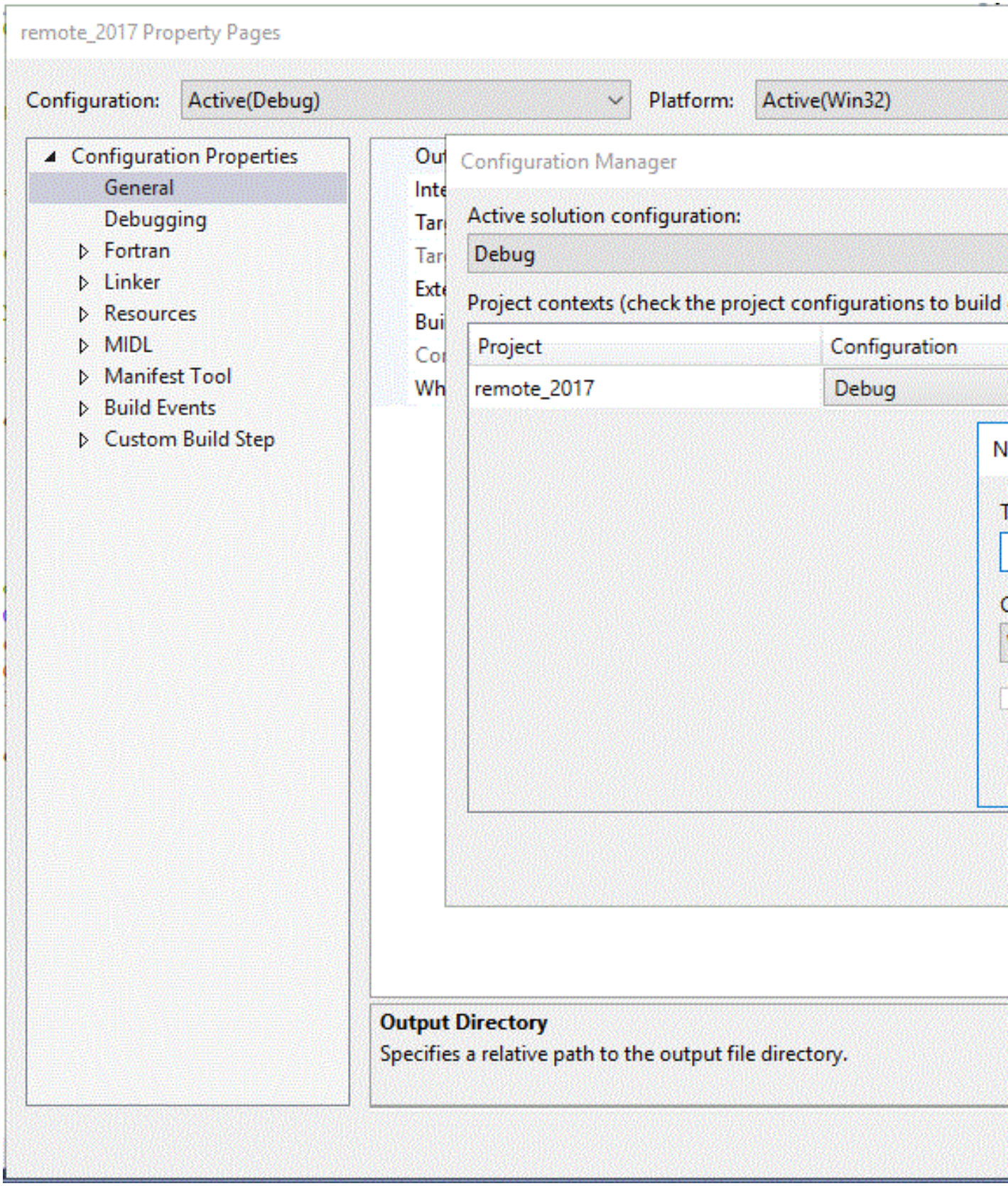
Start Visual Studio* 2017 on the host machine (*HOST_MACHINE*).

2. Create a new console application; for this example, use remote_2017.

3. Add the following lines to the `remote_2017.f90` file:

```
REAL, DIMENSION(1) :: xxx = 0
PRINT *, 'Type one number: '
READ  (*,'(F10.3)') xxx
```

4. Open **Project** > **Properties** and create a new platform configuration for x64 using the **Configuration Manager**.



5. Make sure, that current (target) configuration is **Debug|x64** and build the solution (**Build** > **Build Solution**). This invokes the cross-compiler and causes the 64-bit executable file `remote_2017.exe` to be built.
6. Run the `remote_2017.exe` application to make sure that it is 64-bit code. An error should result, explaining that it is not a valid Win32 application.
7. Run the Remote Desktop Connection program (`%SystemRoot%\System32\mstsc.exe`) and connect to the remote machine (*REMOTE_SERVER*).
8. Create `C:\remote_dir` on *REMOTE_SERVER* and copy `remote_2017.exe` from *HOST_MACHINE* to *REMOTE_SERVER*.
9. On the *HOST_MACHINE* open the `remote_2017.f90` source file in the Visual Studio editor.
10. Set a breakpoint at line 27.

```
1   !--remote_2017.f90-
2   !
3   !--FUNCTIONS:
4   !--remote_2017---Entry-point-of-console-application.
5   !
6
7   !********************************************************************************
8   !
9   !--PROGRAM:-remote_2017
10  !
11  !--PURPOSE:---Entry-point-for-the-console-application.
12  !
13  !********************************************************************************
14
15      program-remote_2017
16
17      implicit-none
18
19      !-Variables
20
21      !-Body-of-remote_2017
22      REAL,-DIMENSION(1)-::-xxx-=-0
23      PRINT-*,-'Type-one-number:-'
24      READ--(*,'(F10.3)')-xxx
25      print-*,-'Hello-World'
26
27      end-program-remote_2017
28
29
```

11. Open **Project** > **Properties** and select **Debugging**. In the Remote Settings section, set the following properties:

    Connection: Remote with Windows authentication

    Remote Machine: *REMOTE_SERVER*

    Remote Command: `C:\remote_dir\remote_2017.exe`
12. Press F5 to start debugging.

    If there is a firewall you will get an `Unable to start debugging` error message. In this case go to *REMOTE_SERVER*, locate, and run the Remote Debugger application (64-but mode). Find `msvsmon.exe` in the location where it was installed, or open the Start menu and search for **Remote Debugger**. Go back to *LOCAL_SERVER* and press F5 to start debugging.
13. Go to *REMOTE_SERVER* and make sure that the `remote_2017` application is started.
14. Type `5` and press Enter. Go back to *HOST_MACHINE*.
15. Verify that the Debugger reaches the breakpoint at the line 27.
16. Open the Locals window and make sure that value `xxx` is 64-bit.

17. Press Shift+F5 to stop debugging and terminate the application on the remote server.
18. Open the project properties again, select **Debugging**, and set **Attach** to **Yes**.
19. Press Ctrl+F5 (remote execution).
20. Press F5 for attaching to a process.
21. Repeat steps 12-17 to check that the attach to process works correctly.
22. Open **Project** > **Properties** and set **Attach** to **No**.
23. Open **Project** > **Properties** and set **Connection** to **Remote with no authentication (Native only).**
24. Go to *REMOTE_SERVER* and switch debugging monitor to No Authentication mode using the **Tools** > **Options** menu.
25. Repeat steps 12-22 to check that Remote with no authentication (Native only) mode works correctly.

Results of Remote Debugging Exercise

The results of this remote debugging exercise for Visual Studio* 2017 are summarized in the table below.

Connection Type	Session	Result	Debugging Monitor status
Remote with Windows authentication	Debug (F5)	Works	Monitor should be run in Windows Authentication mode.
	Execute (Ctrl +F5)	Works	
	Attach to Process	Works	
Remote with no authentication (Native only)	Debug (F5)	Works	Monitor should be run in No Authentication mode.
	Execute (Ctrl +F5)	Works	
	Attach to Process	Works	

Part VI

Program Structure

Using Module (.mod) Files

One way to reduce potential confusion when you use the same source code in several projects is to organize the routines into modules and submodules. A module is a type of program unit that contains specifications of such entities as data objects, parameters, structures, procedures, and operators. These precompiled specifications and definitions can be used by one or more program units. Partial or complete access to the module entities is provided by the program's USE statement. Typical applications of modules are the specification of global data or the specification of a derived type and its associated operations.

Modules are excellent ways to organize programs. You can set up separate modules for:

- Commonly used routines
- Data definitions specific to certain operating systems
- System-dependent language extensions.

Submodules let you separate the interface of procedures from their implementation, making it easier to build applications and perform maintenance.

To USE a module, its source must first be compiled into a .mod file. The name of the .mod file is the module name, not the source file name. Similarly, submodules are compiled into .smod files - their parent modules or submodules must be compiled first. Compiling a module or submodule also creates an object file (.o or .obj) that must be included when linking the application.

When compiling a source that has a USE statement containing a module, the compiler looks for the corresponding .mod file in the same place it looks for include files. Submodule .smod files are used only when a child submodule is compiled.

Some programs require modules located in multiple directories. You can use compiler option `I` when you compile the program to specify the location of the `.mod` and `.smod` files that should be included in the program.

Intrinsic modules, defined by the Fortran standard, are supplied in a system directory alongside the compiler binaries and libraries. These have a .modintr file type that is searched only when the Fortran source has USE, INTRINSIC.

You can use compiler option `module`*path* to specify the directory in which to create the module files. If you do not use this option, module files are created in the current directory.

Directories are searched for `.mod` and `.smod` files in this order:

1. In the directory of the source file that contains the USE statement
2. In the directories specified by compiler option `module`*path*
3. In the current working directory
4. In the directories specified by compiler options `-I`*dir* (Linux* and macOS*) or `/include` (Windows*)
5. In the directories specified with environment variables `CPATH` or `INCLUDE`

6. In the standard system directories

If building as part of a Microsoft Visual Studio* project that has dependent projects, the Fortran build system automatically searches the dependent projects for compiled .mod files.

You need to make sure that the module files are created before they are referenced by another program or subprogram. If the module and its USE statement are in the same source file, the module must appear first. The `gen-dep` compiler option creates a list of compiled module files that a source file depends upon. This list can be used with build utilities such as `make` to specify dependencies and to ensure that files that use a module are recompiled when the module is modified and recompiled.

Compiling Programs with Modules

If a file being compiled has one or more modules defined in it, the compiler generates one or more `.mod` or `.smod` files, along with object files.

For example, consider that file `a.f90` contains modules defined as follows:

a.f90 Module

```
module test
integer:: a
contains
  subroutine f()
  end subroutine
end module test
module payroll
  ...
end module payroll
```

The following compiler command:

```
ifort -ca.f90
```

generates the following files:

- `test.mod`
- `payroll.mod`
- `a.o` (Linux* and macOS*)
- `a.obj` (Windows*)

The `.mod` files contain the necessary information regarding the modules that have been defined in the program `a.f90`.

The following example uses the program `mod_def.f90` which contains a module defined as follows:

mod_def.f90 Module

```
file: mod_def.f90
module definedmod
  ...
end module
```

Compile the program as follows:

```
ifort -c mod_def.f90
```

This produces the object files `mod_def.o` (Linux* and macOS*) or `mod_def.obj` (Windows*) and also the `.mod` file `definedmod.mod`, all in the current directory.

Using .mod files from another directory

```
file: use_mod_def.f90
program usemod
use definedmod
  ...
end program
```

To compile the above program, use compiler option `I` to specify the path to search and locate the `definedmod.mod` file.

See Also

gen-dep compiler option
I compiler option
module compiler option
SUBMODULE statement
USE statement

Using Include Files

Include files are brought into a program with the `#include` preprocessor directive or a Fortran INCLUDE statement.

Directories are searched for include files in this order:

1. In the directory of the source file that contains the include
2. In the current working directory
3. In the directories specified by compiler option `I`.
4. In the directory specified by compiler option `-isystem` (Linux* and macOS* only)
5. In the directories specified with environment variables `CPATH` (Linux* and macOS*) or `INCLUDE` (Windows*)
6. In the standard system directories

The locations of directories to be searched are known as the include file path. More than one directory can be specified in the include file path.

Specifying and Removing an Include File Path

You can use compiler option `I` to indicate the location of include files (and also module files).

To prevent the compiler from searching the default path specified by the `CPATH` or the `INCLUDE` environment variable, use compiler option `-X` or `/noinclude`.

You can specify these options in the configuration file, `ifort.cfg`, or on the command line.

For example, to direct the compiler to search a specified path instead of the default path, use the following command line:

Example

```
// Linux* and macOS*
  ifort -X -I/alt/include newmain.f

// Windows*
   ifort /noinclude /IC:\Project2\include newmain.f
```

See Also

I compiler option
INCLUDE statement
isystem compiler option
X compiler option

Advantages of Internal Procedures

Functions or subroutines that are used in only one program can be organized as internal procedures, following the CONTAINS statement of a program or module.

Internal procedures have the advantage of host association, that is, variables declared and used in the main program are also available to any internal procedure it may contain. For more information on procedures and host association, see Program Units and Procedures.

Internal procedures, like modules, provide a means of encapsulation. Modules can be used to store routines commonly used by many programs; internal procedures separate functions and subroutines whose use is limited or temporary.

See Also

CONTAINS statement
Program Units and Procedures

Implications for Array Copies

Fortran language semantics sometimes require the compiler to make a temporary copy of an array or array slice. Situations where this can occur include:

- Passing a non-contiguous array to a procedure that does not declare it as assumed-shape
- Array expressions, especially those involving intrinsic functions RESHAPE, PACK, and MERGE
- Assignments of arrays where the array appears on both the left and right-hand sides of the assignment
- Assignments of POINTER arrays

By default, these temporary values are created on the stack and, if large, may result in a stack overflow error at runtime. The size of the stack can be increased, but with limitations dependent on the operating system. If you use compiler option `heap-arrays`, it tells the compiler to use heap allocation, rather than the stack, for such temporary copies. Heap allocation adds a small amount of overhead when creating and removing the temporary values, but this is usually inconsequential in comparison to the rest of the code.

Performance can be further improved by entirely eliminating the need for a temporary copy. For the first case in the above list, passing a non-contiguous array to a procedure expecting a contiguous array, enabling compiler option `check:arg_temp_created` will produce a run-time informational message when the compiler determines that the argument being passed is not contiguous. A run-time test is performed and if the argument is contiguous, no copy is made. However, this option will not issue a diagnostic message for other uses of temporary copies.

One way to avoid temporary copies for array arguments is to change the called procedure to declare the array as assumed-shape, with the DIMENSION(:) attribute. Such procedures require an explicit interface to be visible to the caller. This is best provided by placing the called procedure in a module or a CONTAINS section. As an alternative, an INTERFACE block can be declared.

Use of POINTER arrays makes it difficult for the compiler to know if a temporary value can be avoided. Where possible, replace POINTER with ALLOCATABLE, especially as components of derived types. The language definition directs the compiler to assume that ALLOCATABLE arrays are contiguous and that they do not overlap other variables, unlike POINTERs.

Another situation where the temporary values can be created is for automatic arrays, where an array's bounds are dependent on a routine argument, use or host associated variable, or COMMON variable, and the array is a local variable in the procedure. As above, these automatic arrays are created on the stack by default; compiler option `heap-arrays` creates them on the heap. Consider making such arrays ALLOCATABLE instead; local ALLOCATABLE variables that do not have the SAVE attribute are automatically deallocated when the routine exits. For example, replace:

```
SUBROUTINE SUB (N)
INTEGER, INTENT(IN) :: N
REAL :: A(N)
```

with:

```
SUBROUTINE SUB(N)
INTEGER, INTENT(IN) :: N
REAL, ALLOCATABLE :: A(:)
ALLOCATE (A(N))
```

See Also

ALLOCATABLE statement
check compiler option
DIMENSION statement
heap-arrays compiler option
MERGE intrinsic function
PACK Function
POINTER - Fortran statement
RESHAPE intrinsic function
SAVE statement

Part VII

Optimization and Programming Guide

OpenMP* Support

The Intel® Compiler supports most of the OpenMP Version Technical Report 4: Version 5.0 Preview 1. For the complete OpenMP specification, see the OpenMP Application Program Interface Version TR4: Version 5.0 specification, which is available from the OpenMP web site (http://www.openmp.org; see *OpenMP Specifications* on that site). The descriptions of OpenMP language characteristics in this documentation often use terms defined in that specification.

The OpenMP API provides symmetric multiprocessing (SMP) with the following major features:

- Relieves you from implementing the low-level details of iteration space partitioning, data sharing, thread creation, scheduling, or synchronization.
- Provides the benefit of performance available from shared memory multiprocessor and multi-core processor systems on all supported Intel architectures, including those processors with Intel® Hyper-Threading Technology (Intel® HT Technology).

The compiler performs transformations to generate multithreaded code based on your placement of OpenMP directives in the source program, making it simple to add threading to existing software. The Intel compiler compiles parallel programs and supports the industry-standard OpenMP directives.

The compiler provides Intel-specific extensions to the OpenMP specification including run-time library routines and environment variables. A summary of the compiler options appear in the OpenMP Options Quick Reference.

Parallel Processing with OpenMP

To compile with the OpenMP API, add the directives in the form of the Fortran program comments to your code. The compiler processes the code and internally produces a multithreaded version which is then compiled into an executable with the parallelism implemented by threads that execute parallel regions or constructs.

Using Other Compilers

The OpenMP specification does not define interoperability of multiple implementations, so the OpenMP implementation supported by other compilers and OpenMP support in the Intel compiler might not be interoperable. Even if you compile and build the entire application with one compiler, be aware that different compilers might not provide OpenMP source compatibility that enable you to compile and link the same set of application sources with a different compiler and get the expected parallel execution results.

Adding OpenMP* Support to your Application

To add OpenMP* support to your application, do the following:

1. Add the appropriate OpenMP* directives to your source code.
2. Compile the application with the `Qopenmp` (Windows) or `qopenmp` (Linux* and macOS*) option.
3. For applications with large local or temporary arrays, you may need to increase the stack space available at run-time. In addition, you may need to increase the stack allocated to individual threads by using the `OMP_STACKSIZE` environment variable or by setting the corresponding library routines.

You can set other environment variables to control multi-threaded code execution.

OpenMP Directive Syntax

To add OpenMP* support to your application, first add appropriate OpenMP* directives to your source code.

OpenMP* directives use a specific format and syntax. Intel Extension Routines to OpenMP* describes the OpenMP* extensions to the specification that have been added to the Intel® Fortran Compiler.

The following syntax illustrates using the directives in your source.

Example

```
<prefix> <directive> [<clause>[[,]<clause>...]]
```

where:

- <*prefix*> - Required for all OpenMP* directives. For free form source input, the prefix is `!$OMP` only; for fixed form source input, the prefix is `!$OMP` or *C$OMP*.
- <*directive*> - A valid OpenMP* directive. Must immediately follow the prefix; for example: `!$OMP PARALLEL`.
- [<*clause*>] - Optional. Clauses can be in any order and repeated as necessary, unless otherwise restricted.
- [<*newline*>] - A required component of directive syntax. It precedes the structured block which is enclosed by this directive.
- [,]: Optional. Commas between more than one <*clause*> are optional.

The directives are interpreted as comments if you omit the `Qopenmp` (Windows) or `qopenmp` (Linux* and macOS*) option.

The OpenMP* constructs defining a parallel region have one of the following syntax forms:

Example

```
!$OMP <directive>
  <structured block of code>
!$OMP END <directive>
                  # OR
!$OMP <directive>
  <structured block of code>
                  # OR
!$OMP <directive>
```

The following example demonstrates one way of using an OpenMP* directive to parallelize a loop.

Example

```
subroutine simple_omp(a, N)
  use omp_lib
  integer :: N, a(N)
!$OMP PARALLEL DO
  do i = 1, N
```

Example

```
    a(i) = i*2
  end do
end subroutine simple_omp
```

Compile the Application

The `Qopenmp` (Windows) or `qopenmp` (Linux* and macOS*) option enables the parallelizer to generate multi-threaded code based on the OpenMP* directives in the source. The code can be executed in parallel on single processor, multi-processor, or multi-core processor systems.

The `Qopenmp` (Windows) or `qopenmp` (Linux* and macOS*) option works with both `-O0` (Linux* and macOS*) and `/Od` (Windows*) and with any optimization level of `O1`, `O2` and `O3`.

Specifying `-O0` (Linux* and macOS*) or `/Od` (Windows*) with the `Qopenmp` (Windows) or `qopenmp` (Linux* and macOS*) option helps to debug OpenMP* applications.

Compile your application using commands similar to those shown below:

Operating System	Syntax Example
Linux*	`ifort -qopenmp` *`source_file`*
macOS*	`ifort -qopenmp` *`source_file`*
Windows*	`ifort /Qopenmp` *`source_file`*

Assume that you compile the sample above, using commands similar to the following, where the `c` option instructs the compiler to compile the code without generating an executable:

Operating System	Extended Syntax Example
Linux*	`ifort -qopenmp -c parallel.f90`
macOS*	`ifort -qopenmp -c parallel.f90`
Windows*	`ifort /Qopenmp /c parallel.f90`

The compiler might return a message similar to the following:

Example

```
parallel.f90(20) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

Configure the OpenMP* Environment

Before you run the multi-threaded code, you can set the number of desired threads using the OpenMP* environment variable, `OMP_NUM_THREADS`.

See Also

c compiler option
O compiler option
OpenMP* Examples
qopenmp, Qopenmp compiler option

Parallel Processing Model

A program containing OpenMP* API compiler directives begins execution as a single thread, called the initial thread of execution. The initial thread executes sequentially until the first parallel construct is encountered.

In the OpenMP* API, the PARALLEL and END PARALLEL directives define the extent of the parallel construct. When the initial thread encounters a parallel construct, it creates a team of threads, with the initial thread becoming the master of the team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes all statements encountered during the execution of a construct by a thread, including all called routines.

When a thread encounters the end of a structured block enclosed by a parallel construct, the thread waits until all threads in the team have arrived. When that happens the team is dissolved, and only the master thread continues execution of the code following the parallel construct. The other threads in the team enter a wait state until they are needed to form another team. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

The following example illustrates, from a high level, the execution model for the OpenMP* constructs. The comments in the code explain the structure of each construct or section.

Example

```
PROGRAM MAIN           ! Begin serial execution.
  ...                  ! Only the initial thread executes.
 !$OMP PARALLEL        ! Begin a Parallel construct, form a team.
   ...                 ! This code is executed by each team member.
  !$OMP SECTIONS       ! Begin a worksharing construct.
    !$OMP SECTION      ! One unit of work.
     ...               !
    !$OMP SECTION      ! Another unit of work.
     ...               !
   !$OMP END SECTIONS  ! Wait until both units of work complete.
   ...                 ! More Replicated Code.
  !$OMP DO             ! Begin a worksharing construct,
     DO                !   each iteration is a unit of work.
     ...               ! Work is distributed among the team.
     END DO            !
  !$OMP END DO NOWAIT  ! End of worksharing construct, NOWAIT
                       !   is specified (threads need not wait).
                       ! This code is executed by each team member.
  !$OMP CRITICAL       ! Begin critical construct.
     ...               ! One thread executes at a time.
  !$OMP END CRITICAL   ! End the critical construct.
   ...                 ! This code is executed by each team member.
  !$OMP BARRIER        ! Wait for all team members to arrive.
   ...                 ! This code is executed by each team member.
 !$OMP END PARALLEL    ! End of parallel construct, disband team
                       !   and continue with serial execution.
  ...                  ! Possibly more parallel constructs.
END PROGRAM MAIN       ! End serial execution.
```

Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the static extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute portions of your program in parallel with only minimal changes to

the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

Example

```
subroutine F
...
!$OMP PARALLEL...
   call G
...
subroutine G
!$OMP DO... ! This is an orphaned directive.
...
```

This is an orphaned DO directive since the parallel region is not lexically present in subroutine `G` .

Data Environment Controls

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can privatize named global-lifetime objects by using THREADPRIVATE directive, or control data scope attributes by using the data environment clauses for directives that support them.

The data scope attribute clauses are:

- DEFAULT
- FIRSTPRIVATE
- IN_REDUCTION
- LASTPRIVATE
- LINEAR
- PRIVATE
- REDUCTION
- SHARED

The data copying clauses are:

- COPYIN
- COPYPRIVATE

The data motion clause is:

- MAP

The miscellaneous clauses are:

- ALIGNED
- COLLAPSE
- DEPEND
- DEVICE
- FINAL
- IF
- MERGEABLE
- NOWAIT
- PRIORITY
- UNTIED

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them; however, if you do not specify a data scope attribute clause on a directive, the behavior for the variable is determined by the default scoping rules, which are described in the OpenMP* API specification, for the variables affected by the directive.

Determining How Many Threads to Use

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree-based structures, and size of list based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer the number of threads to employ until application run-time when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system.

Avoid simultaneously using more threads than the number of processing units on the system. This situation causes the operating system to multiplex threads on the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has higher-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Use the NUM_THREADS clause on parallel regions to control the number of threads employed and use the IF clause on parallel regions to decide whether to employ multiple threads at all. The OMP_SET_NUM_THREADS routine can also be used, but it also affects parallel regions created by the calling thread. The NUM_THREADS clause is local in its effect, so it does not impact other parallel regions. The disadvantages of explicitly setting a number of threads are:

1. In a system with a large number of processors, your application will use some but not all of the processors.
2. In a system with a small number of processors, your application may force over subscription that results in poor performance.

The Intel OpenMP runtime will create the same number of threads as the available number of logical processors unless you use the OMP_SET_NUM_THREADS routine. To determine the actual limits, use OMP_GET_THREAD_LIMIT() and OMP_GET_MAX_ACTIVE_LEVELS(). Developers should carefully consider their thread usage and nesting of parallelism to avoid overloading the system. The `OMP_THREAD_LIMIT` environment variable limits the number of OpenMP* threads to use for the whole OpenMP* program. The `OMP_MAX_ACTIVE_LEVELS` environment variable limits the number of active nested parallel regions.

Binding Sets

The various binding sets describe which OpenMP constructs can be nested in which other OpenMP constructs and what effect that nesting has.

The binding region for an OpenMP construct is the enclosing region that determines the execution context and the scope of the effects of the directive:

- The binding region for an ORDERED construct is the innermost enclosing DO loop region.
- The binding region for a TASKWAIT construct is the innermost enclosing TASK region.
- For all other constructs for which the binding thread set is the current team or the binding task set is the current team tasks, the binding region is the innermost enclosing PARALLEL region.
- For constructs for which the binding task set is the generating task, the binding region is the region of the generating task.
- A PARALLEL construct need not be active nor explicit to be a binding region.
- A TASK construct need not be explicit to be a binding region.
- A region never binds to any region outside of the innermost enclosing parallel region.

The binding task set for an OpenMP construct is the set of tasks that are affected by, or provide the context for, the execution of a region. The binding task set for a given construct can be all tasks, the current team tasks, or the generating task.

The binding thread set for an OpenMP construct is the set of threads that are affected by, or provide the context for, the execution of a region. The binding thread set for a given construct can be all threads on a device, all threads in a contention group, the current team, or the encountering thread.

Controlling Thread Allocation

The `KMP_HW_SUBSET` and `KMP_AFFINITY` environment variables allow you to control how the OpenMP* runtime uses the hardware threads on the processors. These environment variables allow you to try different thread distributions on the cores of the processors and determine how these threads are bound to the cores. You can use the environment variables to work out what is optimal for your application.

The `KMP_HW_SUBSET` variable controls the allocation of hardware resources and the `KMP_AFFINITY` variable controls how the OpenMP threads are bound to those resources.

Controlling Thread Distribution

The `KMP_HW_SUBSET` variable controls the hardware resource that will be used by the program. This variable specifies the number of sockets to use, how many cores to use per socket and how many threads to assign per core. While specifying two threads per core often yields better performance than one thread per core, specifying three or four threads per core may or may not improve the performance. This variable enables you to conveniently measure the performance of up to four threads per core.

For example, you can determine the effects of assigning 24, 48, 72, or the maximum 96 OpenMP threads in a system with 24 cores by specifying the following variable settings:

To Assign This Number of Threads ...	... Use This Setting
24	`KMP_HW_SUBSET=24c,1t`
48	`KMP_HW_SUBSET=24c,2t`
72	`KMP_HW_SUBSET=24c,3t`
96	`KMP_HW_SUBSET=24c,4t`

NOTE
Take care when using the `OMP_NUM_THREADS` variable along with this variable. Using the `OMP_NUM_THREADS` variable can result in over or under subscription.

Controlling Thread Bindings

The `KMP_AFFINITY` variable controls how the OpenMP threads are bound to the hardware resources allocated by the `KMP_HW_SUBSET` variable. While this variable can be set to several binding or affinity types, the following are the recommended affinity types to use to run your OpenMP threads on the processor:

- *compact*: sequentially distribute the threads among the cores that share the same cache.
- *scatter*: distribute the threads among the cores without regard to the cache.

The following table shows how the threads are bound to the cores when you want to use three threads per core on two cores by specifying `KMP_HW_SUBSET=2c,3t`:

Affinity	OpenMP Threads on Core 0	OpenMP Threads on Core 1
KMP_AFFINITY=compact	0, 1, 2	3, 4, 5
KMP_AFFINITY=scatter	0, 2, 4	1, 3, 5

Determining the Best Setting

To determine the best thread distribution and bindings using these variables, use the following:

1. Ensure that your OpenMP code is working properly before using these environment variables.
2. Establish a baseline with your current OpenMP code to compare to the performance when you allocate the threads to a processor.
3. Measure the performance of distributing one, two, three, or four threads per core by use the KMP_HW_SUBSET variable.
4. Measure the performance of binding the threads to the cores by using the KMP_AFFINITY variable.

See Also

Thread Affinity Interface
Supported Environment Variables

OpenMP* Directives Summary

This is a summary of the OpenMP* directives supported in the Intel® Fortran Compiler. For detailed information about the OpenMP* API, see the *OpenMP Application Program Interface* Version TR4: Version 5.0 specification, which is available from the OpenMP* web site.

In the directive lists below, an OpenMP directive is qualified with the word **Directive** when the name of the directive is also used for one or more elements besides the directive. For example: FLUSH refers to a directive, a statement, and a subroutine.

PARALLEL Directive

Use this directive to form a team of threads and execute those threads in parallel.

Directive	Description
PARALLEL Directive (OpenMP*)	Defines a parallel region.

TASKING Directives

Use this directive for deferring execution.

Directive	Description
TASK	Defines a task region.
TASKLOOP	Specifies that the iterations of one or more associated DO loops should be executed in parallel using OpenMP* tasks. The iterations are distributed across tasks that are created by the construct and scheduled to be executed.

WORKSHARING Directives

Use these directives to share work among a team of threads.

Directive	Description
DO Directive	Identifies an iterative worksharing construct in which the iterations of the associated loop should be divided among threads in a team.
SECTIONS	Specifies that the enclosed `SECTION` directives define blocks of code to be divided among threads in a team. Each section is executed once by a thread in the team.
SINGLE	Specifies that a block of code is to be executed by only one thread in the team at a time.
WORKSHARE	Divides the work of executing a block of statements or constructs into separate units. It also distributes the work of executing the units to threads of the team so each unit is only executed once.

SYNCHRONIZATION Directives

Use these directives to synchronize between threads.

Directive	Description
ATOMIC	Ensures that a specific memory location is updated atomically; this prevents the possibility of multiple, simultaneous reading and writing of threads.
BARRIER	Synchronizes all the threads in a team. It causes each thread to wait until all of the other threads in the team have reached the barrier.
CRITICAL Directive	Restricts access to a block of code to only one thread at a time.
FLUSH Directive	Identifies synchronization points at which the threads in a team must provide a consistent view of memory.
MASTER	Specifies a block of code to be executed by the master thread of the team.
ORDERED	Specifies a block of code that the threads in a team must execute in the natural order of the loop iterations.
TASKGROUP	Specifies a wait for the completion of all child tasks of the current task and all of their descendant tasks.
TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
TASKYIELD	Specifies that the current task can be suspended at this point in favor of execution of a different task.

Data Environment Directive

Use this directive to give threads global private data.

Directive	Description
THREADPRIVATE	Specifies named common blocks to be private (local) to each thread; they are global within the thread.

Offload Target Control Directives

Use these directives to control execution on one or more offload targets. Offload is not supported on Windows* systems.

Directive	Description
DECLARE TARGET	Specifies named routines and variables that are created or mapped to a device.
DISTRIBUTE	Specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a teams construct.
TARGET Directive	Creates a device data environment and executes the construct on that device.
TARGET DATA	Maps variables to a device data environment for the extent of the region.
TARGET ENTER DATA	Specifies that variables are mapped to a device data environment.
TARGET EXIT DATA	Specifies that variables are unmapped from a device data environment.
TEAMS	Creates a league of thread teams inside a target region to execute a structured block in the master thread of each team.
TARGET UPDATE	Makes the list items in the device data environment consistent with their corresponding original list items.

Vectorization Directives

Use these directives to control execution on vector hardware.

Directive	Description
SIMD Directive (OpenMP*)	Transforms the loop into a loop that will be executed concurrently using SIMD instructions. The `EARLY_EXIT` clause is an Intel-specific extension of the OpenMP* specification. `EARLY_EXIT` Allows vectorization of multiple exit loops. When this clause is specified: • Each operation before last lexical early exit of the loop may be executed as if early exit were not triggered within the SIMD chunk. • After the last lexical early exit of the loop, all operations are executed as if the last iteration of the loop was found. • Each list item specified in the `LINEAR` clause is computed based on the last iteration number upon exiting the loop. • The last value for `LINEARs` and conditional `LASTPRIVATEs` are preserved with respect to scalar execution. • The last value for `REDUCTIONs` are computed as if the last iteration in the last SIMD chunk was executed up on exiting the loop. • The shared memory state may not be preserved with regard to scalar execution. • Exceptions are not allowed.

Directive	Description
DECLARE SIMD	Generates a SIMD procedure.

Cancellation Constructs

Directive	Description
CANCEL	Requests cancellation of the innermost enclosing region of the construct specified, and causes the encountering task to proceed to the end of the cancelled construct.
CANCELLATION POINT	Defines a point at which implicit or explicit tasks check to see if cancellation has been requested for the innermost enclosing region of the type specified.

Combined Directives

Use these directives as shortcuts for multiple directives in sequence. A combined construct is a shortcut for specifying one construct immediately nested inside another construct. A combined construct is semantically identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

A composite construct is composed of two constructs but does not have identical semantics to specifying one of the constructs immediately nested inside the other. A composite construct either adds semantics not included in the constructs from which it is composed or the nesting of the one construct inside the other is not conforming.

Directive	Description
DISTRIBUTE PARALLEL DO[1]	Specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.
DISTRIBUTE PARALLEL DO SIMD[1]	Specifies a loop that will be executed in parallel by multiple threads that are members of multiple teams. It will be executed concurrently using SIMD instructions.
DISTRIBUTE SIMD[1]	Specifies a loop that will be distributed across the master threads of the teams region. It will be executed concurrently using SIMD instructions.
DO SIMD[1]	Specifies a loop that can be executed concurrently using SIMD instructions.
PARALLEL DO	Provides an abbreviated way to specify a parallel region containing a single `DO` directive.
PARALLEL DO SIMD	Specifies a loop that can be executed concurrently using SIMD instructions. It provides a shortcut for specifying a PARALLEL construct containing one SIMD loop construct and no other statement.
PARALLEL SECTIONS	Provides an abbreviated way to specify a parallel region containing a single `SECTIONS` directive. The semantics are identical to explicitly specifying a `PARALLEL` directive immediately followed by a `SECTIONS` directive.

Directive	Description
PARALLEL WORKSHARE	Provides an abbreviated way to specify a parallel region containing a single WORKSHARE directive.
TARGET PARALLEL	Creates a device data environment in a parallel region and executes the construct on that device.
TARGET PARALLEL DO	Provides an abbreviated way to specify a TARGET construct that contains a PARALLEL DO construct and no other statement between them.
TARGET PARALLEL DO SIMD	Specifies a TARGET construct that contains a PARALLEL DO SIMD construct and no other statement between them.
TARGET SIMD	Specifies a TARGET construct that contains a SIMD construct and no other statement between them.
TARGET TEAMS	Creates a device data environment and executes the construct on the same device. It also creates a league of thread teams with the master thread in each team executing the structured block.
TARGET TEAMS DISTRIBUTE	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a TEAMS construct.
TARGET TEAMS DISTRIBUTE PARALLEL DO	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct.
TARGET TEAMS DISTRIBUTE PARALLEL DO SIMD	Creates a device data environment and then executes the construct on that device. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams created by a TEAMS construct. The loop will be distributed across the teams, which will be executed concurrently using SIMD instructions.
TARGET TEAMS DISTRIBUTE SIMD	Creates a device data environment and then executes the construct on that device. It also specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a teams construct. It will be executed concurrently using SIMD instructions.
TASKLOOP SIMD [1]	Specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using OpenMP* tasks.
TEAMS DISTRIBUTE	Creates a league of thread teams to execute the structured block in the master thread of each team. It also specifies that loop iterations will be distributed among the master threads of all thread teams in a league created by a TEAMS construct.
TEAMS DISTRIBUTE PARALLEL DO	Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.

Directive	Description
TEAMS DISTRIBUTE PARALLEL DO SIMD	Creates a league of thread teams to execute a structured block in the master thread of each team. It also specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams. The loop will be distributed across the master threads of the teams region, which will be executed concurrently using SIMD instructions.
TEAMS DISTRIBUTE SIMD	Creates a league of thread teams to execute the structured block in the master thread of each team. It also specifies a loop that will be distributed across the master threads of the teams.

Footnotes:

[1] This directive specifies a composite construct.

OpenMP* Library Support

OpenMP* Run-time Library Routines

OpenMP* provides run-time library routines to help you manage your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines let you dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable. These routines are all external procedures.

> **Caution**
> Running OpenMP runtime library routines may initialize the OpenMP runtime environment, which might cause a situation where subsequent programmatic setting of OpenMP environment variables has no effect. To avoid this situation, you can use the Intel extension routine `kmp_set_defaults()` to set OpenMP environment variables.

The compiler supports all the OpenMP* run-time library routines. Refer to the OpenMP* API specification for detailed information about using these routines.

Include the appropriate declarations of the routines in the program unit containing the routine by adding a statement similar to the following:

Example

```
use omp_lib
```

The compiler provides module files in the ../include (Linux* and macOS*) or ..\include (Windows*) directory of your compiler installation.

The integer parameter `openmp_version` in omp_lib.mod (and omp_lib.h, an INCLUDE version of omp_lib.mod) has the decimal value *yyyymm* where *yyyy* and *mm* are the year and month of the version of the OpenMP API supported by the current version of the compiler and libraries. See the predefined preprocessor symbol `_OPENMP` for more information.

NOTE
Some of the routines interfaces have offload equivalents. The offload equivalent takes two additional arguments to specify the target type and target number. For more information, see **Calling Functions on the CPU to Modify the Coprocessor's Execution Environment**.

The following table lists the keys that specify the data types of the dummy arguments for each of the listed routines:

Key	OMP_LIB Kind	BIND(C) Kind	KIND=
INTEGER (*int*)	OMP_INTEGER_KIND	C_INT	4
LOGICAL (*log*)	OMP_LOGICAL_KIND		4
REAL (*dp*)	DOUBLE PRECISION	C_DOUBLE	8
INTEGER(OMP_LOCK_KIND)		C_INTPTR_T	intptr_t <<1>>
INTEGER(OMP_NEST_LOCK_KIND)		C_INTPTR_T	intptr_t <<1>>
INTEGER (OMP_SCHED_KIND)	OMP_INTEGER_KIND	C_INT	4
INTEGER(OMP_PROC_BIND_KIND)	OMP_INTEGER_KIND	C_INT	4
INTEGER(OMP_LOCK_HINT_KIND)		C_INTPTR_T	intptr_t <<1>>

intptr_t is an integer that is large enough to hold a pointer (address). With the Intel® Fortran Compiler, this is INTEGER(4) when building a 32-bit application and INTEGER(8) when building a 64-bit application. It is the value returned by the Intel Fortran intrinsic INT_PTR_KIND().

Execution Environment Routines

Use these routines to monitor and influence threads and the parallel environment.

Routine	Description
`SUBROUTINE OMP_SET_NUM_THREADS(num_threads)` `INTEGER(int) num_threads`	Sets the number of threads to use for subsequent parallel regions created by the calling thread.
`SUBROUTINE OMP_SET_DYNAMIC(dynamic_threads)` `LOGICAL dynamic_threads`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If *dynamic_threads* is `.TRUE.`, dynamic threads are enabled. If *dynamic_threads* is `.FALSE.`, dynamic threads are disabled. Dynamic threads are disabled by default.
`SUBROUTINE OMP_SET_NESTED(nested)` `LOGICAL(log) nested`	Enables or disables nested parallelism. If *nested* is `.TRUE.`, nested parallelism is enabled. If *nested* is `.FALSE.`, nested parallelism is disabled. Nested parallelism is disabled by default.

Routine	Description
`INTEGER(int) FUNCTION OMP_GET_NUM_THREADS()`	Returns the number of threads that are being used in the current parallel region. This function does not necessarily return the value inherited by the calling thread from the `OMP_SET_NUM_THREADS()` function.
`INTEGER(int) FUNCTION OMP_GET_MAX_THREADS()`	Returns the number of threads available to subsequent parallel regions created by the calling thread.
`INTEGER(int) FUNCTION OMP_GET_THREAD_NUM()`	Returns the thread number of the calling thread, within the context of the current parallel region.
`INTEGER(int) FUNCTION OMP_GET_NUM_PROCS()`	Returns the number of processors available to the program.
`LOGICAL(log) FUNCTION OMP_IN_PARALLEL()`	Returns `.TRUE.` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `.FALSE.`.
`LOGICAL(log) FUNCTION OMP_IN_FINAL()`	Returns `.TRUE.` if called within a final task region; otherwise returns `.FALSE.`.
`LOGICAL(log) FUNCTION OMP_GET_DYNAMIC()`	Returns `.TRUE.` if dynamic thread adjustment is enabled, otherwise returns `.FALSE.`.
`LOGICAL FUNCTION OMP_GET_NESTED()`	Returns `.TRUE.` if nested parallelism is enabled, otherwise returns `.FALSE.`.
`INTEGER FUNCTION OMP_GET_THREAD_LIMIT()`	Returns the maximum number of simultaneously executing threads in an OpenMP* program.
`SUBROUTINE OMP_SET_MAX_ACTIVE_LEVELS(max_active_levels)` `INTEGER max_active_levels`	Limits the number of nested active parallel regions. The call is ignored if negative `max_active_levels` specified.
`INTEGER FUNCTION OMP_GET_MAX_ACTIVE_LEVELS()`	Returns the maximum number of nested active parallel regions.
`INTEGER FUNCTION OMP_GET_LEVEL()`	Returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region.
`INTEGER FUNCTION OMP_GET_ACTIVE_LEVEL()`	Returns the number of nested, active parallel regions enclosing the task that contains the call.

Routine	Description
`INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(level) INTEGER level`	Returns the thread number of the ancestor at a given nest level of the current thread.
`INTEGER FUNCTION OMP_GET_TEAM_SIZE(level) INTEGER level`	Returns the size of the thread team to which the ancestor or the current thread belongs for the specified nested level of the current thread..
`SUBROUTINE OMP_SET_SCHEDULE(kind,chunk_size) INTEGER(KIND=omp_sched_kind) kind INTEGER(int) chunk_size`	Determines the schedule of a worksharing loop that is applied when '`runtime`' is used as the schedule kind.
`SUBROUTINE OMP_GET_SCHEDULE(kind,chunk_size) INTEGER(KIND=omp_sched_chunk_size) kind INTEGER(int) chunk_size`	Returns the schedule of a worksharing loop that is applied when the '`runtime`' schedule is used.
`INTEGER(KIND=OMP_PROC_BIND_KIND) OMP_GET_PROC_BIND()`	Returns the currently active thread affinity policy, which is set by environment variable `OMP_PROC_BIND`. This policy is used for subsequent nested parallel regions.
`INTEGER(int) FUNCTION OMP_GET_NUM_PLACES()`	Returns the number of places available to the execution environment in the place list of the initial task, usually threads, cores, or sockets.
`INTEGER(int) FUNCTION OMP_GET_PLACE_NUM_PROCS(place_num) INTEGER(int) place_num`	Returns the number of processors associated with the place numbered `place_num`. The routine returns zero when `place_num` is negative or is greater than or equal to `OMP_GET_NUM_PLACES()`.
`SUBROUTINE OMP_GET_PLACE_PROC_IDS(place_num,ids) INTEGER(int) place_num INTEGER(int) ids(*)`	Returns the numerical identifiers of each processor associated with the place numbered place_num. The numerical identifiers are non-negative and their meaning is implementation defined. The numerical identifiers are returned in the array ids and their order in the array is implementation defined. ids must have at least `OMP_GET_PLACE_NUM_PROCS(place_num)` elements. The routine has no effect when `place_num` is greater than or equal to `OMP_GET_NUM_PLACES()`.
`INTEGER(int) FUNCTION OMP_GET_PLACE_NUM()`	Returns the place number of the place to which the encountering thread is bound. The returned value is between 0 and

Routine	Description
	OMP_GET_NUM_PLACES() - 1, inclusive. When the encountering thread is not bound to a place, the routine returns -1.
INTEGER(*int*) FUNCTION OMP_GET_DEFAULT_DEVICE()	Returns the default device number.
SUBROUTINE OMP_SET_DEFAULT_DEVICE(device_number) INTEGER(*int*) device_number	Sets the default device number.
INTEGER(*int*) FUNCTION OMP_GET_NUM_DEVICES()	Gets the number of target devices.
INTEGER(*int*) FUNCTION OMP_GET_NUM_TEAMS()	Gets the number of teams in the current teams region.
INTEGER(*int*) FUNCTION OMP_GET_TEAM_NUM()	Gets the team number of the calling thread.
LOGICAL(*log*) FUNCTION OMP_GET_CANCELLATION()	Returns .TRUE. if cancellation is enabled; otherwise, .FALSE.. This routine can be affected by the setting for environment variable OMP_CANCELLATION.
LOGICAL(*log*) FUNCTION OMP_IS_INITIAL_DEVICE()	Returns .TRUE. if the current task is running on the host device; otherwise, .FALSE..
INTEGER(*int*) FUNCTION OMP_GET_INITIAL_DEVICE()	Returns the device number of the host device. The value of the device number is implementation defined. If it is between 0 and OMP_GET_NUM_DEVICES()-1, then it is valid in all device constructs and routines; if it is outside that range, then it is only valid in the device memory routines and not in the DEVICE clause.
INTEGER(*int*) FUNCTION OMP_GET_MAX_TASK_PRIORITY()	Returns the maximum value that can be specified in the PRIORITY clause.

Lock Routines

Use these routines to affect OpenMP* locks.

Function	Description
SUBROUTINE OMP_INIT_LOCK(svar) INTEGER (KIND=OMP_LOCK_KIND)svar	Initializes the lock associated with the simple lock variable svar for use in subsequent calls.
SUBROUTINE OMP_INIT_LOCK_WITH_HINT(svar, *hint*) INTEGER (KIND=OMP_LOCK_KIND)svar INTEGER (KIND=OMP_LOCK_HINT_KIND)*hint*	Initializes the lock associated with svar to the unlocked state, optionally choosing a specific lock implementation based on *hint*.

Function	Description
`SUBROUTINE OMP_DESTROY_LOCK(svar)` `INTEGER(KIND=OMP_LOCK_KIND)svar`	Causes the lock specified by svar to become undefined or uninitialized. The lock must be initialized and not locked.
`SUBROUTINE OMP_SET_LOCK(svar)` `INTEGER(KIND=OMP_LOCK_KIND)svar`	Forces the executing thread to wait until the lock associated with svar is available. The thread is granted ownership of the lock when it becomes available. The lock must be initialized.
`SUBROUTINE OMP_UNSET_LOCK(svar)` `INTEGER(KIND=OMP_LOCK_KIND)svar`	Releases the executing thread from ownership of the lock associated with svar. The behavior is undefined if the executing thread does not own the lock associated with svar.
`LOGICAL(`*`log`*`) OMP_TEST_LOCK(svar)` `INTEGER(KIND=OMP_LOCK_KIND)svar`	Attempts to set the lock associated with svar. If successful, returns `.TRUE.`, otherwise returns `.FALSE.`. The lock must be initialized.
`SUBROUTINE OMP_INIT_NEST_LOCK(nvar)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar`	Initializes the nested lock associated with the nested lock variable nvar for use in the subsequent calls.

`SUBROUTINE`
`OMP_INIT_NEST_LOCK_WITH_HINT(nvar, `*`hint`*`)`
`INTEGER (KIND=OMP_NEST_LOCK_KIND)nvar`
`INTEGER (KIND=OMP_LOCK_HINT_KIND)`*`hint`*
	Initializes the nested lock associated with nvar to the unlocked state, optionally choosing a specific lock implementation based on *hint*. The nesting count for nvar is set to zero.

`SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar`	Causes the nested lock associated with nvar to become undefined or uninitialized. The lock must be initialized and not locked.
`SUBROUTINE OMP_SET_NEST_LOCK(nvar)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar`	Forces the executing thread to wait until the nested lock associated with nvar is available. If the thread already owns the lock, then the lock nesting count is incremented.The lock must be initialized.
`SUBROUTINE OMP_UNSET_NEST_LOCK(nvar)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar`	Releases the executing thread from ownership of the nested lock associated with nvar if the nesting count is zero; otherwise, the nesting count is decremented. Behavior is undefined if the executing thread does not own the nested lock associated with nvar.
`INTEGER(`*`int`*`) OMP_TEST_NEST_LOCK(nvar)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)nvar`	Attempts to set the nested lock specified by nvar. If successful, returns the nesting count, otherwise returns zero.

Timing Routines

Function	Description
`REAL (dp) FUNCTION OMP_GET_WTIME()`	Returns a double precision value equal to the elapsed wall clock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.
`REAL (dp) FUNCTION OMP_GET_WTICK()`	Returns a double precision value equal to the number of seconds between successive clock ticks.

The following parameter constants are defined in `OMP_LIB.MOD` and can be set or returned in the `KIND` dummy argument in `OMP_SET_SCHEDULE` and `OMP_GET_SCHEDULE`:

```
integer(omp_sched_kind), parameter :: omp_sched_static  = 1
integer(omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(omp_sched_kind), parameter :: omp_sched_guided  = 3
integer(omp_sched_kind), parameter :: omp_sched_auto    = 4
```

The following parameter constants are defined in `OMP_LIB.MOD` and represent values returned by `OMP_GET_PROC_BIND`:

```
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_false  = 0
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_true   = 1
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_master = 2
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_close  = 3
integer (omp_proc_bind_kind), parameter :: omp_proc_bind_spread = 4
```

The following parameter constants are defined in `OMP_LIB.MOD` and can be set in the `HINT` dummy argument in `OMP_INIT_LOCK_WITH_HINT` and `OMP_INIT_NEST_LOCK_WITH_HINT`:

```
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_none           = 0
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_uncontended    = 1
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_contended      = 2
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_nonspeculative = 4
integer (omp_lock_hint_kind), parameter :: omp_lock_hint_speculative    = 8
```

The hints can be combined by using the + operator in Fortran. The effect of the combined hint is implementation defined. Combining `omp_lock_hint_none` with any other hint is equivalent to specifying the other hint. The following restrictions apply to combined hints:

- Hints `omp_lock_hint_uncontended` and `omp_lock_hint_contended` cannot be combined.
- Hints `omp_lock_hint_nonspeculative` and `omp_lock_hint_nonspeculative` cannot be combined.

See Also

Intel Extension Routines to OpenMP*
Predefined Preprocessor Symbols

Intel® Compiler Extension Routines to OpenMP*

The Intel compiler implements the following group of routines as extensions to the OpenMP* run-time library:

- Get and set the execution environment
- Get and set the stack size for parallel threads
- Memory allocation
- Get and set the thread sleep time for the throughput execution mode

The Intel extension routines described in this section can be used for low-level tuning to verify that the library code and application are functioning as intended. These routines are generally not recognized by other OpenMP-compliant compilers, which may cause the link stage to fail in the other compiler. To execute these OpenMP* routines, use the `[Q]openmp-stubs` option.

In most cases, environment variables can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `OMP_STACKSIZE` environment variable rather than the `KMP_SET_STACKSIZE_S()` library routine.

NOTE
A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

Execution Environment

Function	Description
`SUBROUTINE KMP_SET_DEFAULTS(STRING) CHARACTER*(*) STRING`	Sets OpenMP* environment variables defined as a list of variables separated by "|" in the argument.
`SUBROUTINE KMP_SET_LIBRARY_THROUGHPUT()`	Sets execution mode to throughput, which is the default. Allows the application to determine the runtime environment. Use in multi-user environments.
`SUBROUTINE KMP_SET_LIBRARY_TURNAROUND()`	Sets execution mode to turnaround. Use in dedicated parallel (single user) environments.
`SUBROUTINE KMP_SET_LIBRARY_SERIAL()`	Sets execution mode to serial.
`SUBROUTINE KMP_SET_LIBRARY(LIBNUM) INTEGER (KIND=OMP_INTEGER_KIND) LIBNUM`	Sets execution mode indicated by the value passed to the function. Valid values are: • **1** - serial mode • **2** - turnaround mode • **3** - throughput mode Call this routine before the first parallel region is executed.
`FUNCTION KMP_GET_LIBRARY() INTEGER (KIND=OMP_INTEGER_KIND) KMP_GET_LIBRARY`	Returns a value corresponding to the current execution mode: • **1** - serial • **2** - turnaround • **3** - throughput

Stack Size

Function	Description
`FUNCTION KMP_GET_STACKSIZE_S() INTEGER(KIND=KMP_SIZE_T_KIND) & KMP_GET_STACKSIZE_S`	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with

Function	Description
	KMP_SET_STACKSIZE_S() routine, prior to the first parallel region or via the KMP_STACKSIZE environment variable.
FUNCTION KMP_GET_STACKSIZE() INTEGER KMP_GET_STACKSIZE	Provided for backwards compatibility only. Use KMP_GET_STACKSIZE_S() routine for compatibility across different families of Intel processors.
SUBROUTINE KMP_SET_STACKSIZE_S(*size*) INTEGER (KIND=KMP_SIZE_T_KIND)*size*	Sets to *size* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the KMP_STACKSIZE environment variable. In order for KMP_SET_STACKSIZE_S() to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
SUBROUTINE KMP_SET_STACKSIZE_S(*size*) INTEGER *size*	Provided for backward compatibility only. Use KMP_SET_STACKSIZE_S(*size*) for compatibility across different families of Intel® processors.

Memory Allocation

The Intel compiler implements a group of memory allocation routines as an extension to the OpenMP* run-time library to enable threads to allocate memory from a heap local to each thread. These routines are: KMP_MALLOC(), KMP_CALLOC(), and KMP_REALLOC().

The memory allocated by these routines must also be freed by the KMP_FREE() routine. While you can allocate memory in one thread and then free that memory in a different thread, this mode of operation incurs a slight performance penalty.

Working with the local heap might lead to improved application performance because synchronization is not required.

Function	Description
FUNCTION KMP_MALLOC(*size*) INTEGER(KIND=KMP_POINTER_KIND)KMP_MALLOC INTEGER(KIND=KMP_SIZE_T_KIND)*size*	Allocate memory block of *size* bytes from thread-local heap.
FUNCTION KMP_CALLOC(*nelem*,*elsize*) INTEGER(KIND=KMP_POINTER_KIND)KMP_CALLOC INTEGER(KIND=KMP_SIZE_T_KIND)*nelem* INTEGER(KIND=KMP_SIZE_T_KIND)*elsize*	Allocate array of *nelem* elements of size *elsize* from thread-local heap.
FUNCTION KMP_REALLOC(*ptr*, *size*) INTEGER(KIND=KMP_POINTER_KIND)KMP_REALLOC INTEGER(KIND=KMP_POINTER_KIND) *ptr* INTEGER(KIND=KMP_SIZE_T_KIND)*size*	Reallocate memory block at address *ptr* and *size* bytes from thread-local heap.
SUBROUTINE KMP_FREE(*ptr*) INTEGER (KIND=KMP_POINTER_KIND)*ptr*	Free memory block at address *ptr* from thread-local heap.

Function	Description
	Memory must have been previously allocated with `KMP_MALLOC()`, `KMP_CALLOC()`, or `KMP_REALLOC()`.

Thread Sleep Time

In the throughput OpenMP* Support Libraries, threads wait for new parallel work at the ends of parallel regions, and then sleep, after a specified period of time. This time interval can be set by the `KMP_BLOCKTIME` environment variable or by the `KMP_SET_BLOCKTIME()` function.

Function	Description
`FUNCTION KMP_GET_BLOCKTIME()` `INTEGER KMP_GET_BLOCKTIME`	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the `KMP_BLOCKTIME` environment variable or by `KMP_SET_BLOCKTIME()`.
`FUNCTION KMP_SET_BLOCKTIME(`*`msec`*`)` `INTEGER` *`msec`*	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. This routine affects the block time setting for the calling thread and any OpenMP* team threads formed by the calling thread. The routine does not affect the block time for any other threads.

See Also

openmp-stubs, Qopenmp-stubs compiler option
OpenMP* Run-time Library Routines
OpenMP* Support Libraries

OpenMP* Support Libraries

The Intel® Compiler provides support libraries for OpenMP*. There are several kinds of libraries:

- **Performance:** supports parallel OpenMP* execution.
- **Stubs:** supports serial execution of OpenMP* applications.

Each kind of library is available for both dynamic and static linking on Linux* and macOS* operating systems. Only dynamic linking is supported on Windows* operating systems.

Performance Libraries

To use these libraries, specify the `[Q]openmp` compiler option.

Options that use OpenMP* are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP* constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Operating System	Dynamic Link	Static Link
Linux*	`libiomp5.so`	`libiomp5.a`
macOS*	`libiomp5.dylib`	`libiomp5.a`
Windows*	`libiomp5md.lib` `libiomp5md.dll`	None

Many routines in the OpenMP* support libraries are more optimized for Intel® microprocessors than for non-Intel microprocessors.

Stubs Libraries

To use these libraries, specify `-[Q]openmp-stubs` compiler option. These allow you to compile OpenMP* applications in serial mode and provide stubs for OpenMP* routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux*	`libiompstubs5.so`	`libiompstubs5.a`
macOS*	`libiompstubs5.dylib`	`libiompstubs5.a`
Windows*	`libiompstubs5md.lib` `libiompstubs5md.dll`	None

Execution modes

The Intel® Compiler enables you to run an application under different execution modes specified at run time; the libraries support the turnaround, throughput, and serial modes. Use the `KMP_LIBRARY`environment variable to select the modes at run time.

Mode	Description
throughput (default)	The throughput mode allows the program to yield to other running programs and adjust resource usage to produce efficient execution in a dynamic environment. In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads yield to other threads while waiting for more parallel work. After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Until more parallel work becomes available, sleeping allows processor and resources to be used for other work by non-OpenMP threaded code that may execute between parallel regions, or by other applications. The amount of time to wait before sleeping is set either by the `KMP_BLOCKTIME` environment variable or by the `KMP_SET_BLOCKTIME()` function. A small blocktime value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger blocktime value may be more appropriate if threads are to be reserved solely for use for OpenMP* execution, but may penalize other concurrently-running OpenMP* or threaded applications.

Mode	Description
turnaround	The turnaround mode is designed to keep active all processors involved in the parallel computation, which minimizes execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads (although they are still subject to `KMP_BLOCKTIME` control). In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively use all processors all of the time. **NOTE** Avoid over-allocating system resources. The condition can occur if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.
serial	The serial mode forces parallel applications to run as a single thread.

See Also

openmp, Qopenmp compiler option
openmp-stubs, Qopenmp-stubs compiler option
Supported Environment Variables

Using the OpenMP* Libraries

This section describes the steps needed to set up and use the OpenMP* Libraries from the command line. On Windows* systems, you can also build applications compiled with the OpenMP libraries in the Microsoft Visual Studio* development environment.

For a list of the options and libraries used by the OpenMP* libraries, see OpenMP* Support Libraries.

Set up your environment for access to the Intel® Fortran Compiler to ensure that the appropriate OpenMP* library is available during linking. On Windows* systems, you can either execute the appropriate batch (`.bat`) file or use the command-line window supplied in the compiler program folder that already has the environment set up. On Linux* and macOS* systems, you can source the appropriate script file (`ifortvars` file).

To use the `gfortran` compiler with the Intel OpenMP library along with the OpenMP* API functions, do the following:

1. Use the `use omp_lib` statement to compile the `omp_lib.f90` source file, which is in the Intel®Fortran Compiler `include` directory.
2. Add the `-I` option to the compile command line with appropriate path to the directory containing the resulting module file.

During compilation, ensure that the version of `omp_lib.h` or `omp_lib.mod` used when compiling is the version provided by that compiler.

Caution
Be aware that when using the gcc* or Microsoft* compiler, you may inadvertently use inappropriate header/module files. To avoid this, copy the header/module file(s) to a separate directory and put it in the appropriate `include` path using the `-I` option.

If a program uses data structures or classes that contain members with data types defined in `omp_lib.h` file, then source files that use those data structures should all be compiled with the same `omp_lib.h` file.

The command for the Intel® Fortran Compiler is `ifort`.

For information on the OpenMP* libraries and options used by the Intel® Fortran Compiler, see OpenMP* Support Libraries.

Command-Line Examples, Windows*

To compile and link (build) the entire application with one command using the Compatibility libraries, specify the following Intel® Fortran Compiler command:

Type of File	Commands
Fortran source, dynamic link	`ifort /MD /Qopenmp hello.f90`

When using the Microsoft* Visual C++* compiler, you should link with the Intel OpenMP compatibility library. You need to avoid linking the Microsoft* OpenMP run-time library (`vcomp`) and explicitly pass the name of the Intel OpenMP compatibility library as linker options (following `/link`):

Type of File	Commands
C source, dynamic link	`cl /MD /openmp hello.c /link /nodefaultlib:vcomp libiomp5md.lib`

You can also use the Intel® Fortran Compiler with the Visual C++* compiler to compile parts of the application and create object files (object-level interoperability). In this example, the Intel® Fortran Compiler compiles and links the entire application:

Type of File	Commands

Mixed C and Fortran sources, dynamic link	`cl /MD /openmp /c f1.c f2.c`
	`ifort /MD /Qopenmp /c f3.f f4.f`
	`ifort /MD /Qopenmp f1.obj f2.obj f3.obj f4.obj /Feapp / link /nodefaultlib:vcomp`

The first command produces two object files compiled by Visual C++* compiler, and the second command produces two more object files compiled by the Intel® Fortran Compiler. The final command links all four object files into an application.

Alternatively, the third line below uses the Visual C++* linker to link the application and specifies the Compatibility library `libiomp5md.lib` at the end of the third command:

Type of File	Commands

Mixed C and Fortran sources, dynamic link	`cl /MD /openmp /c f1.c f2.c`
	`ifort /MD /Qopenmp /c f3.f f4.f`
	`link f1.obj f2.obj f3.obj f4.obj /out:app.exe / nodefaultlib:vcomp libiomp5md.lib`

The following example shows the use of interprocedural optimization by the Intel® Fortran Compiler on several files, the Visual C++* compiler compiles several files, and the Visual C++* linker links the object files to create the executable:

Type of File	Commands
Mixed C and Fortran sources, dynamic link	`ifort /MD /Qopenmp /O3 /Qipo /Qipo-c f1.f f2.f f3.f` `cl /MD /openmp /O2 /c f4.c f5.c`

Type of File	Commands
	`cl /MD /openmp /O2 ipo_out.obj f4.obj f5.obj /Feapp /link /nodefaultlib:vcomp libiomp5md.lib`

The first command uses the Intel® Fortran Compiler to produce an optimized multi-file object file named `ipo_out.obj` by default (the `/Fe` option is not required). The second command uses the Visual C++* compiler to produce two more object files. The third command uses the Visual C++* `cl` command to link all three object files using the Intel® Fortran Compiler OpenMP library.

Command-Line Examples, Linux*

To compile and link (build) the entire application with one command using the Intel OpenMP libraries, specify the following Intel® Fortran Compiler command on Linux* platforms:

Type of File	Commands
Fortran source	`ifort -qopenmp hello.f90`

By default, the Intel® Fortran Compiler performs a dynamic link of the OpenMP* libraries. To perform a static link (not recommended), add the option `-qopenmp-link=static`. The Intel® Fortran Compiler option `-qopenmp-link` controls whether the linker uses static or dynamic OpenMP* libraries on Linux* and macOS* systems (default is `-qopenmp-link=dynamic`).

You can also use the Intel® C++ Compiler `icc`/`icpc` with `gcc`/`g++` compilers to compile parts of the application and create object files (object-level interoperability).

In this example, `gcc` compiles the C file `foo.c` (the `gcc` option `-fopenmp` enables OpenMP* support), and the Intel® Fortran Compiler links the application using the Intel OpenMP library:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c`
	`ifort -qopenmp foo.o`

When using `gcc` or `g++` compiler to link the application with the Intel® Fortran Compiler OpenMP compatibility library, you need to explicitly pass the Intel OpenMP library name using the `-l` option, the Linux* `pthread` library using the `-l` option, and path to the Intel® libraries where the Intel® C++ compiler is installed using the `-L` option:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c bar.c`
	`gcc foo.o bar.o -liomp5 -lpthread -L<icc_dir>/lib`

You can mix object files, but it is easier to use the Intel® Fortran Compiler to link the application so you do not need to specify the `gcc-l` option, `-L` option, and the `-lpthread` option:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c`
	`icc -qopenmp -c bar.c` (Linux* and macOS*)
	`ifort -qopenmp foo.o bar.o` (Linux* and macOS*)

You can mix OpenMP* object files compiled with gcc, the Intel® C++ Compiler, and the Intel® Fortran Compiler.

NOTE
You cannot mix object files compiled by the Intel® Fortran compiler and the `gfortran` compiler.

The table illustrates examples of using the Intel® Fortran compiler to link all the objects:

Type of File	Commands

Mixed C and Fortran sources	`icc -qopenmp -c ibar.c`
	`gcc -fopenmp -c gbar.c`
	`ifort -qopenmp -c foo.f`
	`ifort -qopenmp foo.o ibar.o gbar.o`

When using the Intel® Fortran compiler, if the main program does not exist in a Fortran object file that is compiled by the Intel® Fortran Compiler `ifort`, specify the `-nofor-main` option on the `ifort` command line during linking.

NOTE
Do not mix objects created by the Intel® Fortran Compiler (`ifort`) with the GNU Fortran Compiler (`gfortran`); instead, recompile all Fortran sources with the same Fortran compiler. The GNU Fortran Compiler is only available on Linux operating systems.

Similarly, you can mix object files compiled with the Intel® C++ Compiler, the GNU C/C++ compiler, and the GNU Fortran Compiler (`gfortran`), if you link with the GNU Fortran Compiler (`gfortran`). When using GNU `gfortran` compiler to link the application with the Intel® Fortran Compiler OpenMP compatibility library, you need to explicitly pass the Intel OpenMP compatibility library name and the Intel `irc` libraries using the `-l` options, the Linux* `pthread` library using the `-l` option, and path to the Intel® libraries where the Intel® C++ Compiler is installed using the `-L`. option. You do not need to specify the `-fopenmp` option on the link line:

Type of File	Commands

Mixed C and GNU Fortran sources	`icc -qopenmp -c ibar.c`
	`gcc -fopenmp -c gbar.c`
	`gfortran -fopenmp -c foo.f`
	`gfortran foo.o ibar.o gbar.o -lirc -liomp5 -lpthread -lc -L<icc_dir>/lib`

Alternatively, you could use the Intel® Fortran Compiler to link the application, but need to pass multiple `gfortran` libraries using the `-l` options on the link line:

Type of File	Commands

Mixed C and Fortran sources	`gfortran -fopenmp -c foo.f`
	`icc -qopenmp -c ibar.c`
	`ifort -qopenmp foo.o ibar.o -lgfortranbegin -lgfortran`

Command-Line Examples, macOS*

To compile and link (build) the entire application with one command using the Intel OpenMP libraries, specify the following Intel® Fortran Compiler command on macOS* platforms:

Type of File	Commands
Fortran source	`ifort -qopenmp hello.f90`

By default, the Intel® Fortran Compiler performs a dynamic link of the OpenMP* libraries. To perform a static link (not recommended), add the option `-qopenmp-link=static`. The Intel® Fortran Compiler option `-qopenmp-link` controls whether the linker uses static or dynamic OpenMP* libraries on Linux* and macOS* systems (default is `-qopenmp-link=dynamic`).

You can also use the Intel® C++ Compiler `icc`/`icpc`/`icl` with `gcc`/`g++` compilers to compile parts of the application and create object files (object-level interoperability).

NOTE
Mixed compiling using Intel® C++ Compiler with GCC compilers is possible only on older macOS* platforms. The latest macOS* v10.x platforms do not include the GCC compiler, and the Clang compiler included in the latest macOS* 10.x does not support OpenMP* implementation. Future versions of Clang compiler may support OpenMP* implementation.

In this example, `icl` or `icc` compiles the C file `foo.c`, `icpc` compiles the file `ifoo.cpp`, the option `Qopenmp` (Windows*) or `qopenmp` (Linux* or macOS*) enables OpenMP* support, and the Intel® Fortran Compiler links the application using the Intel OpenMP library:

Type of File	Commands
C source	`icc -qopenmp -c foo.c`
C++ source	`icpc -qopenmp -c ifoo.cpp`
	`ifort -qopenmp foo.o ifoo.o`

NOTE
GCC is absent on macOS* v10.9 and later (that is, Xcode 5.x and later). However, you may install GCC along with the Intel® C++ Compiler.

You can mix object files, but it is easier to use the Intel compiler to link the application so you do not need to specify the `gcc-l` option, `-L` option, and the `-lpthread` option:

Type of File	Commands
C source	`icc -qopenmp -c bar.c`
	`ifort -qopenmp -c foo.f90`
	`ifort -qopenmp foo.o bar.o`

When using the Intel® Fortran compiler, if the main program does not exist in a Fortran object file that is compiled by the Intel® Fortran Compiler `ifort`, specify the `-nofor-main` option on the `ifort` command line during linking.

Alternatively, you could use the Intel® Fortran Compiler to link the application, but need to pass multiple `gfortran` libraries using the `-l` options on the link line:

Type of File	Commands
Mixed C and Fortran sources	`gfortran -fopenmp -c foo.f`

Type of File	Commands
	icc -qopenmp -c ibar.c ifort -qopenmp foo.o ibar.o -lgfortranbegin -lgfortran

See Also

openmp, Qopenmp compiler option
Using IPO
OpenMP* Support Libraries
qopenmp-link, Qopenmp-link compiler option

Thread Affinity Interface (Linux* and Windows*)

The Intel® runtime library has the ability to bind OpenMP* threads to physical processing units. The interface is controlled using the KMP_AFFINITY environment variable. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the application speed.

Thread affinity restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect on the execution speed of a program.

Thread affinity is supported on Windows* systems and versions of Linux* systems that have kernel support for thread affinity, but is not supported by macOS*.

The Intel OpenMP runtime library has the ability to bind OpenMP* threads to physical processing units. There are three types of interfaces you can use to specify this binding, which are collectively referred to as the Intel OpenMP Thread Affinity Interface:

- The high-level affinity interface uses an environment variable to determine the machine topology and assigns OpenMP* threads to the processors based upon their physical location in the machine. This interface is controlled entirely by the KMP_AFFINITY environment variable.
- The mid-level affinity interface uses an environment variable to explicitly specifies which processors (labeled with integer IDs) are bound to OpenMP* threads. This interface provides compatibility with the gcc* GOMP_AFFINITY environment variable, but you can also invoke it by using the KMP_AFFINITY environment variable. The GOMP_AFFINITY environment variable is supported on Linux* systems only, but users on Windows* or Linux* systems can use the similar functionality provided by the KMP_AFFINITY environment variable.
- The low-level affinity interface uses APIs to enable OpenMP* threads to make calls into the OpenMP* runtime library to explicitly specify the set of processors on which they are to be run. This interface is similar in nature to sched_setaffinity and related functions on Linux* systems or to SetThreadAffinityMask and related functions on Windows* systems. In addition, you can specify certain options of the KMP_AFFINITY environment variable to affect the behavior of the low-level API interface. For example, you can set the affinity type KMP_AFFINITY to disabled, which disables the low-level affinity interface, or you could use the KMP_AFFINITY or GOMP_AFFINITY environment variables to set the initial affinity mask, and then retrieve the mask with the low-level API interface.

The following terms are used in this section:

- The total number of processing elements on the machine is referred to as the number of *OS thread contexts.*
- Each processing element is referred to as an Operating System processor, or *OS proc*.
- Each OS processor has a unique integer identifier associated with it, called an *OS proc ID*.
- The term *package* refers to a single or multi-core processor chip.
- The term *OpenMP* Global Thread ID* (GTID) refers to an integer which uniquely identifies all threads known to the Intel OpenMP runtime library. The thread that first initializes the library is given GTID 0. In the normal case where all other threads are created by the library and when there is no nested parallelism, then *n-threads-var* - 1 new threads are created with GTIDs ranging from 1 to *ntheads-var* - 1, and each thread's GTID is equal to the OpenMP* thread number returned by function

omp_get_thread_num(). The high-level and mid-level interfaces rely heavily on this concept. Hence, their usefulness is limited in programs containing nested parallelism. The low-level interface does not make use of the concept of a GTID, and can be used by programs containing arbitrarily many levels of parallelism.

Some environment variables are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

The KMP_AFFINITY Environment Variable

NOTE
You must set the KMP_AFFINITY environment variable before the first parallel region, or certain API calls including omp_get_max_threads(), omp_get_num_procs() and any affinity API calls, as described in Low Level Affinity API, below.

The KMP_AFFINITY environment variable uses the following general syntax:

Syntax
KMP_AFFINITY=[<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

For example, to list a machine topology map, specify KMP_AFFINITY=verbose,none to use a *modifier* of verbose and a *type* of none.

The following table describes the supported specific arguments.

Argument	Default	Description
modifier	noverbose respect granularity=core	Optional. String consisting of keyword and specifier. • granularity=<*specifier*> takes the following specifiers: fine, thread, core, and tile • norespect • noverbose • nowarnings • proclist={<*proc-list*>} • respect • verbose • warnings The syntax for <*proc-list*> is explained in mid-level affinity interface. **NOTE** On Windows* with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows*). Otherwise, the respect affinity modifier is used.

Argument	Default	Description
type	none	Required string. Indicates the thread affinity to use. • balanced • compact • disabled • explicit • none • scatter • logical (deprecated; instead use compact, but omit any permute value) • physical (deprecated; instead use scatter, possibly with an offset value) The logical and physical types are deprecated but supported for backward compatibility.
permute	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.
offset	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.

Affinity Types

Type is the only required argument.

type = none (default)

Does not bind OpenMP* threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP* thread affinity interface to determine machine topology. Specify KMP_AFFINITY=verbose,none to list a machine topology map.

type = balanced

Places threads on separate cores until all cores have at least one thread, similar to the scatter type. However, when the runtime must use multiple hardware thread contexts on the same core, the balanced type ensures that the OpenMP* thread numbers are close to each other, which scatter does not do. This affinity type is supported on the CPU only for single socket systems.

NOTE
The OpenMP* environment variable OMP_PROC_BIND=spread is similar to KMP_AFFINITY=balanced and is available on all platforms, including multi-socket CPU systems.

type = compact

Specifying `compact` assigns the OpenMP* thread <*n*>+1 to a free thread context as close as possible to the thread context where the <*n*> OpenMP* thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.

type = disabled

Specifying `disabled` completely disables the thread affinity interfaces. This forces the OpenMP* run-time library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as `kmp_set_affinity` and `kmp_get_affinity`, which have no effect and will return a nonzero error code.

type = explicit

Specifying `explicit` assigns OpenMP* threads to a list of OS proc IDs that have been explicitly specified by using the proclist= modifier, which is required for this affinity type. See Explicitly Specifying OS Proc IDs (GOMP_CPU_AFFINITY).

type = scatter

Specifying `scatter` distributes the threads as evenly as possible across the entire system. `scatter` is the opposite of `compact`; so the leaves of the node are most significant when sorting through the machine topology map.

Deprecated Types: logical and physical

Types `logical` and `physical` are deprecated and may become unsupported in a future release. Both are supported for backward compatibility.

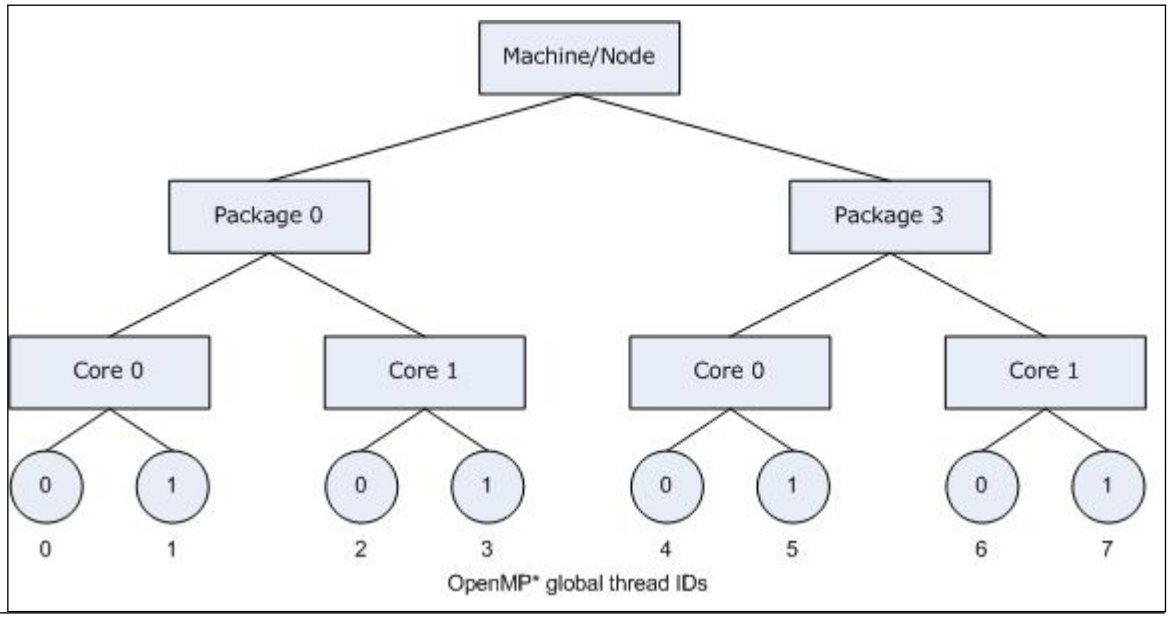
For `logical` and `physical` affinity types, a single trailing integer is interpreted as an `offset` specifier instead of a `permute` specifier. In contrast, with `compact` and `scatter` types, a single trailing integer is interpreted as a `permute` specifier.

- Specifying `logical` assigns OpenMP* threads to consecutive logical processors, which are also called hardware thread contexts. The type is equivalent to `compact`, except that the `permute` specifier is not allowed. Thus, `KMP_AFFINITY=logical,n` is equivalent to `KMP_AFFINITY=compact,0,n` (this equivalence is true regardless of the whether or not a `granularity=fine` modifier is present).
- Specifying `physical` assigns threads to consecutive physical processors (cores). For systems where there is only a single thread context per core, the type is equivalent to `logical`. For systems where multiple thread contexts exist per core, `physical` is equivalent to `compact` with a `permute` specifier of 1; that is, `KMP_AFFINITY=physical,n` is equivalent to `KMP_AFFINITY=compact,1,n` (regardless of the whether or not a `granularity=fine` modifier is present). This equivalence means that when the compiler sorts the map it should permute the innermost level of the machine topology map to the outermost, presumably the thread context level. This type does not support the `permute` specifier.

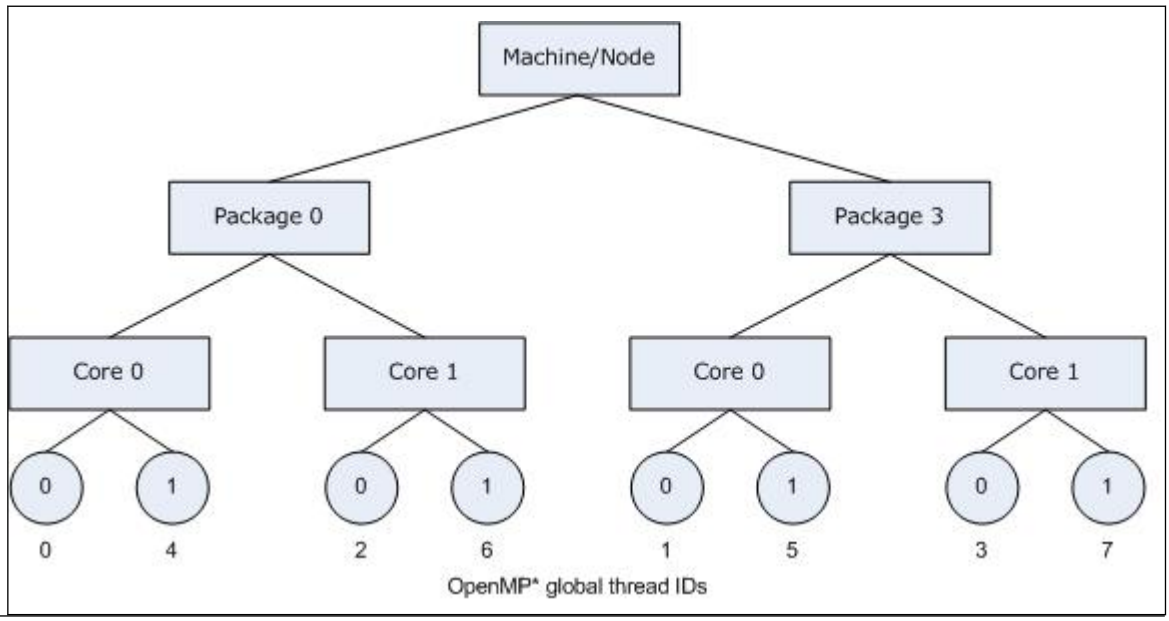
Examples of Types `compact` and `scatter`

The following figure illustrates the topology for a machine with two processors, and each processor has two cores; further, each core has Intel® Hyper-Threading Technology (Intel® HT Technology) enabled.

The following figure also illustrates the binding of OpenMP* thread to hardware thread contexts when specifying `KMP_AFFINITY=granularity=fine,compact`.



Specifying `scatter` on the same system as shown in the figure above, the OpenMP* threads would be assigned the thread contexts as shown in the following figure, which shows the result of specifying `KMP_AFFINITY=granularity=fine,scatter`.



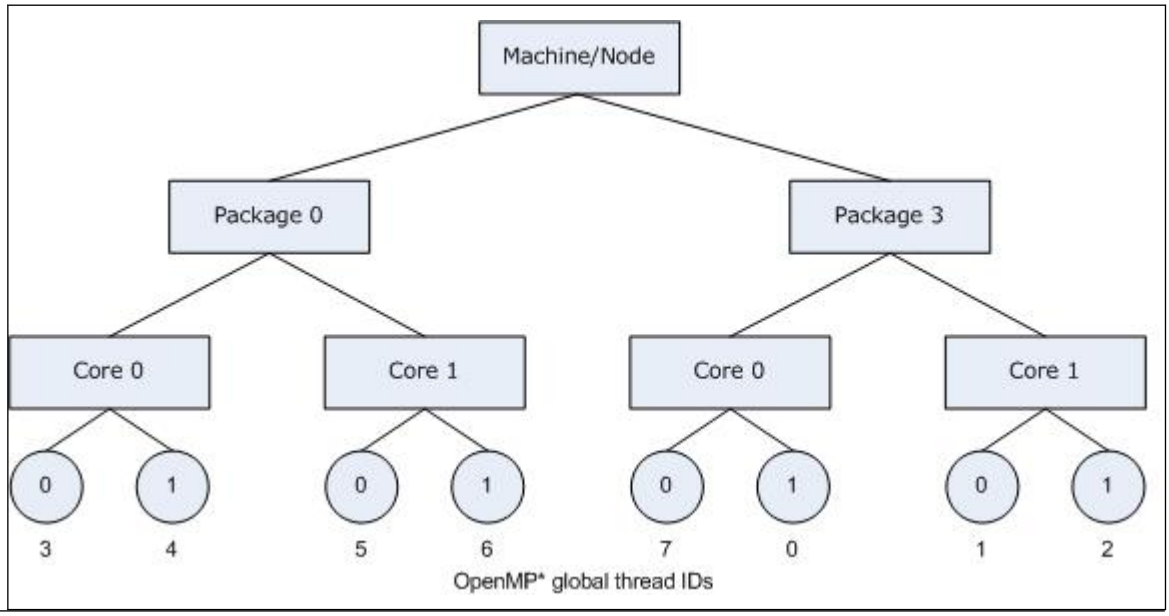
permute and offset combinations

For both `compact` and `scatter`, `permute` and `offset` are allowed; however, if you specify only one integer, the compiler interprets the value as a permute specifier. Both `permute` and `offset` default to 0.

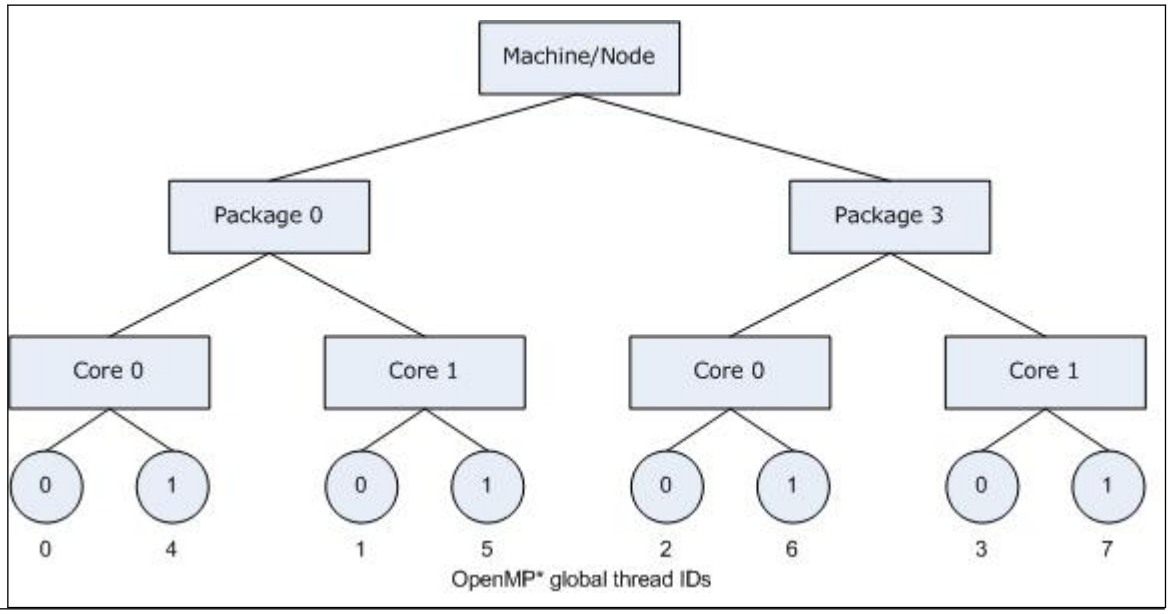
The `permute` specifier controls which levels are most significant when sorting the machine topology map. A value for permute forces the mappings to make the specified number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.

The `offset` specifier indicates the starting position for thread assignment.

The following figure illustrates the result of specifying `KMP_AFFINITY=granularity=fine,compact,0,3`.



Consider the hardware configuration from the previous example, running an OpenMP* application which exhibits data sharing between consecutive iterations of loops. We would therefore like consecutive threads to be bound close together, as is done with `KMP_AFFINITY=compact`, so that communication overhead, cache line invalidation overhead, and page thrashing are minimized. Now, suppose the application also had a number of parallel regions which did not utilize all of the available OpenMP* threads. It is desirable to avoid binding multiple threads to the same core and leaving other cores not utilized, since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core. Since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core, you might want to avoid binding multiple threads to the same core while leaving other cores unused. The following figure illustrates this strategy of using `KMP_AFFINITY=granularity=fine,compact,1,0` as a setting.



The OpenMP* thread *n*+1 is bound to a thread context as close as possible to OpenMP* thread *n*, but on a different core. Once each core has been assigned one OpenMP* thread, the subsequent OpenMP* threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.

Modifier Values for Affinity Types

Modifiers are optional arguments that precede type. If you do not specify a modifier, the `noverbose`, `respect`, and `granularity=core` modifiers are used automatically.

Modifiers are interpreted in order from left to right, and can negate each other. For example, specifying `KMP_AFFINITY=verbose,noverbose,scatter` is therefore equivalent to setting `KMP_AFFINITY=noverbose,scatter`, or just `KMP_AFFINITY=scatter`.

modifier = noverbose (default)

Does not print verbose messages.

modifier = verbose

Prints messages concerning the supported affinity. The messages include information about the number of packages, number of cores in each package, number of thread contexts for each core, and OpenMP* thread bindings to physical thread contexts.

Information about binding OpenMP* threads to physical thread contexts is indirectly shown in the form of the mappings between hardware thread contexts and the operating system (OS) processor (proc) IDs. The affinity mask for each OpenMP* thread is printed as a set of OS processor IDs.

For example, specifying `KMP_AFFINITY=verbose,scatter` on a dual core system with two processors, with Intel® Hyper-Threading Technology (Intel® HT Technology) disabled, results in a message listing similar to the following when then program is executed:

Verbose, scatter message

```
...
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {1}
```

The verbose modifier generates several standard, general messages. The following table summarizes how to read the messages.

Message String	Description
"affinity capable"	Indicates that all components (compiler, operating system, and hardware) support affinity, so thread binding is possible.
"using global cpuid info"	Indicates that the machine topology was discovered by binding a thread to each operating system processor and decoding the output of the `cpuid` instruction.
"using local cpuid info"	Indicates that compiler is decoding the output of the `cpuid` instruction, issued by only the initial thread, and is assuming a machine topology using the number of operating system processors.

Message String	Description
"using /proc/cpuinfo"	Linux* only. Indicates that `cpuinfo` is being used to determine machine topology.
"using flat"	Operating system processor ID is assumed to be equivalent to physical package ID. This method of determining machine topology is used if none of the other methods will work, but may not accurately detect the actual machine topology.
"uniform topology of"	The machine topology map is a full tree with no missing leaves at any level.

The mapping from the operating system processors to thread context ID is printed next. The binding of OpenMP* thread context ID is printed next unless the affinity type is `none`. The thread level is contained in brackets (in the listing shown above). This implies that there is no representation of the thread context level in the machine topology map. For more information, see Determining Machine Topology.

modifier = granularity

Binding OpenMP* threads to particular packages and cores will often result in a performance gain on systems with Intel processors with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled; however, it is usually not beneficial to bind each OpenMP* thread to a particular thread context on a specific core. Granularity describes the lowest levels that OpenMP* threads are allowed to float within a topology map.

This modifier supports the following additional specifiers.

Specifier	Description
`core`	Default. Allows all the OpenMP* threads bound to a core to float between the different thread contexts.
`fine` or `thread`	The finest granularity level. Causes each OpenMP* thread to be bound to a single thread context. The two specifiers are functionally equivalent.
tile	Allows all the OpenMP* threads bound to a tile to float between the different thread contexts of cores the tile consists of.

Specifying `KMP_AFFINITY=verbose,granularity=core,compact` on the same dual core system with two processors as in the previous section, but with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, results in a message listing similar to the following when the program is executed:

Verbose, granularity=core,compact message

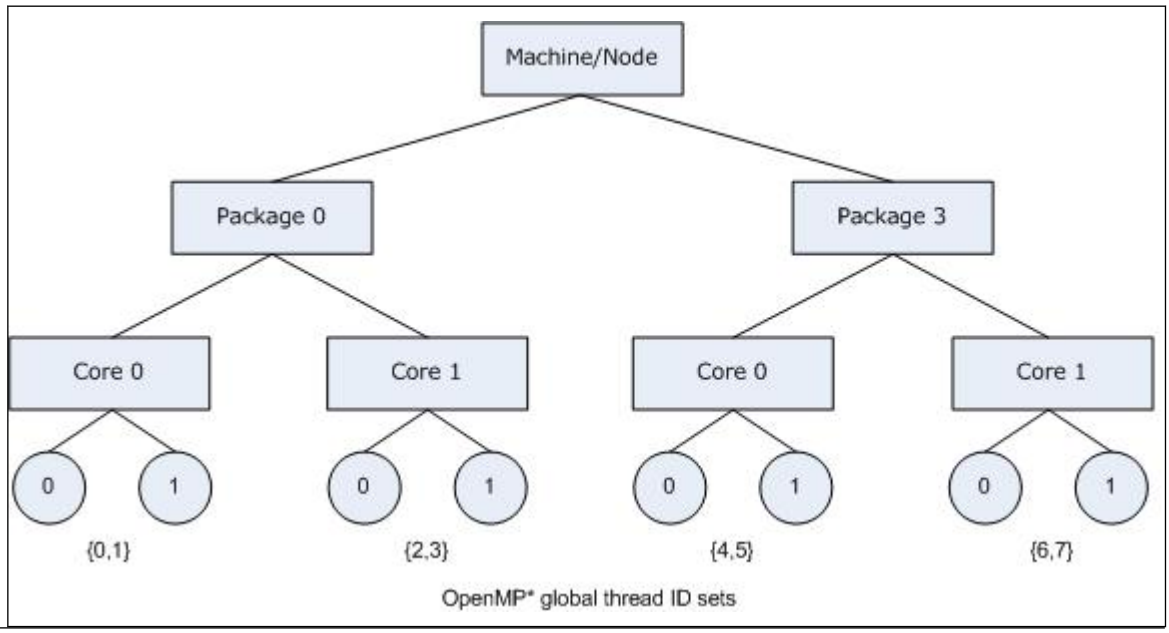
```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP_AFFINITY: 8 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,4}
```

Verbose, granularity=core,compact message

```
KMP_AFFINITY: Internal thread 1 bound to OS proc set {0,4}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2,6}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {2,6}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {1,5}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {1,5}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {3,7}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {3,7}
```

The affinity mask for each OpenMP* thread is shown in the listing (above) as the set of operating system processor to which the OpenMP* thread is bound.

The following figure illustrates the machine topology map, for the above listing, with OpenMP* thread bindings.



In contrast, specifying `KMP_AFFINITY=verbose,granularity=fine,compact` or `KMP_AFFINITY=verbose,granularity=thread,compact` binds each OpenMP* thread to a single hardware thread context when the program is executed:

Verbose, granularity=fine,compact message

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP_AFFINITY: 8 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
```

Verbose, granularity=fine,compact message

```
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {1}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

The OpenMP* to hardware context binding for this example was illustrated in the first example.

Specifying `granularity=fine` will always cause each OpenMP* thread to be bound to a single OS processor. This is equivalent to `granularity=thread`, currently the finest granularity level.

modifier = respect (default)

Respect the process' original affinity mask, or more specifically, the affinity mask in place for the thread that initializes the OpenMP* run-time library. The behavior differs between Linux* and Windows*:

- On Windows*: Respect original affinity mask for the process.
- On Linux*: Respect the affinity mask for the thread that initializes the OpenMP* run-time library.

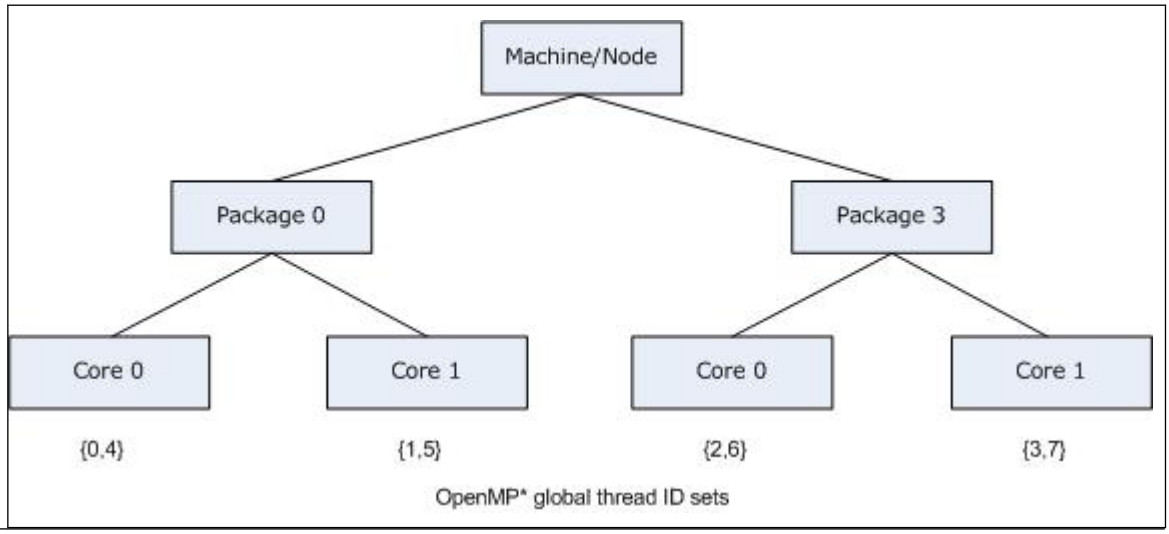
Specifying `KMP_AFFINITY=verbose,compact` for the same system used in the previous example, with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, and invoking the library with an initial affinity mask of {4,5,6,7} (thread context 1 on every core) causes the compiler to model the machine as a dual core, two-processor system with Intel® HT Technology disabled.

Verbose,compact message

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{4,5,6,7}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 [thread 1]
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 [thread 1]
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 [thread 1]
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 [thread 1]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {7}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

Because there are eight thread contexts on the machine, by default the compiler created eight threads for an OpenMP* `parallel` construct.

The brackets around thread 1 indicate that the thread context level is ignored, and is not present in the topology map. The following figure illustrates the corresponding machine topology map.



When using the local `cpuid` information to determine the machine topology, it is not always possible to distinguish between a machine that does not support Intel® Hyper-Threading Technology (Intel® HT Technology) and a machine that supports it, but has it disabled. Therefore, the compiler does not include a level in the map if the elements (nodes) at that level had no siblings, with the exception that the package level is always modeled. As mentioned earlier, the package level will always appear in the topology map, even if there only a single package in the machine.

modifier = norespect

Do not respect original affinity mask for the process. Binds OpenMP* threads to all operating system processors.

In early versions of the OpenMP* run-time library that supported only the `physical` and `logical` affinity types, `norespect` was the default and was not recognized as a modifier.

The default was changed to `respect` when types `compact` and `scatter` were added; therefore, thread bindings for the `logical` and `physical` affinity types may have changed with the newer compilers in situations where the application specified a partial initial thread affinity mask.

modifier = nowarnings

Do not print warning messages from the affinity interface.

modifier = warnings (default)

Print warning messages from the affinity interface (default).

Determining Machine Topology

On IA-32 and Intel® 64 architecture systems, if the package has an APIC (Advanced Programmable Interrupt Controller), the compiler will use the `cpuid` instruction to obtain the `package id`, `core id`, and `thread context id`. Under normal conditions, each thread context on the system is assigned a unique APIC ID at boot time. The compiler obtains other pieces of information obtained by using the *cpuid* instruction, which together with the number of OS thread contexts (total number of processing elements on the machine), determine how to break the APIC ID down into the `package ID`, `core ID`, and `thread context ID`.

There are two ways to specify the APIC ID in the *cpuid* instruction - the legacy method in leaf 4, and the more modern method in leaf 11. Only 256 unique APIC IDs are available in leaf 4. Leaf 11 has no such limitation.

Normally, all `core ids` on a package and all `thread context ids` on a core are contiguous; however, numbering assignment gaps are common for `package ids`, as shown in the figure above.

If the compiler cannot determine the machine topology using any other method, but the operating system supports affinity, a warning message is printed, and the topology is assumed to be `flat`. For example, a flat topology assumes the operating system process *N* maps to package *N*, and there exists only one thread context per core and only one core for each package.

If the machine topology cannot be accurately determined as described above, the user can manually copy `/proc/cpuinfo` to a temporary file, correct any errors, and specify the machine topology to the OpenMP* runtime library via the environment variable `KMP_CPUINFO_FILE=<temp_filename>`, as described in the section `KMP_CPUINFO_FILE` and `/proc/cpuinfo`.

Regardless of the method used in determining the machine topology, if there is only one thread context per core for every core on the machine, the thread context level will not appear in the topology map. If there is only one core per package for every package in the machine, the core level will not appear in the machine topology map. The topology map need not be a full tree, because different packages may contain a different number of cores, and different cores may support a different number of thread contexts.

The package level will always appear in the topology map, even if there only a single package in the machine.

`KMP_CPUINFO_FILE` and /proc/cpuinfo

One of the methods the Intel® C++ Compiler OpenMP runtime library can use to detect the machine topology on Linux* systems is to parse the contents of `/proc/cpuinfo`. If the contents of this file (or a device mapped into the Linux* file system) are insufficient or erroneous, you can consider copying its contents to a writable temporary file `<temp_file>`, correct it or extend it with the necessary information, and set `KMP_CPUINFO_FILE=<temp_file>`.

If you do this, the OpenMP* runtime library will read the `<temp_file>` location pointed to by `KMP_CPUINFO_FILE` instead of the information contained in `/proc/cpuinfo` or attempting to detect the machine topology by decoding the APIC IDs. That is, the information contained in the <*temp_file*> overrides these other methods. You can use the KMP_CPUINFO_FILE interface on Windows* systems, where `/proc/cpuinfo` does not exist.

The content of `/proc/cpuinfo` or `<temp_file>` should contain a list of entries for each processing element on the machine. Each processor element contains a list of entries (descriptive name and value on each line). A blank line separates the entries for each processor element. Only the following fields are used to determine the machine topology from each entry, either in <*temp_file*> or `/proc/cpuinfo`:

Field	Description
processor :	Specifies the OS ID for the processing element. The OS ID must be unique. The `processor` and `physical id` fields are the only ones that are required to use the interface.
physical id :	Specifies the package ID, which is a physical chip ID. Each package may contain multiple cores. The package level always exists in the Intel compiler OpenMP run-time library's model of the machine topology.
core id :	Specifies the core ID. If it does not exist, it defaults to 0. If every package on the machine contains only a single core, the core level will not exist in the machine topology map (even if some of the core ID fields are non-zero).

Field	Description
thread id :	Specifies the thread ID. If it does not exist, it defaults to 0. If every core on the machine contains only a single thread, the thread level will not exist in the machine topology map (even if some thread ID fields are non-zero).
node_*n* id :	This is a extension to the normal contents of `/proc/cpuinfo` that can be used to specify the nodes at different levels of the memory interconnect on Non-Uniform Memory Access (NUMA) systems. Arbitrarily many levels *n* are supported. The node_0 level is closest to the package level; multiple packages comprise a node at level 0. Multiple nodes at level 0 comprise a node at level 1, and so on.

Each entry must be spelled exactly as shown, in lowercase, followed by optional whitespace, a colon (:), more optional whitespace, then the integer ID. Fields other than those listed are simply ignored.

NOTE
It is common for the `thread id` field to be missing from `/proc/cpuinfo` on many Linux* variants, and for a field labeled `siblings` to specify the number of threads per node or number of nodes per package. However, the Intel OpenMP runtime library ignores fields labeled `siblings` so it can distinguish between the `thread id` and `siblings` fields. When this situation arises, the warning message `Physical node/pkg/core/thread ids not unique` appears (unless the `type specified is nowarnings`).

Windows* Processor Groups

On a 64-bit Windows* operating system, it is possible for multiple processor groups to accommodate more than 64 processors. Each group is limited in size, up to a maximum value of sixty-four (64) processors.

If multiple processor groups are detected, the default is to model the machine as a 2-level tree, where level 0 are for the processors in a group, and level 1 are for the different groups. Threads are assigned to a group until there are as many OpenMP* threads bound to the groups as there are processors in the group. Subsequent threads are assigned to the next group, and so on.

By default, threads are allowed to float among all processors in a group, that is to say, granularity equals the group [granularity=group]. You can override this binding and explicitly use another affinity type like compact, scatter, and so on. If you do so, the granularity must be sufficiently fine to prevent a thread from being bound to multiple processors in different groups.

Using a Specific Machine Topology Modeling Method (KMP_TOPOLOGY_METHOD)

You can set the `KMP_TOPOLOGY_METHOD` environment variable to force OpenMP* to use a particular machine topology modeling method.

Value	Description
`cpuid_leaf11`	Decodes the APIC identifiers as specfied by leaf 11 of the *cpuid* instruction.

Value	Description
`cpuid_leaf4`	Decodes the APIC identifiers as specified in leaf 4 of the *cpuid* instruction.
`cpuinfo`	If `KMP_CPUINFO_FILE` is not specified, forces OpenMP* to parse `/proc/cpuinfo` to determine the topology (Linux* only). If `KMP_CPUINFO_FILE` is specified as described above, uses it (Windows* or Linux*).
`group`	Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows* 64-bit only) .
`flat`	Models the machine as a flat (linear) list of processors.
`hwloc`	Models the machine as the Portable Hardware Locality* (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa nodes, packages, cores, hardware threads, caches, and Windows* processor groups.

Explicitly Specifying OS Processor IDs (GOMP_CPU_AFFINITY)

NOTE
You must set the `GOMP_CPU_AFFINITY` environment variable before the first parallel region, or certain API calls including `omp_get_max_threads()`, `omp_get_num_procs()` and any affinity API calls, as described in Low Level Affinity API, below.

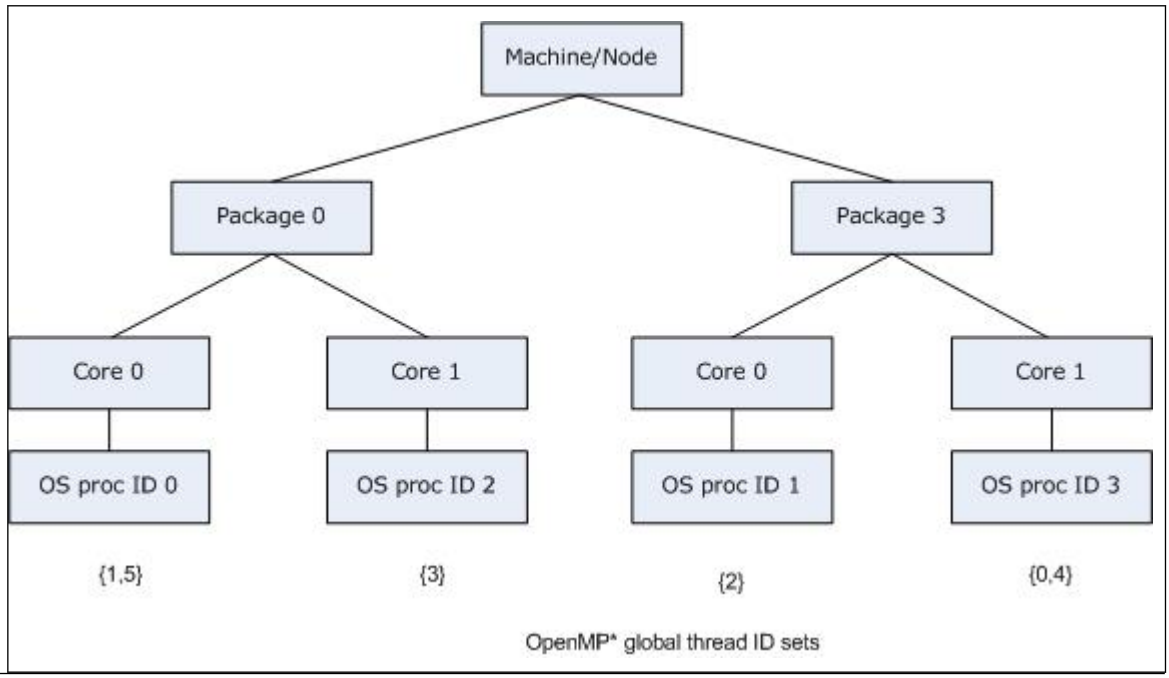
Instead of allowing the library to detect the hardware topology and automatically assign OpenMP* threads to processing elements, the user may explicitly specify the assignment by using a list of operating system (OS) processor (proc) IDs. However, this requires knowledge of which processing elements the OS proc IDs represent.

On Linux* systems, when using the Intel OpenMP compatibility libraries enabled by the compiler option `-qopenmp-lib compat`, you can use the `GOMP_AFFINITY` environment variable to specify a list of OS processor IDs. Its syntax is identical to that accepted by `libgomp` (assume that <*proc_list*> produces the entire `GOMP_AFFINITY` environment string):

Value	Description
<*proc_list*> :=	<entry> | <*elem*> , <list> | <elem> <whitespace> <list>
<*elem*> :=	<*proc_spec*> | <*range*>
<*proc_spec*> :=	<*proc_id*>
<*range*> :=	<*proc_id*> - <*proc_id*> | <*proc_id*> - <*proc_id*> : <int>
<*proc_id*> :=	<positive_int>

OS processors specified in this list are then assigned to OpenMP* threads, in order of OpenMP* Global Thread IDs. If more OpenMP* threads are created than there are elements in the list, then the assignment occurs modulo the size of the list. That is, OpenMP* Global Thread ID *n* is bound to list element *n* mod <*list_size*>.

Consider the machine previously mentioned: a dual core, dual-package machine without Intel® Hyper-Threading Technology (Intel® HT Technology) enabled, where the OS proc IDs are assigned in the same manner as the example in a previous figure. Suppose that the application creates six OpenMP* threads instead of 4 (the default), oversubscribing the machine. If `GOMP_AFFINITY=3,0-2`, then OpenMP* threads are bound as shown in the figure below, just as should happen when compiling with `gcc` and linking with `libgomp`:



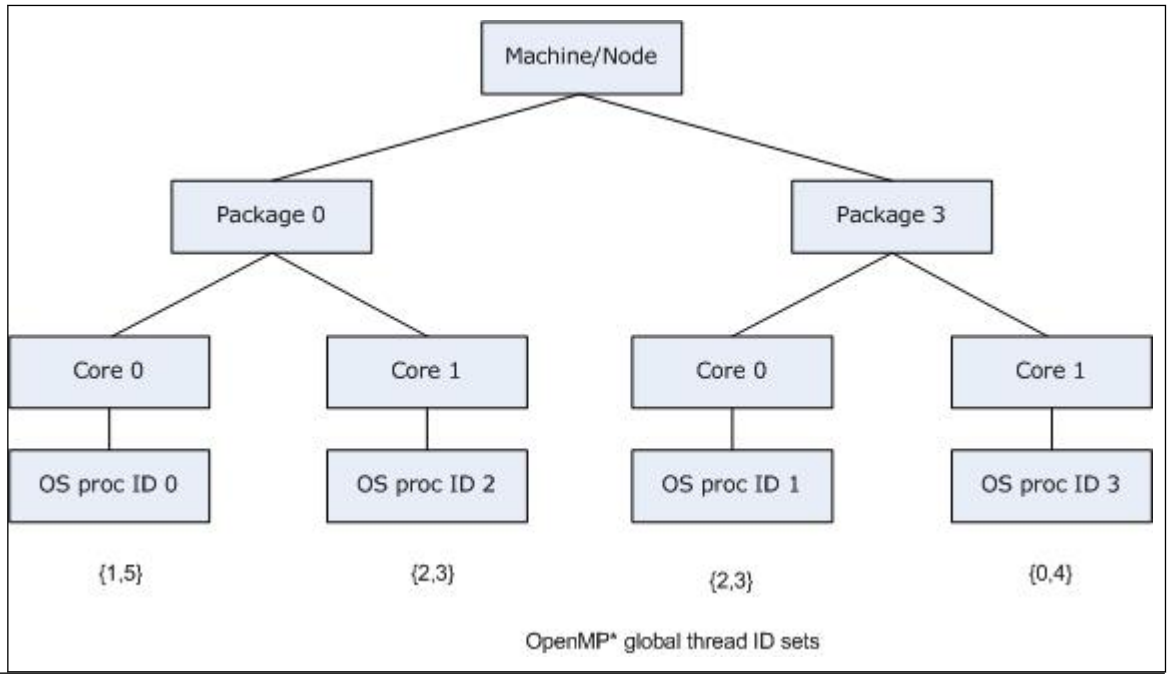
The same syntax can be used to specify the OS proc ID list in the proclist=[<proc_list>] modifier in the `KMP_AFFINITY` environment variable string. There is a slight difference: in order to have strictly the same semantics as in the `gcc` OpenMP* runtime library `libgomp`: the `GOMP_AFFINITY` environment variable implies `granularity=fine`. If you specify the OS proc list in the `KMP_AFFINITY` environment variable without a `granularity=` specifier, then the default `granularity` is not changed. That is, OpenMP* threads are allowed to float between the different thread contexts on a single core. Thus `GOMP_AFFINITY=<proc_list>` is an alias for `KMP_AFFINITY="granularity=fine,proclist=[<proc_list>],explicit"`.

In the `KMP_AFFINITY` environment variable string, the syntax is extended to handle operating system processor ID sets. The user may specify a set of operating system processor IDs among which an OpenMP* thread may execute ("œfloat") enclosed in brackets:

Value	Description
<*proc_list*> :=	<proc_id> | { <float_list> }
<*float_list*> :=	<proc_id> | <proc_id> , <float_list>

This allows functionality similarity to the `granularity= specifier`, but it is more flexible. The OS processors on which an OpenMP* thread executes may exclude other OS processors nearby in the machine topology, but include other distant OS processors. Building upon the previous example, we may allow

OpenMP* threads 2 and 3 to "œfloat" between OS processor 1 and OS processor 2 by using `KMP_AFFINITY="granularity=fine,proclist=[3,0,{1,2},{1,2}],explicit"`, as shown in the figure below:



If `verbose` were also specified, the output when the application is executed would include:

`KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"`

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {1,2}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {0}}
```

Low Level Affinity API

Instead of relying on the user to specify the OpenMP* thread to OS proc binding by setting an environment variable before program execution starts (or by using the `kmp_settings` interface before the first parallel region is reached), each OpenMP* thread can determine the desired set of OS procs on which it is to execute and bind to them with the `kmp_set_affinity` API call.

Caution
When you use this affinity interface you take complete control of the hardware resources on which your threads run. To do that sensibly you need to understand in detail how the logical CPUs, the enumeration of hardware threads controlled by the OS, map to the physical hardware of the specific machine on which you are running. That mapping can be, and likely is, different on different machines, so you risk binding machine-specific information into your code, which can result in explicitly forcing bad affinities when your code runs on a different machine. And if you are concerned with optimization at this level of detail, your code is probably valuable, and therefore will probably move to another machine.

This interface may also allow you to ignore the resource limitations that were set by the program startup mechanism, such as Message Passing Interface (MPI), specifically to prevent multiple OpenMP processes on the same node from using the same hardware threads. Again, this can result in explicitly forcing affinities that cause bad performance, and the OpenMP runtime will neither prevent this from happening, nor warn you when it does. These are expert interfaces and you must use them with caution.

It is recommended, therefore, to use the higher level affinity settings if you possibly can, because they are more portable and do not require this low level knowledge.

The Fortran API interfaces follow, where the type name `kmp_affinity_mask_t` is defined in `omp_lib.h` or `omp_lib.mod`:

NOTE
Some of these interfaces have offload equivalents. The offload equivalent takes two additional arguments to specify the target type and target number. For more information, see **Calling Functions on the CPU to Modify the Coprocessor's Execution Environment**. Offload is not supported on Windows* systems.

Syntax	Description
`integer function kmp_set_affinity(mask)` `integer (kind=kmp_affinity_mask_kind) mask`	Sets the affinity mask for the current OpenMP* thread to `mask`, where `mask` is a set of OS proc IDs that has been created using the API calls listed below, and the thread will only execute on OS procs in the set. Returns either a zero (0) upon success or a nonzero error code.
`integer kmp_get_affinity(mask)` `integer (kind=kmp_affinity_mask_kind) mask`	Retrieves the affinity mask for the current OpenMP* thread, and stores it in `mask`, which must have previously been initialized with a call to `kmp_create_affinity_mask()`. Returns either a zero (0) upon success or a nonzero error code.
`integer function kmp_get_affinity_max_proc()`	Returns the maximum OS proc ID that is on the machine, plus 1. All OS proc IDs are guaranteed to be between 0 (inclusive) and `kmp_get_affinity_max_proc()` (exclusive).
`subroutine kmp_create_affinity_mask(mask)` `integer (kind=kmp_affinity_mask_kind) mask`	Allocates a new OpenMP* thread affinity mask, and initializes `mask` to the empty set of OS procs. The implementation is free to use an object of `kmp_affinity_mask_kind` either as the set itself,

Syntax	Description
	a pointer to the actual set, or an index into a table describing the set. Do not make any assumption as to what the actual representation is.
`subroutine kmp_destroy_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	Deallocates the OpenMP* thread affinity mask. For each call to `kmp_create_affinity_mask()`, there should be a corresponding call to `kmp_destroy_affinity_mask()`.
`integer function kmp_set_affinity_mask_proc(proc, mask) integer proc integer (kind=kmp_affinity_mask_kind) mask`	Adds the OS proc ID proc to the set `mask`, if it is not already. Returns either a zero (0) upon success or a nonzero error code.
`integer function kmp_unset_affinity_mask_proc(proc, mask) integer proc integer (kind=kmp_affinity_mask_kind) mask`	If the OS proc ID proc is in the set `mask`, it removes it. Returns either a zero (0) upon success or a nonzero error code.
`integer function kmp_get_affinity_mask_proc(proc, mask)integer proc integer (kind=kmp_affinity_mask_kind) mask`	Returns 1 if the OS proc ID proc is in the set `mask`; if not, it returns 0.

Once an OpenMP* thread has set its own affinity mask via a successful call to `kmp_set_affinity()`, then that thread remains bound to the corresponding OS proc set until at least the end of the parallel region, unless reset via a subsequent call to `kmp_set_affinity()`.

Between parallel regions, the affinity mask (and the corresponding OpenMP* thread to OS proc bindings) can be considered thread private data objects, and have the same persistence as described in the OpenMP* Application Program Interface. For more information, see the OpenMP* API specification (http://www.openmp.org), some relevant parts of which are provided below:

In order for the affinity mask and thread binding to persist between two consecutive active parallel regions, all three of the following conditions must hold:

- Neither parallel region is nested inside another explicit parallel region.
- The number of threads used to execute both parallel regions is the same.
- The value of the dyn-var internal control variable in the enclosing task region is false at entry to both parallel regions."

Therefore, by creating a parallel region at the start of the program whose sole purpose is to set the affinity mask for each thread, you can mimic the behavior of the `KMP_AFFINITY` environment variable with low-level affinity API calls, if program execution obeys the three aforementioned rules from the OpenMP* specification.

The following example shows how these low-level interfaces can be used. This code binds the executing thread to the specified logical CPU:

Example

```
! Force the executing thread to execute on logical CPU i
! Returns .TRUE. on success, .FALSE. on failure

function forceAffinity (i)
    use omp_lib
    logical forceAffinity
    integer, intent(in) :: i

    integer(kmp_affinity_mask_kind) :: mask

    call kmp_create_affinity_mask(mask)
    forceAffinity = (kmp_set_affinity_mask_proc(i, mask) == 0)
    if (.not. forceAffinity) return
    forceAffinity = (kmp_set_affinity_mask(mask) == 0)
    return
    end function forceAffinity
```

This program fragment was written with knowledge about the mapping of the OS proc IDs to the physical processing elements of the target machine. On another machine, or on the same machine with a different OS installed, the program would still run, but the OpenMP* thread to physical processing element bindings could differ and you might be explicitly force a bad distribution.

OpenMP* Advanced Issues

This topic discusses how to use the OpenMP* library functions and environment variables and discusses some guidelines for enhancing performance with OpenMP*.

OpenMP* provides specific function calls, and environment variables. See the following topics to refresh you memory about the primary functions and environment variable used in this topic:

- OpenMP* Run-time Library Routines
- OpenMP* Environment Variables

To use the function calls, include the `omp_lib.h` header file or specify `use omp_lib` to use the Fortran90 module file. These files are installed in the `INCLUDE` directory during the compiler installation, and compile the application using the `[Q]openmp` option.

The following example, which demonstrates how to use the OpenMP* functions to print the alphabet, also illustrates several important concepts:

1. When using functions instead of directives, your code must be rewritten; rewrites can mean extra debugging, testing, and maintenance efforts.
2. It becomes difficult to compile without OpenMP* support.
3. it is very easy to introduce simple bugs, as in the loop (below) that fails to print all the letters of the alphabet when the number of threads is not a multiple of 26.
4. You lose the ability to adjust loop scheduling without creating your own work-queue algorithm, which is a lot of extra effort. You are limited by your own scheduling, which is mostly likely static scheduling as shown in the example.

Example

```
include "omp_lib.h"
integer i
integer LettersPerThread, ThisThreadNum, StartLetter, EndLetter

call omp_set_num_threads(4)
     !$OMP PARALLEL PRIVATE(i)
```

Example

```
        ! OMP_NUM_THREADS is not a multiple of 26,
        ! which can be considered a bug in this code.
        LettersPerThread = 26 / omp_get_num_threads()
        ThisThreadNum = omp_get_thread_num()
        StartLetter = 'a'+ThisThreadNum*LettersPerThread
        EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread

        DO i = StartLetter, EndLetter - 1
             write( *,FMT='(A)',ADVANCE='NO') char(i)
        END DO

    !$OMP END PARALLEL
write(*,*)
end
```

Debugging threaded applications is a complex process because debuggers change the run-time performance, which can mask race conditions. Even `print` statements can mask issues, because they use synchronization and operating system functions. OpenMP* itself also adds some complications, because it introduces additional structure by distinguishing private variables and shared variables, and inserts additional code. A debugger that supports OpenMP* can help you to examine variables and step through threaded code. You can use Intel® Inspector to detect many hard-to-find threading errors analytically. Sometimes, a process of elimination can help identify problems without resorting to sophisticated debugging tools.

Remember that most mistakes are race conditions. Most race conditions are caused by shared variables that really should have been declared private. Start by looking at the variables inside the parallel regions and make sure that the variables are declared private when necessary. Next, check functions called within parallel constructs.

The `DEFAULT(NONE)` clause, shown below, can be used to help find those hard-to-spot variables. If you specify `DEFAULT(NONE)`, then every variable must be declared with a data-sharing attribute clause.

Example

```
!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(x,y) SHARED(a,b)
```

Another common mistake is using uninitialized variables. Remember that private variables do not have initial values upon entering a parallel construct. Use the `FIRSTPRIVATE` and `LASTPRIVATE` clauses to initialize them only when necessary, because doing so adds extra overhead.

If you still can't find the bug, then consider the possibility of reducing the scope. Try a binary-hunt. Another method is to force large chunks of a parallel region to be critical sections. Pick a region of the code that you think contains the bug and place it within a critical section. Try to find the section of code that suddenly works when it is within a critical section and fails when it is not. Now look at the variables, and see if the bug is apparent. If that still doesn't work, try setting the entire program to run in serial by setting the compiler-specific environment variable `KMP_LIBRARY=serial`.

If the code is still not working, and you are not using any OpenMP* API function calls, compile it without the `[Q]openmp` option to make sure the serial version works. If you are using OpenMP* API function calls, use the `[Q]openmp-stubs` option.

Performance

OpenMP* threaded application performance is largely dependent upon the following things:

- The underlying performance of the single-threaded code.
- CPU utilization, idle threads, and load balancing.
- The percentage of the application that is executed in parallel by multiple threads.
- The amount of synchronization and communication among the threads.

- The overhead needed to create, manage, destroy, and synchronize the threads, made worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions.
- Performance limitations of shared resources such as memory, bus bandwidth, and CPU execution units.
- Memory conflicts caused by shared memory or falsely shared memory.

Performance always begins with a properly constructed parallel algorithm or application. For example, parallelizing a bubble-sort, even one written in hand-optimized assembly language, is not a good place to start. Keep scalability in mind; creating a program that runs well on two CPUs is not as efficient as creating one that runs well on *n* CPUs. With OpenMP*, the number of threads is chosen by the compiler, so programs that work well regardless of the number of threads are highly desirable. Producer/consumer architectures are rarely efficient, because they are made specifically for two threads.

Once the algorithm is in place, make sure that the code runs efficiently on the targeted Intel® architecture; a single-threaded version can be a big help. Turn off the `[Q]openmp` option to generate a single-threaded version, or build with the `[Q]openmp-stubs` option, and run the single-threaded version through the usual set of optimizations.

Once you have gotten the single-threaded performance, it is time to generate the multi-threaded version and start doing some analysis.

Optimizations are really a combination of patience, experimentation, and practice. Make little test programs that mimic the way your application uses the computer resources to get a feel for what things are faster than others. Be sure to try the different scheduling clauses for the parallel sections of code. If the overhead of a parallel region is large compared to the compute time, you may want to use an `if` clause to execute the section serially.

Interoperability with OpenMP* in C/C++

The Intel® Fortran Compiler does not support Fortran calling C/C++ or C/C++ calling Fortran when both caller and callee are using OpenMP* constructs.

Fortran code that uses OpenMP* constructs can call C/C++ as long as the C/C++ code does not use OpenMP* constructs.

See Also

OpenMP* Run-time Library Routines
Supported Environment Variables
openmp, Qopenmp
openmp-stubs, Qopenmp-stubs

OpenMP* Implementation-Defined Behaviors

This topic summarizes the behaviors that are described as implementation defined in the OpenMP* API specification.

NOTE
Internal Control Variables (ICVs) mentioned below are discussed in the OpenMP* API specification.

Name	Description
`single` construct	The first thread that encounters the single construct executes the structured block.
`teams` construct	The number of teams that are created is equal to 1 if you don't specify the `num_teams` clause.

Name	Description
`dist_schedule` clause, `distribute` construct	If you don't specify the `dist_schedule` clause, then the schedule for the `distribute` construct is `static`.
`omp_set_num_threads` routine	If the argument is not a positive integer, then Intel's OpenMP* implementation sets the value of the first element of the `nthreads-var` ICV of the current task to 1.
`omp_set_max_active_levels` routine	If the argument is a negative integer this call is ignored and the last valid setting is used.
`omp_get_max_active_levels` routine	When called from within any explicit parallel region the binding thread set, and binding region, if required, for the `omp_get_max_active_levels` region is the current task region.
`OMP_SCHEDULE` environment variable	If the value of the variable does not conform to the specified format then the value of the `run-sched-var` ICV is set to `static` and the chunk size is set to 1.
`OMP_NUM_THREADS` environment variable	If any value of the list specified in the environment variable is negative then the whole list is ignored. If any value of the list is zero then this value is set to 1.
`OMP_PROC_BIND` environment variable	If the value is not `true`, `false`, or a comma separated list of `master`, `close`, or `spread`, then Intel's OpenMP* implementation sets the value of `bind-var` ICV to false.
`OMP_DYNAMIC` environment variable	If the value is neither `true` nor `false`, then the implementation sets the value of `dyn-var` ICV to false.
`OMP_NESTED` environment variable	If the value is neither `true` nor `false`, then the implementation sets the value of `nest-var` ICV to false.
`OMP_STACKSIZE` environment variable	If the value does not conform to the specified format or the implementation cannot provide a stack of the specified size, then Intel's OpenMP* implementation sets the value of `stacksize-var` ICV to the default size, which is specified as being from1MB to 4MB depending on the architecture.
`OMP_MAX_ACTIVE_LEVELS` environment variable	If the value is a negative integer or is greater than the number of parallel levels an implementation can support, then Intel's OpenMP* implementation sets the value of the `max-active-levels-var` ICV to the maximum number of parallel levels supported on a particular platform.
`OMP_THREAD_LIMIT` environment variable	If the requested value is greater than the number of threads an implementation can support, or if the value is a negative integer, then Intel's OpenMP* implementation sets the value of the `thread-limit-var` ICV to the maximum number of threads

Name	Description
	supported on a particular platform. If the requested value is zero then the implementation sets the value of the `thread-limit-var` ICV to 1.
Runtime library definitions	Intel's OpenMP* implementation provides both the include file `omp_lib.h` and the module `omp_lib`.

OpenMP* Examples

The following examples show how to use several OpenMP* features.

A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to improve load balancing.

The `END DO` has a `NOWAIT` because there is an implicit barrier at the end of the parallel region.

Example

```
subroutine do_1(a,b,n)
  real a(n,n), b(n,n)
  !$OMP PARALLEL SHARED(A,B,N)
     !$OMP DO SCHEDULE(DYNAMIC,1) PRIVATE(I,J)
       do i = 2, n
          do j = 1, i
            b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
          end do
      end do
  !$OMP END DO NOWAIT
!$OMP END PARALLEL
end
```

Two Difference Operators: DO Loop Version

The example uses two parallel loops fused to reduce fork/join overhead. The first `END DO` directive has a `NOWAIT` clause because all the data used in the second loop is different than all the data used in the first loop.

Example

```
subroutine do_2(a,b,c,d,m,n)
  real a(n,n), b(n,n), c(m,m), d(m,m)
  !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
    !$OMP DO SCHEDULE(DYNAMIC,1)
       do i = 2, n
        do j = 1, i
          b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
        end do
      end do
    !$OMP END DO NOWAIT
    !$OMP DO SCHEDULE(DYNAMIC,1)
      do i = 2, m
        do j = 1, i
          d(j,i) = ( c(j,i) + c(j,i-1) ) / 2.0
```

Example

```
        end do
      end do
    !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end
```

Two Difference Operators: SECTIONS Version

The example demonstrates the use of the `SECTIONS` directive. The logic is identical to the preceding `DO` example, but uses `SECTIONS` instead of `DO`. Here the speedup is limited to two because there are only two units of work whereas in the example above there are `(n-1) + (m-1)` units of work.

Example

```
subroutine sections_1(a,b,c,d,m,n)
  real a(n,n), b(n,n), c(m,m), d(m,m)
  !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
    !$OMP SECTIONS
       !$OMP SECTION
          do i = 2, n
             do j = 1, i
                b(j,i)=( a(j,i) + a(j,i-1) ) / 2.0
             end do
          end do
       !$OMP SECTION
          do i = 2, m
             do j = 1, i
                d(j,i)=( c(j,i) + c(j,i-1) ) / 2.0
             end do
          end do
    !$OMP END SECTIONS NOWAIT
  !$OMP END PARALLEL
end
```

Updating a Shared Scalar

This example demonstrates how to use a `SINGLE` construct to update an element of the shared array `a`. The optional `nowait` clause after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the `SINGLE` construct.

Example

```
subroutine sp_1a(a,b,n)
  real a(n), b(n)
  !$OMP PARALLEL SHARED(A,B,N) PRIVATE(I)
    !$OMP DO
       do i = 1, n
         a(i) = 1.0 / a(i)
       end do
    !$OMP SINGLE
      a(1) = min( a(1), 1.0 )
    !$OMP END SINGLE
    !$OMP DO
      do i = 1, n
```

Example

```
        b(i) = b(i) / a(i)
      end do
    !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end
```

Coarrays

Using Coarrays

Coarrays are not supported on macOS* systems.

Coarrays, a data sharing concept standardized in Fortran 2008 and extended in Fortran 2018, enable parallel processing using multiple copies of a single program. Each copy, called an image, has ordinary local variables and also shared variables called coarrays or covariables. A covariable, which can be either an array or a scalar, is a variable whose storage spans all the images in the program. In this Partitioned Global Address Space (PGAS) model, each image can access its own piece of a covariable as a local variable and can access those pieces that live on other images using coindices, which are enclosed in square brackets.

Intel® Fortran supports coarray programs that run using shared memory on a multicore or multiprocessor system. In some products (see the Feature Requirements section), coarray programs can also be built to run using distributed memory across a Linux* or Windows* cluster.

Please refer to the product system requirements in the Release Notes for further details.

NOTE
32-bit coarrays are deprecated and will be removed in a future release.

For more information on how to write programs using coarrays, see books on the Fortran 2008 language or the ISO Fortran 2008 standard.

Using Coarray Program Syntax

The additional syntax required by Fortran 2008 coarrays includes:

- CODIMENSION attribute and "[cobounds]" to declare an object a coarray (covariable)
- [coindices] notation to reference covariables on other images
- SYNC ALL, SYNC IMAGES, and SYNC MEMORY statements to provide points where images must communicate to synchronize shared data
- CRITICAL and END CRITICAL statements to form a block of code executed by one image at a time
- LOCK and UNLOCK statements to control objects called locks, used to synchronize actions on specific images
- ERROR STOP statement to end all images
- ALLOCATE and DEALLOCATE statements may specify coarrays
- Intrinsic procedures IMAGE_INDEX, LCOBOUND, NUM_IMAGES, THIS_IMAGE, and UCOBOUND
- Atomic subroutines ATOMIC_DEFINE and ATOMIC_REF for defining and referencing an atomic variable

The following Fortran 2018 coarray extensions are also supported:

- EVENT POST and EVENT WAIT statements to synchronize execution between two images
- The FAIL IMAGE statement to simulate a failed image
- Intrinsic procedures COSHAPE, EVENT_QUERY, FAILED_IMAGES, IMAGE_STATUS, and STOPPED IMAGES
- Atomic subroutines ATOMIC_ADD, ATOMIC_AND, ATOMIC_CAS, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR, ATOMIC_OR, and ATOMIC_XOR
- Collective subroutines CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE, and CO_SUM

- Optional STAT= and ERRMSG= specifiers on a CRITICAL construct, optional arguments STAT and ERRMSG for the MOVE_ALLOC intrinsic, and optional STAT= specifier on image selectors, and an optional STAT argument to ATOMIC_DEFINE and ATOMIC_REF subroutines

Using the Coarray Compiler Options

You must use the `-coarray` (Linux) or `/Qcoarray` (Windows) compiler option (hereafter referred to as `[Q]coarray`) to enable the compiler to recognize coarray syntax. If you do not specify this compiler option, a program that uses coarray syntax or features produces a compile-time error.

In the list that follows, only one option is valid on the command line; if multiple coarray compiler options are specified, the last one specified is used. An exception to this rule is the `[Q]coarray` compiler option using keyword `single`; if specified, this option takes precedence regardless of where it appears on the command line.

- Using `[Q]coarray` with no keyword is equivalent to running on one node (shared memory).
- Using `[Q]coarray` with keyword `shared` causes the underlying Intel® Message Passing Interface (MPI) parallelization to run on one node with multiple cores or processors with shared memory.
- Using `[Q]coarray` with keyword `distributed` requires a special license to be installed (see the Feature Requirements section) and causes the underlying Intel® MPI Library parallelization to run in a multi-node environment (multiple CPUs with distributed memory).
- Using `[Q]coarray` with keyword `single` creates an executable that will not be replicated, resulting in a single running image. (This is in contrast to the self-replicating behavior that occurs when any other coarray keyword is specified.) This option is useful for debugging purposes.

No special procedure is necessary to run a program that uses coarrays; you simply run the executable file. The underlying parallelization implementation uses the Intel® MPI Library. Installation of the compiler automatically installs the necessary Intel® MPI run-time libraries to run on shared memory. Products supporting clusters will also install the necessary Intel® MPI Library run-time libraries to run on distributed memory. Use of coarray applications with any other MPI implementation, or with OpenMP*, is not supported.

By default, the number of images created is equal to the number of execution units on the current system. You can override this by specifying a number using the `[Q]coarray-num-images` compiler option on the `ifort` command line that compiles the main program. You can also specify the number of images at execution time in the environment variable FOR_COARRAY_NUM_IMAGES.

Using a Configuration File

Use of the config-file option is appropriate only in a limited number of cases:

- You can take advantage of Intel® MPI Library features in the coarray environment. To do so, specify the command line segments used by "mpiexec -config *filename*" in a file named *filename* and pass that file name to the Intel® MPI Library using the `[Q]coarray-config-file` compiler option. If the `[Q]coarray-num-images` compiler option also appears on the command line, it will be overridden by what is in the configuration file. Rules for using an MPI configuration files are as follows:
  - The format of a configuration file is described in the Intel® MPI Library documentation; you will need to add the MPI option "-genv FOR_ICAF_STATUS launched" in the configuration file in order for coarrays to work on multi-node (distributed memory) systems.
  - You can also set the environment variable FOR_COARRAY_CONFIG_FILE to be the filename and path of the Intel® MPI Library configuration file you want to use at execution time.

Examples on Windows*:

- `/Qcoarray:shared /Qcoarray-num-images:8` runs a coarray program on shared memory using 8 images.
- `/Qcoarray:shared /Qcoarray-config-file:`*`filename`* runs a coarray program on shared memory using the MPI configuration detailed in *filename*.
- `/Qcoarray:distributed /Qcoarray-config-file:`*`filename`* runs a coarray program on distributed memory using the Intel® MPI Library configuration detailed in *filename*.

Examples on Linux*:

- `-coarray=shared -coarray-num-images=8` runs a coarray program on shared memory using 8 images.

- `-coarray=distributed -coarray-num-images=8` runs a coarray program on distributed memory across 8 images.

See Also

Feature Requirements

`coarray, Qcoarray` compiler option

`coarray-config-file, Qcoarray-config-file` compiler option

`coarray-num-images, Qcoarray-num-images` compiler option

Coarray constructs

Debugging a Coarray Application (Linux*)

Steps to perform to debug a coarray application.

For the following, you need an application with shared variables or coarrays whose storage spans all the images in a program.

Follow these steps to debug your coarray application:

1. Add a stall loop to your application before the area of code you wish to debug.

```
LOGICAL VOLATILE :: WAIT_FOR_DEBUGGER
LOGICAL, VOLATILE :: TICK
!
! Other code may be here
!
DO WHILE(WAIT_FOR_DEBUGGER)
TICK = .NOT. TICK
END DO
!
! Code you want to debug is here
```

   The use of VOLATILE is required to ensure that the loop will not be removed by the compiler. If the problem is only found on one image, you can wrap the loop in `IF (THIS_IMAGE() .EQ. 4) THEN` or the like.
2. Compile and link with debug enabled (`-g`).
3. Create at least N+1 terminal windows on the machine where the application will be running, where N is the number of images your application will have.
4. In a terminal window, start the application.

```
linuxprompt> ./my_app
```

5. In each of the other terminal windows, set your default directory to be the same as the location of the application executable. Use the `ps` command in one of the windows to find out which processes are running your application:

```
linuxprompt> ps -ef | grep 'whoami' | grep my_app
```

There will be several processes. The oldest is the one you started in step 4 – it has run the Message Passing Interface (MPI) launcher and is now waiting for the others to terminate. Do not debug it.

The others will look like this:

```
<your-user-name> 25653 25650 98 15:06 ? 00:00:49 my_app
<your-user-name> 25654 25651 97 15:06 ? 00:00:48 my_app
<your-user-name> 25655 25649 98 15:06 ? 00:00:49 my_app
```

The first number is the PID of the process (for example, `25653` in the first line). In the steps below, the PIDs of these N processes are referred to as `P1`, `P2`, and so on. When you type the commands, replace the text `<P1>` with the actual value of the PID for process 1, and so on.

6. In each window other than the first, start your debugger and set it to stop processes when attached:

```
linuxprompt> gdb
```

7. Attach to one of the processes. For example, attach to `P1` in window 1, attach to `P2` in window 2, and so on.

```
(gdb) attach <P1>
```

8. Get execution out of the stall loop:

```
(gdb) set WAIT_FOR_DEBUGGER = .false.
```

You are now ready to debug your coarray application by examining the data and code paths in the various images.

Automatic Parallelization

The auto-parallelization feature of the Intel®Fortran Compiler automatically translates serial portions of the input program into equivalent multithreaded code. Automatic parallelization determines the loops that are good worksharing candidates, performs the dataflow analysis to verify correct parallel execution, and partitions the data for threaded code generation as needed in programming with OpenMP* directives. The OpenMP* and auto-parallelization functionality provides the performance gains from shared memory on multiprocessor and dual core systems.

The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops which can safely and efficiently be executed in parallel.

This behavior enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

The guided auto-parallelization feature of the Intel®Fortran Compiler helps you locate portions in your serial code that can be parallelized further. You can invoke guidance for parallelization, vectorization, or data transformation using specified compiler options of the `[Q]guide` series.

Automatic parallelization frees developers from having to:

- Find loops that are good worksharing candidates.
- Perform the dataflow analysis to verify correct parallel execution.
- Partition the data for threaded code generation as is needed in programming with OpenMP* directives.

Although OpenMP* directives enable serial applications to transform into parallel applications quickly, you must explicitly identify specific portions of your application code that contain parallelism and add the appropriate compiler directives. Auto-parallelization, which is triggered by the `[Q]parallel` option, automatically identifies those loop structures that contain parallelism. During compilation, the compiler automatically attempts to deconstruct the code sequences into separate threads for parallel processing. No other effort is needed.

NOTE In order to execute a program that uses auto-parallelization on Linux* or macOS* systems, you must include the `-parallel` compiler option when you compile and link your program.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

Serial code can be divided so that the code can execute concurrently on multiple threads. For example, consider the following serial code example.

Example 1: Original Serial Code

```
subroutine ser(a, b, c)
  integer, dimension(100) :: a, b, c
  do i=1,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine ser
```

The following example illustrates one method showing how the loop iteration space, shown in the previous example, might be divided to execute on two threads.

Example 2: Transformed Parallel Code

```
subroutine par(a, b, c)
  integer, dimension(100) :: a, b, c
  ! Thread 1
  do i=1,50
    a(i) = a(i) + b(i) * c(i)
  enddo
  ! Thread 2
  do i=51,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine par
```

Auto-Vectorization and Parallelization

Auto-vectorization detects low-level operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8, or (up to) 16 elements in one operation, depending on the data type. In some cases, auto-parallelization and vectorization can be combined for better performance results. For example, in the code below, thread-level parallelism can be exploited in the outermost loop, while instruction-level parallelism can be exploited in the innermost loop.

Example

```
DO I = 1, 100     ! Execute groups of iterations in different hreads (TLP)
  DO J = 1, 32    ! Execute in SIMD style with multimedia extension (ILP)
     A(J,I) = A(J,I) + 1
  ENDDO
ENDDO
```

With the relatively small effort of adding OpenMP* directives to existing code you can transform a sequential program into a parallel program. The [Q]openmp option must be specifed to enable the OpenMP directives. The following example shows OpenMP* directives within the code.

Example

```
!OMP$ PARALLEL PRIVATE(NUM), SHARED (X,A,B,C)
! Defines a parallel region
!OMP$ PARALLEL DO
! Specifies a parallel region that
! implicitly contains a single DO directive
DO I = 1, 1000
  NUM = FOO(B(i), C(I))
  X(I) = BAR(A(I), NUM)
! Assume FOO and BAR have no other effect
ENDDO
```

NOTE
Options that use OpenMP* are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP* constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

Using Parallelism Reports

To generate a parallelism report, use the `-opt-report-phase=`*par* (Linux* and macOS*) or the `/Qopt-report-phase:`*par* option along with the `-opt-report=`*n* or `/Qopt-report:`*n* option. By default the auto-parallelism report generates a medium level of detail, where *n*=2. You can use `[Q]opt-report` option along with the `[Q]opt-report-phase` option if you want a greater or lesser level of detail. Specifying a value of '5' generates the maximum diagnostic details.

Run the report by entering commands similar to the following:

Operating System	Command
Linux*	`ifort -c -parallel -opt-report-phase=par -opt-report:5 sample.cpp`
macOS*	`ifort -c -parallel -opt-report-phase=par -opt-report:5 sample.cpp`
Windows*	`ifort sample.cpp /c /Qparallel /Qopt-report-phase=par /Qopt-report:5`

NOTE The `-c` (Linux* and macOS*) or `/c` (Windows*) prevents linking and instructs the compiler to stop compilation after the object file is generated. The example is compiled without generating an executable.

The output, by default, produces a file with the same name as the object file, with `.optrpt` extension, and is written into the same directory as the object file. Using the above command-line entries, you will obtain an output file called `sample.optrpt`. Use the `[Q]opt-report-file` option to specify any other name for the output file that captures the report results. Use the arguments *stdout* or *stderr* to send the optimization report to stdout or stderr.

For more information on options to generate reports see the Optimization Report Options topic.

See Also

Guided Auto-Parallelization

`parallel`, `Qparallel`

compiler option

par-runtime-control, Qpar-runtime-control
compiler option

par-threshold, Qpar-threshold
compiler option

guide, Qguide
compiler option

qopt-report-phase, Qopt-report-phase
compiler option

qopt-report, Qopt-report
compiler option

Enabling Auto-parallelization

To enable the auto-parallelizer, use the [Q]parallel option. This option detects parallel loops capable of being executed safely in parallel, and automatically generates multi-threaded code for these loops.

NOTE You may need to set the KMP_STACKSIZE environment variable to an appropriately large size to enable parallelization with this option.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as /arch (Windows*), -m (Linux* and macOS*), or [Q]x.

An example of the command using auto-parallelization is as follows:

Commanding auto-parallelization in Linux*

```
ifort -c -parallel myprog.f
```

Commanding auto-parallelization in Windows*

```
ifort -c /Qparallel myprog.f
```

Commanding auto-parallelization in macOS*

```
ifort -c -parallel myprog.f
```

Auto-parallelization uses two specific directives: !DIR$ PARALLEL and !DIR$ NOPARALLEL.

The format of an auto-parallelization compiler directive is below:

Syntax

```
!DIR$ <directive>
```

Because auto-parallelization directives begin with an exclamation point, the directives take the form of comments if you omit the [Q]parallel option.

The !DIR$ PARALLEL directive instructs the compiler to ignore dependencies that it assumes may exist and that would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored. In addition, PARALLEL [ALWAYS] overrides the compiler heuristics that estimate the likelihood that parallelization of a loop increases performance. It allows a loop to be parallelized even if the compiler thinks parallelization may not improve performance. If the ASSERT keyword is added, as in !DIR$ PARALLEL [ALWAYS [ASSERT]], the compiler generates an error-level assertion message saying that the compiler analysis and cost model indicate that the loop cannot be parallelized.

The !DIR$ NOPARALLEL directive disables auto-parallelization.

For example, in the following code, the NOPARALLEL directive disables auto-parallelization.

Example

```
program main
parameter (n=100
integer x(n),a(n)
!DIR$ NOPARALLEL
do i=1,n
  x(i) = i
enddo
!DIR$ PARALLEL
do i=1,n
  a( x(i) ) = i
enddo
end
```

See Also

parallel, Qparallel
compiler option

Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP*, such as the worksharing construct (with the PARALLEL DO directive). This section provides details on auto-parallelization.

Guidelines for Effective Auto-parallelization Usage

A loop can be parallelized if it meets the following criteria:

- The loop is countable at compile time: This means that an expression representing how many times the loop will execute (loop trip count) can be generated just before entering the loop.
- There are no FLOW (*READ* after *WRITE*), OUTPUT (*WRITE* after *WRITE*) or ANTI (*WRITE* after *READ*) loop-carried data dependencies. A loop-carried data dependency occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by run-time dependency testing.

The compiler may generate a run-time test for the profitability of executing in parallel for loop, with loop parameters that are not compile-time constants.

Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible; use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, procedure calls, ambiguous indirect references or global references.
- Insert the !DIR$ PARALLEL directive to disambiguate assumed data dependencies.

- Insert the `!DIR$ NOPARALLEL` directive before loops known to have insufficient work to justify the overhead of sharing among threads.

Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

1. **Data flow analysis:** Computing the flow of data through the program.
2. **Loop classification:** Determining loop candidates for parallelization based on correctness and efficiency, as shown by Enabling Auto-parallelization.
3. **Dependency analysis:** Computing the dependency analysis for references in each loop nest.
4. **High-level parallelization:** Analyzing the dependency graph to determine loops that can execute in parallel, and computing run-time dependency.
5. **Data partitioning:** Examining data reference and partition based on the following types of access: *SHARED*, *PRIVATE*, and *FIRSTPRIVATE*.
6. **Multithreaded code generation:** Modifying loop parameters, generating entry/exit per threaded task, and generating calls to parallel run-time routines for thread creation and synchronization.

NOTE
Options that use OpenMP* are available for both Intel® and non-Intel microprocessors, but these options may perform additional optimizations on Intel® microprocessors than they perform on non-Intel microprocessors. The list of major, user-visible OpenMP* constructs and features that may perform differently on Intel® microprocessors than on non-Intel microprocessors includes: locks (internal and user visible), the SINGLE construct, barriers (explicit and implicit), parallel loop scheduling, reductions, memory allocation, and thread affinity and binding.

See Also
Enabling Auto-parallelization

Enabling Further Loop Parallelization for Multicore Platforms

Parallelizing loops for multicore platforms is subject to certain conditions. Three requirements must be met for the compiler to parallelize a loop:

- The number of iterations must be known before entry into a loop to insure that the work can be divided in advance. A `do while` loop, for example, usually cannot be made parallel.
- There can be no jumps into or out of the loop.
- The loop iterations must be independent (no cross-iteration dependencies).

Correct results must not logically depend on the order in which the iterations are executed. There may be slight variations in the accumulated rounding error, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location, for example, if they depend on function arguments, run-time data, or the results of complex calculations.

If the compiler cannot prove that pointers or array references are safe, it will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time.

If you know parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using the !DIR$ PARALLEL directive.

Parallelizing Loops with Cross-iteration Dependencies

Before the compiler can auto-parallelize a loop, it must prove that the loop does not have potential cross-iteration dependencies that prevent parallelization. A cross-iteration dependency exists if a memory location is written to in an iteration of a loop and accessed (read from or written to) in another iteration of the loop. Cross-iteration dependencies often occur in loops that access overlapping array ranges, such as a loop that reads from `a(1:100)` and writes to `a(0:99)`.

Sometimes, even though a loop does not have cross-iteration dependencies, the compiler does not have enough information to prove it and does not parallelize the loop. In such cases, you can assist the compiler by providing additional information about the loop using the !DIR$ PARALLEL directive. Adding the `!DIR$ PARALLEL` directive before a `DO` loop informs the compiler that the loop does not have cross-iteration dependencies. Auto-parallelization analysis ignores potential dependencies that it assumes could exist; however, the compiler still may not parallelize the loop if heuristics estimate parallelization is unlikely to increase performance of the loop.

The Fortran DO CONCURRENT construct also specifies that there are no cross-iteration dependencies.

The `!DIR$ PARALLEL ALWAYS` directive has the same effect to ignore potential dependencies as the `!DIR$ PARALLEL` directive, but it also overrides the compiler heuristics that estimate the likelihood that parallelization of a loop would increase performance. It allows a loop to be parallelized even when the compiler estimates that parallelization might not improve performance.

The `!DIR$ NOPARALLEL` directive prevents auto-parallelization of the immediately following `DO` loop. Unlike `!DIR$ PARALLEL`, which is a hint, the `NOPARALLEL` directive is guaranteed to prevent parallelization of the following loop.

These directives take effect only if auto-parallelization is enabled by the option `[Q]parallel`.

Parallelizing Loops with Private Clauses

When you use the Guided Auto Parallelism feature, the compiler's auto-parallelizer gives you advice on where to alter your program to enhance parallelization. For instance, you may get advice to check if a condition (that the compiler could not prove) is true, and if true, to insert `!DIR$ PARALLEL` in your source code so that the associated loop is parallelized when you recompile.

To specify that it is legal for each thread to create a new, private copy (not visible by other threads) of a variable, and replace the original variable in the loop with the new private variable, use the `!DIR$ PARALLEL` directive with the *private* clause. The *private* clause allows you to list scalar and array type variables and specify the number of array elements to privatize.

Use the *firstprivate* clause to specify private variables that need to be initialized with the original value before entering the parallel loop.

Use the *lastprivate* clause to specify those variables with a value you want to reuse after it exits a parallelized loop. When you use the *lastprivate* clause to handle a particular privatized variable, the value is copied to the original variable when it exits from the parallelized loop.

NOTE
Do not use the same variable in both *private* and *lastprivate* clauses for the same loop. You will get an error message.

Parallelizing Loops with External Function Calls

The compiler can only effectively analyze loops with a relatively simple structure. For example, the compiler cannot determine the thread safety of a loop containing external function calls because it does not know whether the function call might have side effects that introduce dependencies. Fortran90 programmers can

use the PURE attribute to assert that subroutines and functions contain no side effects. You can invoke interprocedural optimization with the [Q]ipo option. Using this option gives the compiler the opportunity to analyze the called function for side effects.

Parallelizing Loops with OpenMP*

When the compiler is unable to automatically parallelize loops you know to be parallel, use OpenMP*. OpenMP* is the preferred solution because you understand the code better than the compiler and can express parallelism at a coarser granularity. Alternatively, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine-grained parallelism using vectorization or software pipelining.

Threshold Parameter to Parallelize Loops

If a loop can be parallelized, it does not necessarily mean that it should be parallelized. The compiler uses a threshold parameter to decide whether to parallelize a loop. The [Q]par-threshold compiler option adjusts this behavior. The threshold ranges from *0* to *100* , where *0* instructs the compiler to always parallelize a safe loop and *100* instructs the compiler to only parallelize those loops for which a performance gain is highly probable. Use the [Q]par-report option to determine which loops were parallelized. The compiler will also report which loops could not be parallelized and indicate probable reason(s) why. See OpenMP* and Parallel Processing Options for more information on the using these compiler options.

The following example illustrates using the options in combination.

Example code

```
subroutine add(k, a, b)
 integer :: k
 real :: a(10000), b(10000)
  DO i = 1, 10000
   a(i) = a(i+k) + b(i)
  end do
end subroutine add
```

Because the compiler does not know the value of *k*, the compiler assumes the iterations depend on each other, for example if *k* equals -1, even if the actual case is otherwise. You can override the compiler by inserting the !DIR$ PARALLEL directive.

Example

```
subroutine add(k, a, b)
  integer :: k
  real :: a(10000), b(10000)
  !DIR$ PARALLEL
   do i = 1, 10000
    a(i) = a(i+k) + b(i)
     end do
end subroutine add
```

Caution
Do not call this function with a value of *k* that is less than 10000; passing a value less than 10000 could lead to incorrect results.

See Also

PARALLEL
directive

OpenMP* and Parallel Processing Options
qopt-report-phase, Qopt-report-phase compiler option
opt-report, Qopt-report
compiler option
par-threshold, Qpar-threshold
compiler option
ipo, Qipo
compiler option

Vectorization

Vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (a series of adjacent values).

Automatic Vectorization

Automatic Vectorization Overview

The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD instructions in the Intel® Streaming SIMD Extensions (Intel® SSE, Intel® SSE2, Intel® SSE3 and Intel® SSE4), Supplemental Streaming SIMD Extensions (SSSE3) instruction sets, and the Intel® Advanced Vector Extensions (Intel® AVX, Intel® AVX2) instruction sets. The vectorizer detects operations in the program that can be done in parallel and converts the sequential operations to parallel; for example, the vectorizer converts the sequential SIMD instruction that processes up to 16 elements into a parallel operation, depending on the data type.

Automatic vectorization occurs when the Intel® Compiler generates packed SIMD instructions to unroll a loop. Because the packed instructions operate on more than one data element at a time, the loop executes more efficiently. This process is referred to as auto-vectorization only to emphasize that the compiler identifies and optimizes suitable loops on its own, without requiring any special action by you. However, it is useful to note that in some cases, certain keywords or directives may be applied in the code for auto-vectorization to occur.

The compiler supports a variety of auto-vectorizing hints that can help the compiler to generate effective vector instructions. Automatic vectorization is supported on IA-32 and Intel® 64 architectures. Intel® Advisor XE, a separate tool included in some editions of Intel® Parallel Studio XE, provides a Vectorization Advisor feature that can analyze the compiler's optimization reports and make recommendations for enhancing vectorization.

NOTE
Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

Programming Guidelines for Vectorization

The goal of including the vectorizer component in the Intel® Fortran Compiler is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help by supplying the compiler with additional information; for example, by using auto-vectorizer hints or directives.

NOTE
Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

Guidelines to Vectorize Innermost Loops

Follow these guidelines to vectorize innermost loop bodies.

Use:

- straight-line code (a single basic block)
- vector data only; that is, arrays and invariant expressions on the right hand side of assignments.

  Array references can appear on the left hand side of assignments.
- only assignment statements.

Avoid:

- function calls (other than math library calls)
- non-vectorizable operations (either because the loop cannot be vectorized, or because an operation is emulated through a number of instructions)
- mixing vectorizable types in the same loop (leads to lower resource utilization)
- data-dependent loop exit conditions (leads to loss of vectorization)

To make your code vectorizable, you will often need to make some changes to your loops. You should only make changes needed to enable vectorization, and avoid these common changes:

- loop unrolling, which the compiler performs automatically
- decomposing one loop with several statements in the body into several single-statement loops

Restrictions

There are a number of restrictions that you should consider. Vectorization depends on two major factors: hardware and style of source code.

Factor	Description
Hardware	The compiler is limited by restrictions imposed by the underlying hardware. In the case of Intel® Streaming SIMD Extensions (Intel® SSE), the vector memory operations are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.
Style of source code	The style in which you write source code can inhibit vectorization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, pointer arithmetic, and memory operations within the loop bodies.

By understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization.

Guidelines for Writing Vectorizable Code

Follow these guidelines to write vectorizable code:

- Use simple `DO` loops. Avoid complex loop termination conditions – the upper iteration limit must be invariant within the loop. For the innermost loop in a nest of loops, you could set the upper limit iteration to be a function of the outer loop indices.
- Write straight-line code. Avoid branches such as `DO`, `GOTO`, or `ASSIGN` statements; most function calls; or `IF` constructs that can not be treated as masked assignments.
- Avoid dependencies between loop iterations or at the least, avoid read-after-write dependencies.
- Try to use array notations instead of the use of pointers. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Wherever possible, use the loop index directly in array subscripts instead of incrementing a separate counter for use as an array address.
- Access memory efficiently:
  - Favor inner loops with unit stride.
  - Minimize indirect addressing.
  - Align your data to 16-byte boundaries (for Intel® SSE instructions).
- Choose a suitable data layout with care. Most multimedia extension instruction sets are rather sensitive to alignment. The data movement instructions of Intel® SSE, for example, operate much more efficiently on data that is aligned at a 16-byte boundary in memory. Therefore, the success of a vectorizing compiler also depends on its ability to select an appropriate data layout which, in combination with code restructuring (like loop peeling), results in aligned memory accesses throughout the program.
- Use aligned data structures: Data structure alignment is the adjustment of any data object in relation with other objects.

  Caution
  Use this hint with care. Incorrect usage of aligned data movements result in an exception when using Intel® SSE.

- Use structure of arrays (SoA) instead of array of structures (AoS): An array is the most common type of data structure that contains a contiguous collection of data items that can be accessed by an ordinal index. You can organize this data as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization is excellent for encapsulation it can be a hindrance for use of vector processing. To make vectorization of the resulting code more effective, you can also select appropriate data structures.

Dynamic Alignment Optimizations

Dynamic alignment optimizations can improve the performance of vectorized code, especially for long trip count loops. Disabling such optimizations can decrease performance, but it may improve bitwise reproducibility of results, factoring out data location from possible sources of discrepancy.

To enable or disable dynamic data alignment optimizations, specify the option `Qopt-dynamic-align[-]` (Windows) or `[no-]qopt-dynamic-align[-]` (Linux).

Using Aligned Data Structures

Data structure alignment is the adjustment of any data object with relation to other objects. The Intel® Fortran Compiler may align individual variables to start at certain addresses to speed up memory access. Misaligned memory accesses can incur large performance losses on certain target processors that do not support them in hardware.

Alignment is a property of a memory address, expressed as the numeric address modulo of powers of two. In addition to its address, a single datum also has a size. A datum is called 'naturally aligned' if its address is aligned to its size, otherwise it is called 'misaligned'. For example, an 8-byte floating-point datum is naturally aligned if the address used to identify it is aligned to eight (8).

A data structure is a way of storing data in a computer so that it can be used efficiently. Often, a carefully chosen data structure allows a more efficient algorithm to be used. A well-designed data structure allows a variety of critical operations to be performed, using as little resources - both execution time and memory space - as possible.

In the example data structure above, if the type `short` is stored in two bytes of memory then each member of the data structure is aligned to a boundary of two bytes. `Data1` would be at offset `0`, `Data2` at offset `2` and `Data3` at offset `4`. The size of this structure is six bytes. The type of each member of the structure usually has a required alignment, meaning that it is aligned on a pre-determined boundary, unless you request otherwise. In cases where the compiler has taken sub-optimal alignment decisions, you can use the declaration `declspec(align(base,offset))`, where `0 <= offset < base` and `base` is a power of two, to allocate a data structure at offset from a certain base.

If the first element of both arrays is aligned at a 16-byte boundary, then either an unaligned load of elements from `b` or an unaligned store of elements into `a` must be used after vectorization.

NOTE
In this case, peeling off an iteration will not help.

However, you can enforce the alignment shown below, which results in two aligned access patterns after vectorization (assuming an 8-byte size for doubles):

Run-time optimization provides a generally effective way to obtain aligned access patterns at the expense of a slight increase in code size and testing. If incoming access patterns are guaranteed to be aligned at a 16-byte boundary, you can avoid this overhead with the hint `__assume_aligned(x, 16);` in the function to convey this information to the compiler.

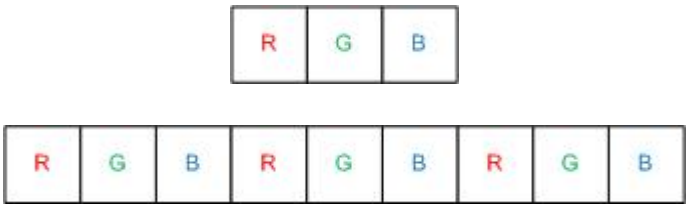
For example, suppose you can introduce an optimization in the case where a block of memory with address `n2` is aligned on a 16-byte boundary. You could use `_assume(n2%16==0)`.

Caution
Use this hint with care. Incorrect use of aligned data movements result in an exception for Intel® SSE.

Using Structure of Arrays versus Array of Structures

The most common and well-known data structure is the array that contains a contiguous collection of data items, which can be accessed by an ordinal index. This data can be organized as an array of structures (AoS) or as a structure of arrays (SoA). While AoS organization works excellently for encapsulation, for vector processing it works poorly.

You can select appropriate data structures to make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array of structures (AoS) arrangement for storing the `r`, `g`, `b` components of a set of three-dimensional points with the alternative structure of arrays (SoA) arrangement for storing this set.



With the AoS arrangement, a loop that visits all components of an RGB point before moving to the next point exhibits a good locality of reference because all elements in the fetched cache lines are utilized. The disadvantage of the AoS arrangement is that each individual memory reference in such a loop exhibits a non-unit stride, which, in general, adversely affects vector performance. Furthermore, a loop that visits only one component of all points exhibits less satisfactory locality of reference because many of the elements in the fetched cache lines remain unused.

In contrast, with the SoA arrangement the unit-stride memory references are more amenable to effective vectorization and still exhibit good locality of reference within each of the three data streams. Consequently, an application that uses the SoA arrangement may ultimately outperform an application based on the AoS arrangement when compiled with a vectorizing compiler, even if this performance difference is not directly apparent during the early implementation phase.

Before you start vectorization, try out some simple rules:

- Make your data structures vector-friendly.
- Make sure that inner loop indices correspond to the outermost (last) array index in your data (row-major order).
- Use structure of arrays over array of structures.

For instance when dealing with three-dimensional coordinates, use three separate arrays for each component (SoA), instead of using one array of three-component structures (AoS). To avoid dependencies between loops that will eventually prevent vectorization, use three separate arrays for each component (SoA), instead of one array of three-component structures (AoS). When you use the AoS arrangement, each iteration produces one result by computing XYZ, but it can at best use only 75% of the SSE unit because the forth component is not used. Sometimes, the compiler may use only one component (25%). When you use the SoA arrangement, each iteration produces four results by computing XXXX, YYYY and ZZZZ, using 100% of the SSE unit. A drawback for the SoA arrangement is that your code will likely be three times as long. On the other hand, the compiler might not be able to vectorize AoS arranged code at all.

If your original data layout is in AoS format, you may even want to consider a conversion to SoA on the fly, before the critical loop. If it gets vectorized, it may be worth the effort!

To summarize:

- Use the smallest data types that gives the needed precision to maximize potential SIMD width. (If only 16-bits are needed, using a `short` rather than an `int` can make the difference between 8-way or four-way SIMD parallelism, respectively.)
- Avoid mixing data types to minimize type conversions.
- Avoid operations not supported in SIMD hardware.
- Use all the instruction sets available for your processor. Use the appropriate command line option for your processor type, or select the appropriate IDE option (Windows* only):
  - **Project** > **Properties** > **Visual Fortran** > **Code Generation** > **Intel Processor-Specific Optimization**, if your application runs only on Intel® processors.
  - **Project** > **Properties** > **Visual Fortran** > **Code Generation** > **Enable Enhanced Instruction Set**, if your application runs on compatible, non-Intel processors.
- Vectorizing compilers usually have some built-in efficiency heuristics to decide whether vectorization is likely to improve performance. The Intel®Fortran Compiler disables vectorization of loops with many unaligned or non-unit stride data access patterns. If experimentation reveals that vectorization improves performance, you can override this behavior using the `!DIR$ VECTOR ALWAYS` hint before the loop; the compiler vectorizes any loop regardless of the outcome of the efficiency analysis (provided, of course, that vectorization is safe).

See Also

Vectorization and Loops

Loop Constructs

opt-dynamic-align, Qopt-dynamic-align
compiler option

Using Automatic Vectorization

Automatic vectorization is supported on IA-32 and Intel® 64 architectures. The information below will guide you in setting up the auto-vectorizer.

Vectorization Speed-up

Where does the vectorization speedup come from? Consider the following sample code fragment, where `a`, `b` and `c` are integer arrays:

Sample Code Fragment

```
do I=1,MAX
    C(I)=A(I)+B(I)
    end do
```

If vectorization is not enabled, that is, you compile using the `O1` or `-no-vec-` (or `/Qvec-`) option, for each iteration, the compiler processes the code such that there is a lot of unused space in the SIMD registers, even though each of the registers could hold three additional integers. If vectorization is enabled (compiled using `O2` or higher options), the compiler may use the additional registers to perform four additions in a single instruction. The compiler looks for vectorization opportunities whenever you compile at default optimization (`O2`) or higher.

NOTE
Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

Tip
To allow comparisons between vectorized and not-vectorized code, disable vectorization using the `/Qvec-` (Windows*) or `-no-vec` (Linux* or macOS*) option; enable vectorization using the `O2` option.

To get information on whether a loop was vectorized or not, enable generation of the optimization report using the options `Qopt-report:1 Qopt-report-phase:vec` (Windows) or `qopt-report=1 qopt-report-phase=vec` (Linux and macOS*) options. These options generate a separate report in an `*.optrpt` file that includes optimization messages. In Visual Studio, the program source is annotated with the report's messages, or you can read the resulting `.optrpt` file using a text editor. A message appears for every loop that is vectorized, such as:

Example: Vectorization Report

```
> ifort /Qopt-report1 matvec.f90
> type matvec.optrpt
…
```

Example: Vectorization Report

```
LOOP BEGIN at C:\Projects\vec_samples\matvec.f90(38,6)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

The source line number (38 in the above example) refers to either the beginning or the end of the loop.

To get details about the type of loop transformations and optimizations that took place, use the `[Q]opt-report-phase` option by itself or along with the `[Q]opt-report` option.

How significant is the performance enhancement? To evaluate performance enhancement yourself, run *vec_samples*:

1. Open an Intel® Compiler command line window.
   - **On Windows*:** Under the **Start** menu item for your Intel product, select an icon under **Compiler and Performance Libraries** > **Command Prompt with Intel Compiler**
   - **On Linux*and macOS*:**Source an environment script such as `compilervars.sh` or the `compilervars.csh` in the `<installdir>/bin` directory and use the attribute appropriate for the architecture.
2. Navigate to the `<install-dir>\Samples\<locale>\Fortran\` directory. On Windows, unzip the sample project `vec_samples.zip` to a writable directory. This small application multiplies a vector by a matrix using the following loop:

   Example: Vector Matrix Multiplication

   ```
       do i=1,size1
          c(i) = c(i) + a(i,j) * b(j)
       end do
   ```

3. Build and run the application, first without enabling auto-vectorization. The default `O2` optimization enables vectorization, so you need to disable it with a separate option. Note the time taken for the application to run.

   Example: Building and Running an Application without Auto-vectorization

   ```
   // (Linux* and macOS*)
   ifort -no-vec  driver.f90 matvec.f90 -o NoVectMult
   ./NoVectMult

   // (Windows*)
   ifort /Qvec- driver.f90 matvec.f90 /exe:NoVectMult
   NoVectMult
   ```

4. Now build and run the application, this time with auto-vectorization. Note the time taken for the application to run.

   Example: Building and Running an Application with Auto-vectorization

   ```
   // (Linux* and macOS*)
   ifort driver.f90 matvec.f90 -o VectMult
   ./VectMult

   // (Windows*)
   ifort driver.f90 matvec.f90 /exe:VectMult
   VectMult
   ```

When you compare the timing of the two runs, you may see that the vectorized version runs faster. The time for the non-vectorized version is only slightly faster than would be obtained by compiling with the `O1` option.

Obstacles to Vectorization

The following do not always prevent vectorization, but frequently either prevent it or cause the compiler to decide that vectorization would not be worthwhile.

- **Non-contiguous memory access:** Four consecutive integers or floating-point values, or two consecutive doubles, may be loaded directly from memory in a single SSE instruction. But if the four integers are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient. The most common examples of non-contiguous memory access are loops with non-unit stride or with indirect addressing, as in the examples below. The compiler rarely vectorizes such loops, unless the amount of computational work is large compared to the overhead from non-contiguous memory access.

  Example: Non-contiguous Memory Access

  ```
  ! arrays accessed with stride 2
  do I=1,SIZE,2; B(I)=B(I)+(A(I)*X(I)); end do

  ! inner loop accesses a A with stride SIZE
  do J=1,SIZE
    do I=1,SIZE
      B(I)=B(I)+(A(J,I)*X(J))
      end do
    end do

  ! indirect addressing of x X using index array INDX
  do I=1,SIZE,2; B(I)=B(I)+(A(I)*X(INDX(I))); end do
  ```

  The typical message from the vectorization report is: `vectorization possible but seems inefficient`, although indirect addressing may also result in the following report: `Existence of vector dependence`.
- **Data dependencies:** Vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation.
  - The simplest case is when data elements that are written (stored to) do not appear in any other iteration of the individual loop. In this case, all the iterations of the original loop are independent of each other, and can be executed in any order, without changing the result. The loop may be safely executed using any parallel method, including vectorization. All the examples considered so far fall into this category.
  - When a variable is written in one iteration and read in a subsequent iteration, there is a "read-after-write" dependency, also known as a flow dependency, as in this example:

    Example: Flow Dependency

    ```
    DO J=2,5
    A(J)=A(J-1)+1
    END DO
    ```

    So the value of j gets propagated to all A(J). This cannot safely be vectorized: if the first two iterations are executed simultaneously by a SIMD instruction, the value of A(2) is used by the second iteration before it has been calculated by the first iteration.
  - When a variable is read in one iteration and written in a subsequent iteration, this is a *write-after-read* dependency, also known as an *anti-dependency*, as in the following example:

Example: Write-after-read Dependency

```
do J=2,MAX; A(J-1)=A(J)+1; end do
          ! this is equivalent to:
   A(1)=A(2)+1
   A(2)=A(3)+1
   A(3)=A(4)+1
   A(4)=A(5)+1
```

This write-after-read dependency is not safe for general parallel execution, since the iteration with the write may execute before the iteration with the read. However, for vectorization, no iteration with a higher value of *j* can complete before an iteration with a lower value of *j*, and so vectorization is safe (that is, it gives the same result as non-vectorized code) in this case. The following example, however, may not be safe, since vectorization might cause some elements of *A* to be overwritten by the first SIMD instruction before being used for the second SIMD instruction.

Example: Unsafe Vectorization

```
do J=1,MAX
  A(J-1)=A(J)+1
  B(J)=A(J)*2
  end do

   ! this is equivalent to:
   A(1)=A(2)+1
   A(2)=A(3)+1
   A(3)=A(4)+1
   A(4)=A(5)+1
```

- Read-after-read situations are not really dependencies, and do not prevent vectorization or parallel execution. If a variable is unwritten, it does not matter how often it is read.
- Write-after-write, or 'output', dependencies, where the same variable is written to in more than one iteration, are in general unsafe for parallel execution, including vectorization.
- One important exception, that apparently contains all of the above types of dependency:

Example: Dependency Exception

```
MYSUM=0
do J=1,MAX; MYSUM = MYSUM + A(J)*B(J); end do
```

Although `MYSUM` is both read and written in every iteration, the compiler recognizes such reduction idioms, and is able to vectorize them safely. The loop in the first example was another example of a reduction, with a loop-invariant array element in place of a scalar.

These types of dependencies between loop iterations are sometimes known as loop-carried dependencies.

The above examples are of proven dependencies. The compiler cannot safely vectorize a loop if there is even a potential dependency. Consider the following example:

Example: Potential Dependency

```
real, pointer :: A(:),B(:),C(:)
…
do I=1,SIZE; C(I)=A(I)*B(I); end do
```

In the above example, the compiler needs to determine whether, for some iteration `I`, `C(I)` might refer to the same memory location as `A(I)` or`B(I)` for a different iteration. Such memory locations are sometimes said to be *aliased*. For example, if `A(I)` pointed to the same memory location as `C(I-1)`, there would be a read-after-write dependency as in the earlier example. If the compiler cannot exclude this possibility, it will not vectorize the loop unless you provide the compiler with hints. You can also avoid this problem by making the arrays ALLOCATABLE instead of POINTER, as the compiler knows these cannot be aliased.

See Also

ansi-alias/Qansi-alias
compiler option
qopt-report, Qopt-report compiler option
qopt-report-phase, Qopt-report-phase compiler option

Vectorization and Loops

This topic provides more information on the interaction between the auto-vectorizer and loops.

Interactions with Loop Parallelization

Combine the [Q]parallel and [Q]x options to instruct the Intel® Fortran Compiler to attempt both Automatic Parallelization and automatic loop vectorization in the same compilation.

NOTE
Using this option enables parallelization for both Intel® microprocessors and non-Intel microprocessors. The resulting executable may get additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The parallelization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

NOTE
Using this option enables vectorization at default optimization levels for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop.

See Programming with Auto-parallelization and Programming Guidelines for Vectorization.

In some rare cases, a successful loop parallelization (either automatically or by means of OpenMP* directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way; for example, in the cases where the `/Qopt-report:2 /Qopt-report-phase:vec` (Windows) or `-qopt-report=2 -qopt-report-phase=vec` (Linux and macOS*) options indicate that loops were not successfully vectorized.

Types of Vectorized Loops

For integer loops, the 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) and the Intel® Advanced Vector Extensions (Intel® AVX) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types, with limited support for the 64-bit integer data type.

Vectorization may proceed if the final precision of integer wrap-around arithmetic is preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Also, note that because the Intel® SSE and the Intel® AVX instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, Intel® SSE provides SIMD instructions for the following arithmetic operators:

- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)

Additionally, Intel® SSE provide SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, and `TAN`) are supported in software in a vector mathematical run-time library that is provided with the Intel® Fortran Compiler.

To be vectorizable, loops must be:

- **Countable:** The loop trip count must be known at entry to the loop at runtime, though it need not be known at compile time (that is, the trip count can be a variable but the variable must remain constant for the duration of the loop). This implies that exit from the loop must not be data-dependent.
- **Innermost loop of a nest:** The only exception is if an original outer loop is transformed into an inner loop as a result of some other prior optimization phase, such as unrolling, loop collapsing or interchange, or an original outermost loop is transformed to an innermost loop due to loop materialization.
- **Without function calls:** Even a `print` statement is sufficient to prevent a loop from getting vectorized. The vectorization report message is typically: non-standard loop is not a vectorization candidate. The two major exceptions are for intrinsic math functions and for functions that may be inlined.

Intrinsic math functions are allowed, because the compiler runtime library contains vectorized versions of these functions. See the table below for a list of these functions; most exist in both float and double versions.

acos	atanh	exp2	tan
acosh	ceiling	floor	tanh
anint	cos	log	
asin	cosh	log10	
asinh	erf	sin	
atan	erfc	sinh	
atan2	exp	sqrt	

Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

Integer Array Operations

The statements within the loop body may be arithmetic or logical operations (again, typically for arrays). Arithmetic operations are limited to such operations as addition, subtraction, ABS, MIN, and MAX. Logical operations include bitwise AND, OR, and XOR operators. You can mix data types but this may potentially cost you in terms of lowering efficiency.

Other Operations

No statements other than the preceding floating-point and integer operations are valid. The loop body cannot contain any function calls other than the ones described above.

Data Dependency

Data dependency relations represent the required ordering constraints on the operations in serial loops. Because vectorization rearranges the order in which operations are executed, any auto-vectorizer must have at its disposal some form of data dependency analysis.

An example where data dependencies prohibit vectorization is shown below. In this example, the value of each element of an array is dependent on the value of its neighbor that was computed in the previous iteration.

Example 1: Data-dependent Loop

```
subroutine dep(data, n)
  real :: data(n)
  integer :: i
 do i = 1, n-1
    data(i) = data(i-1)*0.25 + data(i)*0.5 + data(i+1)*0.25
  end do
end subroutine dep
```

The loop in the above example is not vectorizable because the WRITE to the current element `data(i)` is dependent on the use of the preceding element `data(i-1)`, which has already been written to and changed in the previous iteration. To see this, look at the access patterns of the array for the first two iterations as shown below.

Example 2: Data-dependency Vectorization Patterns

```
I=1: READ DATA(0)
      READ DATA(1)
      READ DATA(2)
      WRITE DATA(1)
I=2: READ DATA(1)
      READ DATA(2)
      READ DATA(3)
      WRITE DATA(2)
```

In the normal sequential version of this loop, the value of `DATA(1)` read from during the second iteration was written to in the first iteration. For vectorization, it must be possible to do the iterations in parallel, without changing the semantics of the original loop.

Example 3: Data Independent Loop

```
  do i=1,100
   a(i)=b(i)
    end do

! which has the following access pattern
    read b(1)
    write a(1)
    read b(2)
    write b(2)
```

Data Dependency Analysis

Data dependency analysis involves finding the conditions under which two memory accesses may overlap. Given two references in a program, the conditions are defined by:

- whether the referenced variables may be aliases for the same (or overlapping) regions in memory.
- for array references, the relationship between the subscripts.

The data dependency analyzer for array references is organized as a series of tests, which progressively increase in power as well as in time and space costs.

First, a number of simple tests are performed in a dimension-by-dimension manner, since independency in any dimension will exclude any dependency relationship. Multidimensional arrays references that may cross their declared dimension boundaries can be converted to their linearized form before the tests are applied.

Some of the simple tests that can be used are the fast greatest common divisor (GCD) test and the extended bounds test. The GCD test proves independency if the GCD of the coefficients of loop indices cannot evenly divide the constant term. The extended bounds test checks for potential overlap of the extreme values in subscript expressions.

If all simple tests fail to prove independency, the compiler will eventually resort to a powerful hierarchical dependency solver that uses Fourier-Motzkin elimination to solve the data dependency problem in all dimensions.

See Also

Automatic Parallelization
Programming with Auto-parallelization
Programming Guidelines for Vectorization
qopt-report, Qopt-report compiler option
qopt-report-phase, Qopt-report-phase compiler option
x, Qx compiler option
parallel, Qparallel compiler option

Loop Constructs

Loops can be formed with the usual `DO-END DO` and `DO WHILE`, or by using an `IF`/`GOTO` and a label. Loops must have a single entry and a single exit to be vectorized. The following examples illustrate loop constructs that can and cannot be vectorized.

Example: Vectorizable structure

```
subroutine vec(a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (i .le. 100)
    a(i) = b(i) * c(i)
      if (a(i) .lt. 0.0) a(i) = 0.0
        i = i + 1
  enddo
end subroutine vec
```

The following example shows a loop that cannot be vectorized because of the inherent potential for an early exit from the loop.

Example: Non-vectorizable structure

```
subroutine no_vec(a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (i .le. 100)
    a(i) = b(i) * c(i)
! The next statement allows early
!
exit from the loop and prevents
! vectorization of the loop.
    if (a(i) .lt. 0.0) go to 10
    i = i + 1
  enddo
  10 continue
end subroutine no_vecN
END
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; in other words, the number of iterations must be expressed as one of the following:

- A constant.
- A loop invariant term.
- A linear function of outermost loop indices.

In the case where a loops exit depends on computation, the loops are not countable. The examples below show loop constructs that are countable and non-countable.

Example: Countable Loop

```
subroutine cnt1 (a, b, c, n, lb)
  dimension a(n), b(n), c(n)
  integer n, lb, i, count
! Number of iterations is "n - lb + 1"
  count = n
  do while (count .ge. lb)
    a(i) = b(i) * c(i)
    count = count - 1
    i = i + 1
  enddo ! lb is not defined within loop
end
```

The following example demonstrates a different countable loop construct.

Example: Countable Loop

```
! Number of iterations is (n-m+2)/2
subroutine cnt2 (a, b, c, m, n)
  dimension a(n), b(n), c(n)
  integer i, l, m, n
  i = 1;
```

Example: Countable Loop

```
  do l = m,n,2
    a(i) = b(i) * c(i)
    i = i + 1
  enddo
end
```

The following examples demonstrates a loop construct that is non-countable due to dependency loop variant count value.

Example: Non-Countable Loop

```
! Number of iterations is dependent on a(i)
subroutine foo (a, b, c)
  dimension a(100),b(100),c(100)
  integer i
  i = 1
  do while (a(i) .gt. 0.0)
    a(i) = b(i) * c(i)
    i = i + 1
  enddo
end
```

Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- By increasing the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- By reducing the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation. In the case of Intel® Streaming SIMD Extensions, this vector or strip-length is reduced by four times: four floating-point data items per single Intel® SSE single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop. For example, assume the compiler attempts to strip-mine the following loop:

Example: Before Vectorization

```
i = 1
do while (i<=n)
  a(i) = b(i) + c(i) ! Original loop code
  i = i + 1
end do
```

The compiler might handle the strip mining and loop cleaning by restructuring the loop in the following manner:

Example: After Vectorization

```
!The vectorizer generates the following two loops
i = 1
do while (i < (n - mod(n,4)))
! Vector strip-mined loop.
  a(i:i+3) = b(i:i+3) + c(i:i+3)
  i = i + 4
end do
do while (i <= n)
  a(i) = b(i) + c(i)     !Scalar clean-up loop
  i = i + 1
end do
```

Loop Blocking

It is possible to treat loop blocking as strip-mining in two or more dimensions. Loop blocking is a useful technique for memory performance optimization. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse.

Consider the following example. The two-dimensional array *A* is referenced in the *j* (column) direction and then in the *i* (row) direction (column-major order); array *B* is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array *A* and *B* for the code would be 1 and `MAX`, respectively. `BS = block_size;MAX` must be evenly divisible by *BS*.

Consider the following loop example code:

Example: Original loop

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX
  DO J = 1, MAX
    A(I,J) = A(I,J) + B(J,I)
  ENDDO
ENDDO
```

The arrays could be blocked into smaller chunks so that the total combined size of the two blocked chunks is smaller than the cache size, which can improve data reuse. One possible way of doing this is demonstrated below:

Example: Transformed Loop after blocking

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX, BS
  DO J = 1, MAX, BS
   DO II = I, I+MAX, BS-1
    DO J = J, J+MAX, BS-1
     A(II,JJ) = A(II,JJ) + B(JJ,II
    ENDDO
   ENDDO
  ENDDO
ENDDO
```

Loop Interchange and Subscripts: Matrix Multiply

Loop interchange is often used for improving memory access patterns. Matrix multiplication is commonly written as shown in the following example:

Example: Typical Matrix Multiplication

```
subroutine matmul_slow(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  do i = 1, n
    do j = 1, n
      do k = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
    end do
  end do
end subroutine matmul_slow
```

The use of *B*(*K*,*J*) is not a stride-1 reference and therefore will not be vectorized efficiently.

If the loops are interchanged, however, all the references will become stride-1 as shown in the following example.

Example: Matrix Multiplication with Stride-1

```
subroutine matmul_fast(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  do j = 1, n
    do k = 1, n
      do i = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
    enddo
  enddo
end subroutine matmul_fast
```

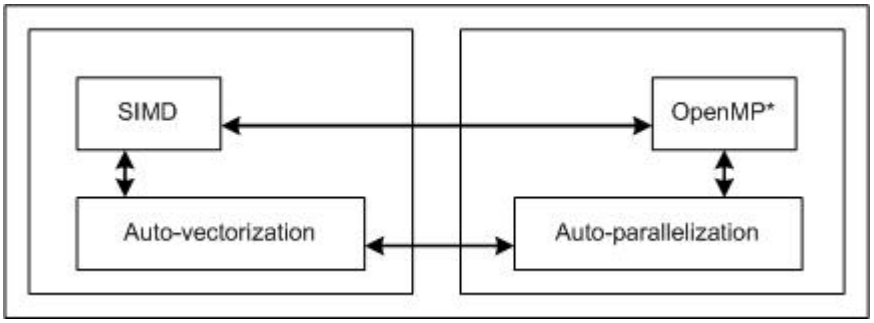
Interchanging is not always possible because of dependencies, which can lead to different results.

Explicit Vector Programming

User-Mandated or SIMD Vectorization

User-mandated or SIMD vectorization supplements automatic vectorization just like OpenMP* parallelization supplements automatic parallelization. The following figure illustrates this relationship. User-mandated vectorization is implemented as a single-instruction-multiple-data (SIMD) feature and is referred to as SIMD vectorization.

NOTE
The SIMD vectorization feature is available for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.



SIMD vectorization uses the `!$OMP SIMD` directive to effect loop vectorization. You must add this directive to a loop and recompile to vectorize the loop using the option `-qopenmp-simd` (Linux and macOS*) or `Qopenmp-simd` (Windows*).

Consider an example in Fortran where the compiler does not automatically vectorize the loop due to the unknown data dependence distance between `I` and `2*I`. If you know that `X` is large enough that data `A(I)` and `A(2*I)` don't overlap within a reasonable number of iterations, perhaps 64, you can enforce vectorization of the loop using `!$OMP SIMD`. If you know that they don't overlap in at least 8 iterations you may additionally specify `!$OMP SIMD SIMDLEN(8)` to avoid vectorization that is too wide, which might lead to overlap.

Example: without !$OMP SIMD

```
[D:/simd] cat example1.f
subroutine add(A, N, X)
integer N, X
real    A(N)
DO I=X, N
  A(I) = A(I) + A(2*I)
ENDDO
End
```

```
[D:/simd] ifort example1.f -c -nologo -Qopt-report2 -Qopt-report-phase=vec -Qopt-report-
file=stderr

Begin optimization report for: ADD

    Report from: Vector optimizations [vec]


LOOP BEGIN at example1.f(5,9)
<Multiversioned v1>
   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First
dependence is shown below. Use level 5 report for details
   remark #15346: vector dependence: assumed FLOW dependence between A(I) (6:11) and A(I*2)
(5:11)
LOOP END

LOOP BEGIN at example1.f(5,9)
<Remainder, Multiversioned v1>
LOOP END

LOOP BEGIN at example1.f(5,9)
<Multiversioned v2>
  remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END
```

Example: without !$OMP SIMD

```
LOOP BEGIN at example1.f(5,9)
<Remainder, Multiversioned v2>
LOOP END
===========================================================================
```

Example: with !$OMP SIMD

```
[D:/simd] cat example1.f
subroutine add(A, N, X)
integer N, X
real    A(N)
!X may be 8 or more, so on overlap with 8 iterations at least
!$OMP SIMD SIMDLEN(8)
DO I=X, N
  A(I) = A(I) + A(2*I)
ENDDO
End
```

```
[D:/simd] ifort example1.f -c -nologo -Qopt-report2 -Qopt-report-phase=vec -Qopt-report-
file=stderr -Qopenp-simd

Begin optimization report for: ADD

    Report from: Vector optimizations [vec]


LOOP BEGIN at example1.f(6,9)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at example1.f(6,9)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at example1.f(6,9)
<Remainder loop for vectorization>
LOOP END
===========================================================================
```

The one big difference between using `!$OMP SIMD` and auto-vectorization hints is that with `!$OMP SIMD`, the compiler generates a warning when it is unable to vectorize the loop. With auto-vectorization hints, actual vectorization is still under the discretion of the compiler, even when you use the `!DIR$ VECTOR ALWAYS` hint.

`!$OMP SIMD` has optional clauses to guide the compiler on how vectorization must proceed. Use these clauses appropriately so that the compiler obtains enough information to generate correct vector code. For more information on the clauses, see the `!$OMP SIMD` description.

Additional Semantics

Note the following points when using the `!$OMP SIMD` directive.

- A variable may belong to zero or one of the following: private, linear, or reduction.
- Within the vector loop, an expression is evaluated as a vector value if it is private, linear, reduction, or it has a sub-expression that is evaluated to a vector value. Otherwise, it is evaluated as a scalar value (that is, broadcast the same value to all iterations). Scalar value does not necessarily mean loop invariant, although that is the most frequently seen usage pattern of scalar value.

- A vector value may not be assigned to a scalar L-value. It is an error.
- A scalar L-value may not be assigned under a vector condition. It is an error.
- The computed `GOTO` statement is not supported.

Using `vector` Declaration

Consider the following Intel® Visual Fortran example code for a program to compare serial and vector computations using a user-defined function, `foo()`.

NOTE All code examples in this section are applicable for Fortran on Windows* only.

Example: Where user-defined function is not vectorized

```
!! file simdmain.f90
program simdtest
use IFPORT
! Test vector function in external file.
implicit none
interface
   integer function foo(a, b)
   integer a, b
   end function foo
end interface

integer, parameter :: M = 48, N = 64

  integer  i, j
  integer, dimension(M,N) :: a1
  integer, dimension(M,N) :: a2
  integer, dimension(M,N) :: s_a3
  integer, dimension(M,N) :: v_a3
logical :: err_flag = .false.

! compute random numbers for arrays
do j = 1, N
  do i = 1, M
   a1(i,j) = rand() * M
   a2(i,j) = rand() * M
  end do
end do

! compute serial results
do j = 1, N
!dir$ novector
  do i = 1, M
   s_a3(i,j) = foo(a1(i,j), a2(i,j))
  end do
end do

! compute vector results
  do j = 1, N
   do i = 1, M
    v_a3(i,j) = foo(a1(i,j), a2(i,j))
   end do
  end do
```

Example: Where user-defined function is not vectorized

```
! compare serial and vector results
do j = 1, N
  do i = 1, M
   if (s_a3(i,j) .ne. v_a3(i,j)) then
    err_flag = .true.
    print *, s_a3(i, j), v_a3(i,j)
   end if
  end do
 end do
if (err_flag .eq. .true.) then
  write(*,*) "FAILED"
   else
  write(*,*) "PASSED"
end if
end program

!! file: vecfoo.f90
integer function foo(a, b)
implicit none
integer, intent(in) :: a, b
  foo = a - b
end function
```

```
[49 C:/temp] ifort -nologo -qopt-report2 -qopt-report-phase=vec -qopt-report-file=stderr
simdmain.f90 vecfoo.f90

Begin optimization report for: SIMDTEST

    Report from: Vector optimizations [vec]


LOOP BEGIN at simdmain.f90(33,3)
   remark #15319: loop was not vectorized: novector directive used
LOOP END

LOOP BEGIN at simdmain.f90(54,2)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at simdmain.f90(47,3)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization. First
dependence is shown below.
Use level 5 report for details
      remark #15346: vector dependence: assumed OUTPUT dependence between at(50:5) and at (50:5)
   LOOP END
LOOP END


Non-optimizable loops:


LOOP BEGIN at simdmain.f90(28,2)
   remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.

   LOOP BEGIN at simdmain.f90(27,3)
```

Example: Where user-defined function is not vectorized

```
      remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.
   LOOP END
LOOP END

LOOP BEGIN at simdmain.f90(36,2)
   remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.
LOOP END

LOOP BEGIN at simdmain.f90(43,3)
   remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.

   LOOP BEGIN at simdmain.f90(42,4)
      remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.
   LOOP END
LOOP END
===========================================================================
```

When you compile the above code, the loop containing the `foo()` function is not auto-vectorized because the auto-vectorizer does not know what `foo()` does unless it is inlined to this call site.

In such cases where the function call is not inlined, you can use the `!DIR$ attributes vector::function-name-list` declaration to vectorize the loop and the function `foo()`. All you need to do is add the vector declaration to the function declaration, and recompile the code. The loop and function are vectorized.

Example: Where loop with user-defined function with vector declaration is auto-vectorized

```
!! file simdmain.f90
program simdtest
! Test vector function in external file.
use IFPORT
implicit none
interface
   integer function foo(a, b)
!$omp declare simd
   integer a, b
   end function foo
end interface

 integer, parameter :: M = 48, N = 64

  integer  i, j
  integer, dimension(M,N) :: a1
  integer, dimension(M,N) :: a2
  integer, dimension(M,N) :: s_a3
  integer, dimension(M,N) :: v_a3
logical :: err_flag = .false.

! compute random numbers for arrays
do j = 1, N
  do i = 1, M
```

Example: Where loop with user-defined function with vector declaration is auto-vectorized

```
   a1(i,j) = rand() * M
   a2(i,j) = rand() * M
  end do
end do

 ! compute serial results
do j = 1, N
!dir$ novector
  do i = 1, M
   s_a3(i,j) = foo(a1(i,j), a2(i,j))
  end do
end do

 ! compute vector results
  do j = 1, N
   do i = 1, M
    v_a3(i,j) = foo(a1(i,j), a2(i,j))
   end do
  end do

 ! compare serial and vector results
do j = 1, N
  do i = 1, M
   if (s_a3(i,j) .ne. v_a3(i,j)) then
    err_flag = .true.
    print *, s_a3(i, j), v_a3(i,j)
   end if
  end do
 end do
if (err_flag .eq. .true.) then
  write(*,*) "FAILED"
   else
  write(*,*) "PASSED"
end if
end program

!! file: vecfoo.f90
integer function foo(a, b)
!$omp declare simd
implicit none
integer, intent(in) :: a, b
  foo = a - b
end function
```

```
[49 C:/temp] ifort -nologo -qopt-report2 -qopt-report-phase=vec -qopt-report-file=stderr
simdmain.f90 vecfoo.f90 -qopenmp

Begin optimization report for: SIMDTEST

    Report from: Vector optimizations [vec]


LOOP BEGIN at simdmain.f90(32,2)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive
```

Example: Where loop with user-defined function with vector declaration is auto-vectorized

```
   LOOP BEGIN at simdmain.f90(34,3)
      remark #15319: loop was not vectorized: novector directive used
   LOOP END
LOOP END

LOOP BEGIN at simdmain.f90(40,3)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at simdmain.f90(41,4)
      remark #15300: LOOP WAS VECTORIZED
   LOOP END
LOOP END

LOOP BEGIN at simdmain.f90(55,2)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at simdmain.f90(48,3)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization. First
dependence is shown below
Use level 5 report for details
      remark #15346: vector dependence: assumed OUTPUT dependence between at (51:5) and at
(51:5)
   LOOP END
LOOP END


Non-optimizable loops:


LOOP BEGIN at simdmain.f90(29,2)
   remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.

   LOOP BEGIN at simdmain.f90(28,3)
      remark #15543: loop was not vectorized: loop with function call not considered an
optimization candidate.
   LOOP END
LOOP END
===========================================================================

Begin optimization report for: FOO..xN4vv

    Report from: Vector optimizations [vec]

remark #15347: FUNCTION WAS VECTORIZED with xmm, simdlen=4, unmasked, formal parameter types:
(vector,vector)
===========================================================================

Begin optimization report for: FOO..xM4vv

    Report from: Vector optimizations [vec]

remark #15347: FUNCTION WAS VECTORIZED with xmm, simdlen=4, masked, formal parameter types:
```

Example: Where loop with user-defined function with vector declaration is auto-vectorized

```
(vector,vector)
=========================================================================.
```

Restrictions on Using `!$omp declare simd` declaration

Vectorization depends on two major factors: hardware and the style of source code. When using the vector declaration, the following features are not allowed:

- Locks, barriers, atomic construct, critical sections (These are allowed inside `!$OMP ORDERED SIMD` blocks).
- Computed and assigned `GOTO` and `SELECT CASE` constructs (in some cases these may be supported and converted to `IF` statements).
- The `GOTO` statement, into or out of a function.
- An `ENTRY` statement.

Non-vector function calls are generally allowed within vector functions but calls to such functions are serialized lane-by-lane and so might perform poorly. Also for SIMD-enabled functions it is not allowed to have side effects except writes by their arguments. This rule can be violated by non-vector function calls, so be careful executing such calls in SIMD-enabled functions and subroutines.

Formal parameters must be of the following data types:

- (un)signed 8, 16, 32, or 64-bit integer
- 32- or 64-bit floating point
- 64- or 128-bit complex

See Also

SIMD Directive (OpenMP* API)
Function Annotations and the SIMD Directive for Vectorization

Function Annotations and the SIMD Directive for Vectorization

This topic presents specific Fortran language features that better help to vectorize code.

NOTE
The SIMD vectorization feature is available for both Intel® microprocessors and non-Intel microprocessors. Vectorization may call library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The vectorization can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x`.

The `!DIR$ ATTRIBUTES ALIGN:`*N* directive enables you to overcome hardware alignment constraints. The `restrict` qualifier and the auto-vectorization hints address the stylistic issues due to lexical scope, data dependency, and ambiguity resolution. The SIMD feature's directive allows you to enforce vectorization of loops.

You can use the `!DIR$ ATTRIBUTES VECTOR[:`*clauses*`]::`*function-name-list*directive to vectorize user-defined functions and loops. For SIMD usage, a function with the `VECTOR` attribute is called from a loop that is being vectorized.

You can use the and declarations to provide a user-defined vector implementation for a function.

The usage model of the `VECTOR` attribute is that the code generated for the function actually takes a small section (*VECTORLENGTH*) of the array, by value, and exploits SIMD parallelism, whereas the implementation of task parallelism is done at the call site.

The following table summarizes the language features that help vectorize code.

Language Feature	Description
`!DIR$ ATTRIBUTES ALIGN : n :: var`	Directs the compiler to align the variable to an *n*-byte boundary. Address of the variable is `address mod n=0`.
`!DIR$ ATTRIBUTES VECTOR [: clauses] :: function-name-list`	Provides data parallel semantics by combining with the vectorized operations or loops at the call site. When multiple instances of the vector declaration are invoked in a parallel context, the execution order among them is not sequenced. The clauses are: • PROCESSOR • VECTORLENGTH • LINEAR • UNIFORM • [NO]MASK

Auto-Vectorization Hints	
`!DIR$ IVDEP`	Instructs the compiler to ignore assumed vector dependencies.
`!DIR$ VECTOR [ALWAYS]`	Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored. Using the `ASSERT` keyword with the `VECTOR [ALWAYS]` directive generates an error-level assertion message if the compiler efficiency heuristics indicate that the loop cannot be vectorized. Use `DIR$ IVDEP` to ignore the assumed dependencies.
`!DIR$ NOVECTOR`	Specifies that the loop should never be vectorized.

NOTE
Some directives are available for both Intel® microprocessors and non-Intel microprocessors, but may perform additional optimizations for Intel® microprocessors than for non-Intel microprocessors.

User-Mandated Directive	
`!DIR$ SIMD`	Enforces vectorization of loops.
SIMD Directive (OpenMP*)	Requires and controls SIMD vectorization of loops.

See Also

IVDEP directive
VECTOR ALWAYS and NOVECTOR directive
ATTRIBUTE directive
User-mandated or SIMD Vectorization

Guided Auto Parallelism

NOTE This feature has been deprecated.

The Guided Auto Parallelism (GAP) feature of the Intel®Fortran Compiler is a tool that offers selective advice to improve the performance of serially-coded applications by suggesting changes that take advantage of the compiler's ability to automatically vectorize and parallelize code and improve the efficiency of data operations. Despite having the words "auto parallelism" in the name, this tool does not require a threaded code implementation to improve the execution performance of the code, or require that the code is already threaded or parallel.

Advanced optimization techniques, such as inter-procedural analysis or profile-guided feedback, are not needed to use this feature. Using the `[Q]guide` set of options in addition to the compiler options normally used is sufficient to enable the GAP feature, with the requirement that you must compile with `O2` or higher optimization levels. The compiler does not generate any object files or executables during the GAP run.

In debug mode (`/Zi` on Windows*, `-g` on Linux*), the compiler's optimization level defaults to `/Od` (on Windows*) or `-O0` (on Linux* and macOS*); thus `O2` (or a higher level optimization) must be specified explicitly on the command-line.

NOTE Use the `[Q]diag-disable` option along with the `[Q]guide` option to direct the compiler to suppress specific diagnostic messages.
For example, the options: `//` **`(Windows*)`**`/Qguide, /Qdiag-disable:30534` and `//` **`(Linux* and macOS*)`** `-guide, -diag-disable:30534` tell the compiler not to emit the 30534 diagnostic. The `[Q]diag-disable` mechanism works the same way as it does for compiler-warnings.

If you decide to follow the advice offered by the GAP tool by making the suggested code changes and/or using the suggested compiler options, you must then recompile the program without the `[Q]guide` option.

Any advice generated by the compiler when using the GAP tool is optional; it can be implemented or rejected. The advice typically falls under three broad categories:

- **Advice for source modifications:** The compiler advises you to make source changes that are localized to a loop-nest or a routine. For example, the tool may recommend that you use a local-variable for the upper-bound of a loop, or that you should initialize a local variable unconditionally at the top of the loop-body, .
- **Advice to apply directives:** The compiler advises you to apply a new directive on a certain loop-level if the directive semantics can be satisfied (you must verify this). In many cases, you may be able to apply the directive (thus implicitly asserting new program/loop properties) that the compiler can take advantage of to perform enhanced optimizations.
- **Advice to add compiler options:** The compiler advises you to add command-line options that assert new properties; .

NOTE These suggested compiler options apply to the entire file. It is your responsibility to check that the properties asserted by these options are valid for the entire file, and not just the loop in question.

If you use GAP options along with option `[Q]parallel`, the compiler may suggest options to further parallelize your application. The compiler may also offer advice on enabling other optimizations of your application, including vectorization.

If you use the GAP options without enabling auto parallelism (without using the `[Q]parallel` option), the compiler may only suggest enabling optimizations such as vectorization for your application. This approach is recommended when you wish to improve the performance of a single-threaded code without the use of parallelization or when you want to improve the performance of threaded applications that do not rely on the compiler for auto parallelism.

See Also

Using Guided Auto Parallelism

`diag`
compiler option

`g`
compiler option

Using Guided Auto Parallelism

The Guided Auto Parallelism feature of the Intel® Fortran Compiler is a tool offering selective advice to improve the performance of serially-coded applications. The tool suggests changes that take advantage of the compiler's ability to automatically vectorize and parallelize code as well as improve the efficiency of data operations. The tool does not require that you implement threaded code to improve the execution performance of your code, nor does it require that your code is already threaded or parallel code.

To invoke this tool, use the compiler option `[Q]guide[=n]`. Using this option causes the compiler to generate messages suggesting ways to optimize the performance of your application. You can also use more specific compiler options such as `[Q]guide-vec`, `[Q]guide-par`, and `[Q]guide-data-trans` , to perform individual guided optimizations for vectorizing, parallelizing, and data transformation of your application.

When any guided auto parallelism option is used, the compiler provides only diagnostic advice. Object files or executables are not created in this mode. See the table below for descriptions of the options.

Syntax	Description
`[Q]guide`	Allows you to set a level of guidance for auto-vectorization and data transformation analysis. To obtain guidance for auto parallelism, you must use the `[Q]parallel` option along with the `[Q]guide` option.
`[Q]guide-par`	Allows you to set a level of guidance only for auto parallelism analysis. **NOTE** You must use the `[Q]parallel` option along with the `[Q]guide-par` option to get this advice.
`[Q]guide-vec`	Allows you to set a level of guidance for auto-vectorization analysis only.
`[Q]guide-data-trans`	Allows you to set a level of guidance for data transformation analysis only.

For all of the above options, the optional argument *n* specifies the level of guidance. The argument *n* takes the values `1-4`. When *n* is not specified, the default is `4`. If you specify *n*=`1` or `2`, a standard level of guidance is provided.

When you use *n*=`3` or *n*=`4`, you may get advanced messages. For example, you may get messages about how to optimize a particular loop-nest or get a message on how exception-handling inside a loop-nest affects optimizations for that loop-nest. Or you may get a message on how to provide extra information to the compiler on cost-modeling (expected values of trip-counts, and so on).

If you simultaneously specify a level of guidance for the general `[Q]guide` option and also for one or more of the other specific guide options, the level of guidance (*n*) for the specific guide option overrides the general `[Q]guide` option setting.

If you do not specify a level of guidance for the general `[Q]guide` option, but do set a level of guidance for one or more of the specific guide options, the `[Q]guide` option is set equal to the greatest value passed to the specific guide options.

Capturing Guidance Messages

The guided auto parallelism tool analyzes all of your serial code or individual parts of your code and generates advisory messages. By default, messages that are generated by the guided auto parallelism tool are output to `stderr`.

To capture messages in a file, use the options listed in the following table.

NOTE
The options listed in the following table must be used with the `[Q]guide`, `[Q]guide-par`, `[Q]guide-vec`, or `[Q]guide-data-trans` options. If not, they are ignored.

Syntax	Description
`[Q]guide-file`	Gathers all messages generated during a guided auto-parallelization run into the specified file.
`[Q]guide-file-append`	Allows you to specify the file into which all messages generated during a guided auto parallelism run should be appended.

For the above options, the *file_name* argument can also include a path. If a path is not specified, the file is created in the current working directory. If there is already a file named *file_name*, it is overwritten when you use the `[Q]guide-file` option. If you do not include an extension as part of the *file_name*, the extension `.guide` is appended.

Configuring Code Regions for GAP Messages

To limit guided auto parallelism analysis to specific regions (hotspots) in your application, use the option mentioned in the table below.

Syntax	Description
`[Q]guide-opts`	Allows you to analyze specified code elements, identified by *string*.

You must use the `[Q]guide-opts` option along with one of the guided auto parallelism options, such as `[Q]guide`, `[Q]guide-vec`, `[Q]guide-par`, and `[Q]guide-data-trans` . Use the *string* parameter to provide information about known areas of interest (hotspots). The *string* parameter takes one or more of the following variables: *filename*, *routine*, *range*. The compiler parses the *string* parameter and generates syntax errors if there are any.

Windows* Syntax

`/Qguide-opts:`*`string`*

Linux* and macOS* Syntax

`-guide-opts=`*`string`*

Follow these guidelines when using the *string* parameter:

- Use only valid file names, routine names, and line numbers. The guided auto parallelism tool ignores invalid values and issues a diagnostic message stating what was ignored.
- Enclose routine names within single quotation marks. Specify original source names (demangled names) as routine names. A routine name alone may not always be sufficient to uniquely identify a routine. You may need to specify additional parameter information to uniquely identify the routine.

  For any specified routine name, the GAP tool first tries to uniquely identify the routine using specified routine information. If that is not possible, then it selects all routines with the specified routine name. The GAP tool uses the parameter information, if specified, to narrow the selection.
- When inlining is involved, use the callee line numbers. The generated messages also use the callee line numbers.

See Also

`guide, Qguide` compiler option
`guide-par, Qguide-par` compiler option
`guide-vec, Qguide-vec` compiler option

`guide-data-trans, Qguide-data-trans` compiler option
`guide-file, Qguide-file` compiler option
`guide-file-append, Qguide-file-append` compiler option
`guide-opts, Qguide-opts` compiler option

See Also

Guided Auto Parallelism Messages

The Guided Auto Parallelism (GAP) messages provide advice that should improve optimizations.

The messages provide suggestions for:

- Automatic parallelization of loop nests
- Automatic vectorization of inner loops
- Data transformation

You must decide whether to follow a particular suggestion. For example, if the advice is to apply a particular directive, you must understand the semantics of the directive and carefully consider whether it can be safely applied to the loop (or loop nest) in question.

If you apply the directive improperly, the compiler may generate incorrect code, causing the application to execute incorrectly.

If you do not fully understand the suggested advice, please refer to the relevant topics in the compiler documentation before applying that advice.

Once you apply the suggested advice, the compiler assumes that it is correct and it does not perform any checks or issue any warnings.

In general, messages that relate to loops tend to target vectorization and/or parallelization of loops. If you are not familiar with loop optimizations, please refer to the compiler documentation on this kind of optimization.

Optimizations can be helped or hindered when data is fetched from or stored to memory. Sometimes these fetch and store operations are referred to as reads and writes, but do not confuse these with `READ` and `WRITE` input/output statements. A `READ` statement (input) is only one of many ways to cause a store to memory, and a `WRITE` statement (output) is only one of many ways to cause a fetch from memory. Values in an expression must be fetched from (read from) memory before the expression can be evaluated. Variables on the left-hand side of an assignment will be the target of a store to (write to) memory. Actual arguments passed to called procedures can be fetched from and stored to memory.

See Also

Using Guided Auto Parallelism

Guided Auto Parallelism
Enabling Auto-parallelization
Enabling Further Loop Parallelization for Multicore Platforms
Loop Constructs

GAP Message (Diagnostic ID 30506)

Message

If the following operations(s) can be safely performed unconditionally, the loop at line %d will be vectorized by adding a "%s ivdep" statement right before the loop: %s.

Advice

Add "!DIR$ IVDEP" before the specified loop.

This directive enables the vectorization of the loop at the specified line. Insure that any conditional divide, sqrt, and inverse sqrt operations will not alter the exception semantics expected by the program when they are performed unconditionally.

Example

Consider the following:

```
subroutine foo(a, n)
  integer n
  real a(n)
  do i=1,n
    if (a(i) .gt. 0) then
      a(i) = 1 / a(i)
    endif
  enddo
end
```

In this case, the compiler is unable to vectorize the loop since the condition "a(i) .gt. 0" may be guarding floating-point exceptions for the divide.

If you determine it is safe to do so, you can add the directive as follows:

```
subroutine foo(a, n)
  integer n
  real a(n)
!dir$ ivdep
  do i=1,n
    if (a(i) .gt. 0) then
      a(i) = 1 / a(i)
    endif
  enddo
end
```

Verify

Confirm that the program operands have safe values for all iterations.

GAP Message (Diagnostic ID 30513)

Message

Insert a "%s ivdep" statement right before the loop at line %d to vectorize the loop.

Advice

Add "!DIR$ IVDEP" before the specified loop. This directive enables the vectorization of the loop at the specified line by ignoring some of the assumed cross-iteration data dependencies.

Example

Consider the following:

```
subroutine foo(a, m, n)
  integer m, n
        real a(n)
  do i=1,n
    a(i) = a(i+m) + 1
  enddo
end
```

In this case, the compiler is unable to vectorize the loop because m could be -1, where each iteration is dependent on the previous iteration. If m is known to be positive, you can vectorize this loop.

If you determine it is safe to do so, you can add the directive as follows:

```
subroutine foo(a, m, n)
  integer m, n
        real a(n)
!dir$ ivdep
  do i=1,n
    a(i) = a(i+m) + 1
  enddo
end
```

Verify

Confirm that any arrays in the loop do not have unsafe cross-iteration dependencies. A cross-iteration dependency exists if a memory location is modified in an iteration of the loop and accessed by a fetch or store operation in another iteration of the loop. Make sure that there are no such dependencies, or that any cross-iteration dependencies can be safely ignored.

GAP Message (Diagnostic ID 30515)

Message

Assign a value to the variable(s) "%s" at the beginning of the body of the loop in line %d. This will allow the loop to be vectorized.

Advice

You should unconditionally initialize the scalar variables at the beginning of the specified loop. This allows the vectorizer to privatize those variables for each iteration and vectorize the loop. You must ensure that all the uses of those variables see the same values before and after the source code change.

Example

Consider the following:

```
subroutine foo(a, n)
  integer n
  do i=1,n
    if (a(i) .gt. 0) then
      b = a(i)
      a(i) = 1 / a(i)
    endif
    if (a(i) .gt. 1) then
      a(i) = a(i) + b
    endif
  enddo
end
```

In this case, the compiler is unable to vectorize the loop because it failed to privatize the variable *b*. Vectorization is assisted when assignment to *b* occurs in each iteration where the value of *b* is used. One of the ways to do this is to assign the value in every iteration.

If you determine it is safe to do so, you can modify the program code as follows:

```
subroutine foo(a, n)
  integer n
  real a(n), b
```

```
  do i=1,n
    b = a(i)
    if (a(i) .gt. 0) then
      a(i) = 1 / a(i)
    endif
    if (a(i) .gt. 1) then
      a(i) = a(i) + b
    endif
  enddo
end
```

Verify

Confirm that in the original program, any variables fetched in any iteration of the loop have been defined earlier in the same iteration.

GAP Message (Diagnostic ID 30519)

Message

Insert a "%s parallel" statement right before the loop at line %d to parallelize the loop.

Advice

Add "!DIR$ PARALLEL" before the specified loop. This directive enables the parallelization of the loop at the specified line by ignoring assumed cross-iteration data dependencies.

Example

Consider the following:

```
subroutine foo(a, m, n)
  real a(n)
  do i=1,n
    a(i) = a(i+m) + 1
  enddo
end
```

In this case, the compiler is unable to parallelize the loop without further information about *m*. For example, if *m* is negative, then each iteration will be dependent on the previous iteration. However, if *m* is known to be greater than *n*, you can parallelize the loop.

If you determine it is safe to do so, you can add the directive as follows:

```
subroutine foo(a, m, n)
  real a(n)
!dir$ parallel
  do i=1,n
    a(i) = a(i+m) + 1
  enddo
end
```

Verify

Confirm that any arrays in the loop do not have cross-iteration dependencies. A cross-iteration dependency exists if a memory location is modified in an iteration of the loop and accessed by a fetch or store operation in another iteration of the loop.

GAP Message (Diagnostic ID 30521)

Message

Assign a value to the variable(s) "%s" at the beginning of the body of the loop in line %d. This will allow the loop to be parallelized.

Advice

Check to see if you can unconditionally initialize the scalar variables at the beginning of the specified loop. If so, do the code change for such initialization (standard), or list the variables in the private clause of a parallel directive (advanced). This allows the parallelizer to privatize those variables for each iteration and to parallelize the loop.

Example

Consider the following:

```
    subroutine foo (A, B, cond1, cond2)
      integer, parameter :: N = 100000
      integer I
      real(8) A(N), B(N), T
      logical cond1, cond2

      do I = 1, N
         if (cond1) T = i + 1
         if (cond2) T = i - 1
         A(I) = T
      end do
    end
```

In this case, the compiler does not parallelize the loop because it cannot privatize the variable *t* without further information. If you know that cond1 or cond2 is true, or both cond1 and cond2 are true, then you can assist the parallelizer by ensuring that any iteration that uses *t* also writes to *t* before its use in the same iteration. One of the ways to do this is to assign a value to *t* at the top of each iteration.

If you determine it is safe to do so, you can modify the program code as follows:

```
    subroutine foo (A, B, cond1, cond2)
      integer, parameter :: N = 100000
      integer I
      real(8) A(N), B(N), T
      logical cond1, cond2

      do I = 1, N
         T = 0
         if (cond1) T = i + 1
         if (cond2) T = i - 1
         A(I) = T
      end do
    end
```

Verify

Confirm that in the original program, any variables fetched in any iteration of the loop have been defined earlier in the same iteration.

See Also

GAP Message (Diagnostic ID 30523)

GAP Message (Diagnostic ID 30522)

Message

Insert a "%s parallel private(%s)" statement right before the loop at line %d to parallelize the loop.

Advice

Add "!DIR$ PARALLEL PRIVATE" before the specified loop. This directive enables the parallelization of the loop at the specified line.

Example

Consider the following:

```
subroutine foo (A, B, C, n, m1, m2)
 real (4) A(10000, 10), B(10000, 10), C(10000, 10)
 integer n, m1, m2
 real W(1000)
 integer i, j

 do i=1,n
   do j=1,m1
     W(j) = A(j,i) * B(j,i)
   end do
   do j=1,m2
     C(j,i) = W(j) + 1.0
   end do
end
```

In this case, the compiler does not parallelize the loop since it cannot determine whether m1 >= m2.

If you know that this property is true, and that no element of W is fetched before it is written to after the loop, then you can use the recommended directive.

If you determine it is safe to do so, you can add the directive as follows:

```
subroutine foo (A, B, C, n, m1, m2)
 real (4) A(10000, 10), B(10000, 10), C(10000, 10)
 integer n, m1, m2
 real W(1000)
 integer i, j

!DIR$ PARALLEL PRIVATE (W)
 do i=1,n
   do j=1,m1
     W(j) = A(j,i) * B(j,i)
   end do
   do j=1,m2
     C(j,i) = W(j) + 1.0
   end do
end
```

Verify

Before an element of an array can be fetched in the loop, there must have been a previous store to it during the same loop iteration. In addition, if an element is fetched after the loop, there must have been a previous write to it before the fetch after the loop.

GAP Message (Diagnostic ID 30523)

Message

Assign a value to the variable(s) "%s" at the beginning of the body of the loop in line %d. This will allow the loop to be parallelized.

Advice

Check to see if you can unconditionally initialize the scalar variables at the beginning of the specified loop. If so, do the code change for such initialization (standard), or list the variables in the private clause of a PARALLEL directive (advanced). This allows the parallelizer to privatize those variables for each iteration and to parallelize the loop.

Verify

Confirm that in the original program, any variables fetched in any iteration of the loop have been defined earlier in the same iteration or have been privatized by means of the private clause of a PARALLEL directive.

See Also

GAP Message (Diagnostic ID 30521)

GAP Message (Diagnostic ID 30525)

Message

Insert a "%s loop count min(%d)" statement right before the loop at line %d to parallelize the loop.

Advice

Add "!DIR$ LOOP COUNT" before the specified loop. This directive indicates the minimum trip count (number of iterations) of the loop that enables the parallelization of the loop.

Example

Consider the following:

```
      subroutine foo (n)
      integer, parameter :: N2 = 10000
      real (8) :: A(N2), B(N2)
      integer :: i, n
      do i =1, n
         A(i) = B(i) * B(i)
      end do
      end subroutine foo
```

In this case, if the trip count of the loop at line 5 is greater than 128, then use the LOOP COUNT directive to parallelize this loop.

If you determine it is safe to do so, you can add the directive as follows:

```
      subroutine foo (n)
      integer, parameter :: N2 = 10000
      real (8) :: A(N2), B(N2)
      integer :: i, n
      !dir$ loop count min(128)
      do i =1, n
```

```
      A(i) = B(i) * B(i)
   end do
   end subroutine foo
```

Make sure that the loop has a minimum of 128 iterations.

Verify

Confirm that the loop has the minimum number of iterations, as specified in the diagnostic message.

See Also

Guided Auto Parallelism Messages provides advice for improving optimizations

GAP Message (Diagnostic ID 30526)

Message

To parallelize the loop at line %d, annotate the routine %s with %s.

Advice

If the loop contains a call to a function, the compiler cannot parallelize the loop without more information about the function being called.

However, if the function being called in the loop is an elemental function or a function specified with `ATTRIBUTES CONCURRENCY_SAFE`, then the call does not inhibit parallelization of the loop.

Example

Consider the following:

```
   subroutine foo ()
   integer, parameter :: N2 = 10000
   real (8) :: A(N2), B(N2)
   interface
      function bar (k)
      integer :: bar, k
      end function bar
   end interface

   integer :: i
   do i =1, N2
      A(i) = B(i) * bar(N2)
   end do
   end subroutine foo
```

In this case, to parallelize the loop at line 11, declare the routine `bar` as elemental.

If you determine it is safe to do so, an alternative way you can modify the program code is as follows:

```
   subroutine foo ()
   integer, parameter :: N2 = 10000
   real (8) :: A(N2), B(N2)
   interface
      function bar (k)
   !dir$ attributes concurrency_safe : profitable :: bar
      integer :: bar, k
      end function bar
   end interface
```

```
integer :: i
do i =1, N2
   A(i) = B(i) * bar(N2)
end do
end subroutine foo
```

Verify

Confirm the routine satisfies the semantics of this annotation. A weaker annotation able to achieve a similar effect is `ATTRIBUTES CONCURRENCY_SAFE(PROFITABLE)`.

See Also

GAP Message (Diagnostic ID 30528)

GAP Message (Diagnostic ID 30528)

Message

Add "%s" to the declaration of routine "%s" in order to parallelize the loop at line %d. Adding "%s" achieves a similar effect.

Advice

Confirm that the routine specified is indeed an elemental function or an ATTRIBUTES CONCURRENCY_SAFE function before following the advice to add the annotation.

If the routine is not one of these kinds of functions, try to inline it with a FORCEINLINE RECURSIVE directive. This action may or may not be beneficial.

Example

Consider the following:

```
subroutine foo ()
integer, parameter :: N2 = 10000
real (8) :: A(N2), B(N2)
interface
   function bar (k)
   integer :: bar, k
   end function bar
end interface

integer :: i
do i =1, N2
   A(i) = B(i) * bar(N2)
end do
end subroutine foo
```

In this case, to parallelize the loop at line 11, declare the routine `bar` as elemental.

If you determine it is safe to do so, an alternative way you can modify the program code is as follows:

```
subroutine foo ()
integer, parameter :: N2 = 10000
real (8) :: A(N2), B(N2)
interface
   function bar (k)
!dir$ attributes concurrency_safe : profitable
   integer :: bar, k
   end function bar
```

```
    end interface

    integer :: i
    do i =1, N2
       A(i) = B(i) * bar(N2)
    end do
    end subroutine foo
```

Verify

Confirm that the routine satisfies the semantics of this declaration. Another way to help the loop being parallelized is to inline the routine with the FORCEINLINE RECURSIVE directive, but this method does not guarantee parallelization.

See Also

GAP Message (Diagnostic ID 30526)

GAP Message (Diagnostic ID 30531)

Message

Store the value of the upper-bound expression of the loop at line %d into a temporary local variable, and use this variable as the new upper-bound expression of the loop. To do this, insert a statement of the form "temp = upper-bound of loop" right before the loop, where "temp" is the newly created local variable. Choose a variable name that is unique, then replace the loop's original upper-bound expression with "temp".

Advice

Use a local-variable for the loop upper-bound if the upper-bound does not change during the execution of the loop. This enables the compiler to recognize the loop as a proper counted do loop, which enables various loop optimizations including vectorization and parallelization.

This message appears when the compiler cannot output the exact upper-bound variable to be replaced.

Verify

Confirm that the value of the upper-bound expression does not change throughout the entire execution of the loop.

See Also

GAP Message (Diagnostic ID 30532)

GAP Message (Diagnostic ID 30532)

Message

Store the value of the upper-bound expression (%s) of the loop at line %d into a temporary local variable, and use this variable as the new upper-bound expression of the loop. To do this, insert a statement of the form "temp = %s" right before the loop, where "temp" is the newly created local variable. Choose a variable name that is unique, then replace the loop's original upper-bound expression with "temp".

Advice

Use a local-variable for the loop upper-bound if the upper-bound does not change during the execution of the loop. This enables the compiler to recognize the loop as a proper counted do loop, which in turn enables various loop optimizations including vectorization and parallelization.

This message appears when the compiler can output the exact upper-bound variable to be replaced.

Verify

Confirm that the value of the upper-bound expression does not change throughout the entire execution of the loop.

See Also

GAP Message (Diagnostic ID 30531)

GAP Message (Diagnostic ID 30533)

Message

Compile with the %s option to vectorize and/or parallelize the loop at line %d.

Advice

Use the `[q or Q]opt-subscript-in-range` option for the specified file during compilation.

This option helps the compiler vectorize and parallelize the loop at the specified line. You must verify that the loops in the file do not contain very large integers and are not likely to generate very large integers in intermediate computations. A very large integer is loosely defined as follows: On an *n*-bit machine, a very large integer is typically >= 2^{n-2}. For example, on a 32-bit machine, a very large integer would be >= 2^{30}.

Verify

Confirm that no loop in the program contains or generates very large integers (typically very large integers are >= 2^{30}).

GAP Message (Diagnostic ID 30538)

Message

Moving the block of code that consists of a function-call (line %d), if-condition (line %d), and an early return (line %d) to outside the loop may enable parallelization of the loop at line %d.

Advice

Move the function call and an associated return from inside the loop (perhaps by inserting them before the loop) to help parallelize the loop.

This kind of function-leading-to-return inside a loop usually handles some error-condition inside the loop. If this error check can be done before starting the execution of the loop without changing the program semantics, the compiler may be able to parallelize the loop thus improving performance.

Verify

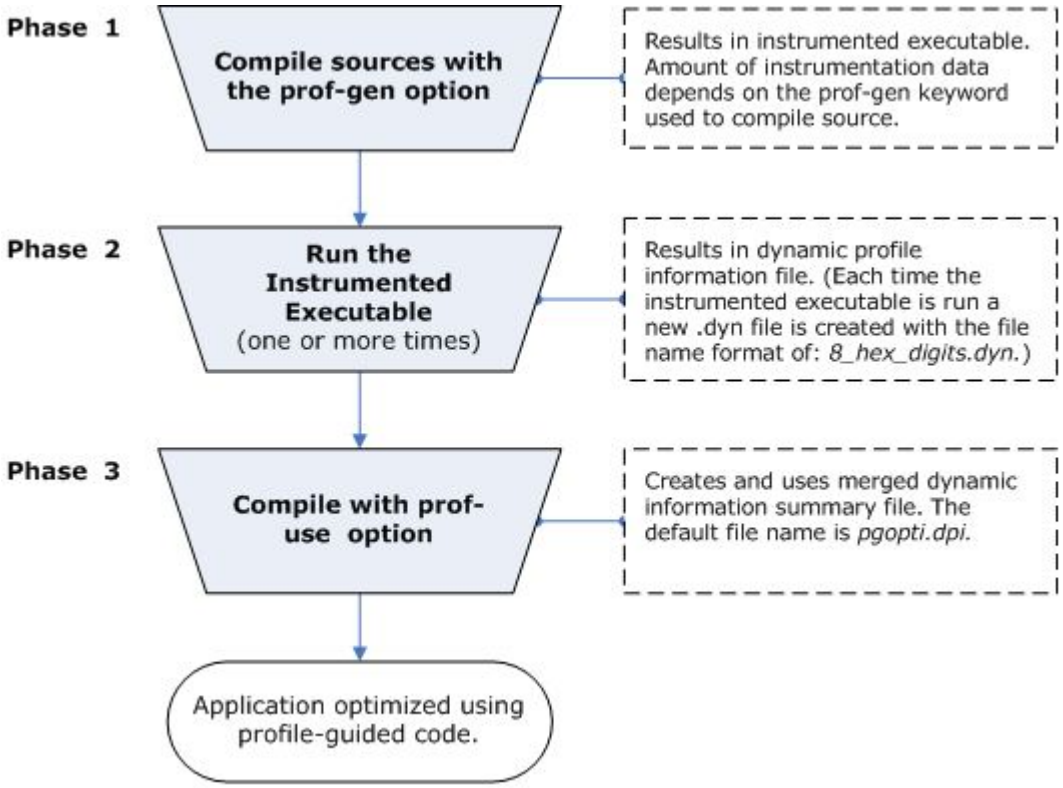
Confirm that the function call does not rely on any computation inside the loop and that restructuring the code as suggested above, retains the original program semantics.

Profile-Guided Optimization (PGO)

Profile-guided Optimization (PGO) improves application performance by shrinking code size, reducing branch mispredictions, and reorganizing code layout to reduce instruction-cache problems. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

PGO consists of three phases or steps.

1. Instrument the program. The compiler creates and links an instrumented program from your source code and special code from the compiler.
2. Run the instrumented executable. Each time you execute the instrumented code, the instrumented program generates a dynamic information file, which is used in the final compilation.
3. Final compilation. When you compile a second time, the dynamic information files are merged into a summary file. Using the summary of the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.



See Profile-guided Optimization Options for information about the supported options and Profile an Application for specific details about using PGO from the command line.

PGO provides the following benefits:

- Use profile information for register allocation to optimize the location of spill code.
- Improve branch prediction for indirect function calls by identifying the most likely targets. Some processors have longer pipelines, which improves branch prediction and translates into high performance gains.
- Detect and do not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Interprocedural optimization (IPO) and PGO can affect each other; using PGO can often enable the compiler to make better decisions about inline function expansion, which increases the effectiveness of interprocedural optimizations. Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This variability is due to the unique characteristics of each program, which often include different profiles and different opportunities for optimizations.

Performance Improvements with PGO

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is the code with intensive error-checking in which the error conditions are false most of the time. The infrequently executed (cold) error-handling code can be relocated so the branch is rarely predicted incorrectly. Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

When you use PGO, consider the following guidelines:

- Minimize changes to your program after you execute the instrumented code and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated. If you modify your program, the compiler can issue a warning that the dynamic information does not correspond to a modified function when PGO remarks are enabled or found in the optimization report.
- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.
- Know the sections of your code that are the most heavily used. If the data set provided to your program is very consistent and displays similar behavior on every execution, then PGO can probably help optimize your program execution.
- Different data sets can result in different algorithms being called. The difference can cause the behavior of your program to vary for each execution. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits. If it takes multiple data sets to accurately characterize application performance, execute the application with all data sets then merge the dynamic profiles; this technique should result in an optimized application.

You must insure that the benefit of the profiled information is worth the effort required to maintain up-to-date profiles.

Profile-Guided Optimization via HW counters

A lightweight profiling mechanism is available that can be used to achieve many of the benefits of instrumentation based profiling, but without the overhead of inserting instrumentation into the application binary. This mode of operation can be beneficial in cases where increase in code/data size or changes in run time due to instrumentation may make regular Performance-Guided Optimization (PGO) infeasible. This approach requires the use of Intel® VTune™ Amplifier to collect information from the hardware counters. The information is collected with minimal overhead, and combined with debug information produced by the compiler to identify the primary code path for optimizations.

Follow these steps to use this method:

Phase 1: Compile the application with the option `prof-gen-sampling`.

This option will instruct the compiler to generate additional debug information for the application, which is used to map the information collect by the hardware counters to specific source code. However, use of the option does not affect the generated instruction sequence in the way instrumented PGO would. Optimizations may be enabled during this build, however it is recommended to disable function inlining during this build.

Phase 2: Run the generated executable on one or more representative workloads with the Intel VTune Amplifier tool:

```
<installation-root>/bin64/amplxe-pgo-report.sh <your application and command line>
```

Additional information regarding options for data collection can be found in the Intel VTune Amplifier documentation. This step will generate files of the form `rNNNpgo_icc.pgo` (where *NNN* is a 3 digit number) which will be used as input to the following phases.

Phase 3: (optional) Merge the report files produced during phase 2.

The tool `profmergesampling` can be used to produce an indexed file of results that will speed up the processing of the data during the next phase.

```
profmergesampling -file <input-file[:input_file]*> -out <output_name>
```

Phase 4: Compile the application with the option `prof-use-sampling:input-file[:input_file]*`

In phase 4, one or more result files produced during phase 2 (or an indexed file from phase 3) can be fed into the compiler to direct the optimizations.

See Also

prof-gen-sampling
compiler option

prof-use-sampling
compiler option

Profile an Application with Instrumentation

Profiling an application includes the following three phases:

- Instrumentation compilation and linking
- Instrumented execution
- Feedback compilation

This topic provides detailed information on how to profile an application by providing sample commands for each of the three phases (or steps).

1. Instrumentation compilation and linking

Use `[Q]prof-gen` to produce an executable with instrumented information included. Use `/Qcov-gen` (Windows) option to obtain minimum instrumentation only for code coverage.

Operating System	Commands
Linux and macOS*	`ifort -prof-gen -prof-dir /usr/profiled a1.f90 a2.f90 a3.f90` `ifort -oa1 a1.o a2.o a3.o`
Windows	`ifort /Qprof-gen /Qprof-dir:c:\profiled a1.f90 a2.f90 a3.f90` `ifort a1.obj a2.obj a3.obj`
Windows	`ifort /Qcov-gen /Qcov-dir:c:\cov_data a1.f90 a2.f90 a3.f90` `ifort a1.obj a2.obj a3.obj`

Use the `[Q]prof-dir` or `/Qcov-dir` (Windows) option if the application includes the source files located in multiple directories; using the option insures the profile information is generated in one consistent place. The example commands demonstrate how to combine these options on multiple sources.

The compiler gathers extra information when you use the `-prof-gen=srcpos` (Linux and macOS*) or `/Qprof-gen:srcpos` (Windows) option; however, the extra information is collected to provide support for specific Intel tools, including the code coverage Tool. If you do not expect to use such tools, do not specify `-prof-gen=srcpos` (Linux and macOS*) or `/Qprof-gen:srcpos` (Windows); the

extended option does not provide better optimization and could slow parallel compile times. If you are interested in using the instrumentation only for code coverage, use the `/Qcov-gen` (Windows) option, instead of the `/Qprof-gen:srcpos` (Windows) option, to minimize instrumentation overhead.

PGO data collection is optimized for collecting data on serial applications at the expense of some loss of precision on areas of high parallelism. However, you can specify the `threadsafe` keyword with the `-prof-gen` (Linux* and macOS*) or the `/Qprof-gen` (Windows) compiler option for files or applications that contain parallel constructs using OpenMP* features, for example. Using the `threadsafe` keyword produces instrumented object files that support the collection of PGO data on applications that use a high level of parallelism but may increase the overhead for data collection.

NOTE
Unlike serial programs, parallel programs using OpenMP* may involve dynamic scheduling of code paths, and counts collected may not be perfectly reproducible for the same training data set.

2. Instrumented execution

Run your instrumented program with a representative set of data to create one or more dynamic information files.

Operating System	Command
Linux and macOS*	`./a1.out`
Windows	`a1.exe`

Executing the instrumented applications generates a dynamic information file that has a unique name and .dyn suffix. A new dynamic information file is created every time you execute the instrumented program.

You can run the program more than once with different input data.

By default, the `.dyn` filename follows this naming convention: `<timestamp>_<pid>.dyn`. The `.dyn` file is either placed into a directory specified by an environment variable, a compile-time specified directory, or the current directory.

To make it easy to distinguish files from different runs, you can specify a prefix for the `.dyn` filename in the environment variable, `INTEL_PROF_DYN_PREFIX`. In such a case, executing the instrumented application generates a `.dyn` filename as follows: `<prefix>_<timestamp>_<pid>.dyn`, where `<prefix>` is the identifier that you have specified. Be sure to set the `INTEL_PROF_DYN_PREFIX` environment variable prior to starting your instrumented application.

NOTE
The value specified in `INTEL_PROF_DYN_PREFIX` environment variable must not contain `< > : " / \ | ? *` characters. The default naming scheme will be used if an invalid prefix is specified.

3. Feedback compilation

Before this step, copy all .dyn and .dpi files into the same directory. Compile and link the source files with `[Q]prof-use`; the option instructs the compiler to use the generated dynamic information to guide the optimization:

Operating System	Examples
Linux and macOS*	`ifort -prof-use -ipo -prof-dir /usr/profiled a1.f90 a2.f90 a3.f90`
Windows	`ifort /Qprof-use /Qipo /Qprof-dir:c:\profiled a1.f90 a2.f90 a3.f90`

This final phase compiles and links the sources files using the data from the dynamic information files generated during instrumented execution (phase 2).

In addition to the optimized executable, the compiler produces a pgopti.dpi file.

Most of the time, you should specify the default optimizations,`O2`, for phase 1, and specify more advanced optimizations, `[Q]ipo`, during the phase 3 (final) compilation. The example shown above used`O2` in step 1 and `[Q]ipo` in step 3.

NOTE
The compiler ignores the `[Q]ipo` or `[Q]ip` option during phase 1 with `[Q]prof-gen`.

Profile-Guided Optimization Report

The PGO report can help identify where and how the compiler used profile information to optimize the source code. The PGO report can also identify where profile information was discarded due to source code changes made between the time of instrumentation and feedback steps. The PGO report is most useful when combined with the PGO compilation steps outlined in the topic, Profile an Application with Instrumentation. Without the profiling data generated during the application profiling process the report will generally not provide useful information.

Combine the final PGO step with the reporting options by including `-prof-use` (Linux* and macOS*) or `/Qprof-use` (Windows*). The following syntax examples demonstrate how to run the report using the combined options.

Operating System	Syntax Examples
Linux*	`ifort -prof-use -qopt-report-phase=pgo pgotools_sample.c`
macOS*	`icpc -prof-use -qopt-report-phase=pgo pgotools_sample.c` `ifort -prof-use -qopt-report-phase=pgo pgotools_sample.c`
Windows*	`ifort pgotools_sample.c /Qprof-use /Qopt-report-phase=pgo`

By default the PGO report generates a medium level of detail (where the `[q or Q]opt-report` argument *n*=2). You can use the `-qopt-report=`*n* (Linux and macOS*) or `/Qopt-report:`*n* option along with the `[q or Q]opt-report-phase` option if you want a greater or lesser level of diagnostic detail.

The output, by default, comes out to a file with the same name as the object file but with an `.optrpt` extension and is written into the same directory as the object file. Using the entries in the example above, the output file will be `pgotools_sample.optrpt`. Use the `-qopt-report-file` (Linux and macOS*) or the `/Qopt-report-file` (Windows) option to specify any other name for the output file that captures the report results, or to specify that the output should go to `stdout` or `stderr`.

See Also

`qopt-report-phase`, `Qopt-report-phase`
compiler option

`qopt-report`, `Qopt-report`
compiler option

`qopt-report-file`, `Qopt-report-file`
compiler option

`prof-use`, `Qprof-use`
compiler option

Profile an Application

High-Level Optimization (HLO)

High-level Optimizations (HLO) exploit the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. While the default optimization level, option `O2` , performs some high-level optimizations, specifying the `O3` option provides the best chance for performing loop transformations to optimize memory accesses.

NOTE
Loop optimizations may result in calls to library routines that can result in additional performance gain on Intel® microprocessors than on non-Intel microprocessors. The optimizations performed can also be affected by certain options, such as `/arch` (Windows*), `-m` (Linux* and macOS*), or `[Q]x` options. Additional HLO transformations may be performed for Intel® microprocessors than for non-Intel microprocessors.

Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Predicate Optimization
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Memset Combining, Memory Layout Change
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests
- Multiversioning: Checks include Dependency of Memory References, and Trip Counts
- Loop Collapsing

Interprocedural Optimization (IPO)

Interprocedural Optimization (IPO) is an automatic, multi-step process that allows the compiler to analyze your code to determine where you can benefit from specific optimizations.

The compiler may apply the following optimizations:

- Address-taken analysis
- Array dimension padding
- Alias analysis
- Automatic array transposition
- Automatic memory pool formation
- Common block variable coalescing
- Common block splitting
- Constant propagation

- Dead call deletion
- Dead formal argument elimination
- Dead function elimination
- Formal parameter alignment analysis
- Forward substitution
- Indirect call conversion
- Inlining
- Mod/ref analysis
- Partial dead call elimination
- Passing arguments in registers to optimize calls and register usage
- Points-to analysis
- Routine key-attribute propagation
- Specialization
- Stack frame alignment
- Structure splitting and field reordering
- Symbol table data promotion
- Un-referenced variable removal
- Whole program analysis

IPO Compilation Models

IPO supports two compilation models - single-file compilation and multi-file compilation.

Single-file compilation uses the `[Q]ip` compiler option, and results in one, real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

The compiler performs some single-file interprocedural optimization at the `O2` default optimization level; additionally the compiler may perform some inlining for the `O1` optimization level, such as inlining functions marked with inlining directives.

Multi-file compilation uses the `[Q]ipo` option, and results in one or more mock object files rather than normal object files. (See the *Compilation* section below for information about mock object files.) Additionally, the compiler collects information from the individual source files that make up the program. Using this information, the compiler performs optimizations across functions and procedures in different source files.

NOTE
Inlining and other optimizations are improved by profile information. For a description of how to use IPO with profile information for further optimization, see Profile an Application.

Compiling with IPO

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file. The mock object files contain the IR instead of the normal object code. Mock object files can be ten times or more larger than the size of normal object files.

During the IPO compilation phase only the mock object files are visible.

Linking with IPO

When you link with the `[Q]ipo` compiler option the compiler is invoked a final time. The compiler performs IPO across all mock object files. The mock objects must be linked with the Intel compiler or by using the Intel linking tools. While linking with IPO, the Intel compilers and other linking tools compile mock object files as well as invoke the real/true object files linkers provided on the user's platform.

Link-time optimization using the `-ffat-lto-objects` compiler option is provided for GCC compatibility. During IPO compilation, you can specify `-ffat-lto-objects` option, for the compiler to generate a fat link-time optimization (LTO) object that has both a real/true object and a discardable intermediate language section. This enables both link-time optimization (LTO) linking and normal linking.

You can specify the `-fno-fat-lto-objects` option for the compiler to generate a link-time optimization (LTO) object that only has a discardable intermediate language section; no real/true object is generated. These files are inserted into archives in the form in which they were created. Using this option may improve compilation time and save space for objects.

If you use `ld` rather than `xild` to link objects or `ar` instead of `xiar` to create an archive, the real/true object, generated during fat link-time optimization guarantees that there will be no impediment to linking/building the archive. However, cross-file optimizations are lost in this case. The extra true object also takes additional space and takes compile time to generate it, so using `-fno-fat-lto-objects` compiler option is an advantage provided that you link the IPO mock object files with `xild` and archive them with `xiar`.

Whole Program Analysis

The compiler supports a large number of IPO optimizations that can be applied or have its effectiveness greatly increased when the whole program condition is satisfied.

During the analysis process, the compiler reads all Intermediate Representation (IR) in the mock file, object files, and library files to determine if all references are resolved and whether or not a given symbol is defined in a mock object file. Symbols that are included in the IR in a mock object file for both data and functions are candidates for manipulation based on the results of whole program analysis.

There are two types of whole program analysis - object reader method and table method. Most optimizations can be applied if either type of whole program analysis determines that the whole program conditions exists; however, some optimizations require the results of the object reader method, and some optimizations require the results of table method.

Object reader method

In the object reader method, the object reader emulates the behavior of the native linker and attempts to resolve the symbols in the application. If all symbols are resolved, the whole program condition is satisfied. This type of whole program analysis is more likely to detect the whole program condition.

Table method

In the table method the compiler analyzes the mock object files and generates a call-graph.

The compiler contains detailed tables about all of the functions for all important language-specific libraries, like the Fortran runtime libraries. In this second method, the compiler constructs a call-graph for the application. The compiler then compares the function table and application call-graph. For each unresolved function in the call-graph, the compiler attempts to resolve the calls by finding an entry for each unresolved function in the compiler tables. If the compiler can resolve the functions call, the whole program condition exists.

See Also

ax, Qax

Inline Expansion of Functions

Interprocedural Optimization (IPO) Options

ip, Qip

ipo, Qipo

ipo-c, Qipo-c

Linking Tools and Options

O

x, Qx

Using IPO

Using IPO

This topic discusses how to use IPO from the command line.

Compiling and Linking Using IPO

To enable IPO, you first compile each source file, then link the resulting source files.

First, compile your source files with `[Q]ipo` compiler as shown below:

Operating System	Example Command
Linux* and macOS*	`ifort -ipo -c a.f90 b.f90 c.f90`
Windows*	`ifort /Qipo /c a.f90 b.f90 c.f90`

The output of the above example command differs according to operating system:

- Linux and macOS*: The commands produce `a.o`, `b.o`, and `c.o` object files.
- Windows: The commands produce `a.obj`, `b.obj`, and `c.obj` object files.

Use the `c` compiler option to stop compilation after generating `.o` or `.obj` files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files.

Second, link the resulting files. The following example command will produce an executable named `app`:

Operating System	Example Command
Linux and macOS*	`ifort -o app a.o b.o c.o`
Windows	`ifort /exe:app a.obj b.obj c.obj`

The command invokes the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xild` (Linux and macOS*) or `xilink` (Windows) tool, with the appropriate linking options.

Combining the Steps

The separate compile and link commands demonstrated above can be combined into a single command, as shown in the following examples:

Operating System	Example Command
Linux and macOS*	`ifort -ipo -o app a.f90 b.f90 c.f90`
Windows	`ifort /Qipo /exe:app a.f90 b.f90 c.f90`

The `ifort` command, shown in the examples above, calls `GCC ld` (Linux and macOS*) or `link.exe` (Windows only) to link the specified object files and produce the executable application, which is specified by the `-o` (Linux and macOS*) or `/exe` (Windows) option.

Capturing Intermediate IPO Output

The `[Q]ipo-c` and `[Q]ipo-S` compiler options are useful for analyzing the effects of multi-file IPO, or when experimenting with multi-file IPO between modules that do not make up a complete program.

- Use the `[Q]ipo-c` compiler option to optimize across files and produce an object file. The option performs optimizations as described for the `[Q]ipo` option but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.o` (Linux and macOS*) or `ipo_out.obj` (Windows).
- Use the `[Q]ipo-S` compiler option to optimize across files and produce an assembly file. The option performs optimizations as described for `[Q]ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s` (Linux) or `ipo_out.asm` (Windows).

For both options, you can use the `-o` (Linux and macOS*) or `/exe` (Windows) option to specify a different name.

These options generate multiple outputs if multi-object IPO is being used. The name of the first file is taken from the value of the `-o` (Linux and macOS*) or `/exe` (Windows) option.

The names of subsequent files are derived from the first file with an appended numeric value to the file name. For example, if the first object file is named `foo.o` (Linux and macOS*) or `foo.obj` (Windows), the second object file will be named `foo1.o` or `foo1.obj`.

You can use the object file generated with the `[Q]ipo-c` option, but you will not get the full benefit of whole program optimizations if you use this option.

The object file created using the `[Q]ipo-c` option is a real object file, in contrast to the mock file normally generated using IPO; however, the generated object file is significantly different than the mock object file. Whole program optimizations, which require a knowledge of how the real object file will be linked in with other files to produce and object, are not applied.

The compiler generates a message indicating the name of each object or assembly file it generates. These files can be added to the real link step to build the final application.

See Also

`c`
compiler option

`o`
compiler option

`ipo`, `Qipo`
compiler option

`ipo-c`, `Qipo-c`
compiler option

`ipo-S`, `Qipo-S`
compiler option

`O`
compiler option

IPO-Related Performance Issues

There are some general optimization guidelines for using IPO that you should keep in mind:

- Using IPO on very large programs might trigger internal limits of other compiler optimization phases.

- Applications where the compiler does not have sufficient intermediate representation (IR) coverage to do whole program analysis might not perform as well as those where IR information is complete.

In addition to these general guidelines, there are some practices to avoid while using IPO. The following list summarizes the activities to avoid:

- Do not use the link phase of an IPO compilation using mock object files produced for your application by a different compiler. The Intel® Compiler cannot inspect mock object files generated by other compilers for optimization opportunities.
- Update make files to call the appropriate Intel linkers when using IPO from scripts. For Linux and macOS*, replace all instances of `ld` with `xild`; for Windows, replace all instances of `link` with `xilink`.

See Also

IPO for Large Programs

O

prof-use, Qprof-use

IPO for Large Programs

In most cases, IPO generates a single true object file for the link-time compilation. This behavior is not optimal for very large programs, perhaps even making it impossible to use `[Q]ipo` compiler option on the application.

The compiler provides two methods to avoid this problem. The first method is an automatic size-based heuristic, which causes the compiler to generate multiple true object files for large link-time compilations. The second method is to manually instruct the compiler to perform multi-object IPO.

- Use the `[Q]ipo`*N* compiler option and pass an integer value in the place of *N*.
- Use the `[Q]ipo-separate` compiler option.

  The number of true object files generated by the link-time compilation is invisible to you unless the `[Q]ipo-c` or `[Q]ipo-S` compiler option is used.

Regardless of the method used, it is best to use the compiler defaults first and examine the results. If the defaults do not provide the desired results then experiment with generating a different number of object files.

You can use the `[Q]ipo-jobs` compiler option to control the number of processes, or jobs, executed during parallel IPO builds.

Using [Q]ipo*N* to Create Multiple Object Files

If you specify `[Q]ipo0`, which is the same as not specifying a value, the compiler uses heuristics to determine whether to create one or more object files based on the expected size of the application. The compiler generates one object file for small applications, and two or more object files for large applications. If you specify any value greater than 0, the compiler generates that number of object files, unless the value you pass a value that exceeds the number of source files. In that case, the compiler creates one object file for each source file then stops generating object files.

The following example commands demonstrate how to use `[Q]ipo2` option to compile large programs.

Operating System	Example Command
Windows*	`ifort /Qipo2 /c a.f90 b.f90`
Linux*	`ifort -ipo2 -c a.f90 b.f90`

Operating System	Example Command
macOS*	`ifort -ipo2 -c a.f90 b.f90`

In executing the above commands, the compiler generates object files using an OS-dependent naming convention. On Linux* and macOS*, the example command results in object files named `ipo_out.o`, `ipo_out1.o`, and `ipo_out2.o`. On Windows*, the file names follow the same convention; however, the file extensions will be `.obj`.

Link the resulting object files as shown in Using IPO or Linking Tools and Options..

Creating the Maximum Number of Object Files

Using `[Q]ipo-separate` allows you to force the compiler to generate the maximum number of true object files that the compiler will support during multiple object compilation. The maximum number of true object files is the equal to the number of mock object files passed on the link line.

For example, you can pass example commands similar to the following:

Operating System	Example Command
Windows*	`ifort a.obj b.obj c.obj /Qipo-separate /Qipo-c`
Linux*	`ifort a.o b.o c.o -ipo-separate -ipo-c`
macOS*	`ifort a.o b.o c.o -ipo-separate -ipo-c`

The compiler generates multiple object files that use the same naming convention discussed above.

Link the resulting object files as shown in Using IPO or Linking Tools and Options.

See Also

`ipo, Qipo`
compiler option

`ipo-c, Qipo-c`
compiler option

`ipo-jobs, Qipo-jobs`
compiler option

`ipo-S, Qipo-S`
compiler option

`ipo-separate, Qipo-separate`
compiler option

Understanding Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout. The analysis performed by the compiler during multi-file IPO determines a layout order for all of the routines for which it has intermediate representation (IR) information. For a multi-object IPO compilation, the compiler must tell the linker about the desired order.

The compiler first puts each routine in a named text section that varies depending on the operating system:

Windows:

- The first routine is placed in `.text$00001`, the second is placed in `.text$00002`, and so on.

Linux:

- The first routine is placed in `.text00001`, the second is placed in `.text00002`, and so on.

See Also

ipo-c, Qipo-c

ipo-S, Qipo-S

Creating a Library from IPO Objects

Linux* and macOS*

Libraries are often created using a library manager such as `xiar` for Linux*/macOS* or `xilib` for Windows. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

Example

```
xiar cru user.a a.o b.o
```

The above command creates a library named `user.a` containing the `a.o` and `b.o` objects.

If the objects have been created using `[Q]ipo -c` then the archive will not only contain a valid object, but the archive will also contain intermediate representation (IR) for that object file. For example, the following example will produce `a.o` and `b.o` that may be archived to produce a library containing both object code and IR for each source file.

Example

```
ifort -ipo -c a.f90 b.f90
```

The commands generate mock object files, which when placed in archive will also be accompanied by a true object file.

Using `xiar` is the same as specifying `xild -lib`.

macOS* Only

When using `xilibtool`, specify `-static` to generate static libraries, or specify `dynamic` to create dynamic libraries. For example, the following command will create a static library named `mylib.a` that includes the `a.o`, `b.o`, and `c.o` objects.

Example

```
xilibtool -static -o mylib.a a.o b.o c.o
```

Alternately, the following example command will create a dynamic library named `mylib.dylib` that includes the `a.o`, `b.o`, and `c.o` objects.

Example

```
xilibtool -dynamic -o mylib.dylib a.o b.o c.o
```

Specifying `xilibtool` is the same as specifying `xild -libtool`.

Windows* Only

Create libraries using `xilib` or `xilink -lib` to create libraries of IPO mock object files and link them on the command line.

For example, assume that you create three mock object files by using a command similar to the following:

Example

```
ifort /c /Qipo a.obj b.obj c.obj
```

Further assume `a.obj` contains the main subprogram. You can enter commands similar to the following to create a library.

Example

```
xilib -out:main.lib b.obj c.obj
or
xilink -lib -out:main.lib b.obj c.obj
```

You can link the library and the main program object file by entering a command similar to the following:

Example

```
xilink -out:result.exe a.obj main.lib
```

See Also

`dynamiclib`
compiler option

`ipo-c`, `Qipo-c`
compiler option

`static`
compiler option

Requesting Compiler Reports with the xi* Tools

The compiler options `qopt-report` (Linux* and macOS*) and `[Q]opt-report` (Windows*) generate optimization reports with different levels of detail. Related compiler options, listed under Optimization Report Options, allow you to specify the phase, direct output to a file (instead of `stderr`), and request reports from all routines with names containing a specific string as part of their name.

The xi* tools are used with inter-procedural optimization (IPO) during the final stage of IPO compilation. You can request compiler reports to be generated during the final IPO compilation by using certain options. The supported xi* tools are:

- Linker tools: `xilink` (Windows*) and `xild` (Linux* and macOS*)
- Library tools: `xilib` (Windows*), `xiar` (Linux* and macOS*), `xilibtool` (macOS*)

The following tables lists the compiler report options that can be used with the xi* tools during the final IPO compilation.

Optimization Report Option	Description
`-qopt-report`[=*n*] (Linux* and macOS*) `/Qopt-report`[:*n*] (Windows*)	Enables optimization report generation with different levels of detail. Valid values for *n* are 0 through 5. By default, when you specify this option without passing a value the compiler will generate a report with a medium level of detail. Higher numbers give greater levels of detail.

Optimization Report Option	Description
`-qopt-report-file=`*filename* (Linux* and macOS*) `/Qopt-report-file:`*filename* (Windows*)	Generates an optimization report and directs the report output to the specified *file* name. If you omit the path, the file is created in the current directory. To create the file in a different directory, specify the full path to the output file and its file name.
`-qopt-report-phase`[=*list*] (Linux* and macOS*) `/Qopt-report-phase`[:*list*] (Windows*)	Specifies a comma separated *list* of optimization phases to use when generating reports. If you do not specify a phase the compiler defaults to all. You can request a list of all available phases by using the `[Q]opt-report-help` option. To generate a report for the IPO phase, use the `-qopt-report-phase=`*ipo* (Linux* and macOS*) or `/Qopt-report-phase:`*ipo* (Windows) option.
`-qopt-report-routine=`*substring* (Linux* and macOS*) `/Qopt-report-routine:`*substring* (Windows*)	Generates reports for all routines with names containing *substring* as part of their name. You can also specify a sequence of substrings separated by commas. If you do this, the compiler generates an optimization report for each of the routines whose name contains one or more of these substrings. If *substring* is not specified, the compiler generates reports on all routines.
`-qopt-report-filter=`*string* (Linux* and macOS*) `/Qopt-report-filter:`*string* (Windows*)	Tells the compiler to find the indicated parts of your application specified by *string*, and generate optimization reports for them. If both `-qopt-report-routines=`*string1* and `qopt-report-filter=`*string2* are specified, it is treated as `-qopt-report-filter=`*string1;string2*.
`-qopt-report-help` (Linux* and macOS*) `/Qopt-report-help` (Windows*)	Displays the optimization phases available to use when using the `-qopt-report-phase` (Linux* and macOS*) or `[q or Q]opt-report-phase` (Windows*) or option.
`-qopt-report-names` (Linux* and macOS*) `/Qopt-report-names` (Windows*)	Specifies whether mangled or unmangled names appear in the optimization report. If this option is not specified, unmangled names are used by default. If you specify mangled, encoding (also known as decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the assembly listing. If you specify unmangled, no encoding (or decoration) is added to names in the optimization report. This is appropriate when you want to match annotations with the source listing. If you use this option, you do not have to specify option `-qopt-report` (Linux* OS and macOS*) or `/Qopt-report` (Windows* OS).

See Also

`qopt-report`, `Qopt-report`
compiler option

`qopt-report-file`, `Qopt-report-file`
compiler option

`qopt-report-help`, `Qopt-report-help`
compiler option

`qopt-report-phase`, `Qopt-report-phase`
compiler option

`qopt-report-routine`, `Qopt-report-routine`

compiler option

qopt-report-filter, Qopt-report-filter
compiler option

Inline Expansion of Functions

Inline function expansion does not require that the applications meet the criteria for whole program analysis normally required by IPO; so this optimization is one of the most important optimizations done in Interprocedural Optimization (IPO). For function calls that the compiler believes are frequently executed, the Intel® compiler often decides to replace the instructions of the call with code for the function itself.

In the compiler, inline function expansion is performed on relatively small user functions more often than on functions that are relatively large. This optimization improves application performance by performing the following:

- Removing the need to set up parameters for a function call
- Eliminating the function call branch
- Propagating constants

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the compiler can examine the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

Specifying the `[Q]ip` compiler option, single-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined within the current source file; in contrast, specifying the `[Q]ipo` compiler option, multi-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined in other files.

Caution
Using the `[Q]ip` and `[Q]ipo` (Windows*) options can, in some cases, significantly increase compile time and code size.

The Intel compiler does a certain amount of inlining at the default level. Although such inlining is similar to what is done when you use the [Q]ip option, the amount of inlining done is generally less than when you use the option.

Selecting Routines for Inlining

The compiler attempts to select the routines whose inline expansions provide the greatest benefit to program performance. The selection is done using default heuristics. The inlining heuristics used by the compiler differ based on whether or not you use options for Profile-Guided Optimizations (PGO): `[Q]prof-use` compiler option.

When you use PGO with `[Q]ip` or `[Q]ipo`, the compiler uses the following guidelines for applying heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.

Using IPO with PGO

Combining IPO and PGO typically produces better results than using IPO alone. PGO produces dynamic profiling information that can usually provide better optimization opportunities than the static profiling information used in IPO.

The compiler uses characteristics of the source code to estimate which function calls are executed most frequently. It applies these estimates to the PGO-based guidelines described above. The estimation of frequency, based on static characteristics of the source, is not always accurate.

See Also

fpic

ip, Qip

ipo, Qipo

prof-use, Qprof-use

Compiler Directed Inline Expansion of Functions

Without directions from the user, the compiler attempts to estimate what functions should be inlined to optimize application performance. See Inline Expansion of Functions for more information.

The following options are useful in situations where an application can benefit from user function inlining but does not need specific direction about inlining limits.

Option	Effect
`inline-level`(Linux* and macOS*) or `Ob` (Windows*)	Specifies the level of inline function expansion. Note that the option /Ob2 on Windows* is equivalent to -inline-level=2 on Linux* and macOS*. Allowed values are 0, 1, and 2.
`[Q]ip-no-inlining`	Disables only inlining enabled by the `[Q]ip`, `[Q]ipo`, or `Ob2` options.
`[Q]ip-no-pinlining`	Disables partial inlining enabled by the `[Q]ip` or `[Q]ipo` options. No other IPO optimizations are disabled.
setting `inline-debug-info` for the `debug` option	Indicates that the source position information for an inlined function should be retained, rather than replaced, by that of the call which is being inlined.

See Also

debug (Linux* and macOS*)

debug (Windows*)

Zi, Z7

inline-level, Ob

ip, Qip

ip-no-pinlining, Qip-no-pinlining

ipo, Qipo

Developer Directed Inline Expansion of User Functions

In addition to the options that support compiler directed inline expansion of user functions, the compiler also provides compiler options and directives that allow you to more precisely direct when and if inline function expansion should occur.

The compiler measures the relative size of a routine in an abstract value of intermediate language units, which is approximately equivalent to the number of instructions that will be generated. The compiler uses the intermediate language unit estimates to classify routines and functions as relatively small, medium, or large functions. The compiler then uses the estimates to determine when to inline a function; if the minimum criteria for inlining is met and all other things are equal, the compiler has an affinity for inlining relatively small functions and not inlining relative large functions.

Typically, the compiler targets functions that have been marked for inlining based on the following:

- **Procedure-specific inlining directives:** indicates to the compiler to inline calls within the targeted procedure if it is legal to do so. For example, `!DIR$ ATTRIBUTES INLINE`, `!DIR$ ATTRIBUTES FORCEINLINE`.

The following developer directed inlining options and directives provide the ability to change the boundaries used by the inliner to distinguish between small and large functions.

In general, you should use the `[Q]inline-factor` option before using the individual inlining options listed below; this single option effectively controls several other upper-limit options.

If your code hits an inlining limit, the compiler issues a warning at the highest warning level. The warning specifies which of the inlining limits have been hit, and the compiler option and/or directives needed to get a full report. For example, you could get a message as follows:

```
Inlining inhibited by limit max-total-size.  Use -qopt-report -qopt-report-phase=ipo for full
report.
```

Messages in the report refer directly to the command line options or directives that can be used to overcome the limits.

The following table lists the options you can use to fine-tune inline expansion of functions. The directives associated with the options are documented in the **Effect** column.

Option	Effect
`[Q]inline-factor`	Controls the multiplier applied to all inlining options that define upper limits: `inline-max-size`, `inline-max-total-size`, `inline-max-per-routine`, and `inline-max-per-compile`. While you can specify an individual increase in any of the upper-limit options, this single option provides an efficient means of controlling all of the upper-limit options with a single command. By default, this option uses a multiplier of 100, which corresponds to a factor of 1. Specifying 200 implies a factor of 2, and so on. Experiment with the multiplier carefully. You could increase the upper limits to allow too much inlining, which might result in your system running out of memory.
`[Q]inline-force-inline`	Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so. Without this option, the compiler treats functions declared with the `ATTRIBUTES INLINE` directive as merely being recommended for inlining. When this option is used, it is as if they were declared with the `ATTRIBUTES FORCEINLINE` directive.
`[Q]inline-min-size`	Redefines the maximum size of small routines; routines that are equal to or smaller than the value specified are more likely to be inlined.

Option	Effect
[Q]inline-max-size	Redefines the minimum size of large routines; routines that are equal to or larger than the value specified are less likely to be inlined.
[Q]inline-max-total-size	Limits the expanded size of inlined functions. You can also use !DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:"INLINE_MAX_TOTAL_SIZE=N" to control the size an individual routine can grow through inlining.
[Q]inline-max-per-routine	Limits the number of times inlining can be applied within a routine. You can also use !DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:"INLINE_MAX_PER_ROUTINE=N" to control the number of times inlining may be applied to a routine.
[Q]inline-max-per-compile	Limits the number of times inlining can be applied within a compilation unit. The compilation unit limit depends on the whether or not you specify the [Q]ipo compiler option. If you enable IPO, all source files that are part of the compilation are considered one compilation unit. For compilations not involving IPO each source file is considered an individual compilation unit.

See Also

inline-factor, Qinline-factor
inline-forceinline, Qinline-forceinline
inline-max-per-compile, Qinline-max-per-compile
inline-max-per-routine, Qinline-max-per-routine
inline-max-total-size, Qinline-max-total-size
inline-max-size, Qinline-max-size
inline-min-size, Qinline-min-size
ipo, Qipo

Inlining Report

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the compiler examines the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

The Inlining Report is part of the Opt Report. The compiler options `-qopt-report` (Linux* and macOS*) and `/Qopt-report` (Windows*) generate optimization reports with different levels of detail. Related compiler options, listed under Optimization Report Options, allow you to specify the phase, direct output to a specific file, `stdout` or `stderr`, and request reports from all routines with names containing a specific string as part of their name.

The inlining report is a description of the inlining choices that were made for each routine that is compiled in the program. It is produced as part of the opt report. To restrict the opt report to contain ONLY the inlining report, use the option `-qopt-report-phase=ipo` (Linux* and macOS*) or `/Qopt-report-phase:ipo` (Windows*).

The user can control the amount of information by specifying a level for the inlining report. The level is shown by a number from 1 to 5. Level 1 contains the smallest amount of information, and each level adds information to the report. Level 2 is the default report.

Level	Summary
Level 1	Shows each call that was inlined
Level 2 (default report)	Shows the values of the key inlining options
Level 3	Shows the calls to routines with external linkage
Level 4	Shows: • Whole program information • Size (sz) of the each routine inlined and the increase in application size (isz) due to each instance of inlining • Routine percentages • Calls that are not inlined
Level 5	Shows inlining footnotes, which contain advice on how to change the inlining to potentially improve application performance

The inlining report gives you an in-depth overview of the compiler's inlining decisions, which occur within five levels of granularity. You can specify levels with `-qopt-report=1`, `-qopt-report=2`, etc., (Linux* and macOS*) or `/Qopt-report=1`, `/Qopt-report=2`, etc. (Windows*). See below for specific level details.

Level 1

The Inlining Report is activated when you run the Optimization Report, using `[q or Q]qopt-report`.

For each routine you compile, you get one report with the title INLINE REPORT that shows the calls inlined into that routine.

Example: Inlining Report Level 1 - Typical Routine

```
INLINE REPORT: (APPLU)
  -> INLINE: (295,12) SETBV
    -> INLINE: (398,18) EXACT
    -> INLINE: (399,18) EXACT
    -> INLINE: (409,18) EXACT
    -> INLINE: (410,18) EXACT
    -> INLINE: (420,18) EXACT
    -> INLINE: (421,18) EXACT
  -> INLINE: (299,12) SETIV
  -> (303,12) ERHS
  -> (307,12) SSOR
  -> INLINE: (311,12) ERROR
    -> INLINE: (1518,24) EXACT
    -> INLINE: (1552,24) EXACT
  -> INLINE: (315,12) PINTGR
  -> (319,12) VERIFY
```

The report gives the name of the compiled routine (APPLU), and contains one line for each call that the compiler decided to inline or not inline. In the above report, the compiler made 15 inlining decisions for calls in the routine APPLU. It decided to inline 12 of the calls. These decisions are indicated by the lines which start with `-> INLINE`. It decided not to inline three of the calls. These decisions are indicated by the lines without the word INLINE.

On each line, the position of the call in the source code is given in parentheses, followed by the name of the routine being called. For example:

```
-> INLINE: (398,18) EXACT
```

This refers to a call at line 398 column 18 to a routine called EXACT.

Level 2

Level 2 includes the values of important compiler options related to inlining. Unless the user specifies one of these values by using the option on the command line, the default value of the option in shown. You can read more about the meaning of the individual inlining options in the Inlining Options section.

Example: Inlining Report Level 2 - Values of Inlining Options

```
INLINING OPTION VALUES:
  -inline-factor: 100
  -inline-min-size: 30
  -inline-max-size: 230
  -inline-max-total-size: 2000
  -inline-max-per-routine: 10000
  -inline-max-per-compile: 500000
```

Level 3

Level 3 contains one additional line for each call to an external routine made in the application. Such calls are not candidates for inlining, because the code for these routines is not present in the file or files being compiled.

Example: Inlining Report Level 3 - External Linkage

```
Begin optimization report for: APPLU
    Report from: Interprocedural optimizations [ipo]
INLINE REPORT: (APPLU) [1] applu.f (1,16)
   -> EXTERN: (1,16) for_set_reentrancy
   -> EXTERN: (80,7) for_read_seq_lis
```

Level 4

Level 4 adds four additional pieces of information. The specifics are shown below:

- Whole Program values:

  Example: Whole Program

  ```
  WHOLE PROGRAM [SAFE] [EITHER METHOD]: false
  WHOLE PROGRAM [SEEN] [TABLE METHOD]: true
  WHOLE PROGRAM [READ] [OBJECT READER METHOD]: false
  ```

  An application for which whole program is determined is subject to a higher degree of optimization than one which is not. The Intel compiler uses two methods of determining whole program, a TABLE METHOD and an OBJECT READER METHOD.

- The size of the routine [sz] and the inlined size of the routine [isz]. Usually isz is less than sz:

  Example: Size of the Routine (sz) vs. Inlined Size of the Routine (isz)

  ```
  -> INLINE: (295,12) SETBV (isz = 752) (sz = 755)
    -> INLINE: (398,18) EXACT (isz = 98) (sz = 109)
  ```

  In the above example, the routine SETBV was inlined into the routine that called it. The size of SETBV, before inlining, was 755 units. After inlining, the calling routine was increased by 752 units. The increase in the size of the calling routine is slightly less than the size of SETBV, because some of the overhead of calling SETBV was removed when SETBV was inlined.
- The percentage of time that has passed in the process of compiling the file:

Example: Percentage of Time Passed During Compilation

```
INLINE REPORT: (APPLU) [1/16=6.2%] applu.f (1,16)
```

For example, on the line above, [1/16 = 6.2%] indicates that APPLU is the first routine out of 16 to be compiled, and when this routine is done being compiled, 6.2% of the compilation is finished. You can use these numbers to estimate how long the compilation is going to take.

- The calls that did not get inlined and the reason why they did not get inlined. The reason is shown in double brackets [[ ]].

Example: Calls That Are Not Inlined

```
-> (303,12) ERHS (isz = 2125) (sz = 2128)
   [[ Inlining would exceed -inline-max-size value (2128>253)]]
```

In the above example, the routine ERHS is not inlined, because the size of the routine (2128 units) is larger than the allowable size (253 units). If you wish to inline routines that are this large, you can use the option `-inline-max-size=2128` (or larger).

Level 5

Level 5 adds the inlining footnotes.

Example: Use of Footnote

```
-> (303,12) ERHS (isz = 2058) (sz = 2061)
   [[ Inlining would exceed -inline-max-size-value (2061>230) <1>]]
```

The inlining footnotes explain the text found in the double brackets [[ ]]. They include a description for why an inlining call did not happen, and what you can do to make the inlining of this call happen.

The footnote annotation <1> refers to the first footnote in the INLINING FOOTNOTES section at the bottom of the inlining report, which is produced when the user selects Level 5. For example, the footnote produced for annotation <1> above is:

Example: Footnote Text

```
<1> The subprogram is larger than the inliner would normally inline. Use the
option -inline-max-size to increase the size of any subprogram that would
normally be inlined, add "!DIR$ATTRIBUTES FORCELINE" to the
declaration of the called function, or add "!DIR$ FORCELINE" before
the call site.
```

Fortran Language Extensions

Intel® Fortran provides a number of additional implementation features designed to simplify or enhance application development. The features in the section are varied; your choice to employ each feature depends on your application and development needs.

The *Intel® Fortran Language Reference* contains a section showing a summary of all the language extensions (non-standard features).

64-bit Addressing Support (Linux*)

This topic only applies to Linux* systems.

Applications designed to take advantage of Intel® 64 architecture can be built with one of three memory models:

- `small (-mcmodel=small)`

  This causes code and data to be restricted to the first 2GB of address space so that all accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.
- `medium (-mcmodel=medium)`

  This causes code to be restricted to the first 2GB; however, there is no restriction on data. Code can be addressed with IP-relative addressing, but access of data must use absolute addressing.
- `large (-mcmodel=large)`

  There are no restrictions on code or data; access to both code and data uses absolute addressing.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. This can affect code size and performance. (IP-relative addressing is somewhat faster.)

Additional Notes on Memory Models and on Large Data Objects

- When you specify the medium or large memory models, you must also specify the compiler option `-shared-intel` to ensure that the correct dynamic versions of the Intel run-time libraries are used.
- When you build shared objects (.so), Position-Independent Code (PIC) is specified (that is, `-fpic` is added by the compiler driver) so that a single .so can support all three memory models. However, code that is to be placed in a static library, or linked statically, must be built with the proper memory model specified. Note that there is a performance impact to specifying the medium or large memory models.
- The use of the memory model (medium, large) option and the `-shared-intel` option is required as a by-product of the code models stipulated in the 64-bit Application Binary Interface (ABI), which is written specifically for processors with the 64-bit memory extensions. Both the compiler and the GNU linker (ld) are responsible for generating the proper code and necessary relocations on this platform according to the chosen memory model.
- The 2GB restriction on Intel® 64 architecture involves not only arrays greater than 2GB, but also COMMON blocks and local data with a total size greater than 2GB. The Compiler Options reference contains additional discussion of the supported memory models and offers details about the 2GB restrictions for each model (see the description for option mcmodel).
- If, during linking, you fail to use the appropriate memory model and dynamic library options, an error message in this format occurs:

```
<some lib.a library>(some .o): In Function <function>:
  : relocation truncated to fit: R_X86_64_PC32 <some symbol>
```

Traceback

When a Fortran program terminates due to a severe error condition, the Fortran run-time system displays additional diagnostic information after the run-time message.

The Fortran run-time system attempts to walk back up the call chain and produce a report of the calling sequence leading to the error as part of the default diagnostic message report. This is known as traceback. The minimum information displayed includes:

- The standard Fortran run-time Run-Time Message Display and Format text that explains the error condition.
- A tabular report that contains one line per call stack frame. This information includes at least the image name and a hexadecimal PC in that image.

The information displayed under the Routine, Line, and Source columns depends on whether your program was compiled with the `traceback` option.

For example, if the `traceback` option is specified, the displayed information might resemble the following:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image              PC        Routine         Line     Source
libifcorert.dll    1000A3B2  Unknown         Unknown  Unknown
libifcorert.dll    1000A184  Unknown         Unknown  Unknown
libifcorert.dll    10009324  Unknown         Unknown  Unknown
libifcorert.dll    10009596  Unknown         Unknown  Unknown
libifcorert.dll    10024193  Unknown         Unknown  Unknown
teof.exe           004011A9  AGAIN                21  teof.for
teof.exe           004010DD  GO                   15  teof.for
teof.exe           004010A7  WE                   11  teof.for
teof.exe           00401071  HERE                  7  teof.for
teof.exe           00401035  TEOF                  3  teof.for
teof.exe           004013D9  Unknown         Unknown  Unknown
teof.exe           004012DF  Unknown         Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown         Unknown  Unknown
```

If the same program is *not* compiled with the `traceback` option:

- The Routine name, Line number, and Source file columns would be reported as "Unknown."
- A link map file is usually needed to locate the cause of the error.

The `traceback` option provides program counter (PC) to source file line correlation information to appear in the displayed error message information, which simplifies the task of locating the cause of severe run-time errors.

For Fortran objects generated with the `traceback` option, the compiler generates additional information used by the Fortran run-time system to automatically correlate PC values to the routine name in which they occur, Fortran source file, and line number in the source file. This information is displayed in the run-time error diagnostic report.

Automatic PC correlation is only supported for Fortran code. For non-Fortran code, only the hexadecimal PC locations are reported.

See Also

`traceback` compiler option

Tradeoffs and Restrictions in Using Traceback

This topic describes tradeoffs and restrictions that apply to using traceback.

Effect on Image Size

Using the `traceback` option to get automatic PC correlation increases the size of an image. For any application, the developer must decide if the increase in image size is worth the benefit of automatic PC correlation or if manually correlating PCs with a map file is acceptable.

The approach of providing automatic correlation information in the image was used so that no run-time penalty is incurred by building the information "on the fly" as your application executes. No run-time diagnostic code is invoked unless your application is terminating due to a severe error.

C Compiler Omit Frame Pointer Option on Systems Using IA-32 Architecture

The following routines are used to walk the stack:

- For Windows*, the Windows API routine StackWalk( ) in imagehlp.dll
- For Linux* and for macOS*, _Unwind_ForcedUnwind( ), _Unwind_GetIP( ), _Unwind_GetRegionStart( ) and _Unwind_GetGr( ) routines in libunwind.so

In an environment using IA-32 architecture, there are no firm software calling standards documented. Compiler developers are under no constraints to use machine registers in any particular way or to hook up procedures in any particular way. The stack walking routines listed above use a set of heuristics to determine

how to walk the call stack. That is, they make a "best guess" to determine how a program reached a particular point in the call chain. With C code that has been compiled with Visual C++* using the Omit Frame Pointer option -- either `-fomit-frame-pointer` (Linux and macOS*) or `/Oy` (Windows) -- this "best guess" is not usually the correct one.

If you are mixing Fortran and C code and you are concerned about stack tracing, consider disabling the `-fomit-frame-pointer` or `/Oy` option in your C compilations. Otherwise, traceback will most likely not work for you.

Inlined Routines Will Not Be Displayed

Inlining can cause some routines not to be realized as separate stack frames, so the traceback will not show inlined routines.

Stack Trace Failure

Programs can fail for a number of reasons, often with unpredictable consequences. Memory corruption by erroneously executing code is one possibility. Stack memory can be corrupted in such a way that the attempt to trace the call stack will result in access violations or other undesirable consequences. The stack-tracing run-time code is guarded with a local exception filter. If the traceback attempt fails due to a hard detectable condition, the run-time will report this in its diagnostic output message as:

```
Stack trace terminated abnormally
```

Be forewarned, however: It is also possible for memory to be corrupted in such a way that a stack trace can seem to complete successfully with no hint of a problem. The bit patterns it finds in corrupted memory where the stack used to be, and then uses to access memory, may constitute perfectly valid memory addresses for the program to be accessing. They just do not happen to have any connection to what the stack used to look like. So, if it appears that the stack walk completed normally, but the reported PCs make no sense to you, then consider ignoring the stack trace output in diagnosing your problem.

Another condition that will disable the stack trace process is your program exiting because it has exhausted virtual memory resources.

The stack trace can fail if the run-time system cannot dynamically load libunwind.so (Linux and macOS*) or imagehlp.dll (Windows) or cannot find the necessary routines from that library. In this case, you still get the basic run-time diagnostic message; you will not get any call stack information.

Linker /incremental:no Option on Windows Operating Systems

The following applies to Windows operating systems only.

When incremental linking is enabled, automatic PC correlation does not work. Use of incremental linking always disables automatic PC correlation even if you specify `/traceback` during compilation.

When you use incremental linking, the default hexadecimal (hex) PC values will still appear in the output. To correlate from the hexadecimal PC values to routine containing the PC addresses requires use of a linker map file. However, if you request a map file during linking, incremental linking becomes disabled. Thus to allow any PC values generated for a run-time problem to be helpful, incremental linking must be disabled.

In the integrated development environment, you can use the Call stack display, so incremental linking is not a problem.

Sample Programs and Traceback Information

The following sections provide sample programs that show the use of traceback to locate the cause of the error:

- Example: End-of-File Condition, Program teof
- Example: Machine Exception Condition, Program ovf

Note that the hex PC's and contents of registers displayed in these program outputs are meant as representative examples of typical output. The PC's will change over time, as the libraries and other tools used to create an image change.

Example: End-of-File Condition, Program teof

In the following example, a READ statement creates an End-Of-File error, which the application has not handled:

```
 program teof
 integer*4 i,res
 i=here( )
 end
integer*4 function here( )
 here = we( )
 end
integer*4 function we( )
 we = go( )
 end
integer*4 function go( )
 go = again( )
 end
integer*4 function again( )
 integer*4 a
 open(10,file='xxx.dat',form='unformatted',status='unknown')
 read(10) a
 again=a
 end
```

The diagnostic output that results when this program is built with traceback enabled, optimization disabled, and linked against the shared Fortran run-time library on the Intel® 64 architecture platform is similar to the following:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image            PC                Routine            Line     Source
libifcorert.dll  000007FED2B232D9  Unknown            Unknown  Unknown
libifcorert.dll  000007FED2B6CEE0  Unknown            Unknown  Unknown
teof.exe         000000013F931193  AGAIN                   17  teof.f90
teof.exe         000000013F93109B  GO                      12  teof.f90
teof.exe         000000013F931072  WE                       9  teof.f90
teof.exe         000000013F931049  HERE                     6  teof.f90
teof.exe         000000013F93101E  MAIN__                   3  teof.f90
teof.exe         000000013F96ADCE  Unknown            Unknown  Unknown
teof.exe         000000013F96B64C  Unknown            Unknown  Unknown
kernel32.dll     0000000076CC59BD  Unknown            Unknown  Unknown
ntdll.dll        0000000076EFA2E1  Unknown            Unknown  Unknown
```

If optimization is not disabled (`/Od` on Windows* or `-O0` on Linux* and macOS*), procedure inlining may collapse the call stack and make it more difficult to locate a problem.

The first line of the output is the standard Fortran run-time error message. What follows is the result of walking the call stack in reverse order to determine where the error originated. Each line of output represents a call frame on the stack. Since the application was compiled with the `traceback` option, the PCs that fall in Fortran code are correlated to their matching routine name, line number and source module. PCs that are not in Fortran code are not correlated and are reported as "Unknown."

The first two frames show the calls to routines in the Fortran run-time library (in reverse order). Since the application was linked against the shared version of the library, the image name reported is either libifcore.so (Linux* and macOS*) or libifcorert.dll (Windows*). These are the run-time routines that were called to do

the READ and upon detection of the EOF condition, were invoked to report the error. In the case of an unhandled I/O programming error, there will always be a few frames on the call stack down in run-time code like this.

The stack frame of real interest to the Fortran developer is the first frame in image teof.exe, which shows that the error originated in the routine named AGAIN in source module `teof.f90` at line 17. Looking in the source code at line 21, you can see the Fortran READ statement that incurred the end-of-file condition.

The next four frames show the trail of calls in the Fortran user code that led to the routine that got the error (TEOF->HERE->WE->GO->AGAIN).

Finally, the bottom four frames are routines which handled the startup and initialization of the program.

If this program had been linked against the static Fortran run-time library, the output would then look like:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image              PC                Routine            Line     Source
teof.exe           000000013F941FFB  Unknown            Unknown  Unknown
teof.exe           000000013F9380A0  Unknown            Unknown  Unknown
teof.exe           000000013F931193  AGAIN                   17  teof.f90
teof.exe           000000013F93109B  GO                      12  teof.f90
teof.exe           000000013F931072  WE                       9  teof.f90
teof.exe           000000013F931049  HERE                     6  teof.f90
teof.exe           000000013F93101E  MAIN__                   3  teof.f90
teof.exe           000000013F96ADCE  Unknown            Unknown  Unknown
teof.exe           000000013F96B64C  Unknown            Unknown  Unknown
kernel32.dll       0000000076CC59BD  Unknown            Unknown  Unknown
ntdll.dll          0000000076EFA2E1  Unknown            Unknown  Unknown
```

Notice that the initial two stack frames now show routines in image `teof.exe`, not libifcore.so (Linux and macOS*) or `libifcorert.dll` (Windows). The routines are the same two run-time routines as previously reported for the shared library case but since the application was linked against the archive library libifcore.a (Linux and macOS*) or the static Fortran run-time library `libifcore.lib` (Windows), the object modules containing these routines were linked into the application image (`teof.exe`). You can use a Generating Listing and Map Files to determine the locations of uncorrelated PCs.

Now suppose the application was compiled *without* traceback enabled and, once again, linked against the static Fortran library. The diagnostic output would appear as follows:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image              PC                Routine            Line     Source
teof.exe           000000013F851FFB  Unknown            Unknown  Unknown
teof.exe           000000013F8480A0  Unknown            Unknown  Unknown
teof.exe           000000013F841193  Unknown            Unknown  Unknown
teof.exe           000000013F84109B  Unknown            Unknown  Unknown
teof.exe           000000013F841072  Unknown            Unknown  Unknown
teof.exe           000000013F841049  Unknown            Unknown  Unknown
teof.exe           000000013F84101E  Unknown            Unknown  Unknown
teof.exe           000000013F87ADCE  Unknown            Unknown  Unknown
teof.exe           000000013F87B64C  Unknown            Unknown  Unknown
kernel32.dll       0000000076CC59BD  Unknown            Unknown  Unknown
ntdll.dll          0000000076EFA2E1  Unknown            Unknown  Unknown
```

Without the correlation information in the image that option `traceback` previously supplied, the Fortran run-time system cannot correlate PC's to routine name, line number, and source file. You can still use the Generating Listing and Map Files to at least determine the routine names and what modules they are in.

Remember that compiling with the `traceback` option increases the size of your application's image because of the extra PC correlation information included in the image. You can see if the extra traceback information is included in an image (checking for the presence of a `.trace` section) by entering:

```
objdump -h your_app.exe   ! Linux OS
otool -l your_app.exe     ! macOS*
link -dump -summary your_app.exe   ! Windows OS
```

Build your application with and without traceback and compare the file size of each image. Check the file size with a simple directory command.

For this simple `teof.exe` example, the traceback correlation information adds about 512 bytes to the image size. In a real application, this would probably be much larger. For any application, the developer must decide if the increase in image size is worth the benefit of automatic PC correlation or if manually correlating PC's with a map file is acceptable.

If an error occurs when traceback was requested during compilation, the run-time library will produce the correlated call stack display.

If an error occurs when traceback was disabled during compilation, the run-time library will produce the uncorrelated call stack display.

If you do not want to see the call stack information displayed, you can set the Supported Environment Variable`FOR_DISABLE_STACK_TRACE` to true. You will still get the Fortran run-time error message:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
```

Example: Machine Exception Condition, Program ovf

The following program generates a floating-point overflow exception when compiled with the `fpe` option value 0:

```
 program ovf
 real*4 a
 a=1e37
 do i=1,10
   a=hey(a)
 end do
 print *, 'a= ', a
 end
real*4 function hey(b)
 real*4 b
 hey = watch(b)
 end
real*4 function watch(b)
 real*4 b
 watch = out(b)
 end
real*4 function out(b)
 real*4 b
 out = below(b)
 end
real*4 function below(b)
 real*4 b
 below = b*10.0e0
 end
```

Assume this program is compiled with the following:

- Option `fpe` value 0

- Option `traceback`
- Option `-O0` (Linux and macOS*) or `/Od` (Windows)

On a system based on IA-32 architecture, the traceback output is similar to the following:

```
forrtl: error (72): floating overflow
Image              PC        Routine              Line     Source
ovf.exe            001211A3  _BELOW                    23  ovf.f90
ovf.exe            0012116F  _OUT                      19  ovf.f90
ovf.exe            0012113D  _WATCH                    15  ovf.f90
ovf.exe            0012110B  _HEY                      11  ovf.f90
ovf.exe            0012104E  _MAIN__                    5  ovf.f90
ovf.exe            0015B31F  Unknown              Unknown  Unknown
ovf.exe            0015BADD  Unknown              Unknown  Unknown
kernel32.dll       7515338A  Unknown              Unknown  Unknown
ntdll.dll          770E9902  Unknown              Unknown  Unknown
ntdll.dll          770E98D5  Unknown              Unknown  Unknown
```

Notice that unlike the previous example of an unhandled I/O programming error, the stack walk can begin right at the point of the exception. There are no run-time routines on the call stack to dig through. The overflow occurs in routine BELOW at PC 001211A3, which is correlated to line 23 of the source file `ovf.f90`.

When the program is compiled at a higher optimization level of `O2`, along with option `fpe` value 0 and the `traceback` option, the traceback output appears as follows:

```
forrtl: error (72): floating overflow
Image              PC        Routine              Line     Source
ovf.exe            00AE1059  _MAIN__                    7  ovf.f90
ovf.exe            00B1B75F  Unknown              Unknown  Unknown
ovf.exe            00B1BADD  Unknown              Unknown  Unknown
kernel32.dll       7515338A  Unknown              Unknown  Unknown
ntdll.dll          770E9902  Unknown              Unknown  Unknown
ntdll.dll          770E98D5  Unknown              Unknown  Unknown
```

With `O2`, the entire program has been inlined.

The main program, `OVF`, no longer calls routine `HEY`. While the output is not quite what one might have expected intuitively, it is still entirely correct. You need to keep in mind the effects of compiler optimization when you interpret the diagnostic information reported for a failure in a release image.

If the same image were executed again, this time with the environment variable called `TBK_ENABLE_VERBOSE_STACK_TRACE` set to True, you would also see a dump of the exception context record at the time of the error. Here is an excerpt of how that might appear on a system using IA-32 architecture:

```
forrtl: error (72): floating overflow
Hex Dump Of Exception Record Context Information:
Exception Context:  Processor Control and Status Registers.
EFlags:  00010212
CS:  0000001B  EIP:  00401161  SS:   00000023  ESP:  0012FE38  EBP:  0012FE60
Exception Context:  Processor Integer Registers.
EAX: 00444488  EBX:  00000009  ECX:  00444488  EDX:  00000002
ESI: 0012FBBC  EDI:  F9A70030
Exception Context:  Processor Segment Registers.
DS:  00000023  ES:   00000023  FS:   00000038  GS:   00000000
Exception Context:  Floating Point Control and Status Registers.
ControlWord:  FFFF0262  ErrorOffset:    0040115E  DataOffset:    0012FE5C
StatusWord:   FFFFF8A8  ErrorSelector:  015D001B  DataSelector:  FFFF0023
TagWord:      FFFF3FFF  Cr0NpxState:    00000000
Exception Context:  Floating Point RegisterArea.
RegisterArea[00]:  4080BC143F4000000000  RegisterArea[10]:  F7A0FFFFFFFF77F9D860
RegisterArea[20]:  00131EF0000800060012  RegisterArea[30]:  00000012F7C002080006
```

```
RegisterArea[40]:  0208000600000000000  RegisterArea[50]:  0000000000000012F7D0
RegisterArea[60]:  0000000000000300000  RegisterArea[70]:  FBBC000000300137D9EF
...
```

See Also

`traceback` compiler option
`fpe` compiler option
`O` compiler option
`Od` compiler option

Allocating Common Blocks

Use the `[Q]dyncom` option to dynamically allocate common blocks at run time.

This option designates a common block to be dynamic. The space for its data is allocated at run time rather than at compile time. On entry to each routine containing a declaration of the dynamic common block, a check is performed to see whether space for the common block has been allocated. If the dynamic common block is not yet allocated, space is allocated at that time.

> **NOTE**
> On macOS* systems, to successfully enable dynamic allocation of common blocks, you must specify the link-time option `-undefined dynamic_lookup` as well as option `-dyncom`.

The following command-line example specifies the dynamic common option with the names of the common blocks to be allocated dynamically at run time:

```
ifort -dyncom "blk1,blk2,blk3" test.f ! Linux and macOS*
ifort /Qdyncom"BLK1,BLK2,BLK3" test.f ! Windows
```

where `BLK1`, `BLK2`, and `BLK3` are the names of the common blocks to be made dynamic.

Guidelines for Using the [Q]dyncom Option

The following are some limitations that you should be aware of when using the `[Q]dyncom` option:

- An entity in a dynamic common cannot be initialized in a DATA statement.
- Only named common blocks can be designated as dynamic COMMON.
- An entity in a dynamic common block must not be used in an EQUIVALENCE expression with an entity in a static common block or a DATA-initialized variable.

Why Use a Dynamic Common Block?

A main reason for using dynamic common blocks is to enable you to control the common block allocation by supplying your own allocation routine. To use your own allocation routine, you should link it ahead of the Fortran run-time library.

The C function prototype is:

```
void _FTN_ALLOC(void **mem, int *size, char *name);
```

where

- *mem* is the location of the base pointer of the common block that must be set by the routine to point to the block of memory allocated.
- *size* is the integer number of bytes of memory that the compiler has determined are necessary to allocate for the common block as it was declared in the program.

You can ignore this value and use whatever value is necessary for your purpose. You must return the size in bytes of the space you allocate. The library routine that calls the default or Fortran run-time function ensures that all other occurrences of this common block fit in the space you allocated. You can return the size in bytes of the space you allocate by modifying *size*.

- *name* is the name of the common block being dynamically allocated.

The equivalent Fortran INTERFACE specification is:

```
interface
subroutine my_ftn_alloc (mem, size, name) bind (C, name="_FTN_ALLOC")
    use, intrinsic               :: ISO_C_BINDING
    implicit none

    type(C_PTR),    intent(OUT)         :: mem
    integer(C_INT), intent(INOUT)       :: size
    character, dimension(*), intent(IN) :: name
end subroutine my_ftn_alloc
end interface
```

You can also use the Fortran run-time function `FOR__SET_FTN_ALLOC` to specify your own allocation routine. This method is especially helpful when you are using shared libraries. If you choose this method, it overrides the default routine `_FTN_ALLOC`.

Allocating Memory to Dynamic Common Blocks

The run-time library routine, `f90_dyncom`, performs memory allocation. The compiler calls this routine at the beginning of each routine in a program that contains a dynamic common block. In turn, this library routine calls `_FTN_ALLOC` to allocate memory. By default, the compiler passes the size in bytes of the common block as declared in each routine to `f90_dyncom`, and then on to `_FTN_ALLOC`. The Fortran run-time library contains a default version of `_FTN_ALLOC`, which simply allocates the requested number of bytes and returns.

If you use the nonstandard extension having the common block of the same name declared with different sizes in different routines, you may get a run-time error depending on the order in which the routines containing the common block declarations are invoked.

You can now create your own routine to dynamically allocate common blocks, which is especially useful when you are sharing libraries. To use your own routine, you must specify the run-time function FOR__SET_FTN_ALLOC.

See Also

`dyncom, Qdyncom` compiler option

FOR__SET_FTN_ALLOC

Generating Listing and Map Files

Compiler-generated assembler output listings and linker-generated map files can help you understand the effects of compiler optimizations and see how your application is laid out in memory. They may also help you interpret the information provided in a stack trace at the time of an error.

How to Generate Assembler Output

When compiling from the command line, specify the `S` option:

```
ifort -S file.f90 ! Linux and macOS*
ifort file.f90 /S ! Windows OS
```

On Linux* and macOS* systems, the resulting assembly file name has a .s suffix. On Windows* systems, the resulting assembly file name has an .asm suffix.

On Windows*, you can also use the Microsoft Visual Studio* integrated development environment:

1. Select **Project > Properties**.
2. Click the **Fortran** tab.
3. In the **Output Files** category, change the **Assembler Output** settings according to your needs. You can choose from a number of options such as **No Listing**, **Assembly-only Listing**, and **Assembly, Machine Code and Source**.

How to Generate a Link Map (.map) File

When compiling from the command line, specify the `-Xlinker` and `-M` options (Linux and macOS*) or the `/map` (Windows) option:

```
ifort file.f90 -Xlinker -M  ! Linux and macOS*
ifort file.f90 /map ! Windows
```

On Windows systems, you can also use the Visual Studio integrated development environment:

1. Select **Project>Properties**.
2. Click the **Linker** tab.
3. In the **Debug** category, select **Generate Map File**.

How to Generate a Source File Listing

You can use the `list` compiler option to create a listing of the source file. The listing can contain the following information: a display of INCLUDE files, a symbol list with a line number cross reference for each routine, and a list of compiler options used for the compilation.

Use the `show` compiler option to control the contents of the listing file.

Additionally, on Windows systems, you can use the Visual Studio integrated development environment to create a source file listing:

1. Select **Project > Properties**.
2. Click the **Fortran** tab.
3. In the **Output Files** category, change the **Source Listing** setting to **Yes**.

See Also

`show` compiler option
`list` compiler option

Ability to Create Shared Libraries

You can create a shareable library by using one of the following compiler options.

Compiler Option	Library
`-shared` (Linux*)	`mylib.so`
`/libs:dll` (Windows*)	`mylib.dll`
`-dynamiclib` (macOS*)	`mylib.dylib`

When you use any of these options, do not specify the `c` option.

See Also

`shared` compiler option
`dynamiclib` compiler option
`libs:dll` compiler option
Linux* and macOS*: Creating Shared Libraries
Storing Routines in Shareable Libraries
macOS*: Using Shared Libraries on macOS*

Specifying Alternative Tools and Locations

The default tools are summarized in the table below.

Tool	Default	Provided with Intel® Fortran Compiler?
Assembler for IA-32 architecture-based applications and Intel® 64 architecture-based applications	MASM* (Windows*)	No
	operating system assembler, as (Linux* and macOS*)	No
Linker	Microsoft* linker (Windows)	No
	System linker, ld(1) (Linux and macOS*)	No

The Intel® Fortran Compiler lets you specify alternatives to default tools and locations for preprocessing, compilation, assembly, and linking. In addition, you can invoke options specific to the alternate tools on the command line. This functionality is provided by the `Qlocation` and `Qoption` options.

See Also

`Qlocation` compiler option

`Qoption` compiler option

Temporary Files Created by the Compiler or Linker

Temporary files created by the compiler or linker reside in the directory used by the operating system to store temporary files.

To store temporary files, the driver first checks for the TMP environment variable. If defined, the directory that TMP points to is used to store temporary files.

If the TMP environment variable is not defined, the driver then checks for the TMPDIR environment variable. If defined, the directory that TMPDIR points to is used to store temporary files.

If the TMPDIR environment variable is not defined, the driver then checks for the TEMP environment variable. If defined, the directory that TEMP points to is used to store temporary files.

For Windows*, if the TEMP environment variable is not defined, the current working directory is used to store temporary files. For Linux* and macOS*, if the TEMP environment variable is not defined, the `/tmp` directory is used to store temporary files.

Temporary files are usually deleted. Use the `[Q]save-temps` compiler option to save temporary files created by the compiler in the current working directory. This option only saves intermediate files that are normally created during compilation.

For performance reasons, use a local drive (rather than a network drive) to contain temporary files.

To view the file name and directory where each temporary file is created, use the `all` keyword for the `watch` option.

To create object files in your current working directory, use the `c` option.

Any object files that you specify on the command line are retained.

See Also

`save-temps, Qsave-temps` compiler option

`watch` compiler option

`c` compiler option

Using the Intel® Fortran COM Server (Windows*)

This topic only applies to Windows* operating systems.

The Component Object Model (COM) supports a model of client server interaction between a user of an object, the client, and the implementor of the object, the server.

This section discusses creating a COM server using Intel Fortran.

The following section, Using the Intel(R) Fortran Module Wizard (COM Client), discusses using COM and Automation objects from an Intel® Fortran application, using the Fortran COM Server.

Advantages of a COM Server (Windows*)

This topic only applies to Windows* operating systems.

A COM server consists of the implementation of one or more *object classes*. An object class is a type that describes the complete public calling interface ("signature") of an object. It describes the functionality that you want to make available to the users of the object. The COM server creates *instances* of the class, called *objects*, at the request of clients.

Some of the advantages of implementing your Fortran code as a COM server include:

- A COM server is a reusable component, which allows multiple applications to use the server. The classes specified by the server define a "contract" between the server and its clients. The server can change the specific implementation of the functionality without breaking the contract. That is, without requiring clients to be changed or rebuilt.
- A COM server is programming-language independent. Multiple development tools can be used to access the server's functionality, including Microsoft Visual Basic*, C++, and Fortran.
- A COM server is self-describing. The server provides a type library that describes the classes and interfaces. Many tools can take advantage of this information and relieve the client programmer from needing to understand low-level invocation details, such as calling conventions. This is a great improvement over multi-language programming with DLLs, where the client programmer has to understand the details of data types and calling conventions.
- A COM server is self-registering. The clients do not need to worry about where the server is located on their system, as COM finds this information in the system registry.
- A COM server can be implemented as an in-process server. Like a DLL, it is loaded into the client's process. A COM server can also be implemented as a separate application and can even reside on a separate machine.

Intel Fortran provides the Fortran COM Server application wizard and special editor for modifying the COM Server structure and generating Fortran code necessary to implement a COM server. This allows you to concentrate on the code that is specific to the functionality that your server provides to its clients.

As explained in Understanding COM and Automation Objects, COM supports two types of servers: COM servers and Automation servers. The Fortran COM Server Wizard can only create a COM server or a server that supports dual interfaces. The wizard cannot create an Automation-only interface.

For more information about creating COM servers, see Creating the Fortran COM Server.

Understanding COM Server Concepts (Windows*)

This topic only applies to Windows* operating systems.

The Fortran COM Server application wizard generates an initial Visual Studio project and creates a COM Hierarchy file, which describes the infrastructure of a COM server. You can modify this file with the COM Server Hierarchy Editor to define the implementation of one or more *object classes*, including its interface(s), and method(s).

When you save the hierarchy file, the Fortran source files are generated. You then need to write code to implement each method.

What Information You Need to Provide

When you create the COM Server project, you will need to describe the COM server classes that you want to implement.

A class implements one or more COM *interfaces*. In COM terminology, an interface is a semantically related set of functions. The interface as a whole represents a "feature", or set of related functionality, that is implemented by the class. An interface contains *methods*, otherwise known as member functions. A method is a routine that performs one of the actions that make up the feature. As far as the Fortran COM Server is concerned, methods are Fortran functions that take arguments and return a value like any other Fortran function.

Consider a simple example of a class that you will create using the Fortran COM Server Application Wizard, called *AddingMachine*. The class contains a single interface that we call *IAdd*. By convention, all interface names begin with a capital letter "I". You can define three methods in the IAdd interface:

- *Clear*, which takes no arguments and sets the current value of the adding machine to 0.
- *Add*, which takes a single REAL argument, the amount to add to the current value.
- *GetValue*, which returns the current value.

These methods allow you to perform specific, distinct tasks with the IAdd interface from any language that supports a COM client. The Fortran COM Server Editor provides a user interface to enter this information about the class (in this case the AddingMachine class), which is discussed later in Creating the Fortran COM Server.

An interface can also contain *properties*. Properties are method pairs that set or return information about the state of an object. Properties must currently be implemented using the get_Method and put_Method and the same DISPID property.

In terms of the data associated with an object, a key concept of object-oriented programming is *encapsulation*. Encapsulation means that all of the details about how the object is implemented, including the data that it uses and the logic that it uses to perform its work, is hidden from the client. The client's only access to the object is through the interfaces that the object supports.

You need to define the data that the object uses and code the logic of the methods. For the data, the Fortran COM Server Application Wizard uses the model that each instance of the object has an associated instance of a Fortran derived-type. The code generated by the wizard takes care of creating and destroying the instances of the derived-type as objects are created and destroyed.

You define the fields of the derived-type. For example, with our AddingMachine, each AddingMachine object needs to store the current value. The derived-type associated with each AddingMachine object would contain a single field of type REAL. We name it CurrentValue. Note that each instance of the AddingMachine object has its own instance of the derived-type and its own CurrentValue. This means that the server could support multiple clients simultaneously and each client would see its own AddingMachine. That is, each client is unaffected by the existence of other clients. The derived-type associated with each object is discussed in detail in Creating the Fortran COM Server.

To summarize, at a high level, what you need to do to create a COM server:

- Create the COM Server project using the COM Server Application Wizard.
- Define the class(es), interface(s), method(s), and properties using the COM Server Hierarchy Editor to create the Hierarchy file.
- Define the fields in the derived-type that is associated with each instance of a class.
- Write code that initializes the fields in the derived-type (if necessary), and code that releases any resources used by the fields in the derived-type (if necessary).
- Write the code that implements the methods.

What the COM Server Will Provide

The Fortran COM Server Editor generates source files that implement all of the infrastructure, or "plumbing", of the COM server. The generated files take care of such tasks as:

- Defining the GUIDs that uniquely identify your classes and interfaces. For a discussion of GUIDs, see Getting a Pointer to an Objects Interface.
- Registering the server and your classes and interfaces on the system.
- Implementing the class factory that creates instances of your object.
- Creating the Interface Definition Library (IDL) and type library that describes your classes and interfaces. For a discussion of type libraries, see Using the Module Wizard to Generate Code.
- Implementing the IUnknown and IDispatch interfaces for your object. All of the interfaces created by the Fortran COM Server Editor are derived from IUnknown or IDispatch (dual interfaces).
- Creating and destroying instances of the derived-type associated with your object.
- Invoking the Fortran routines that implement your object's methods.

The majority of these source files are generated fully by the Fortran COM Server Editor and are not modified by you. Other files contain the skeleton or template of your derived-type and methods. You edit these Fortran source files to fill in your implementation. A walk-through of the AddingMachine example will show you how it's done.

Creating the Fortran COM Server (Windows*)

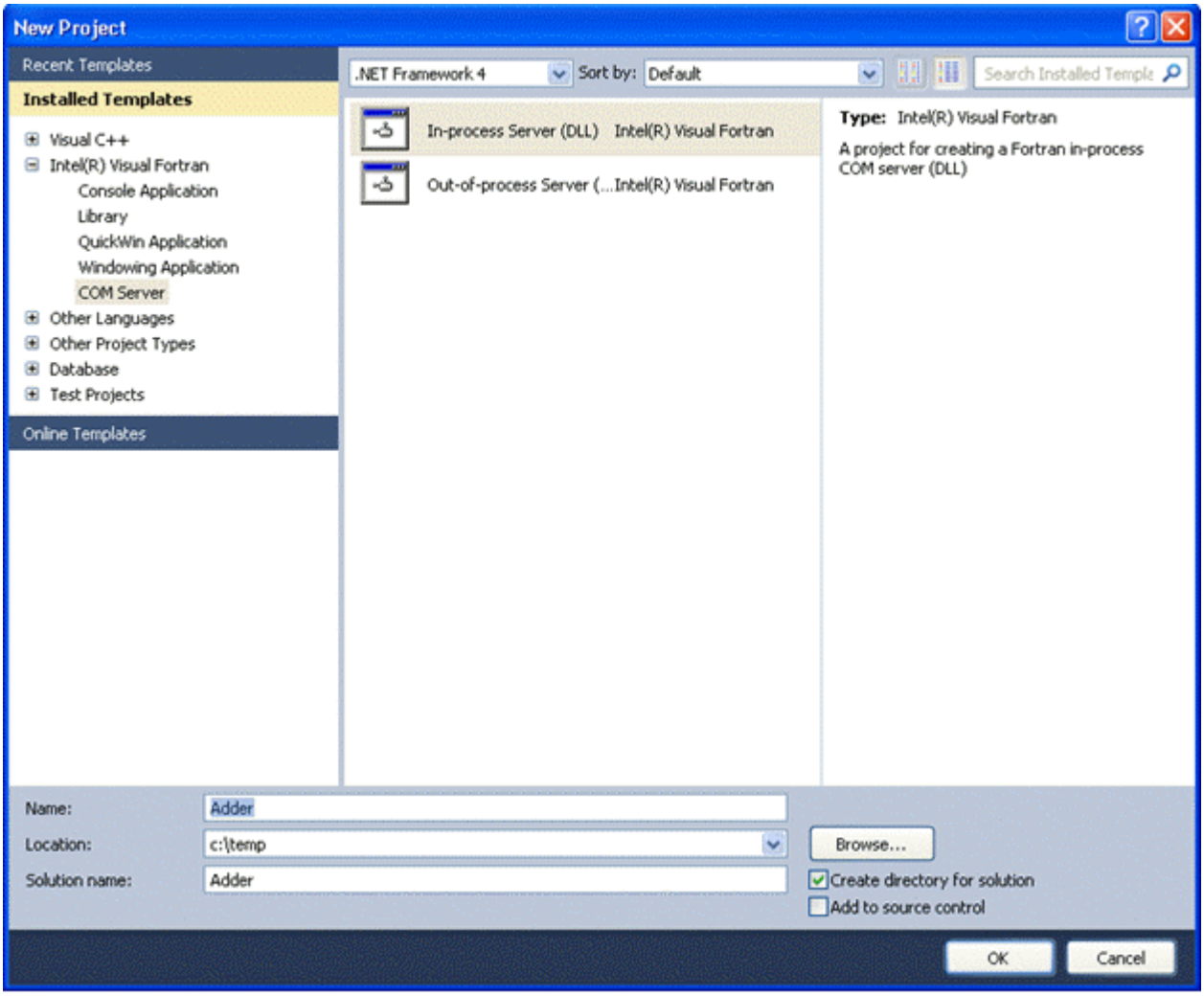
This topic only applies to Windows* operating systems.

This section shows how to create a project, specify the COM server characteristics, generate code for, and implement a sample Fortran COM server project called Adder.

The first step in creating a Fortran COM Server is to create a new project.

Creating a Fortran COM Server Project using the Visual Studio* IDE:

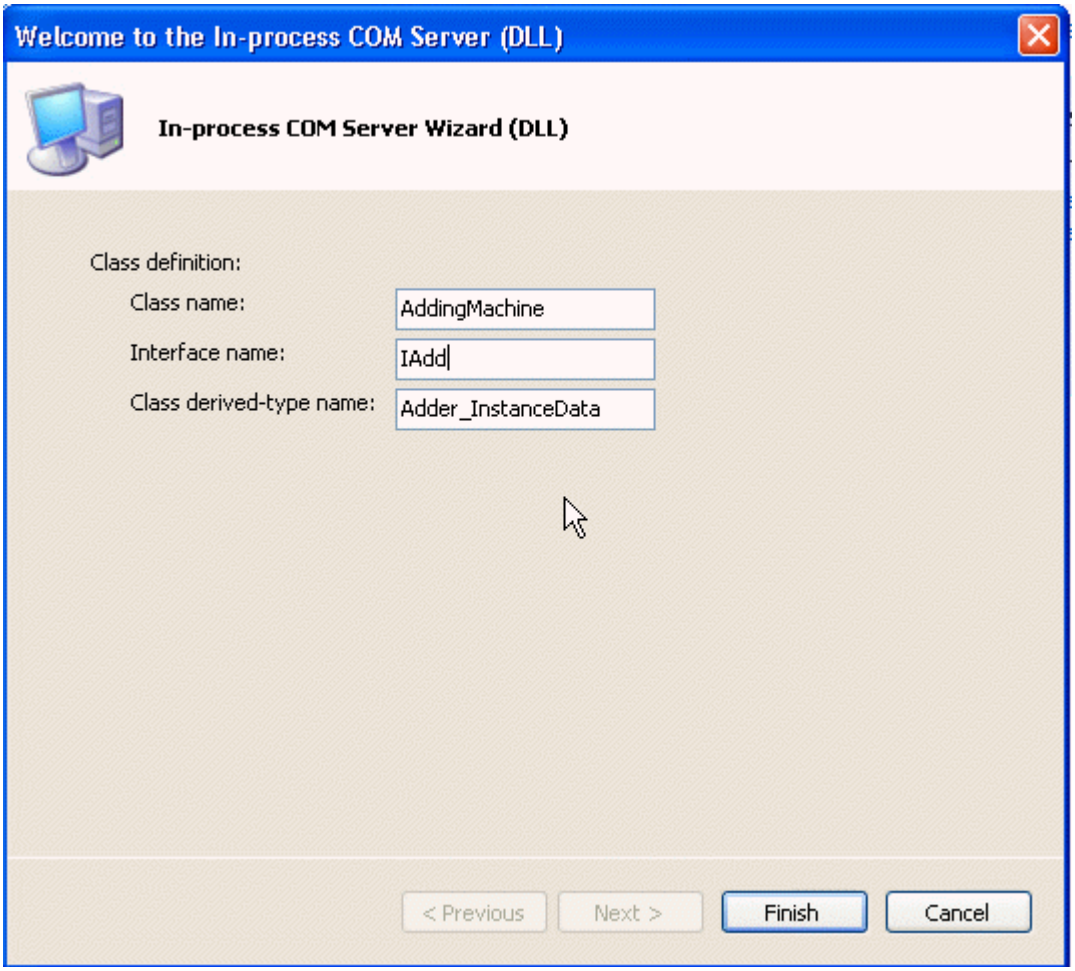
1. Start the Visual Studio IDE. Choose **File > New > Project**.
2. In the **New Project** dialog box, expand the Intel(R) Visual Fortran item and click **COM Server**.



To create the sample project, choose **In-process Server (DLL**). Now do the following:

1. Enter **Adder** as the name of the project.
2. Accept or modify the project folder location.
3. Click **OK**.

You can use the project AppWizard once per project to create the project files and skeleton template. The Fortran COM Server AppWizard displays default text for the interface name and class derived type name.



Now do the following:

1. Enter class name `AddingMachine`.
2. Shorten the default interface name to `IAdd`.
3. Accept the default class derived type name.
4. Click **Finish**. (If you click **Cancel**, project creation is terminated.)

The project is created and the COM Server Hierarchy file (`adder.hie`) is opened in the COM Hierarchy Editor.

Using the COM Hierarchy Editor to Define your COM Server

The COM Hierarchy Editor lets you interactively define the attributes (for example, classes, interfaces, and methods) for your COM server. The user interface contains two panes:

- The top pane is a tree control that displays attribute information in a hierarchy.
- The bottom pane is a panel that displays the IDL definition for a selected element in the hierarchy.

You can use "right-click" context menus to insert, delete, and modify items in the hierarchy. Context menus also let you change item properties. For example, you can right-click the `IAdd` element and select **Properties** to see the `IAdd` interface properties. You could then change the `Is Dual` property value to **False**; this indicates that the `Adder` example only supports a COM server interface.

An explanation of the hierarchy follows:

- The root is the COM server itself.

- The immediate children of the COM server are classes. The hierarchy initially contains a single class, `AddingMachine`. You can add additional classes to the COM server.
- The immediate children of a class are interfaces and the class derived-type. In the example, the hierarchy contains the interface `IAdd` and the class derived-type `AddingMachine_InstanceData`. Each class contains one, and only one, class derived-type. You can add additional interfaces to the class.
- The immediate children of an interface are methods (one or more).
- The immediate children of a method are the method arguments (one or more).

To add a class, interface, method, property, or argument:

1. Select the parent item, right-click and select the appropriate **Add...** item.

To delete an item:

1. Select the item to be deleted, right-click and select **Delete**.
2. Click the **Delete** button.

Continuing with the Adder COM server example, add methods to the `IAdd` interface:

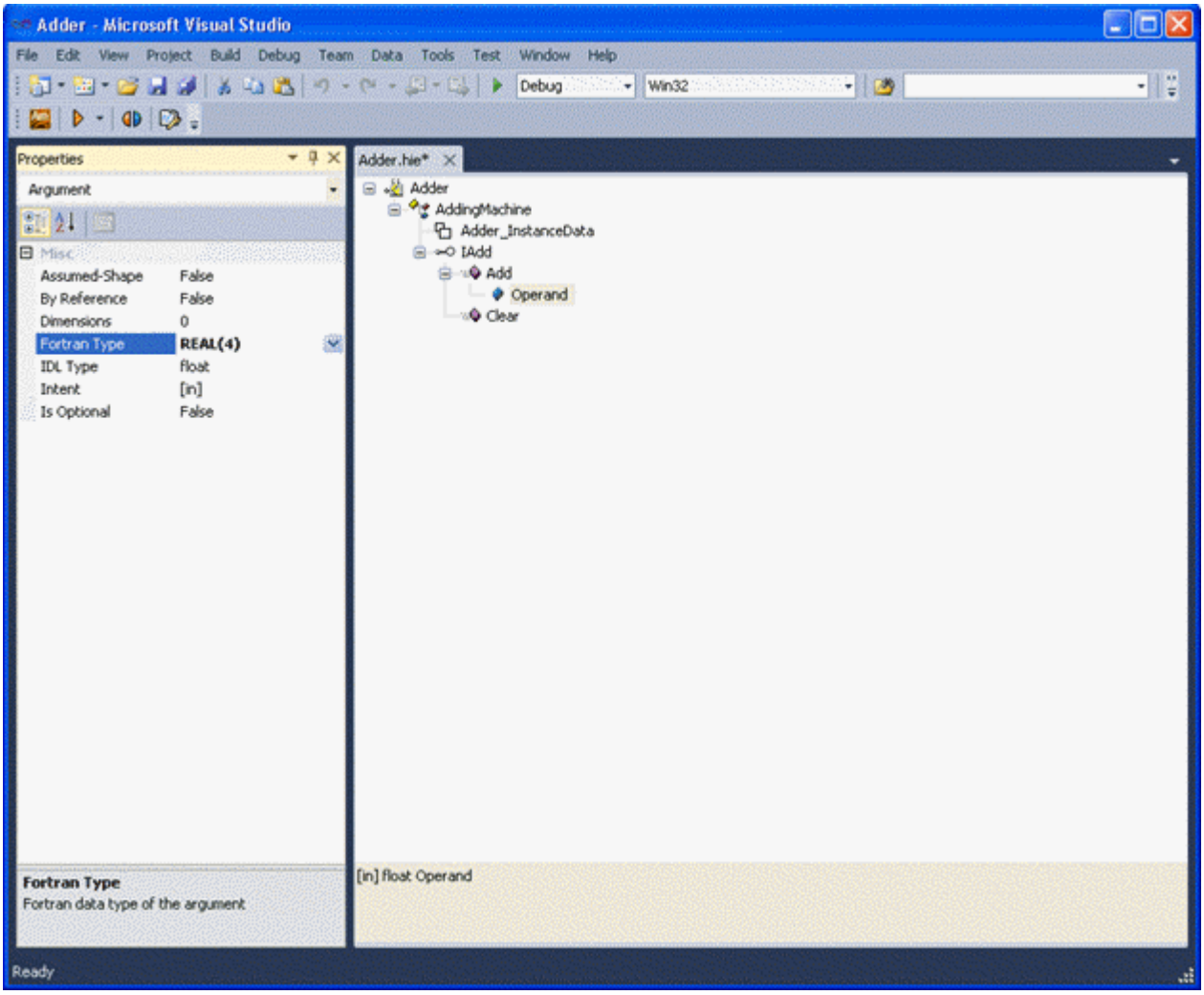
1. Right-click the `IAdd` interface and select **Add Method**.
2. Enter `Clear` as the name of the method.
3. Repeat steps 1 and 2, this time entering the name of `Add`.

The Add method requires an argument. To add the Operand argument named `Operand`:

1. Right-click on the **Add** method and select **Add Argument**.
2. Enter the argument name, `Operand`.

The default data type of an argument is INTEGER(4).

Proceeding with this example, use the Properties Window to change the Fortran Type property value to `REAL(4)`:



To add a third (and in this example, the last) method:

1. Right-click on the `IAdd` interface and select **Add Method**.
2. Enter `GetValue` as the name for the method.

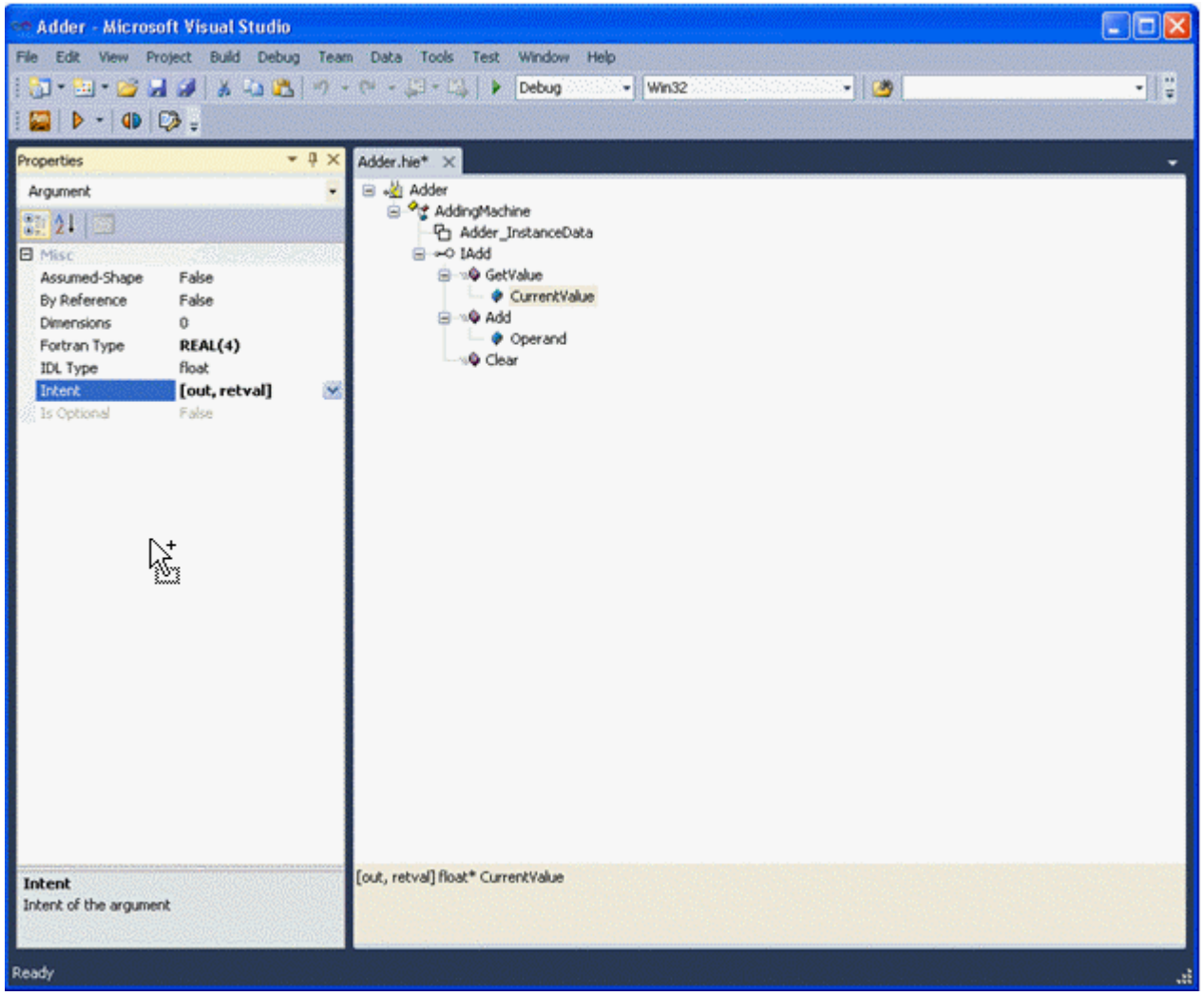
The GetValue method requires an argument, `CurrentValue`. To add the `CurrentValue` argument:

1. Right-click on the `GetValue` method and select **Add Argument**.
2. Enter `CurrentValue` as the name of the argument.

For the `CurrentValue` argument:

1. Set **Fortran Type** property value to `REAL(4)`.
2. Set **Intent** property values to `[out,retval]`.

The screen appears as follows:



The server definition is now complete for this example. When you save the file using **File > Save adder.hie**, the Fortran sources will be regenerated.

For detailed information about the property pages (right pane) for the server, class, interface, method, argument, or instance properties, see Description of Property Pages.

To change the definition of your server, open `adder.hie` in the Editor.

Working with the Hierarchy Pane

As you have seen in the `AddingMachine` example, the hierarchy pane represents the definition of your server; that is, the classes, interfaces, methods, and so on. The hierarchy pane's user interface supports the following functionality:

Expand/ contract an area	Click the plus sign (+) next to an item to display its children and the - to hide the children.
Add a new entry	Select the entry that will precede the new entry in the hierarchy, right-click and select one of the **New...** menu options.
Delete an entry	Select the item to be deleted, right-click and select **Delete**. All of an item's children are deleted when the item is deleted.

Rename a member	Select the item to be renamed, right-click and select **Rename**.
Change the order of items	The hierarchy pane supports drag-and-drop to allow you to change the order of items. The order of some of the entries in the hierarchy is very important. In particular: • The order of the methods in an interface defines the order of the methods in the class' `VTBL`. Changing the order of the methods will break an existing client. • The order of the arguments in a method define the order of the arguments in the method's interface. Changing the order of the arguments will break an existing client.

Description of Property Pages

Property pages appear in the **Properties** Window. Property pages are available for the following:

- Server Properties
- Class Properties
- Interface Properties
- Method Properties
- Argument Properties
- Instance Type Properties

Server Properties:

Type Library GUID	The unique identifier of the server's type library. There is usually no reason to change this from the default value generated by the Fortran COM Server Wizard.
Type Library Version	The current version of the Type Library.
HelpString	A string used to set the library's help string attribute in the IDL file.

Class Properties:

ProgID	The version independent program ID (or text alias) for the class. The ProgID can be used in calls such as COMCreateObjectByProgID.
Version	The current version of the class. It is appended to the ProgID to define the version-specific ProgID.
Short name	A short name for the class. It is used in some of the generated file names.
Description	A string used as the default value of the class' ProgID keys in the registry. This string is often used by tools, such as the OLE-COM Object Viewer, that display a list of the objects that are registered on the system.
Help String	A string used to set the class' help string attribute in the IDL file.
Threading model	The threading model of the class. The two choices are Apartment and Single. See Threading Models in Advanced COM Server Topics for information about the implications of this choice.
CLSID	The unique identifier of the class. There is usually no reason to change this from the default value generated by the Wizard.

Interface Properties:

Is Dual	If true, then the dual interface attribute is set in the IDL file. A dual interface supports both COM and Automation clients.

Is OleAutomation	If true, then the interface uses only Automation-compatible data types as described in Fortran COM Server Interface Design Considerations.
Is Default	If true, then the default interface attribute is set for this interface in the IDL file. The default attribute represents the default programmability interface of the object, and is intended for use by macro languages.
Help String	A string used to set the interface's help string attribute in the IDL file.
IID	The unique identifier of the interface. There is usually no reason to change this from the default value generated by the Wizard.

Method Properties:

DISPID	The identifier of the method used by Automation clients.
Help string	A string used to set the method's help string attribute in the IDL file.
Property Method	If checked, then the method is the get_ or put_ method of a property.

Argument Properties:

Fortran data type	The Fortran data type of the argument. Select one of the data types from the list, or type in the data type. See Fortran COM Server Interface Design Considerations for a discussion of the implications of your choice.
IDL type	The IDL data type. If you select one of the Fortran data types from the predefined list, then this field defaults to the corresponding IDL data type. Select one of the data types from the list, or type in the data type. See Fortran COM Server Interface Design Considerations for a discussion of the implications of your choice
Intent	The INTENT of the argument, one of the following list: • [in] - the argument value is read but not modified by the method • [out] - the argument value is not read, but is modified by the method • [in, out] - the argument value is both read and modified by the method • [out, retval] - the argument represents the return value of the method
By Reference	Indicates that an argument is passed by reference rather than by value. Only valid with Intent In. Intent Out and Intent InOut are automatically passed by reference.
Is Optional	If true, then the argument is optional. Optional arguments are passed using the Variant data type.
Dimensions	Number of array dimensions (0 if the argument is scalar).

Instance Type Properties:

Module name	The name used for the module defined in the U*classname*TY.f90 file.
Constructor name	The name used for the class constructor defined in the U*classname*TY.f90 file.
Destructor name	The name used for the class destructor defined in the U*classname*TY.f90 file.

Modifying the Generated Code

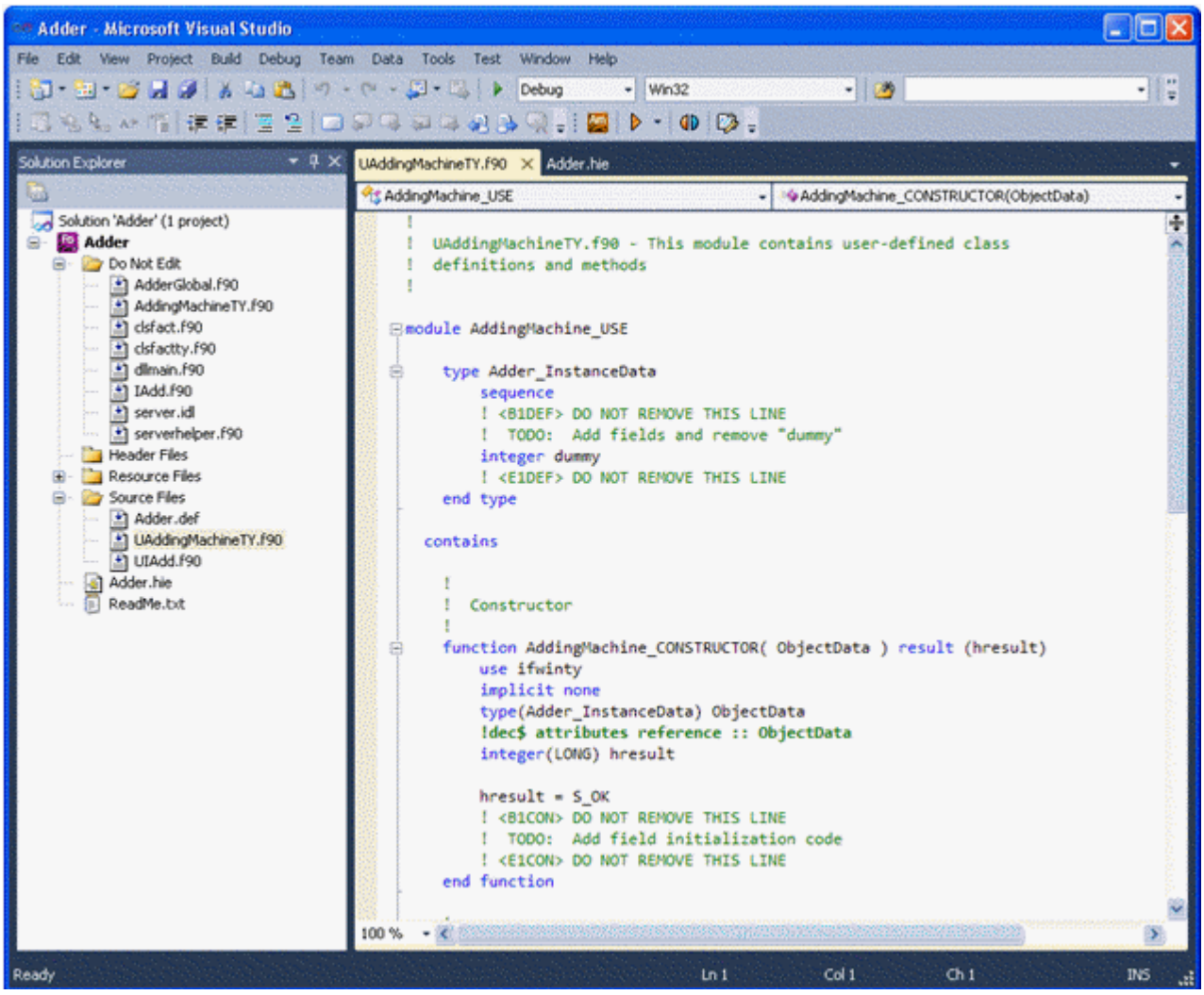
When you save the hierarchy file, Fortran sources are generated (or regenerated). Source files are stored as follows:

- Source files that you will need to modify are placed in the **Source Files** folder.
- Files that are complete and need no modification are placed in the **Do Not Edit** folder.

Modify the following files in the Source Files folder:

- `UAddingMachineTY.f90`, which contains a module that defines your class derived-type.
- `UIAdd.f90`, which contains the implementation of the `IAdd` methods.

Initially, the file `UAddingMachineTY.f90` contains the following code:



The file contains a module named `AddingMachine_USE` (in the form *classname*_USE). There are three places in this module where you may need to add code specific to your class:

- The first entry in the module, marked by `<BnDEF>` and `<EnDEF>`, is your class derived-type. Initially it contains an "integer dummy" field to allow the module to be compiled without error. If your class has per-object data, remove the "integer dummy" field line and add your data to the derived-type. For the AddingMachine class, we add the following where the object stores the current value:

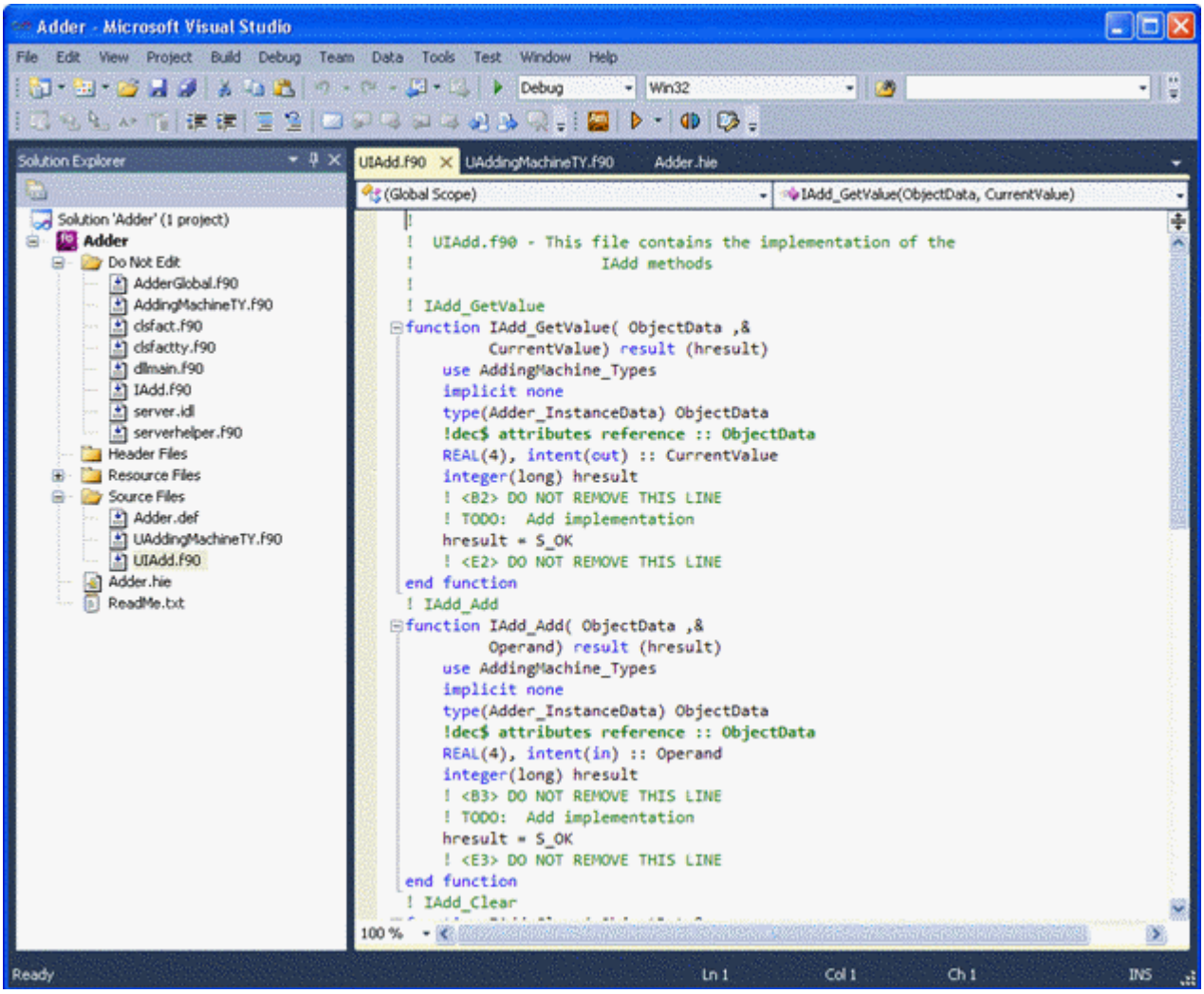
```
real (4) CurrentValue
```

- The module also contains two module procedures, `AddingMachine_CONSTRUCTOR` and `AddingMachine_DESTRUCTOR` (referred to as *classname*_CONSTRUCTOR and *classname*_DESTRUCTOR below). The former delineated with `<BnCON>` and `<EnCON>`; the latter is delineated with `<BnDES>` and `<EnDES>`.
- The *classname*_CONSTRUCTOR procedure is called immediately after an instance of the class derived-type is created because of the creation of a new object. This function is where you initialize the fields of the class derived-type, if necessary. The new derived-type is passed as an argument to the function. For the `AddingMachine` class, initialize the current value to 0 by adding the following statement:

```
ObjectData%CurrentValue = 0
```

- The *classname*_DESTRUCTOR procedure is called immediately before an instance of the class derived-type is destroyed because an object is being destroyed. This function is where you release any resources used by the fields of the class derived-type, if necessary. The derived-type is passed as an argument to the function. For the `AddingMachine` class, there is nothing that needs to be added.

Now modify the other source file called `UIAdd.f90`. The original file `UIAdd.f90` contains the following code:



A file by the name U*interfacename*.f90 (for example `UIadd.f90`) is created for each interface defined by the class. The file contains the methods of the class. Each method is named *interfacename_methodname*, for example: `IAdd_Clear`. Each method is a function that is passed to the class derived-type as the first

argument. This gives the function access to the per-object data. Each function returns a 32-bit COM status code called an HRESULT. S_OK is a parameter that defines a success status. For additional information on COM status codes, see Fortran COM Server Interface Design Considerations.

Replace the `! TODO: Add implementation` line in each method with the code for the method. For the `IAdd` interface, the implementation of its three methods is as follows:

For IAdd_Clear: `ObjectData%CurrentValue = 0`

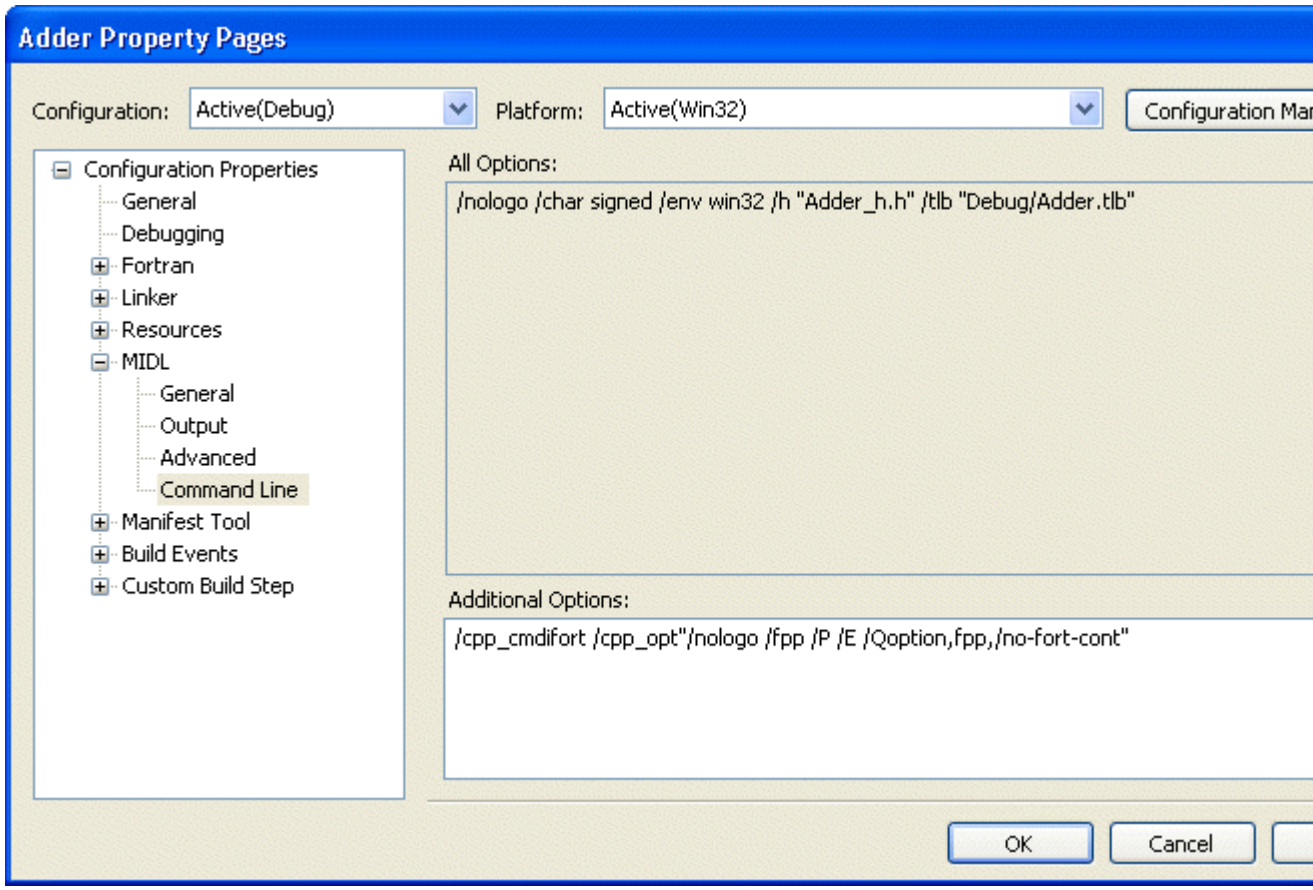
For IAdd_Add: `ObjectData%CurrentValue = ObjectData%CurrentValue + Operand`

For IAdd_GetValue: `CurrentValue = ObjectData%CurrentValue`

Save the file and, from the **Build** menu, click **Build Solution** to build the server. The COM server is now complete.

COM Server Projects and the MIDL Compiler

During the creation of a COM Server project -- either In-process Server (DLL) or Out-of-process Server (EXE), as described in this section -- the application wizard determines whether Visual C++* is installed. If it is installed, the MIDL compiler uses the Visual C++ C preprocessor. This is both the default and recommended setting. If Visual C++ is not installed, the application wizard specifies that the MIDL compiler use the Fortran preprocessor for preprocessing MIDL source files. It does this by setting the **Additional Options** property of the MIDL compiler to include the following: `/cpp_cmdifort /cpp_opt"/nologo /fpp /P /E /Qoption,fpp,/no-fort-cont"`.



If, later, you install Visual C++ and want to specify that MIDL use the Visual C++ C preprocessor, remove these options from the MIDL compiler **Additional Options** property.

Fortran COM Server Interface Design Considerations (Windows*)

This topic only applies to Windows* operating systems.

This section provides information that should be considered when designing a Fortran COM server. It contains the following topics:

- Method and Property Data Types
- COM Status Codes: HRESULT
- Visual Basic*, Visual C++*, and Intel® C++ Client Notes

Method and Property Data Types

COM places some restrictions on the data types used in COM methods and properties. The reason for the restrictions is that COM can pass arguments between threads, processes and machines. This raises issues that are not present in older technologies, such as DLLs, that always run in the same address space as the caller.

COM defines set of data types called Automation-compatible data types. These are the only data types that can be used in Automation and Dual interfaces. There are two advantages to restricting your COM interface to these data types:

- Your server will be usable from clients written in the largest set of languages, including Intel® Fortran, Visual Basic, and Visual C++.
- COM will automatically handle the passing of arguments between threads, processes and machines. This is called *Type Library Marshalling*. For more information, see Marshalling, Proxies and Stubs in Advanced COM Server Topics.

To restrict your server to Automation-compatible data types:

1. Select Use only Automation data types on the Interface property page. When defining a dual interface, this is automatically set.
2. Use only the following combinations of Fortran data type and Interface data type on the Argument property page.

   Note that Intel Fortran does not support the Currency, Decimal, or User Defined Type, Automation-compatible data types.

Fortran Date Type	Interface Data Type
INTEGER(1)	unsigned char
INTEGER(2)	short
INTEGER(4)	long
	SCODE
	Int
INTEGER(INT_PTR_KIND())	IUnknown*
	IDispatch*
REAL(4)	float
REAL(8)	double
	DATE
LOGICAL(2)	VARIANT_BOOL
LOGICAL(4)	long

Fortran Date Type	Interface Data Type
CHARACTER(1)	unsigned char
CHARACTER(*)	BSTR
BYTE	unsigned char
TYPE(VARIANT)	VARIANT (containing one of the above types or SafeArray)

If you decide not to restrict your interface to Automation-compatible data types, the next approach is to restrict your interface to data types that can be described in the Interface Description Language (IDL).

The Fortran COM Server Application Wizard automatically generates the IDL file from the description of your server. The MIDL compiler compiles the IDL file into a type library. MIDL can also automatically generate the code needed handle the passing of arguments between threads, processes and machines. Note, however, that a C compiler is required to use this option. For more information, see Discussion of Wizard Code Generation in Advanced COM Server Topics.

If you decide not to restrict your interface to IDL data types, your only remaining options are:

- Implement Custom Marshalling - For more information, see Marshalling, Proxies and Stubs in Advanced COM Server Topics.
- Decide that you will never use your server across apartment, thread, or process boundaries. The server will not be usable in this manner because there is no way to pass the arguments across these boundaries.

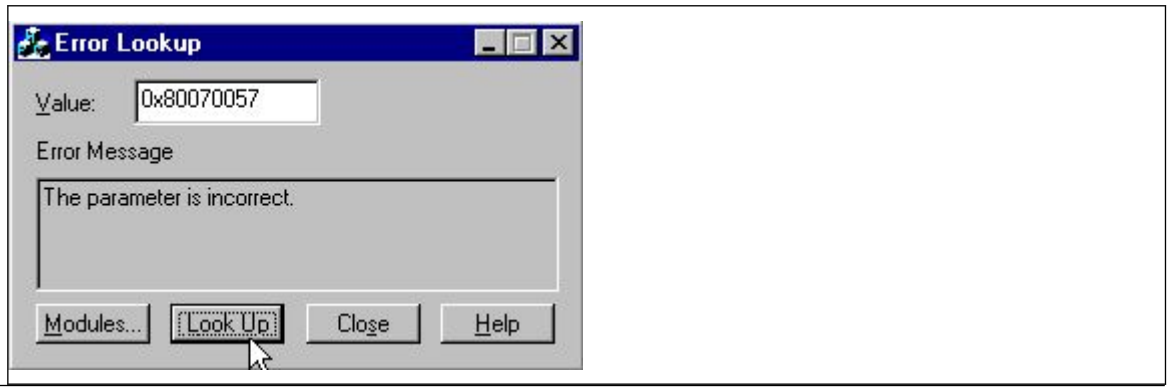
COM Status Codes: HRESULT

Each function returns a 32-bit COM status code called an HRESULT. An HRESULT is divided into fields:

- The top bit (31) indicates whether the function succeeded or failed. The bit is set if the function failed. In Fortran, you can compare the HRESULT to < 0 as a test to see if a function failed.
- Bits 16 to 27 contain a facility code to indicate the facility that issued the HRESULT. Microsoft pre-defines a number of facility codes for its own use. If you create your own status codes, use FACILITY_ITF. All other facility codes are reserved by Microsoft.
- The first 16 bits, or low word, contain the error code specific to the error that occurred.

A typical HRESULT error value could be a value such as 0x80070057. The first hex digit, 8, indicates that bit 31 is set and that this is an error value. Bits 16 to 27 contain the value 7. This indicates the facility FACILITY_WIN32. The low word contains the value 0057. This is the specific code that identifies the error as E_INVALIDARG.

To view the text description that corresponds to a system HRESULT value, use the Error Lookup tool in the Intel® Fortran program folder. For example, entering the value 0x80070057 retrieves the text message "The parameter is incorrect ", as shown below:



You can also search for HRESULT values in the `WINERROR.H` file in the `\VC\INCLUDE` directory.

Visual Basic*, Visual C++*, and Intel® C++ Client Notes

To use an object from Visual Basic, you must add a "reference" to the object to the Visual Basic project. Use the References item in the Project menu to display a list of the registered objects. Select the object in the list to inform Visual Basic that you will be using the object.

Here are some points to be aware of when writing a server that can be used with Visual Basic clients:

- Use only the Automation-compatible data types (see Method and Property Data Types).
- Arguments to a method can be passed ByVal, ByRef, or they can be the function return value.
- An argument to be passed ByVal, must be defined with Intent set to In.
- An argument to be passed ByRef, must be defined with Intent set to InOut.
- A function return value must be defined with Intent set to Out. It must have the Return Value field checked. The argument must be the final argument of the method.
- An array is always passed as a SafeArray ByRef. Therefore it must be defined with Intent set to InOut.
- To use an argument of the Visual Basic Boolean data type, set the Fortran data type to LOGICAL(2) and set the Interface data type to VARIANT_BOOL.

To use an object from C++, use the #import directive. The syntax of the #import directive is:

```
#import "filename" [attributes]
#import <filename> [attributes]
```

The filename is the name of the file containing the type library information. The directive makes the information in the type library available to your source file as a set of C++ classes.

Advanced COM Server Topics (Windows*)

This topic only applies to Windows* operating systems.

Advanced topics about Fortran COM Servers described in this section include:

- Choosing Between DLL or EXE COM Servers
- DLL Server Surrogates
- Discussion of Wizard Code Generation
- Threading Models
- Marshalling, Proxies and Stubs
- A Map of the Generated "Do Not Edit" Code

Choosing Between DLL or EXE COM Servers

The basic tradeoff in choosing between a DLL (in-process) COM server and an EXE (out-of-process) COM server is one of performance vs. robustness:

- A DLL server provides the advantage of performance over an EXE server. Since the DLL server is loaded into the client's address space, there is less overhead involved in method calls. If the client code and the server object live in the same COM apartment, method calls are as efficient as DLL routine calls.
- An EXE server provides the advantage of robustness over a DLL server. Since the server object lives in a separate address space, the object cannot be affected by client memory handling bugs, and vice-versa. If the server crashes, the client doesn't necessarily crash – as long as the client checks the results of all method calls and takes steps to recover from a "dead" object.

In addition to the tradeoff between performance and robustness, the following factors should also be considered:

- With an EXE server, the object can run in a separate security context from the client. With a DLL server, the code of the object's methods executes using the client's access token.
- An EXE server can be run on a remote machine using COM distributed object support.

You can load a DLL server into a surrogate to gain the benefits of an EXE server. This is explained in the next section.

DLL Server Surrogates

An in-process DLL server can be run in a separate process with the help of a *surrogate*. A surrogate runs as a separate process, loads the DLL server, and provides all of the mechanism that allows the DLL server to act as a local server. Windows provides a standard surrogate named DLLHOST.EXE. The primary advantage of using a surrogate is fault isolation. That is, if the server crashes it does not crash the client, and vice versa. The disadvantage is performance. There is significant performance overhead in executing methods in a separate application, as opposed to in a DLL.

A DLL server is associated with a surrogate via entries in the system registry. This is done by associating the DLL server with an AppID. An AppID is a GUID. When using the standard surrogate, you can use the CLSID of a class in the DLL as the AppID, or a newly generated GUID. To generate a new GUID, use GUIDGEN.EXE in the Developer Studio Tools subdirectory:

1. Under the HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx} key of the class, add an AppID entry with the value of the CLSID. Using the AddingMachine class as an example, the registry key would be HKEY_CLASSES_ROOT\CLSID\{904245FC-DD6D-11D3-9835-0000F875E193} and the AppID value would be {904245FC-DD6D-11D3-9835-0000F875E193}.
2. Under the HKEY_CLASSES_ROOT\AppID key, add a key using the AppID. Using the AddingMachine class as an example, the registry key would be HKEY_CLASSES_ROOT\AppID\{904245FC-DD6D-11D3-9835-0000F875E193}. Use the class name for the default value of the key, for example, "AddingMachine Class". Add a "DllSurrogate" entry with an empty string for the value.

The client must request CLSCTX_LOCAL_SERVER rather than CLSCTX_INPROC_SERVER to use the surrogate rather than loading the DLL server in-process. Using a surrogate requires that the DLL server have a proxy/stub registered since the method invocations are between different processes. For information on proxies/stubs, see Marshalling, Proxies and Stubs.

It is also possible to write a custom surrogate. See the Windows Platform SDK documentation for information.

Discussion of Wizard Code Generation

The COM Server Application Wizard generates the code for your project from the files in the subdirectory of your project named `templates`. The *project-name*.hie file contains the definition of your COM server in an undocumented text language. You should not manually edit the *project-name*.hie file; the Wizard does this for you.

Most of the other files in the `templates` directory are templates of the source files generated for your project. These templates contain source code that is copied "as-is" to the generated sources, and embedded directives that guide the Wizard in generating the code specific to the COM server that you define. The directives use the information in the *project-name*.hie file. The directives are undocumented and subject to change.

When you create a new Fortran COM Server project, the AppWizard creates the `templates` directory and copies the templates from the Fortran COM Server templates directory, `...\Intel Fortran\Templates\COMServer`.

The files in the templates directory may change with each release, but the templates in your `templates` directory are never automatically updated. For example, if you create a COM server using Intel® Fortran Version 13.0 and the next release of Intel® Fortran (such as Version 13.1) contains updated templates, the templates for your COM server are not automatically updated to the new 13.1 templates. If you modify the definition of your server, your project continues to use the templates that it was created with. This has the advantage of not introducing different code into a project that you have developed and tested.

However, there are two cases where you may want to modify the templates that are used by your project:

- A new release of Intel® Fortran may contain additional features that can be used in your COM server project. Some of the new features may depend upon the new templates. These features will not be available to a pre-existing project unless you update the templates in the `templates` directory. You do

this by copying all of the files in the Fortran COM Server templates directory `...\Intel Fortran \Templates\COMServer` into the `templates` directory, replacing all files with the same name. Your project will then use the new templates the next time that the definition of your server is modified and the project sources are regenerated.

- When you want to modify the code generated by the Wizard. You may edit a template in your `templates` directory. Editing code that is copied "as-is" is straightforward. Attempting to modify the embedded directives is unsupported and could cause the Wizard to fail when using the modified template. The embedded directives begin with an @ (at-sign) character. The next character determines the type of directive. The end of many directives is also delimited by an @ character followed by the matching end character for the type of directive. For example, @[and @] are the delimiters for one type of directive.

The advantage of modifying a template is that you can customize the code generated by the Wizard. The disadvantages of modifying a template are:

- You must be very careful not to modify a template in a manner that causes the Wizard to generate bad code or fail.
- You will not be able to update the project templates with the templates from a new release of Intel® Fortran without having to re-apply your modifications to the new templates.

Threading Models

The Wizard supports two COM threading models for the classes that you create in a DLL COM server, Apartment and Single. The Wizard uses the Apartment threading model (also known as the Single Threaded Apartment model "STA") by default. The basic rules of the Apartment threading model are:

- If two objects, A and B, are created in the same STA thread, and A is processing a method call, no other client can call either A or B until A completes.
- If B is created in a different thread from A, B can accept a method call while A is still processing and vice versa.

This means that if the class shares any global data among its objects, the global data must be protected from simultaneous access using thread synchronization primitives. This is because two instances of the class could be running in different threads. However the per-object instance data, that is, the fields in the class derived-type, are protected from simultaneous access by COM mechanisms. This is true except in the case where an object calls out to another object that triggers a reentrant invocation of the first object.

A class using the Single threading model need not worry about simultaneous access to class global data, as well as per-object data. All objects of the class are created in a single thread and therefore only a single object of the class can be executing at any time.

An EXE COM server generated by the wizard is single threaded. All method invocations are serialized by the server's message queue. Therefore, with an EXE server, you need not worry about simultaneous access to class global data, as well as per-object data.

Explaining the COM threading models in detail is beyond the scope of this documentation. See the Windows Platform SDK documentation for additional information.

Marshalling, Proxies and Stubs

COM supports the use of objects in separate processes (as when using an EXE server or a DLL surrogate), and the threading models described above, by the use of marshalling, proxies and stubs. This section presents an overview of marshalling, proxies and stubs.

Marshalling is the process of reading the parameters for a method call and preparing them for transmission to another execution context (for example, thread, process, or machine). Marshalling is done by a *proxy*. From the client's perspective, a proxy has the same interface as the object itself. The proxy's job is to make the object look like an object in the same execution context as the client.

Proxies allow client code to be unconcerned about where the object actually lives. A proxy marshalls method parameters and transmits them to a *stub* associated with the object in the server. The stub unmarshalls the parameters and invokes the method in the server. From the server's perspective, this is no different than when it is called from a client in the same execution context. A server that is not an in-process DLL server always requires a proxy/stub pair. An in-process DLL server requires a proxy/stub pair when the client and object are in different apartments.

There are three ways to assign a proxy/stub pair to a server:

Type Library Marshalling:	If you use only Automation-compatible data types in your methods and properties, COM can automatically use the "Universal Marshaller" as the proxy/stub for the server. The Universal Marshaller uses the description of the server in the type library to decide how to marshall the parameters. This is known as type library marshalling. Using type library marshalling requires no effort on your part, other than restricting your server to Automation-compatible data types.
MIDL-based Marshalling:	Your project uses the MIDL compiler to compile the IDL description of the server into a type library. At the same time, the MIDL compiler also generates the C source code necessary to build a proxy/stub DLL for the server. You must have a C compiler to build a proxy/stub DLL from the MIDL generated code. The proxy/stub DLL is itself an in-process DLL server and needs to be registered with the system. You would need to use MIDL-based marshalling if your server uses non-Automation-compatible data types, and is used by a client in a different execution context.
Custom Marshalling:	Your server can implement its own marshalling by implementing the IMarshall interface. This approach typically involves a lot of work and is done for performance reasons.

A Map of the Generated "Do Not Edit" Code

This section presents an overview of which parts of the COM server functionality are implemented as "Do Not Edit" source files generated by the application wizard:

File Name	Description
server.idl	Contains the IDL description of the server. It is compiled by the MIDL compiler to produce the server's type library.
*servername*global.f90	Contains the global data and functions for the server.
dllmain.f90 (DLL server)	Contains the exported functions that are required of all COM Server DLLs. This includes DllMain, DLLRegisterServer, DLLUnregisterServer, DllGetClassObject, and DllCanUnloadNow.
exemain.f90 (EXE server)	Contains the main entry point of an EXE server. It also processes the command-line argument.
serverhelper.f90	Contains helper functions for the server.
clsfactty.f90	Contains definitions of the IClassFactory interface that is used to create instances of the classes defined by the server.
clsfact.f90	Contains methods of the IClassFactory interface that is used to create instances of the classes defined by the server.

File Name	Description
*classname*TY.f90	Defines a module that contains definitions of parameters and types used in the implementation of the class. It also contains the implementation of the IUnknown methods of the class. A separate instance of this file is generated for each class defined in the server.
interfacename.f90	Defines a module that contains the Fortran interfaces of the methods in the interface. It also contains the implementation of the Fortran "wrappers" that are called directly from the class' VTBL and call the methods implemented by the user. A separate instance of this file is generated for each interface defined in the class.

Deploying the COM Server on Another System (Windows*)

This topic only applies to Windows* operating systems.

When you have finished developing the COM server, you may want to deploy it on another system. Besides the server itself, you need to install the Fortran run-time DLLs and register the server:

1. Install (or copy) the COM server to an appropriate directory.
2. Register the server:
   - To register a DLL server, use the REGSVR32.EXE system tool, a command-line tool. To register the DLL server, specify the path to the DLL and its file name (*dll_path*) by typing the following command:

```
REGSVR32 dll_path
```

   - The file REGSVR32.EXE is in your system directory, such as the `\Winnt\System32` on Windows NT* or Windows 2000 systems. If this directory is not in your PATH, include its path before the REGSVR32 command.
   - To register a an EXE server, run the server with the /REGSERVER command-line opton. For example:

```
exe_path /REGSERVER
```

If you are using MIDL-based Marshalling or Custom Marshalling, then the marshalling DLL also needs to be installed and registered. For information on marshalling, see Advanced COM Server Topics.

Using the Intel® Fortran Module Wizard (COM Client) (Windows*)

This topic only applies to Windows* operating systems.

Intel® Fortran provides a wizard to simplify the use of Component Object Model (COM) and Automation (formerly called OLE Automation) objects. The Intel® Fortran Module Wizard generates Standard Fortran modules that simplify calling COM and Automation services from Fortran programs. This Fortran code lets you invoke methods of an Automation object and member functions of a Component Object Model (COM) object.

The following sections describe the use of COM and Automation objects as clients with Intel Fortran.

Understanding COM and Automation Objects (Windows*)

This topic only applies to Windows* operating systems.

This section provides a brief overview of:

- COM Objects
- Automation Objects
- Object Servers

COM Objects

The Component Object Model (COM) provides mechanisms for creating reusable software components. COM is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or *components* to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems.

With COM, components interact with each other and with the system through collections of function calls, also known as methods or member functions or requests, called *interfaces*. An interface is a semantically related set of member functions. The interface as a whole represents a feature of an object. The member functions of an interface represent the operations that make up the feature. In general, an object can support multiple interfaces and you can use COMQueryInterface (W*32 W*64) to get a pointer to any of them.

The Intel Fortran COM routines provide a Fortran interface to basic COM functions.

Automation Objects

The capabilities of an *Automation object*resemble those of a COM object. An Automation object is in fact a COM object that implements the interface IDispatch. An Automation object exposes:

- *Methods*, which are functions that perform an action on an object. These are very similar to the member functions of COM objects.
- *Properties*, which hold information about the state of an object. A property can be represented by a pair of methods; one for getting the property's current value, and one for setting the property's value.

The Intel® Visual Fortran AUTO routines provide a Fortran interface to invoking an automation object's methods and setting and getting its properties.

Object Servers

COM and Automation objects are made available to users by COM and Automation *servers*. A COM or Automation server can be implemented either as:

- A DLL that is loaded into your process
- A separate executable program. The separate executable program can reside on the same system as your application or on a different system.

The Role of the Module Wizard (Windows*)

This topic only applies to Windows* operating systems.

To use COM and Automation objects from a Fortran program:

1. Find or install the object server on the system. COM and Automation objects can be registered:

   - By other programs you install.
   - By creating the object server yourself, for example, by using Visual C++*, or Visual Basic*.

   For example, the Microsoft visual development environment registers certain objects during installation (see the documentation on the Common Environment Object Model).

   Creating an object server involves deciding what type of object and what type of interfaces or methods should be available. The object's server must be designed, coded, and tested like any other application. For information about object server creation, see Creating the Fortran COM Server.
2. Determine:

   - Whether the object has a COM interface, Automation interface, or both.
   - Where the object's type information is located.

   You should be able to obtain this information from the object's documentation. You can use the OLE/COM Object Viewer tool from the development environment Tools menu to determine the characteristics of an object on your system.
3. Use the Intel® Fortran Module Wizard to generate code.

   The module wizard is an application that allows you to select a COM or Automation object and set generated code options. The information collected by the module wizard is used in the generated code. To learn about using the Intel® Fortran module wizard, see Using the Module Wizard to Generate Code.
4. Write a Fortran program to invoke the code generated by the Intel® Fortran module wizard.

To understand more about calling the interfaces and jacket routines created by the module wizard, see Calling the Routines Generated by the Module Wizard.

Using the Module Wizard to Generate Code (Windows*)

This topic only applies to Windows* operating systems.

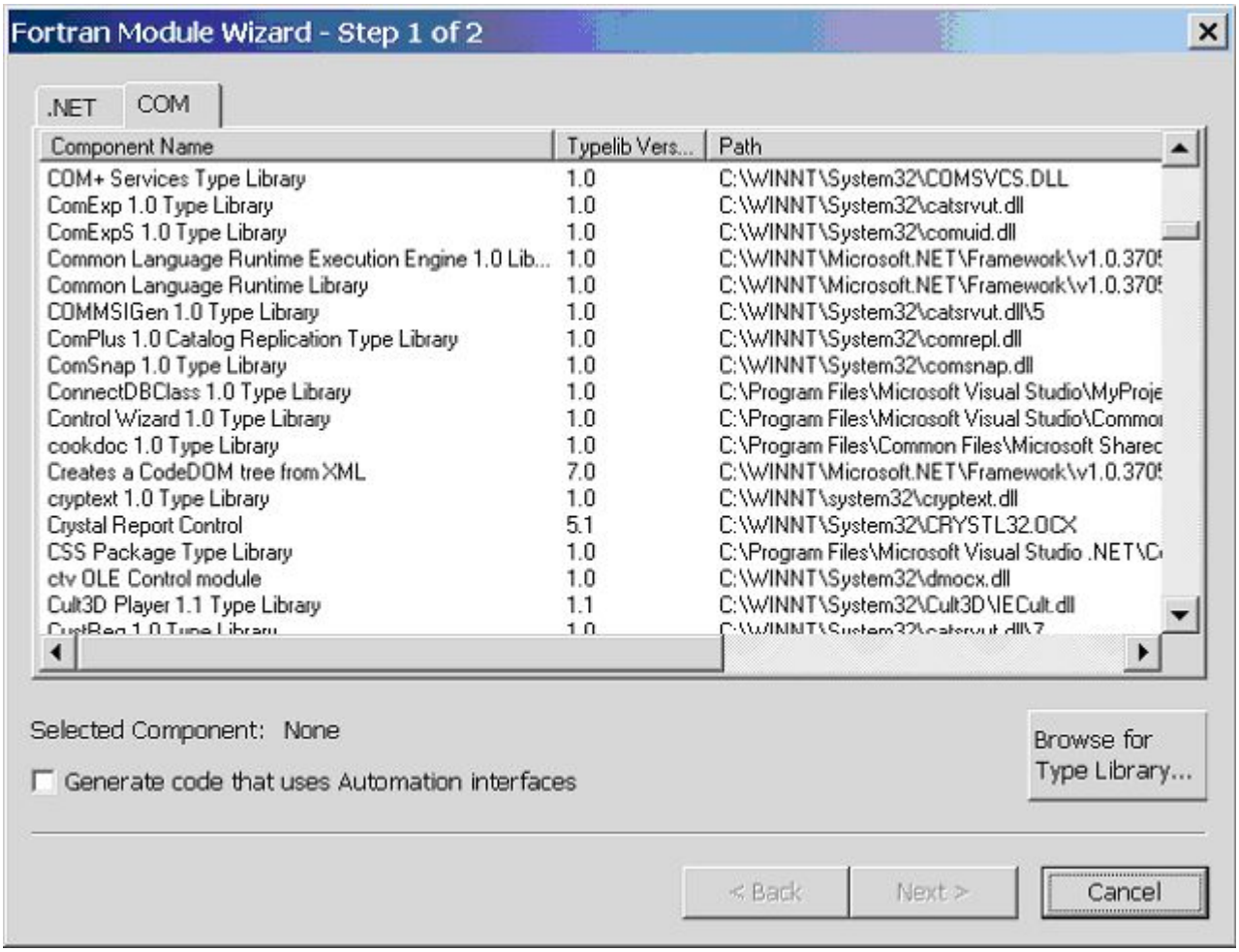
To run the Intel® Fortran Module Wizard, select **Tools > Intel Compiler > Intel® Fortran Module Wizard**.

The module wizard displays a series of dialog boxes allowing you to select the COM or Automation object and provide additional information.

COM and automation objects are self-describing, using type information. An object's type information contains programming language independent descriptions of the object's interfaces. Type information can be obtained from the object's type library.

A type library is a collection of type information for any number of object classes, interfaces, and so on. You can store a type library in a file of its own (usually with an extension of `.TLB`) or it can be part of another file. For example, the type library that describes a DLL can be stored in the DLL itself.

After you start the module wizard, the COM tab of the dialog box displays the list of type libraries that are registered on your system. (There is also a .NET tab, which is explained in Using .NET Components.)



The list has three columns:

- The component name
- The component version number
- The path to the type library

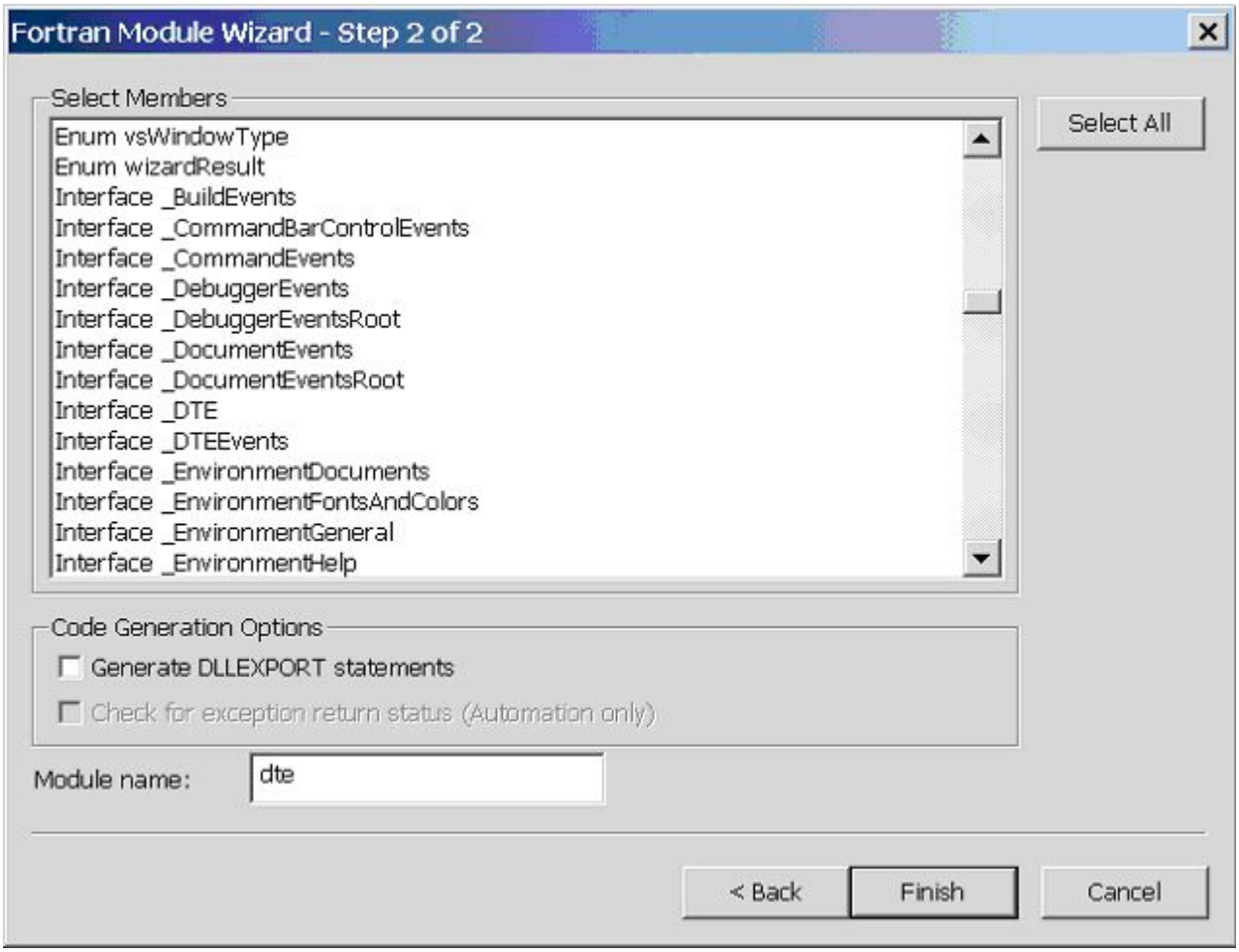
You may select a single component from the list, or click the **Browse for Type Library...** button to find a type library on your system.

The dialog box also provides the Generate code that uses **Automation interfaces** option. If the option is not selected, the Wizard will generate code to use the COM interfaces of the component.

The documentation of the component should tell you whether the component implements COM interfaces, Automation interfaces, or dual interfaces. A dual interface supports both COM and Automation interfaces. For a component that supports dual interfaces, you can check the Generate code that uses Automation interfaces option, or leave it unchecked. The COM interfaces tend to be more efficient (better run-time performance).

ActiveX controls implement an Automation interface. When using an ActiveX control, check the **Generate code** that uses Automation interfaces option.

After you select a component from the list or by browsing using the **Browse for Type Library...** button, click **Next>**.



This dialog box lists the members that are defined in the type library. Click the **Select All** button to select all of the members of the type library, or select individual members from the list. The Wizard uses the selected members.

This dialog box also contains two options regarding the code to be generated by the Wizard:

- Select **Generate DLLEXPORT statements** if you plan to place the generated module in a DLL for use by other projects. The Wizard will generate the DLLEXPORT statements to make the routines in the module visible outside of the DLL.
- **Check for exception return status (Automation only)** is active if you checked the Generate code that uses Automation interfaces option in the first dialog box. Select this option if you want the Wizard to generate code to check the return status of each Automation method or property invocation, and to display a dialog box when an error is generated.

Change the module name if desired and select **Finish**.

Using .NET Components

The .NET tab is available on the first dialog box. The module wizard cannot directly read the type information of a .NET component. However, the .NET Framework Tools (provided with Visual Studio*) contain the Type Library Exporter (`Tlbexp.exe`). The Type Library Exporter generates a COM type library that describes the types defined in a .NET component (also known as a common language runtime assembly).

Select the .NET tab to view the assemblies that are registered in your system's Global Assembly Cache (GAC). The list contains the same three columns as the COM tab, as well as a Browse for Assembly button. Select a .NET component from the list, or use the **Browse for Assembly** button to select a .NET component on your system.

Press **Next >** to run the Type Library Exporter tool to create a type library from the .NET component. If the tool returns an error message, the message is displayed in a dialog box and the Wizard returns to its first dialog box. In this case, no Fortran module is generated. If the tool succeeds, the second Wizard dialog box is displayed with the members found in the type library. There are restrictions on the .NET components that can be used with the Type Library Exporter tool. For example, a common error message returned by Type Library Exporter tool is:

```
You cannot use Tlbexp.exe to produce a type library from an assembly that was imported
using the Type Library Importer (Tlbimp.exe). Instead, you should refer to the original
type library that was imported with Tlbimp.exe.
```

In this case, the .NET assembly was created from the information in a type library, and you must obtain the original type library from the assembly provider in order to use it with the Module Wizard.

See the .NET Framework Tools documentation for details.

The Wizard runs the Type Library Exporter tool with the `/nologo/silent` options. You can also run the tool to create a type library before running the Module Wizard if you prefer.

Intel® Fortran Module Wizard Command Line Interface

The Intel® Fortran Module Wizard also has a command-line interface. The MODWIZ command has the following form:

```
MODWIZ [options] typeinfo-name
```

For a list of `MODWIZ` command options and an explanation of *typeinfo-name*, type the following command in a Fortran command prompt (available from the Intel® Fortran Compiler program folder):

```
MODWIZ /?
```

See Also

Using .NET Components

Calling the Routines Generated by the Module Wizard (Windows*)

This topic only applies to Windows* operating systems.

Although Standard Fortran does not support objects, it does provide Standard Fortran *modules*. A module is a set of declarations that are grouped together under a global name, and are made available to other program units by using the USE statement.

The Intel® Fortran Module Wizard generates a source file containing one or more modules. The types of information placed in the modules include:

- *Derived-type definitions* are Fortran equivalents of data structures that are found in the type information.
- *Constant definitions* are Fortran PARAMETER declarations that include identifiers and enumerations found in the type information.

- *Procedure interface definitions* are Fortran interface blocks that describe the procedures found in the type information.
- *Procedure definitions* are Fortran functions and subroutines that are jacket routines for the procedures found in the type information.

The jacket routines make the external procedures easier to call from Fortran by handling data conversion and low-level invocation details.

The use of modules allows the Intel® Fortran Module Wizard to encapsulate the data structures and procedures exposed by an object or DLL in a single place. You can then share these definitions in multiple Fortran programs.

The appropriate USE statement needs to be added in your program, as well as function invocations or subroutine calls.

The routines generated by the Intel® Fortran Module Wizard are designed to be called from Fortran. These routines in turn call the appropriate system routines (not designed to be called from Fortran), thereby simplifying the coding needed to use COM and Automation objects.

Intel® Visual Fortran provides a set of run-time routines that present to the Fortran programmer a higher level abstraction of the COM and Automation functionality. The Fortran interfaces that the wizard generates hide most of the differences between Automation objects and COM objects.

Depending on the options specified, the following routines can be present in the generated code:

IFCOM Routines (COM*xxxxx*)	
COMAddObjectReference	Adds a reference to an object's interface.
COMCLSIDFromProgID	Passes a programmatic identifier and returns the corresponding class identifier.
COMCLSIDFromString	Passes a class identifier string and returns the corresponding class identifier.
COMCreateObject	A generic routine that executes either COMCreateObjectByProgID or COMCreateObjectByGUID.
COMCreateObjectByGUID	Passes a class identifier and creates an instance of an object. It returns a pointer to the object's interface.
COMCreateObjectByProgID	Passes a programmatic identifier and creates an instance of an object. It returns a pointer to the object's IDispatch interface.
COMGetActiveObjectByGUID	Pass a class identifier and returns a pointer to the interface of a currently active object.
COMGetActiveObjectByProgID	Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.
COMInitialize	Initializes the COM library. You must initialize the library before calling any other COM or AUTO routine.
COMIsEqualGUID	Determines if two GUIDs are the same.
COMGetFileObject	Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
COMQueryInterface	Passes an interface identifier and it returns a pointer to an object's interface.
COMReleaseObject	Indicates that the program is done with a reference to an object's interface.
COMStringFromGUID	Passes a GUID and returns the corresponding string representation.

IFCOM Routines (COM*xxxxx*)	
COMUninitialize	Uninitializes the COM library. This must be the last COM routine that you call.

IFAUTO Automation Routines (AUTO*xxxxx*)	
AUTOAddArg (W*32 W*64)	Passes an argument name and value and adds the argument to the argument list data structure.
AUTOAllocateInvokeArgs (W*32 W*64)	Allocates an argument list data structure that holds the arguments that you will pass to AUTOInvoke.
AUTODeallocateInvokeArgs (W*32 W*64)	Deallocates an argument list data structure.
AUTOGetExceptInfo (W*32 W*64)	Retrieves the exception information when a method has returned an exception status.
AUTOGetProperty (W*32 W*64)	Passes the name or identifier of the property and gets the value of the Automation object's property.
AUTOGetPropertyByID (W*32 W*64)	Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.
AUTOGetPropertyInvokeArgs (W*32 W*64)	Passes an argument list data structure and gets the value of the Automation object's property specified in the argument list's first argument.
AUTOInvoke (W*32 W*64)	Passes the name or identifier of an object's method and an argument list data structure. It invokes the method with the passed arguments.
AUTOSetProperty (W*32 W*64)	Passes the name or identifier of the property and a value. It sets the value of the Automation object's property.
AUTOSetPropertyByID (W*32 W*64)	Passes the member ID of the property and sets the value of the Automation object's property using the argument list's first argument.
AUTOSetPropertyInvokeArgs (W*32 W*64)	Passes an argument list data structure and sets the value of the Automation object's property specified in the argument list's first argument.

The following code shows an annotated version of a portion of the code generated by the Intel® Fortran Module Wizard. This code is generated from the COM type information for the Save method of the IGeneric Document interface.

```
INTERFACE
!  Saves the document to disk.         1
INTEGER*4 FUNCTION IGenericDocument_Save($OBJECT, vFilename, &
                   vBoolPrompt, pSaved)         2
USE IFWINTY
INTEGER (INT_PTR_KIND()), INTENT(IN) :: $OBJECT     ! Object Pointer

!DIR$ ATTRIBUTES VALUE    :: $OBJECT                                    3
TYPE (VARIANT), INTENT(IN), OPTIONAL :: vFilename   ! (Optional Arg)         4
!DIR$ ATTRIBUTES REFERENCE      :: vFilename
TYPE (VARIANT), INTENT(IN), OPTIONAL :: vBoolPrompt ! (Optional Arg)
!DIR$ ATTRIBUTES REFERENCE      :: vBoolPrompt
INTEGER, INTENT(OUT) :: pSaved          ! DsSaveStatus               5
!DIR$ ATTRIBUTES REFERENCE  :: pSaved
!DIR$ ATTRIBUTES STDCALL    :: IGenericDocument_Save
```

```
END FUNCTION IGenericDocument_Save
END INTERFACE
POINTER(IGenericDocument_Save_PTR, IGenericDocument_Save) ! routine pointer 6
```

Notes for this example:

1 If the type information provides a comment that describes the member function, then the comment is placed before the beginning of the procedure.

2 The first argument to the procedure is always $OBJECT. It is a pointer to the object's interface. The remaining argument names are determined from the type information. For information on how to get a pointer to an object's interface, see Getting a Pointer to an Objects Interface.

3 This is an example of an ATTRIBUTES directive statement used to specify the calling convention of an argument.

4 A VARIANT is a data structure that can contain any type of Automation data. It contains a field that identifies the type of data and a union that holds the data value. The use of a VARIANT argument allows the caller to use any data type that can be converted into the data type expected by the member function.

5 Nearly every COM member function returns a status of type HRESULT. Because of this, if a COM member function produces output, it uses output arguments to return the values. In this example, the "pSaved" argument returns a routine specific status value.

6 The interface of a COM member function looks very similar to the interface for a dynamic link library function with one major exception. Unlike a DLL function, the address of a COM member function is never known at program link time. You must get a pointer to an object's interface at run-time, and the address of a particular member function is computed from that.

The following code shows an annotated version of the wrapper generated by the Intel® Fortran Module Wizard for the "Save" function. The name of a wrapper is the same as the name of the corresponding member function, prefixed with a "$" character.

```
! Saves the document to disk.
 INTEGER*4 FUNCTION $IGenericDocument_Save($OBJECT, vFilename, &  1
           vBoolPrompt, pSaved)
 IMPLICIT NONE
 INTEGER(INT_PTR_KIND()), INTENT(IN) :: $OBJECT    ! Object Pointer
 !DIR$ ATTRIBUTES VALUE :: $OBJECT
 TYPE (VARIANT), INTENT(IN), OPTIONAL :: vFilename
 !DIR$ ATTRIBUTES REFERENCE :: vFilename
 TYPE (VARIANT), INTENT(IN), OPTIONAL :: vBoolPrompt
 !DIR$ ATTRIBUTES REFERENCE :: vBoolPrompt
 INTEGER, INTENT(OUT)    :: pSaved    ! DsSaveStatus
 !DIR$ ATTRIBUTES REFERENCE :: pSaved
 INTEGER(4) $RETURN
 INTEGER(INT_PTR_KIND()) $VTBL            ! Interface Function Table    2
 POINTER($VPTR, $VTBL)
 TYPE (VARIANT) :: $VAR_vFilename
 TYPE (VARIANT) :: $VAR_vBoolPrompt
 IF (PRESENT(vFilename)) THEN                                    3
   $VAR_vFilename = vFilename
 ELSE
   $VAR_vFilename = OPTIONAL_VARIANT
 END IF
 IF (PRESENT(vBoolPrompt)) THEN
   $VAR_vBoolPrompt = vBoolPrompt
 ELSE
  $VAR_vBoolPrompt = OPTIONAL_VARIANT
 END IF
 $VPTR = $OBJECT           ! Interface Function Table     4
 $VPTR = $VTBL + 84    ! Add routine table offset
 IGenericDocument_Save_PTR = $VTBL
```

```
$RETURN = IGenericDocument_Save($OBJECT, $VAR_vFilename, &
$VAR_vBoolPrompt, pSaved)
$IGenericDocument_Save = $RETURN
END FUNCTION $IGenericDocument_Save
```

Notes for this example:

1 The wrapper takes the same argument names as the member function interface.

2 The wrapper computes the address of the member function from the interface pointer and an offset found in the interface's type information. In implementation terms, an interface pointer is a pointer to a pointer to an array of function pointers called an "Interface Function Table."

3 Arguments to a COM or Automation routine can be optional. The wrapper handles the invocation details for specifying an optional argument that is not present in the call.

4 The offset of the "Save" member function is 84. The code assigns the computed address to the function pointer IGenericDocument_Save_PTR, which was declared with the interface shown above, and then calls the function.

The following code shows an annotated version of a portion of the code generated by the Intel® Fortran Module Wizard from Automation type information for the Rebuild All method of the IApplication interface.

```
! Rebuilds all files in a specified configuration.
SUBROUTINE IApplication_RebuildAll($OBJECT, Configuration, $STATUS) 1
IMPLICIT NONE
INTEGER(INT_PTR_KIND()), INTENT(IN)            :: $OBJECT        ! Object Pointer
!DIR$ ATTRIBUTES VALUE           :: $OBJECT
TYPE (VARIANT), INTENT(IN), OPTIONAL :: Configuration
!DIR$ ATTRIBUTES REFERENCE          :: Configuration
INTEGER(4), INTENT(OUT), OPTIONAL    :: $STATUS       ! Method status
!DIR$ ATTRIBUTES REFERENCE          :: $STATUS
INTEGER(4) $$STATUS
INTEGER (INT_PTR_KIND) invokeargs
invokeargs = AUTOALLOCATEINVOKEARGS()                          2
IF (PRESENT(Configuration)) CALL AUTOADDARG(invokeargs, '$ARG1', &
           Configuration, AUTO_ARG_IN)
$$STATUS = AUTOINVOKE($OBJECT, 28, invokeargs)                 3
IF (PRESENT($STATUS)) $STATUS = $$STATUS                       4
CALL AUTODEALLOCATEINVOKEARGS (invokeargs)                     5
END SUBROUTINE IApplication_RebuildAll
```

Notes for this example:

1The first argument to the procedure is always $OBJECT. It is a pointer to an Automation object's IDispatch interface. The last argument to the procedure is always $STATUS. It is an optional argument that you can specify if you wish to examine the return status of the method. The IDispatch Invoke member function returns a status of type HRESULT. An HRESULT is a 32-bit value. It has the same structure as a Windows error code. In between the $OBJECT and $STATUS arguments are the method arguments' names determined from the type information. Sometimes, the type information does not provide a name for an argument. The Fortran Module Wizard creates a "$ARGn" name in this case.

2 AUTOAllocateInvokeArgs allocates a data structure that is used to collect the arguments that you will pass to the method. AUTOAddArg adds an argument to this data structure.

3 AUTOInvoke invokes the named method passing the argument list. This returns a status result.

4 If the caller supplied a status argument, the code copies the status result to it.

5AUTODeallocateInvokeArgs deallocates the memory used by the argument list data structure.

Getting a Pointer to an Object's Interface (Windows*)

This topic only applies to Windows* operating systems.

To get a pointer to an object's interface, you need to know the object's unique identifier.

Object Identification

Object identification enables the use of COM objects created by disparate groups of developers. To provide a method of uniquely identifying an object class regardless of where it came from, COM uses *globally unique identifiers* (GUIDs). A GUID is a 16-byte integer value that is guaranteed (for all practical purposes) to be unique across space and time. COM uses GUIDs to identify object classes, interfaces, and other things that require unique identification.

To create an instance of an object, you need to tell COM what the GUID of the object is. While using 16-byte integers for identification is fine for computers, it poses a challenge for the typical developer. So, COM also supports the use of a less precise, textual name called a *programmatic identifier* (ProgID). A ProgID takes the form:

```
application_name .object_name .object_version
```

Obtaining the Pointer to an Object's Interface

To use the routines generated by the Module Wizard, your application must get a pointer to an object's interface. This pointer is used as the value of the $OBJECT argument, which is the first argument of every interface generated by the Module Wizard.

Typically, your application obtains its first pointer to an object's interface by calling the COM routine COMCreateObject. COMCreateObject creates a new instance of an object class and returns a pointer to it. COMCreateObject is the generic name of the two subroutines COMCreateObjectByProgID and COMCreateObjectByGUID.

- Use COMCreateObjectByProgID to create Automation objects. It accepts the progID of an object and returns a pointer to the object's IDispatch interface.
- Use COMCreateObjectByGUID to create both COM and Automation objects. It takes a GUID and returns a pointer to the object's interface (either Automation or COM - see below)

The arguments to COMCreateObjectByGUID are as follows:

- The first argument is a class identifier (CLSID) that uniquely identifies the object's class. The Module Wizard defines a GUID parameter for each class selected from the Type library. These are given the name in the form: CLSID_*class-name*.
- The second argument allows you to limit the type(s) of server that the call will accept. Most of the time you can use CLSCTX_ALL to accept any type of server.
- The third argument is an interface identifier (IID) that specifies the particular object interface you are requesting:
  - To request an Automation interface, use IID_IDispatch.
  - To request a COM interface, the Module Wizard defines a GUID parameter for each interface selected from the type library. These are given a name in the form: IID_*interface-name*.
- The fourth argument is an output parameter that returns the object's interface pointer.

The COMCreateObjectByProgID and COMCreateObjectByGUID subroutines return an interface pointer of an object that the server has defined as being externally creatable. However, not all objects are externally creatable. Often, a server implements a hierarchy of objects, or *object model*. COMCreateObject is called to obtain a pointer to an interface of the root object in the hierarchy. Methods and/or properties of the root object are used to obtain pointers to child objects, and so on, down the hierarchy.

All objects must implement the IUnknown interface. Every object also implements one or more additional interfaces. You can always get a pointer to any of the object's interfaces from any of the object's interface pointers by using the COMQueryInterface subroutine. It is important that you have a pointer to the correct object interface when calling a routine generated by the Module Wizard. If not, your application will likely terminate unexpectedly.

Releasing the Pointer to an Object's Interface

When you have finished using an object's interface pointer, you must call COMReleaseObject with the pointer. This includes object pointers that you have received using any method, including COMCreateObject, COMQueryInterface, or by calling an object's method.

Additional Resources about COM and Automation (Windows*)

This topic only applies to Windows* operating systems.

There are a number of published books and articles about COM and Automation. Some of these are listed here as additional resources to assist customers who want to learn more about the subject matter. This list does not comment -- either negatively or positively -- on any documents listed or not yet listed. Books and related resources about COM and Automation include:

- *Inside COM+ Base Services* by Guy Eddon, Henry Eddon. Published by Microsoft Press (Redmond, Washington) 1999
- *Understanding COM+* by David S. Platt. Published by Microsoft Press (Redmond, Washington) 1999
- *Inside Distributed COM* by Guy Eddon; Henry Eddon. Published by Microsoft Press (Redmond, Washington) 1998
- *Inside COM* by Dale Rogerson. Published by Microsoft Press (Redmond, Washington) 1996
- *Understanding ActiveX and OLE* by David Chappell. Published by Microsoft Press (Redmond, Washington) 1996
- *Automation Programmer's Reference* by Microsoft. Published by Microsoft Press (Redmond, Washington) 1997
- *ActiveX Controls Inside Out*, Second Edition by Adam Denning. Published by Microsoft Press (Redmond, Washington) 1997
- *Platform SDK* online version. Relevant titles include *Platform SDK, COM and ActiveX Object Services*.
- *Visual C++ User's Guide* online version

IFPORT Portability Library

Intel® Fortran includes functions and subroutines that ease porting of code to or from a PC, or allow you to write code on a PC that is compatible with other platforms.

The portability library is called `LIBIFPORT.LIB` (Windows*) or `libifport.a` (Linux* and macOS*). Frequently used functions are included in a portability module called IFPORT.

The portability library also contains IEEE* POSIX library functions. These functions are included in a module called IFPOSIX.

You can use the portability library in one of two ways:

- Add the statement USE IFPORT to your program. This statement includes the IFPORT module.
- Call portability routines using the correct parameters and return value.

The portability library is passed to the linker by default during linking. To prevent this, specify the `fpscomp` compiler option with the `nolibs` keyword.

Using the IFPORT mod file provides interface blocks and parameter definitions for the routines, as well as compiler verification of calls.

See Also

Portability Library Routines

`fpscomp`
compiler option

fpp Preprocessing

The Intel® Fortran Compiler preprocessor, fpp, is provided as part of the Intel® Fortran product. When you use a preprocessor for Intel® Fortran source files, the generated output files are used as input source files by the compiler.

Preprocessing performs such tasks as preprocessor symbol (macro) substitution, conditional compilation, and file inclusion. Intel® Fortran predefined symbols are described in Using Predefined Preprocessor Symbols.

fpp has some of the capabilities of the ANSI C preprocessor and supports a similar set of preprocessor directives. Preprocessor directives must begin in column 1 of any Fortran source files. Preprocessor directives are not part of the Fortran language and not subject to the rules for Fortran statements. Syntax for preprocessor directives is based on that of the C preprocessor.

The compiler includes a limited conditional compilation capability, based on directives, that does not require use of fpp. The IF directive construct provides the capability of limited conditional compilation.

Note that you can also specify an alternate Fortran preprocessor instead of the Intel® Fortran fpp preprocessor. For more information, see option fpp-name.

Automatic Preprocessing by the Compiler

By default, the preprocessor is not run on files before compilation. However, the Intel® Fortran Compiler automatically calls fpp when compiling source files that have the .fpp, .F, .F90, .FOR, .FTN, or .FPP. For example, the following command preprocesses a source file that contains fpp preprocessor directives, then passes the preprocessed file to the compiler and linker:

```
ifort source.fpp
```

If you want to preprocess files that have other Fortran extensions than those listed, you have to explicitly specify the preprocessor with the `fpp` compiler option.

The fpp preprocessor can process both free-form and fixed-form Fortran source files. By default, filenames with the suffix of .F, .f, .for, or .fpp are assumed to be fixed form. Filenames with a suffix of .F90 or .f90 (or any other suffix not specifically mentioned here) are assumed to be free form. You can use the `free` compiler option to specify free form and the `fixed` compiler option to explicitly specify fixed form.

The fpp preprocessor recognizes tab format in a source line in fixed form.

Running fpp to Preprocess Files

You can explicitly run fpp in these ways:

- On the `ifort` command line, use the `ifort` command with the `fpp` compiler option. By default, the specified files are then compiled and linked. To retain the intermediate (.i or .i90) file, specify the `[Q]save-temps` compiler option.
- On the command line, use the fpp command. In this case, the compiler is not invoked. When using the fpp command line, you need to specify the input file and the intermediate (.i or .i90) output file. For more information, type `fpp -help` (Linux and macOS*) or `fpp /help` (Windows) on the command line.
- In the Microsoft Visual Studio* IDE, set the **Preprocess Source File** option to **Yes** in the **Fortran Preprocessor Option** Category. To retain the intermediate files, add `/Qsave-temps` to **Additional Options** in the **Fortran Command Line Category**.

The following lists some common cpp features that are supported by fpp; it also shows common cpp features that are not supported.

Supported cpp features:	Unsupported cpp features:
#define, #undef, #ifdef, #ifndef, #if, #elif, #else, #endif, #include, #error, #warning, #line	#pragma and #ident
# (stringsize) and ## (concatenation) operators	spaces or tab characters preceding the initial "#" character
	# followed by empty line
! as negation operator	\ backslash-newline

Unlike cpp, fpp does not merge continued lines into a single line when possible.

You do not usually need to specify preprocessing for Fortran source programs unless your program uses fpp preprocessing commands, such as those listed above.

Caution
Using a preprocessor that does not support Fortran can damage your Fortran code, especially with "FORMAT (\\I4)" with cpp changes the meaning of the program because the double backslash "\\" indicates end-of-record with most C/C++ preprocessors.

fpp Source Files

A source file can contain fpp tokens in the following forms:

- fpp preprocessor directive names

  For more information on fpp preprocessor directives, see **Using fpp Preprocessor Directives**.
- symbolic names including Fortran keywords

  fpp permits the same characters in names as Fortran. For more information on symbolic names, see **Using Predefined Preprocessor Symbols**.
- constants

  Integer, real, double, and quadruple precision real, binary, octal, hexadecimal (including alternate notation), character, and Hollerith constants are allowed.
- special characters, space, tab, and newline characters
- comments, including:
  - Fortran language comments. A fixed form source line containing one of the symbols C, c, *, d, or D in the first position is considered a comment line. The ! symbol is interpreted as the beginning of a comment extending to the end of the line except when the ! occurs within a constant-expression in an #if or #elif directive. Within such comments, macro expansions are not performed, but they can be switched on by `-f-com=no`.
  - fpp comments between `/*` and `*/`. They are excluded from the output and macro expansions are not performed within these symbols. fpp comments can be nested: for each `/*` there must be a corresponding `*/`. fpp comments are useful for excluding from the compilation large portions of source instead of commenting every line with a Fortran comment symbol.
  - C++ -like line comments that begin with `//` (double-slash).

A string that is a token can occupy several lines, but only if its input includes continued line characters using the Fortran continuation character `&`. fpp will not merge such lines into one line.

Identifiers are always placed on one line by fpp. For example, if an input identifier occupies several lines, it will be merged by fpp into one line.

fpp Output

Output consists of a modified copy of the input, plus lines of the form:

```
#line_number  file_name
```

These are inserted to indicate the original source line number and filename of the output line that follows. Use the fpp preprocessor option `P` to disable the generation of these lines.

Diagnostics

There are three kinds of fpp diagnostic messages:

- warnings: preprocessing of source code is continued and the fpp return value is 0
- errors: fpp continues preprocessing but sets the return value to a nonzero value which is the number of errors
- fatal errors: fpp stops preprocessing and returns a nonzero return value.

The messages produced by fpp are intended to be self-explanatory. The line number and filename where the error occurred are displayed along with the diagnostic on stderr.

See Also

fixed compiler option
fpp compiler option
fpp-name compiler option
free compiler option
IF Directive Construct
save-temps, Qsave-temps compiler option
Understanding File Extensions

Using fpp Preprocessor Directives

All fpp preprocessor directives start with the number sign (`#`) as the first character on a line. White space (blank or tab characters) can appear after the initial "#" for indentation.

fpp preprocessor directives can be placed anywhere in source code, in particular before a Fortran continuation line. However, fpp preprocessor directives within a macro call can not be divided among several lines by means of continuation symbols.

fpp preprocessor directives can be grouped according to their purpose.

Preprocessor directives for string substitution

The following fpp preprocessor directives cause substitutions in your program:

Preprocessor Directive	Result
`__FILE__`	Replaces a string with the input file name (a character string literal).
`__LINE__`	Replaces a string with the current line number in the input file (an integer constant).
`__DATE__`	Replaces a string with the date that fpp processed the input file (a character string literal in the form `Mmm dd yyyy`).
`__TIME__`	Replaces a string with the time that fpp processed the input file (a character string literal in the form `hh:mm:ss`).

Preprocessor Directive	Result
__TIMESTAMP__	Replaces a string with the timestamp that fpp processed the input file (a character string literal in the form "day date time year", where *day* is a 3-letter abbreviation, *date* is `Mmm dd`, *time* is `hh:mm:ss` and *year* is `yyyy`).

Preprocessor directive for inclusion of external files

To include external files, preprocessor directive #include can be specified in one of the following forms:

```
#include "filename"
```

```
#include <filename>
```

#include reads in the contents of the named file into the specified or default location in the source. The lines read in from the file are processed by fpp just as if they were part of the current file.

When the <*filename*> notation is used, *filename* is only searched for in the standard "include" directories. For more information, see the fpp `-I` and the `-Y` options in **Using Fortran Preprocessor Options**. No additional tokens are allowed on the directive line after the final "" or ">".

For `#include "filename"`, filenames are searched for in the following order:

- In the directory in which the source file resides
- In the directories specified by the `-I` or `-Y` option
- In the default directory

For `#include <filename>`, filenames are searched for in the following order:

- In the directories specified by the `-I` or `-Y` option
- In the default directory

Preprocessor directive for line control

The preprocessor directive #line-number generates line control information for the Fortran compiler. It takes the following form:

```
#line-number "filename"
```

#line-number is an integer constant that is the line number of the next line. "*filename*" is the name of the file containing the line. If "*filename*" is not provided, the current filename is assumed.

Preprocessor directive for fpp variable and macro definitions

The preprocessor directive #define can be used to define both simple string variables and more complicated macros. It can take two forms.

- Definition of an fpp variable:

  ```
  #define nametoken-string
  ```

  In the above, occurrences of *name* in the source file will be replaced by *token-string*.
- Definition of an fpp macro:

  ```
  #define name(argument[,argument] ... ) token-string
  ```

  In the above, occurrences of the macro *name* followed by the comma-separated list of actual arguments within parentheses will be replaced by *token-string*. Each occurrence of *argument* in *token-string* is replaced by the token sequence representing the corresponding "actual" argument in the macro call.

An error occurs if the number of macro call arguments is not the same as the number of arguments in the corresponding macro definition. For example, consider this macro definition:

```
#define INTSUB(m, n, o) call mysub(m, n, o)
```

Any use of the macro `INTSUB` must have three arguments. In macro definitions, spaces between the macro name and the open parenthesis "(" are prohibited to prevent the directive from being interpreted as an fpp variable definition with the rest of the line beginning with the open parenthesis "(" being interpreted as its token-string.

An fpp variable or macro definition can be of any length and is limited only by the newline symbol. It can be defined in multiple lines by continuing it to the next line with the insertion of "\". For example:

```
#define long_macro_name(x,\
   y) x*y
```

The occurrence of a newline without a macro-continuation signifies the end of the macro definition.

The scope of a definition begins from the #define and encloses all the source lines (and source lines from #include files) to the end of the current file, except for:

- Files included by Fortran INCLUDE statements
- fpp and Fortran comments
- Fortran IMPLICIT statements that specify a single letter
- Fortran FORMAT statements
- Numeric, typeless, and character constants

Preprocessor directive for undefining a macro

The preprocessor directive #undef takes the following form:

```
#undef name
```

This preprocessor directive removes any definition for *name* produced by the `D` options, the #define preprocessor directives, or by default. No additional tokens are permitted on the directive line after *name*.

If *name* has not been previously defined, then #undef has no effect.

Preprocessor directive for macro expansion

If, during expansion of a macro, the column width of a line exceeds column 72 (for fixed format) or column 132 (for free format), fpp inserts appropriate Fortran continuation lines.

For fixed format, there is a limit on macro expansions in label fields (positions 1-5):

- A macro call (together with possible arguments) should not extend beyond column 5.
- A macro call whose name begins with one of the Fortran comment symbols is considered to be part of a comment.
- A macro expansion may produce text extending beyond column 5. In this case, a warning will be issued.

In fixed format, when the fpp option `-Xw` has been specified, an ambiguity may occur if a macro call occurs in a statement position and a macro name begins or coincides with a Fortran keyword. For example, consider the following:

```
#define callp(x)   call f(x)
  call p(0)
```

fpp cannot determine how to interpret the `callp` token sequence above. It could be considered to be a macro name. The current implementation does the following:

- The longer identifier is chosen (`callp` in this case)

- From this identifier the longest macro name or keyword is extracted
- If a macro name has been extracted a macro expansion is performed. If the name begins with some keyword, fpp issues an appropriate warning
- The rest of the identifier is considered as a whole identifier

In the previous example, the macro expansion is performed and the following warning is produced:

```
warning: possibly incorrect substitution of macro callp
```

This situation appears only when preprocessing fixed-format source code and when the space symbol is not interpreted as a token delimiter.

In the following case, a macro name coincides with a beginning of a keyword:

```
 #define INT  INTEGER*8
              INTEGER k
```

The INTEGER keyword will be found earlier than the INT macro name. There will be no warning when preprocessing such a macro definition.

Preprocessor directives for conditional selection of source text

The following three preprocessor directives are conditional constructs that you can use to select source text.

- #if preprocessor directive

  When #if is specified, subsequent lines up to the matching #else, #elif, or #endif preprocessor directive appear in the output only if *condition* evaluates to .TRUE.. The following shows an example:

```
#if condition_1block_1
#elif condition_2block_2
#elif ...
#else
  block_n
#endif
```

- #ifdef preprocessor directive

  When #ifdef is specified, subsequent lines up to the matching #else, #elif, or #endif preprocessor directive appear in the output only if *name* has been defined, either by a #define preprocessor directive or by the `D` compiler option, with no intervening #undef preprocessor directive. No additional tokens are permitted on the preprocessor directive line after *name*. The following shows an example:

```
#ifdef nameblock_1
#elif conditionblock_2
#elif ...
#else
  block_n
#endif
```

- #ifndef preprocessor directive

  When #ifndef is specified, subsequent lines up to the matching #else, #elif, or #endif preprocessor directive appear in the output only if *name* has not been defined, or if its definition has been removed with an #undef preprocessor directive. No additional tokens are permitted on the directive line after *name*. The following shows an example:

```
#ifndef nameblock_1
#elif conditionblock_2
#elif ...
#else
  block_n
#endif
```

The #else, #elif, or #endif preprocessor directives are optional. They can be used in the above preprocessor directives.

Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if all of the following occur:

- The condition in the preceding #if directive evaluates to .FALSE. or the name in the preceding #ifdef directive is not defined, or the name in the preceding #ifndef directive is defined.
- The conditions in all of the preceding #elif directives evaluate to .FALSE.
- The condition in the current #elif evaluates to .TRUE.

Any condition allowed in an #if directive is allowed in an #elif directive. Any number of #elif directives may appear between an #if, #ifdef, or #ifndef directive and a matching #else or #endif directive.

Conditional expressions

condition_1, *condition_2*, etc. are logical expressions involving fpp constants, macros, and intrinsic functions. The following items are permitted in conditional expressions:

- C language operations: <, >, ==, !=, >=, <=, +, -, /, *, %, <<, >>, &, ~, |, &&, and ||

  They are interpreted by fpp in accordance to the C language semantics. This facility is provided for compatibility with "old" Fortran programs using cpp.
- Fortran language operations: .AND., .OR., .NEQV., .XOR., .EQV., .NOT., .GT., .LT., .LE., .GE., .NE., .EQ., ** (power).
- Fortran logical constants: .TRUE. , .FALSE.
- The fpp intrinsic function "`defined`": `defined(name)` or `defined name` which returns .TRUE. if name is defined as an fpp variable or a macro. It returns .FALSE. if the name is not defined.

#ifdef is shorthand for `#if defined(name)` and #ifndef is shorthand for `#if .not. defined(name)`.

Only these items, integer constants, and names can be used within a constant expression. A name that has not been defined with the `-D` option, a #define preprocessor directive, or defined by default, has a value of 0. The C operation `!=` (not equal) can be used in the #if or #elif preprocessor directive, but not in the #define preprocessor directive, where the symbol `!` is considered to be the Fortran comment symbol by default.

See Also

D compiler option

fpp compiler option

Using Fortran Preprocessor Options

Using Predefined Preprocessor Symbols

Preprocessor symbols (macros) let you substitute values in a program before it is compiled. The substitution is performed in the preprocessing phase.

The preprocessor symbols shown in the table below are predefined by the compiler system and are available to compiler directives and to fpp. If you want to use other symbol names, you need to specify them on the command line.

You can use the `D` compiler option to define the symbol names to be used during preprocessing. This option performs the same function as the #define preprocessor directive.

Preprocessing with fpp replaces every occurrence of the defined symbol name with the specified value. Preprocessing compiler directives only allow IF and IF DEFINED.

If you want to disable symbol replacement (also known as macro expansion) during the preprocessor step, you can specify the `macro=no` preprocessor option for the `fpp` compiler option.

Disabling preprocessor symbol replacement is useful for running fpp to perform conditional compilation (using #ifdef, etc.) without replacement.

You can use the U preprocessor option to suppress an automatic definition of a preprocessor symbol. This option suppresses any symbol definition currently in effect for the specified *name*. This option performs the same function as an #undef preprocessor directive.

The symbols in the following table can be used in both fpp and the Fortran compiler conditional compilations.

Symbol	Description
__APPLE__ (macOS*)	Defined as '1'.
__AVX512BW__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Byte and Word instructions. It is also defined as '1' when option [Q]xCORE-AVX512 or higher processor-targeting options are specified.
__AVX512CD__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Conflict Detection instructions. It is also defined as '1' when option [Q]xCORE-AVX512, [Q]xCOMMON-AVX512, or higher processor-targeting options are specified.
__AVX512DQ__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Doubleword and Quadword instructions. It is also defined as '1' when option [Q]xCORE-AVX512 or higher processor-targeting options are specified.
__AVX512ER__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Exponential and Reciprocal instructions.
__AVX512F__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions. It is also defined as '1' when option [Q]xCORE-AVX512, [Q]xCOMMON-AVX512, or higher processor-targeting options are specified.
__AVX512PF__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Prefetch instructions.
__AVX512VL__ (Windows*, Linux, macOS*)	Defined as '1' for processors that support Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Vector Length extensions. It is also defined as '1' when option [Q]xCORE-AVX512 or higher processor-targeting options are specified.
_DEBUG (Windows)	Defined as '1' only if options /dbglibs, /MT[d], or /MD[d] are specified.
_DLL (Windows)	Defined as '1' only if one of the following options is specified or implied:

Symbol	Description
	• /libs:dll • /MD • /MDd • /MDs
__ELF__ (Linux)	Defined as '1' at the start of compilation.
__gnu_linux__ (Linux)	Defined as '1' at the start of compilation.
__i386__ __i386 i386 (Linux, macOS*)	Defined as '1' for compilations targeting IA-32 architecture. IA-32 is not available on macOS*.
__INTEL_COMPILER (Windows, Linux, macOS*)	The Intel compiler version in the form VVSS, where *VV* is the major version and *SS* is the minor version. For example, Version 16.0 is indicated by a value of 1600.
__INTEL_COMPILER_BUILD_DATE (Windows*, Linux, macOS*)	The Intel compiler build date. It takes the form YYYYMMDD, where *YYYY* is the year, *MM* is the month, and *DD* is the day.
__INTEL_COMPILER_UPDATE (Windows, Linux)	The Intel compiler update number within a version (such as 1 for Update 1, etc.). See also symbol __INTEL_COMPILER.
__linux__ __linux linux (Linux)	Defined as '1' at the start of compilation.
_M_AMD64 (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
_M_IX86=700 (Windows)	Defined as '1' while building code targeting IA-32 architecture. IA-32 is not available on macOS*.
_M_X64 (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.
__MACH__ (macOS*)	Defined as '1'.
_MT (Windows)	Defined as '1' only if option /threads or /MT is specified.
_OPENMP=201611 (Windows, Linux, macOS*)	Defined when OpenMP* processing has been requested (that is, option [q or Q]openmp has been specified along with option fpp).

Symbol	Description
	The value takes the form YYYYMM, where *YYYY* is the year and *MM* is the month of the supported OpenMP* Fortran specification. The currently supported OpenMP* API is version TR4: Version 5.0, dated November 2016.

`__PIC__`
`__pic__`
(Linux)
	Defined as '1' only if the code was requested to be compiled as position-independent code.

`_WIN32` (Windows)	Defined as '1' while building code targeting IA-32 or Intel® 64 architecture. IA-32 is not available on macOS*.
`_WIN64` (Windows)	Defined as '1' while building code targeting Intel® 64 architecture.

`__x86_64`
`__x86_64__`
(Linux, macOS*)
	Defined as '1' while building code targeting Intel® 64 architecture.

See Also

arch compiler option
march compiler option
m compiler option
D compiler option
U compiler option
dll compiler option
fpp compiler option
libs compiler option
qopenmp, Qopenmp compiler option
threads compiler option
x, Qx compiler option
qoffload compiler option

Using Fortran Preprocessor Options

The Fortran preprocessor (fpp) may be invoked automatically or by specifying the `fpp` compiler option.

The following preprocessor options are available if you use the `fpp` compiler option. On Linux* and macOS* systems, the preprocessor options must start with a dash (`-`); for example, `-macro=no`. On Windows* systems, the preprocessor options must start with a slash (`/`); for example, `/macro=no`.

Preprocessor Option	Description
`B`	Specifies that C++-style comments should not be recognized.

Preprocessor Option	Description
C	Specifies that C-style comments should not be recognized. This is the same as specifying c_com=no.
c_com={yes|no}	Determines whether C-style comments are recognized. If you specify c_com=no or C, C-style comments are not recognized. By default, C-style comments are recognized; that is, c_com=yes.
D*name*	Defines the preprocessor variable *name* as 1 (one). This is the same as if a Dname=1 option appeared on the fpp command line. This is the same as specifying the following line in the source file processed by fpp: `#define name 1`
D*name=val*	Defines *name* as if by a #define directive. This is the same as specifying the following line in the source file processed by fpp: `#define name var` The D*name=val* will be ignored if there are any non-alphabetic, non-numeric characters in *name*. The D option has lower precedence than the U option. That is, if the same name is used in both a U option and a D option, the name will be undefined regardless of the order of the options.
e	For fixed-form source files, tells the compiler to accept extended source lines, up to 132 characters long.
e80	For fixed-form source files, tells the compiler to accept extended source lines, up to 80 characters long.

f_com={yes|no}	Determines whether Fortran-style end-of-line comments are recognized or ignored by fpp. If you specify f_com=no, Fortran style end-of-line comments are processed as part of the preprocessor directive.
	By default, Fortran style end-of-line comments are identified by fpp on preprocessor lines and are then ignored by fpp; that is, f_com=yes. For example:
	`#define max 100 ! max number`
	`do i = 1, max + 1`
	If you specify f_com=yes, fpp will output the following:
	`do i = 1, 100 + 1`

Preprocessor Option	Description
	If you specify `f_com=no`, fpp will output the following: `do i = 1, 100 ! max number + 1`
`f77`	Tells the compiler to assume FORTRAN 77 language input source.
`f90`	Tells the compiler to assume Fortran 90 language input source.
`fixed`	Tells the compiler to assume fixed format in the source file.
`free`	Tells the compiler to assume free format in the source file.
`help`	Displays information about fpp options.
`I`*dir*	Inserts directory *dir* into the search path for files with names not beginning with "/". The #include *dir* is inserted ahead of the standard list of "include" directories so that #include files with names enclosed in double-quotes (") are searched for first in the directory of the file with the #include line, then in directories named with `I` options, and lastly, in directories from the standard list. For #include files with names enclosed in angle-brackets (< >), the directory of the file with the #include line is not searched.
`M`	Generates make dependencies.
`m`	Expands macros everywhere. This is the same as specifying `macro=yes`.
`macro={yes|no_com|no}`	Determines the behavior of macro expansion. If you specify `macro=no_com`, macro expansion is turned off in comments. If you specify `macro=no`, no macro expansion occurs anywhere. By default, macros are expanded everywhere; that is, `macro=yes`.
`macro_expand={vc|cpp}`	Determines the mode of macro expansion. If you specify `macro_expand=vc`, macros are expanded in Microsoft Visual C/C++ order. If you specify `macro_expand=cpp`, macros are expanded in GNU CPP order.
`MF=`*file*	Makes fpp append dependencies to *file*.
`noB`	Specifies that C++-style comments should be recognized.

Preprocessor Option	Description
noC	Specifies that C-style comments should be recognized. This is the same as c_com=yes.
noJ	Specifies that F90-style comments should be recognized in a #define line. This is the same as f_com=no.
no-fort-cont	Specifies that IDL-style format should be recognized. This option is only for the IDE. Note that macro arguments in IDL may have a C-like continuation character \ which is different from the Fortran continuation character &. The fpp preprocessor should recognize the C-like continuation character and process some other non-Fortran tokens so that the IDL processor can recognize them.
P	Tells the compiler that line-numbering directives should not be added to the output file. This line-numbering directive appears as `#line-number file-name`
U*name*	Removes any initial definition of name, where name is an fpp variable that is predefined on a particular preprocessor. The following shows an example of symbols that may be predefined, depending upon the architecture of the system: Operating System: __APPLE__, __unix, and __linux Hardware: __i386, __x86_64
undef	Removes initial definitions for all predefined symbols.
V	Displays the fpp version number.
w[0]	Prevents warnings from being output. By default, warnings are output to standard error stream (stderr).
what	Displays detailed version information.
Xu	Converts uppercase letters to lowercase, except within character-string constants. The default is to leave the case as is.
Xw	Tells the compiler that in fixed-format source files, the blank or space symbol " " is insignificant. By default, the space symbol is the delimiter of tokens for this format.
Y*dir*	Adds directory *dir* to the end of the system include paths.

Methods to Optimize Code Size

This section provides some guidance on how to achieve smaller object and smaller executable size when using the optimizing features of Intel compilers.

To begin, there are two compiler options that are designed to prioritize code size over performance:

Favors size over speed	Linux* and macOS*: `-Os` Windows*: `/Os`	This option enables optimizations that do not increase code size; it produces smaller code size than option `O2`. Option `Os` disables some optimizations that may increase code size for a small speed benefit.
Minimizes code size	Linux* and macOS*: `-O1` Windows*: `/O1`	Compared to option `Os`, option `O1` disables even more optimizations that are generally known to increase code size. Specifying option `O1` implies option `Os`. As an intermediate step in reducing code size, you can replace option `O3` with option `O2` before specifying option `O1`. Option `O1` may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.

For more information about the above options, see their full descriptions in the Compiler Reference.

The rest of this section briefly discusses methods that may help you further improve code size even when compared to the default behaviors of options `Os` and `O1`.

The following table summarizes the topics in this section.

Most Common Methods to Reduce Code Size:
Disable or Decrease the Amount of Inlining
Strip Symbols from Your Binaries
Dynamically Link Intel-Provided Libraries
Disable Inline Expansion of Standard Library or Intrinsic Functions
Methods to Use Only When Code Size is Very Important:
Disable Passing Arguments in Registers Instead of On the Stack
Disable Loop Unrolling
Disable Automatic Vectorization
Method to Use Under Special Circumstances:
Avoid Unnecessary 16-Byte Alignment

The following are important considerations:

- Some of these methods may already be applied by default when options `Os` and `O1` are specified. All the methods mentioned in subsequent topics can be applied at higher optimization levels.

- Some of the options referred to in these topics will not necessarily cause code size reduction, and they may provide varying results (good, bad, or neutral) based on the characteristics of the target code. Still, these are the recommended things to try to see if they cause your binaries to become smaller while maintaining acceptable performance.
- You should read the full description of compiler options that are mentioned in these topics to get complete information about their behavior, syntax, and target platforms.

See Also

- `O` compiler option
- `Os` compiler option

Disable or Decrease the Amount of Inlining

Inlining replaces a call to a function with the body of the function. This lets the compiler optimize the code for the inlined function in the context of its caller, usually yielding more specialized and better performing code. This also removes the overhead of calling the function at run-time.

However, replacing a call to a function by the code for that function usually increases code size. The code size increase can be substantial. To eliminate this code size increase, at the cost of the potential performance improvement, inlining can be disabled.

As an alternative to completely disabling inlining, the default amount of inlining can be decreased by using an inline factor less than the default value of 100. It corresponds to scaling the default values of the main inlining parameters by *n*%.

Options to specify:

To disable inlining:	Linux* and macOS*: `-fno-inline` Windows*: `/Ob0`
To reduce inlining and factor the main inlining parameters:	Linux* and macOS*: `-inline-factor=`*n* Windows*: `/Qinline-factor:`*n*
To fine tune the main inlining parameters:	Linux* and macOS*: • `-inline-factor` • `-inline-max-per-compile` • `-inline-max-per-routine` • `-inline-max-size` • `-inline-max-total-size` • `-inline-min-size` Windows*: • `/Qinline-factor` • `/Qinline-max-per-compile` • `/Qinline-max-per-routine` • `/Qinline-max-size` • `/Qinline-max-total-size` • `/Qinline-min-size`

For further details about the compiler options, see the compiler option descriptions.

Advantages of this method:	Disabling or reducing this optimization can reduce code size.
Disadvantages of this method:	Performance is likely to be sacrificed by disabling or reducing inlining especially for applications with many small functions.

Strip Symbols from Your Binaries

You can specify a compiler option to omit debugging and symbol information from the executable without sacrificing its operability.

Options to specify:

Linux* and macOS*:	`-Wl,--strip-all`
Windows*:	None

Advantages of this method:	This method noticeably reduces the size of the binary.
Disadvantages of this method:	It may be very difficult to debug a stripped application.

Dynamically Link Intel-Provided Libraries

By default, some of the Intel support and performance libraries are linked statically into an executable. As a result, the library codes are linked into every executable being built. This means that codes are duplicated.

It may be more profitable to link them dynamically.

Options to specify:

Linux* and macOS*:	`-shared-intel`
Windows*:	`/MD` or /libs:dll

Advantages of this method:	• Performance of the resulting executable is normally not significantly affected. • Library codes that are otherwise linked in statically into every executable will not contribute to the code size of each executable with this option. These codes will be shared between all executables using them, and will be available independent of those executables.
Disadvantages of this method:	• The libraries on which the resulting executable depends must be re-distributed with the executable in order for it to work properly. • When libraries are linked statically, only library content that is actually used is linked into the executable. Dynamic libraries, on the other hand, contain all the library content. Therefore, it may not be beneficial to use this option if you only need to build and/or distribute a single executable. • The executable itself may be much smaller when linked dynamically, compared to a statically linked executable. However, the total size of the executable plus shared libraries or DLLs may be much larger than the size of the statically linked executable.

Disable Inline Expansion of Standard Library or Intrinsic Functions

In some cases, disabling the inline expansion of standard library or intrinsic functions may noticeably improve the size of the produced object or binary.

Options to specify:

Linux* and macOS*:	`-nolib-inline`
Windows*:	None

Disable Passing Arguments in Registers Instead of On the Stack

You can specify an option that causes the compiler to pass arguments in registers rather than on the stack. This can yield faster code.

However, doing this may require the compiler to create an additional entry point for any function that can be called outside the code being compiled.

In many cases, this will lead to an increase in code size. To prevent this increase in code size, you can disable this optimization.

Options to specify:

Linux* and macOS*:	`-qopt-args-in-regs=none`
Windows*:	`/Qopt-args-in-regs:none`

Advantages of this method:	Disabling this optimization can reduce code size.
Disadvantages of this method:	The amount of code size saved may be small when compared to the corresponding performance loss of disabling the optimization.

Notes:

If you do not specify "none" for option [q or Q]opt-args-in-regs, the default behavior for the option is that parameters are passed in registers when they are passed to routines whose definition is seen in the same compilation unit.

Depending on code characteristics, this option can sometimes increase binary size.

Disable Loop Unrolling

Unrolling a loop increases the size of the loop proportionally to the unroll factor.

Disabling (or limiting) this optimization may help reduce code size at the expense of performance.

Options to specify:

Linux* and macOS*:	`-unroll=0`
Windows*:	`/Qunroll:0`

Advantages of this method:	Code size is reduced.
Disadvantages of this method:	Performance of otherwise unrolled loops may noticeably degrade because this limits other possible loop optimizations.

Notes:

This option is already the default if you specify option `Os` or option `O1`.

Disable Automatic Vectorization

The compiler finds possibilities to use SIMD (SSE/AVX) instructions to improve performance of applications. This optimization is called automatic vectorization.

In most cases, this optimization involves transformation of loops and increases code size, in some cases significantly.

Disabling this optimization may help reduce code size at the expense of performance.

Options to specify:

Linux* and macOS*:	`-no-vec`
Windows*:	`/Qvec-`

Advantages of this method:	Compile-time is also improved significantly.
Disadvantages of this method:	Performance of otherwise vectorized loops may suffer significantly. If you care about the performance of your application, you should use this option selectively to suppress vectorization on everything except performance-critical parts.

Notes:

Depending on code characteristics, this option can sometimes increase binary size.

Avoid Unnecessary 16-Byte Alignment

This method should only be used in certain situations that are well understood. It can potentially cause correctness issues when linking with other objects or libraries that aren't built with this option. This topic only applies to Linux systems on IA-32 architecture.

The 32-bit Linux ABI states that stacks need only maintain 4-byte alignment. However, for performance reasons in modern architectures, GCC and ICC maintain an alignment of 16-bytes on the stack. Maintaining 16-byte alignment may require additional instructions to adjust the stack on function entries where no stack adjustment would otherwise be needed. This can impact code size, especially in code that consists of many small routines.

You can specify a compiler option that will revert ICC back to maintaining 4-byte alignment, which can eliminate the need for extra stack adjust instructions in some cases.

Use this option *only* if one of the following is true:

- Your code does not call any other object or library that can be built without this option and, therefore, may rely on the stack being aligned to 16-bytes when called.
- Your code is targeted for architectures that do not have or support SSE instructions; therefore, it would never need 16-byte alignment for correctness reasons.

Options to specify:

Linux*:	`-falign-stack=assume-4-byte`
macOS*:	None
Windows*:	None

Advantages of this method:	• Code size can be smaller because you do not need extra instructions to maintain 16-byte alignment when not needed. • This method can improve performance in some cases because of this reduction of instructions.
Disadvantages of this method:	This method can cause incompatibility when linked with other objects or libraries that rely on the stack being 16-byte aligned across the calls.

Notes:

Depending on code characteristics, this option can sometimes increase binary size.

National Language Support (NLS) Routines

Intel® Fortran provides a complete National Language Support (NLS) library of language-localization routines and multibyte-character routines. You can use these routines to write applications in many different languages.

In many languages, the standard ASCII character set is not enough because it lacks common symbols and punctuation (such as the British pound sign), or because the language uses a non-ASCII script (such as Cyrillic for Russian) or because the language consists of too many characters for each to be represented by a single byte (such as Chinese).

In the case of many non-ASCII languages, such as Arabic and Russian, an extended single-byte character set is sufficient. You need only change the language locale and codepage, which can be done at a system level or within your program. However, Eastern languages such as Japanese and Chinese use thousands of separate characters that cannot be encoded as single-byte characters. Multibyte characters are needed to represent them.

Character sets are stored in tables called code sets. There are three components of a code set: the locale, which is a language and country (since, for instance, the language Spanish may vary among countries), the codepage, which is a table of characters to make up the computer's alphabet, and the font used to represent the characters on the screen. These three components can be set independently. Each computer running Windows operating systems comes with many code sets built into the system, such as English, Arabic, and Spanish. Multibyte code sets, such as Chinese and Japanese, are not standard but come with special versions of the operating system (for instance, Windows NT-J comes with the Japanese code set).

The default code set is obtained from the operating system when a program starts up. When you install your operating system, you should install the system supplied code sets. Thereafter, they are always available. You can switch among them by:

- Open the Control Panel (available from Settings)
- Click the Regional Settings icon
- Choose from the dropdown list of available locales (languages and countries).

When you select a new locale, it becomes the default system locale, and will remain the default locale until you change it. Each locale has a default codepage associated with it, and a default currency, number, and date format.

NOTE
The default codepage does not change when you select a new locale until you reboot your computer.

You can change the currency, number, and date format in the International dialog box or the Regional Setting dialog box independently of the locale.

The locale determines the character set available to the user. The locale you select becomes the default for the NLS routines described in this section, but the NLS routines allow you to change locales and their parameters from within your programs. These routines are useful for creating original foreign-language programs or different versions of the same program for various international markets. Changes you make to the locale from within a program affect only the program. They do not change the system default settings.

The codepage you select, which can be set independently, controls the multibyte (MB routines) character routines described in this section. Only users with special multibyte-character code sets installed on their computers need to use MB routines. The standard code sets all use single-byte character code sets.

Note that in Intel Fortran source code, multibyte characters can be used only in character strings and source comments. They cannot be used within variable names or statements. Like program changes to the locale, program changes to codepages affect only the program, not the system defaults.

To access the routines, the following statement should be present in any program unit that uses NLS or MB routines:

```
USE IFNLS
```

See Also

Overview of NLS and MCBS Routines
Program Units and Procedures

Understanding Single and Multibyte Character Sets (Windows*)

The ASCII character set defines the characters from 0 to 127 and an extended set from 128 to 255. Several alternative single-byte character sets, primarily European, define the characters from 0 to 127 identically to ASCII, but define the characters from 128 to 255 differently. With this extension, 8-bit representation is sufficient for defining the needed characters in most European-derived languages. However, some languages, such as Japanese Kanji, include many more characters than can be represented with a single byte. These languages require multibyte coding.

A multibyte character set consists of both one-byte and two-byte characters. A multibyte-character string can contain a mix of single and double-byte characters. A two-byte character has a lead byte and a trail byte. In a particular multibyte character set, the lead and trail byte values can overlap, and it is then necessary to use the byte's context to determine whether it is a lead or trail byte.

Tools

PGO Tools

PGO Tools Overview

This section describes the tools that take advantage of or support the Profile-guided Optimizations (PGO) available in the compiler.

- Code coverage Tool
- Test prioritization Tool
- Profmerge and proforder Tools

Code Coverage Tool

The code coverage tool provides software developers with a view of how much application code is exercised when a specific workload is applied to the application. To determine which code is used, the code coverage tool uses Profile-guided Optimization (PGO) options and optimizations. The major features of the code coverage tool are listed below:

- Visually presenting code coverage information for an application with a customizable code coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

The information about using the code coverage tool is separated into the following sections:

- Code coverage tool Requirements

- Visually Presenting Code Coverage for an Application
- Excluding Code from Coverage Analysis
- Exporting Coverage Data

The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate an HTML-formatted report and export data in both text-, and XML-formatted files. The reports can be further customized to show color-coded, annotated, source-code listings that distinguish between used and unused code.

The code coverage tool is available on all supported Intel architectures on Linux*, Windows*, and macOS* operating systems.

You can use the tool in a number of ways to improve development efficiency, reduce defects, and increase application performance:

- During the project testing phase, the tool can measure the overall quality of testing by showing how much code is actually tested.
- When applied to the profile of a performance workload, the code coverage tool can reveal how well the workload exercises the critical code in an application. High coverage of performance-critical modules is essential to taking full advantage of the Profile-Guided Optimizations that Intel Compilers offer.
- The tool provides an option useful for both coverage and performance tuning, enabling developers to display the dynamic execution count for each basic block of the application.
- The code coverage tool can compare the profile of two different application runs. This feature can help locate portions of the code in an application that are unrevealed during testing but are exercised when the application is used outside the test space, for example, when used by a customer.

Code Coverage Tool Requirements

To run the code coverage tool on an application, you must have following items:

- The application sources.
- The .spi file generated by the Intel® compiler when compiling the application for the instrumented binaries using the `-prof-gen=srcpos` (Linux and macOS*) or `/Qprof-gen:srcpos` (Windows) option.

> **NOTE**
> Use the `-[Q]prof-gen:srcpos` option if you intend to use the collected data for code coverage and profile feedback. If you are only interested in using the instrumentation for code coverage, use the `/Qcov-gen` option. Using the `/Qcov-gen` option saves time and improves performance. This option can be used only on Windows platform for all architectures.

- A `pgopti.dpi` file that contains the results of merging the dynamic profile information (.dyn) files, which is most easily generated by the profmerge tool. This file is also generated implicitly by the Intel® compilers when compiling an application with `[Q]prof-use` options with available .dyn and .dpi files.

Using the Tool

The tool uses the following syntax:

Tool Syntax
`codecov [-codecov_option]`

where `-codecov_option` is one or more optional parameters specifying the tool option passed to the tool. The available tool options are listed in the code coverage tools Options section. If you do not use any additional tool options, the tool will provide the top-level code coverage for the entire application.

In general, you must perform the following steps to use the code coverage tool:

1. Compile the application using `-prof-gen=srcpos` (Linux and macOS*) or `/Qprof-gen:srcpos` (Windows), and/or `/Qcov-gen` (Windows) option.

This step generates an instrumented executable and a corresponding static profile information (`pgopti.spi`) file when the `[Q]prof-gen=srcpos` option is used. When the `/Qcov-gen` option is used, minimum instrumentation only for code coverage and generation of .spi file is enabled.

NOTE
You can specify both the `/Qprof-gen=srcpos` and `/Qcov-gen` options on the command line. The higher level of instrumentation needed for profile feedback is enabled along with the profile option for generating the .spi file, regardless of the order the options are specified on the command line.

2. Run the instrumented application.

   This step creates the dynamic profile information (.dyn) file. Each time you run the instrumented application, the compiler generates a unique .dyn file either in the current directory or the directory specified in by the -prof-dir (Linux or macOS*) or /Qprof-dir (Windows) option, or PROF_DIR environment variable. On Windows, you can use the /Qcov-dir or COV_DIR environment variable. These have the same meaning as /Qprof-dir and PROF_DIR.
3. Use the profmerge tool to merge all the .dyn files into one .dpi (`pgopti.dpi`) file.

   This step consolidates results from all runs and represents the total profile information for the application, generates an optimized binary, and creates the dpi file needed by the code coverage tool.

   You can use the profmerge tool to merge the .dyn files into a .dpi file without recompiling the application. The profmerge tool can also merge multiple .dpi files into one .dpi file using the `profmerge -a` option. Select an alternate name for the output .dpi file using the `profmerge -prof_dpi` option.

   Caution
   The profmerge tool merges all .dyn files that exist in the given directory. Confirm that unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code.

4. Run the code coverage tool. The valid syntax and tool options are shown below.

   This step creates a report or exported data as specified. If no other options are specified, the code coverage tool creates a single HTML file (CODE_COVERAGE.HTML) and a sub-directory (CodeCoverage) in the current directory. Open the file in a web browser to view the reports.

   NOTE
   Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

The code coverage tool allows you to name the project and specify paths to specific, necessary files. The following example demonstrates how to name a project and specify .dpi and .spi files to use:

Example: specify .dpi and .spi files

```
codecov -prj myProject -spi pgopti.spi -dpi pgopti.dpi
```

The tool can add a contact name and generate an email link for that contact at the bottom of each HTML page. This provides a way to send an electronic message to the named contact. The following example demonstrates how to add specify a contact and the email links:

Example: add contact information

```
codecov -prj myProject -mname JoeSmith -maddr js@company.com
```

This following example demonstrates how to use the tool to specify the project name, specify the dynamic profile information file, and specify the output format and file name.

Example: export data to text

```
codecov -prj test1 -dpi test1.dpi -txtbcvrg test1_bcvrg.txt
```

Code Coverage Tool Options

Option	Default	Description
`-bcolor`*color*	#FFFF99	Specifies the HTML color name for code in the uncovered blocks.
`-beginblkdsbl`*string*		Specifies the comment that marks the beginning of the code fragment to be ignored by the coverage tool.
`-blockcounts`		When used with `-txtlcov`, reports individual bloc counts for lines that involved multiple blocks.
`-ccolor`*color*	#FFFFFF	Specifies the HTML color name or code of the covered code.
`-comp`*file*		Specifies the file name that contains the list of files being (or not) displayed.
`-counts`		Generates dynamic execution counts.
`-demang`		Demangles both function names and their arguments.
`-dpi`*file*	`pgopti.dpi`	Specifies the file name of the dynamic profile information file (.dpi).
`-endblkdsbl`*string*		Specifies the comment that marks the end of the code fragment to be ignored by the coverage tool.
`-fcolor`*color*	#FFCCCC	Specifies the HTML color name for code of the uncovered functions.
`-help`, `-h`		Prints tool option descriptions.
`-icolor`*color*	#FFFFFF	Specifies the HTML color name or code of the information lines, such as basic-block markers and dynamic counts.
-include-nonexec		Block details will also be listed for functions that did not execute, when used with `-xmlbcvrg[full]` or `-txtbcvrg[full]` options.
`-maddr`*string*	Nobody	Sets the email address of the web-page owner
`-mname`*string*	Nobody	Sets the name of the web-page owner.
`-nopartial`		Treats partially covered code as fully covered code.
`-nopmeter`		Turns off the progress meter. The meter is enabled by default.

Option	Default	Description
-nounwind		Ignores compiler-generated unwind handlers for exception handling cleanup when computing and displaying basic block coverage.
`-onelinedsblstring`		Specifies the comment that marks individual lines of code or the whole functions to be ignored by the coverage tool.
`-pcolor`*color*	#FAFAD2	Specifies the HTML color name or code of the partially covered code.
`-prj`*string*		Sets the project name.
`-ref`		Finds the differential coverage with respect to ref_dpi_file.
-showdirnames		Displays the full path name for source files in the HTML report, instead of just the base filename.
`-spi`*file*	`pgopti.spi`	Specifies the file name of the static profile information file (.spi).

`-srcroot`*dir*		Specifies a different top level project directory than was used during compiler instrumentation run to use for relative paths to source files in place of absolute paths.
		
		NOTE
		In order for the substitution to take place, the sources need to be compiled with one of the following options: `[Q]prof-src-root`, `[Q]prof-src-root-cwd`. This option specifies the base directory that is to be treated as the project root directory.
		
		An example of use is:
		
		`C:> ifort -Qprof-gen:srcpos -Qprof-src-root c:\workspaces\orig_project_dir test1.f90 test2.f90`
		`C:> test1.exe`
		`C:> profmerge`
		`C:> cd \workspaces\`
		`C:> mv orig_project_dir new_project_dir`
		`C:> cd new_project_dir\src`
		`C:> codecov -srcroot C:\workspaces\new_project_dir`
		
		Now, "C:\workspaces\new_project_dir" will be substituted for "c:\workspaces\orig_project_dir" when looking for the source files.
		
		For use of `[Q]prof-src-root`, `[Q]prof-src-root-cwd` options, refer to prof-src-root/Qprof-src-root, prof-src-root-cwd/Qprof-src-root-cwd

`-txtbcvrg`*file*		Export block-coverage for covered functions as text format. The file parameter must be in the form of a valid file name.

Option	Default	Description
`-txtbcvrgfull`*file*		Export block-coverage for entire application in text and HTML formats. The file parameter must be in the form of a valid file name.
`-txtdcg`*file*		Generates the dynamic call-graph information in text format. The file parameter must be in the form of a valid file name.
`-txtfcvrg`*file*		Export function coverage for covered function in text format. The file parameter must by in the form of a valid file name.
`-txtlcov`*file*		Generates line coverage in text format output files, instead of block coverage in HTML output files.
`-ucolor`*color*	#FFFFFF	Specifies the HTML color name or code of the unknown code.
`-xcolor`*color*	#90EE90	Specifies the HTML color of the code ignored.
`-xmlbcvrg`*file*		Export the block-coverage for the covered function in XML format. The file parameter must by in the form of a valid file name.
`-xmlbcvrgfull`*file*		Export function coverage for entire application in XML format in addition to HTML output. The file parameter must be in the form of a valid file name.
`-xmlfcvrg`*file*		Export function coverage for covered function in XML format. The file parameter must be in the form of a valid file name.

Visually Presenting Code Coverage for an Application

Based on the profile information collected from running the instrumented binaries when testing an application, the Intel® compiler will create HTML-formatted reports using the code coverage tool. These reports indicate portions of the source code that were or were not exercised by the tests. When applied to the profile of the performance workloads, the code coverage information shows how well the training workload covers the application's critical code. High coverage of performance-critical modules is essential to taking full advantage of the profile-guided optimizations.

The code coverage tool can create two levels of coverage:

- Top level (for a group of selected modules)
- Individual module source views

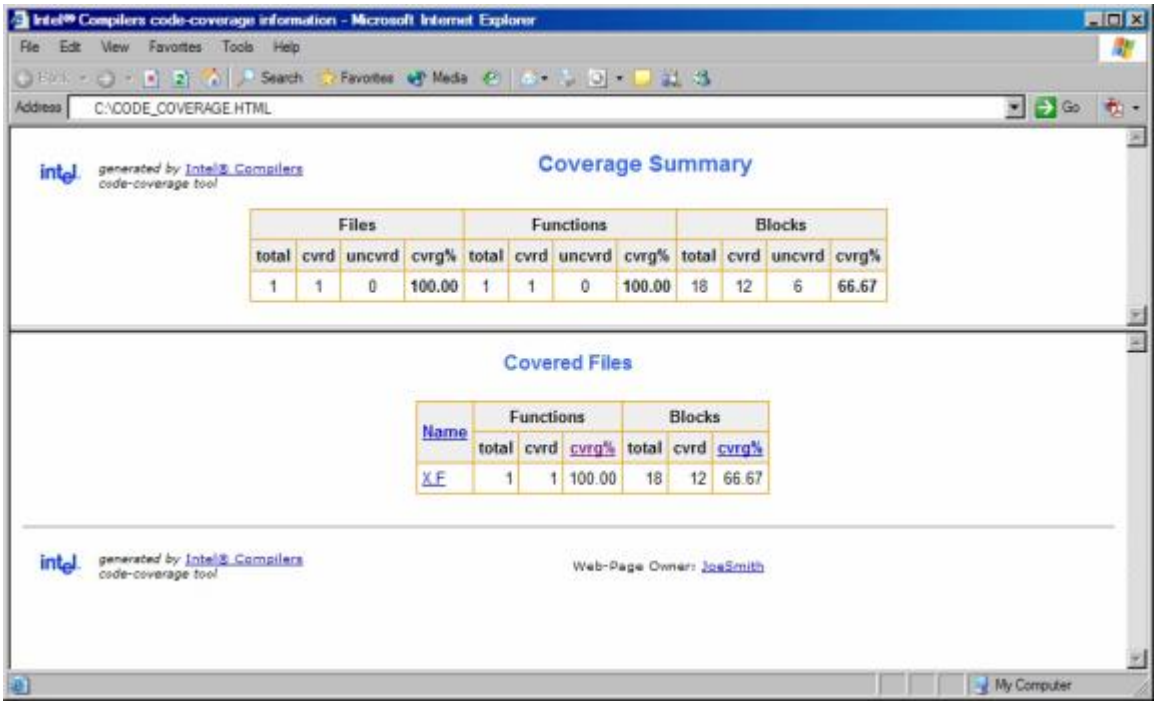
Top-Level Coverage

The top-level coverage reports the overall code coverage of the modules that were selected. The following options are provided:

- Select the modules of interest.
- For the selected modules, the tool generates a list with their coverage information. The information includes the total number of functions and blocks in a module and the portions that were covered.
- By clicking on the title of columns in the reported tables, the lists may be sorted in ascending or descending order based on:
  - Basic-block coverage
  - Function coverage
  - Function name

By default, the code coverage tool generates a single HTML file (CODE_COVERAGE.HTML) and a subdirectory (CodeCoverage) in the current directory. The HTML file defines a frameset to display all of the other generated reports. Open the HTML file in a web-browser. The tool places all other generated report files in a CodeCoverage subdirectory.

If you choose to generate the html-formatted version of the report, you can view coverage source of that particular module directly from a browser. The following figure shows the top-level coverage report.



The coverage tool creates a frame set that allows quick browsing through the code to identify uncovered code. The top frame displays the list of uncovered functions while the bottom frame displays the list of covered functions. For uncovered functions, the total number of basic blocks of each function is also displayed. For covered functions, both the total number of blocks and the number of covered blocks as well as their ratio (that is, the coverage rate) are displayed.

For example, 66.67(4/6) indicates that four out of the six blocks of the corresponding function were covered. The block coverage rate of that function is thus 66.67%. These lists can be sorted based on the coverage rate, number of blocks, or function names. Function names are linked to the position in source view where the function body starts. With one click you can see the least-covered function in the list, and with another click you can see the body of the function. You can scroll down in the source view and browse through the function body.

Individual Module Source View

Within the individual module source views, the tool provides the list of uncovered functions as well as the list of covered functions. The lists are reported in two distinct frames that provide easy navigation of the source code. The lists can be sorted based on:

- Number of blocks within uncovered functions
- Block coverage in the case of covered functions
- Function names

Setting the Coloring Scheme for the Code Coverage

The tool provides a visible coloring distinction of the following coverage categories: covered code, uncovered basic blocks, uncovered functions, partially covered code, and unknown code. The default colors that the tool uses for presenting the coverage information are shown in the tables that follows:

Category	Default	Description
Covered code	#FFFFFF	Indicates code was exercised by the tests. You can override the default color with the `-ccolor` tool option.
Uncovered basic block	#FFFF99	Indicates the basic blocks that were not exercised by any of the tests. However, these blocks were within functions that were executed during the tests. You can override the default color with the `-bcolor` tool option.
Uncovered function	#FFCCCC	Indicates functions that were never called during the tests. You can override the default color with the `-fcolor` tool option.
Partially covered code	#FAFAD2	Indicates that more than one basic block was generated for the code at this position. Some of the blocks were covered and some were not. You can override the default color with the `-pcolor` tool option.
Ignored code	#90EE90	Indicates code that was specifically marked to be ignored. You can override this default color using the `-xcolor` tool option.
Information lines	#FFFFFF	Indicates basic-block markers and dynamic counts. You can override the default color with the `-icolor` tool option.
Unknown	#FFFFFF	Indicates that no code was generated for this source line. Most probably, the source at this position is a comment, a header-file inclusion, or a variable declaration. You can override the default color with the `-ucolor` tool option.

The default colors can be customized to be any valid HTML color name or hexadecimal value using the options mentioned for each coverage category in the table above.

For code coverage colored presentation, the coverage tool uses the following heuristic: source characters are scanned until reaching a position in the source that is indicated by the profile information as the beginning of a basic block. If the profile information for that basic block indicates that a coverage category changes, then the tool changes the color corresponding to the coverage condition of that portion of the code, and the coverage tool inserts the appropriate color change in the HTML-formatted report files.

NOTE
You need to interpret the colors in the context of the code. For example, comment lines that follow a basic block that was never executed would be colored in the same color as the uncovered blocks.

Dynamic Counters

The coverage tool can be configured to generate the information about the dynamic execution counts. This ability can display the dynamic execution count of each basic block of the application and is useful for both coverage and performance tuning.

The custom configuration requires using the `-counts` option. The counts information is displayed under the code after a "^" sign precisely under the source position where the corresponding basic block begins.

If more than one basic block is generated for the code at a source position (for example, for macros), then the total number of such blocks and the number of the blocks that were executed are also displayed in front of the execution count. For example, line 11 in the code is an `IF` statement:

Example

```
11   IF ((N .EQ. 1).OR. (N .EQ. 0))
     ^ 10 (1/2)
12      PRINT N
        ^ 7
```

The coverage lines under code lines 11 and 12 contain the following information:

- The `IF` statement in line 11 was executed 10 times.
- Two basic blocks were generated for the `IF` statement in line 11.
- Only one of the two blocks was executed, resulting in the partial coverage color.
- Only seven out of the ten times variable n had a value of 0 or 1.

In certain situations, it may be desirable to consider all the blocks generated for a single source position as one entity. In such cases, it is necessary to assume that all blocks generated for one source position are covered when at least one of the blocks is covered. This assumption can be configured with the `-nopartial` option. When this option is specified, decision coverage is disabled, and the related statistics are adjusted accordingly. The code lines 11 and 12 indicate that the `print` statement in line 12 was covered. However, only one of the conditions in line 11 was ever true. With the `-nopartial` option, the tool treats the partially covered code (like the code on line 11) as covered.

Differential Coverage

Using the code coverage tool, you can compare the profiles from two runs of an application: a reference run, and a new run identifying the code that is covered by the new run but not covered by the reference run. Use this feature to find the portion of the applications code that is not covered by the applications tests but is executed when the application is run by a customer. It can also be used to find the incremental coverage impact of newly added tests to an applications test space.

Generating Reference Data

Create the dynamic profile information for the reference data, which can be used in differential coverage reporting later, by using the `-ref` option. The following command demonstrate a typical command for generating the reference data:

Example: generating reference data

```
codecov -prj Project_Name -dpi customer.dpi -ref appTests.dpi
```

The coverage statistics of a differential-coverage run shows the percentage of the code exercised on a new run but missed in the reference run. In such cases, the tool shows only the modules that included the code that was not covered. Keep this in mind when viewing the coloring scheme in the source views.

The code with the same coverage property (covered or not covered) on both runs is considered covered code. Otherwise, if the new run indicates that the code was executed, while in the reference run the code was not executed, then the code is treated as uncovered. Alternately, if the code is covered in the reference run but not covered in the new run, the differential-coverage source view shows the code as covered.

Running Differential Coverage

To run the code coverage tool for differential coverage, you must have the application sources, the .spi file, and the .dpi file, as described in the code coverage tool Requirements section (above).

Once the required files are available, enter a command similar to the following begin the process of differential coverage analysis:

Example

```
codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi
```

Specify the .dpi and .spi files using the `-spi` and `-dpi` options.

Excluding Code from Coverage Analysis

The code coverage tool allows you to exclude portions of your code from coverage analysis. This ability can be useful during development; for example, certain portions of code might include functions used for debugging only. The test case should not include tests for functionality that will be unavailable in the final application.

Another example of code that can be excluded is code that might be designed to deal with internal errors unlikely to occur in the application. In such cases, lack of a test case is preferred. You may want to ignore infeasible (dead) code in the coverage analysis. The code coverage tool provides several options for marking portions of the code infeasible and ignoring the code at the file level, function level, line level, and arbitrary code boundaries indicated by user-specific comments. The following sections explain how to exclude code at different levels.

Including and Excluding Coverage at the File Level

The code coverage tool provides the ability to selectively include or exclude files for analysis. Create a component file and add the appropriate string values that indicate the file and directory name for code you want included or excluded. Pass the file name as a parameter of the `-comp` option. The following example shows the general command:

Example: specifying a component file

```
codecov -comp file
```

where *file* is the name of a text file containing strings that act as file/directory name masks for including and excluding file-level analysis. For example, assume the following:

- You want to include all files with the string "source" in the file name or directory name.
- You create a component text file named `myComp.txt` with the selective inclusion string "source".

Once you have a component file, enter a command similar to the following:

Example

```
codecov -comp myComp.txt
```

In this example, filenames with string "source" (like source1.f and source2.f) and all files within directories where the directory name contains the string "source" (like source/file1.f and source2\file2.f) are included in the analysis.

To exclude files or directories, add the tilde (~) prefix to the string. You can specify inclusion and exclusion in the same component file. For example, assume you want to analyze all individual files or files in a directory where the file/directory name includes the string "source", and you want to exclude all individual files and files in directories where the file/directory name includes the string "skip". You add content similar to the following to the component file (`myComp.txt`) and pass it to the `-comp` option:

Example: inclusion and exclusion strings in the myComp.txt file

```
source
~skip
```

Entering the `codecov -comp myComp.txt` command with both instructions in the component file, `myComp.txt`, instructs the tool to:

- Include files with filename containing "source" (like source1.f and source2.f)
- Include all files in directories with the directory name containing "source" (like source/file1.f and source2\file2.f)

- Exclude all files with filename containing "skip" (like skipthis1.f and skipthis2.f)
- Exclude all files in directories with the directory name containing "skip" (like skipthese1\debug1.f and skipthese2\debug2.f)

Excluding Coverage at the Line and Function Level

You can mark individual lines for exclusion my passing string values to the `-onelinedsbl` option. For example, assume that you have some code similar to the following:

Sample code

```
print*, "ERROR: n  = ", n  ! NO_COVER
print*, "       n2 = ", n2  ! INF IA-32 architecture
```

If you wanted to exclude all functions marked with the comments NO_COVER or INF IA-32 architecture, you would enter a command similar to the following.

Example

```
codecov -onelinedsbl NO_COVER -onelinedsbl "INF IA-32 architecture"
```

You can specify multiple exclusion strings simultaneously, and you can specify any string values for the markers; however, you must remember the following guidelines when using this option:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

An entire function can be excluded from coverage analysis using the same methods. For example, the following function will be ignored from the coverage analysis when you issue example command shown above.

Sample code

```
subroutine dumpInfo (n)
integer n ! NO_COVER
...
end subroutine
```

Additionally, you can use the code coverage tool to color the infeasible code with any valid HTML color code by combining the `-onelinedsbl` and `-xcolor` options. The following example commands demonstrate the combination:

Example: combining tool options

```
codecov -onelinedsbl INF -xcolor lightgreen

codecov -onelinedsbl INF -xcolor #CCFFCC
```

Excluding Code by Defining Arbitrary Boundaries

The code coverage tool provides the ability to arbitrarily exclude code from coverage analysis. This feature is most useful where the excluded code either occur inside of a function or spans several functions.

Use the `-beginblkdsbl` and `-endblkdsbl` options to mark the beginning and end (respectively) of any arbitrarily defined boundary to exclude code from analysis. Remember the following guidelines when using these options:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

For example assume that you have the following code:

Sample code

```
integer n, n2
n = 123
n2 = n*n
if (n2 .lt. 0) then
! /* BINF */
  print*, "ERROR: n  = ", n
  print*, "       n2 = ", n2
! // EINF
else if (n2 .eq. 0) then
  print*, "zero: n = ", n, " n2 = ", n2
else
  print*, "positive: n = ", n, " n2 = ", n2
endif
end
```

The following example commands demonstrate how to use the `-beginblkdsbl` option to mark the beginning and the `-endblkdsbl` option to mark the end of code to exclude from the sample shown above.

Example: arbitrary code marker commands

```
codecov -xcolor #ccFFCC -beginblkdsbl BINF -endblkdsbl EINF

codecov -xcolor #ccFFCC -beginblkdsbl "BEGIN_INF" -endblkdsbl "END_INF"
```

Notice that you can combine these options in combination with the `-xcolor` option.

Exporting Coverage Data

The code coverage tool provides specific options to extract coverage data from the dynamic profile information (.dpi files) that result from running instrumented application binaries under various workloads. The tool can export the coverage data in various formats for post-processing and direct loading into databases: the default HTML, text, and XML. You can choose to export data at the function and basic block levels.

There are two basic methods for exporting the data: quick export and combined export. Each method has associated options supported by the tool

- **Quick export:** The first method is to export the data coverage to text- or XML-formatted files without generating the default HTML report. The application sources need not be present for this method. The code coverage tool creates reports and provides statistics only about the portions of the application executed. The resulting analysis and reporting occurs quickly, which makes it practical to apply the coverage tool to the dynamic profile information (the .dpi file) for every test case in a given test space instead of applying the tool to the profile of individual test suites or the merge of all test suites. The `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg` and `-txtbcvrg` options support the first method.
- **Combined export:** The second method is to generate the default HTML and simultaneously export the data to text- and XML-formatted files. This process is slower than first method since the application sources are parsed and reports generated. The `-xmlbcvrgfull` and `-txtbcvrgfull` options support the second method.

These export methods provide the means to quickly extend the code coverage reporting capabilities by supplying consistently formatted output from the code coverage tool. You can extend these by creating additional reporting tools on top of these report files.

Quick Export

The profile of covered functions of an application can be exported quickly using the `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg`, and `-txtbcvrg` options. When using any of these options, specify the output file that will contain the coverage report. For example, enter a command similar to the following to generate a report of covered functions in XML formatted output:

Example: quick export of function data

```
codecov -prj test1 -dpi test1.dpi -xmlfcvrg test1_fcvrg.xml
```

The resulting report will show how many times each function was executed and the total number of blocks of each function, together with the number of covered blocks and the block coverage of each function. The following example shows some of the content of a typical XML report.

XML-formatted report example

```
<PROJECT name = "test1">
   <MODULE name = "D:\SAMPLE.F">
      <FUNCTION name="f0" freq="2">
         <BLOCKS total="6" covered="5" coverage="83.33%"></BLOCKS>
      </FUNCTION>
      ...
   </MODULE>
   <MODULE name = "D:\SAMPLE2.F">
      ...
   </MODULE>
</PROJECT>
```

In the above example, we note that function f0, which is defined in file sample.f, has been executed twice. It has a total number of six basic blocks, five of which are executed, resulting in an 83.33% basic block coverage.

You can also export the data in text format using the `-txtfcvrg` option. The generated text report, using this option, for the above example would be similar to the following example:

Text-formatted report example

```
Covered Functions in File: "D:\SAMPLE.F"
"f0"    2       6       5        83.33
"f1"    1       6       4        66.67
"f2"    1       6       3        50.00
...
```

In the text formatted version of the report, the each line of the report should be read in the following manner:

Column 1	Column 2	Column 3	Column 4	Column 5
Function name	Execution frequency	Line number of the start of the function definition	Column number of the start of the function definition	Percentage of basic-block coverage of the function

Additionally, the tool supports exporting the block level coverage data using the `-xmlbcvrg` option. For example, enter a command similar to the following to generate a report of covered blocks in XML formatted output:

Example: quick export of block data to XML

```
codecov -prj test1 -dpi test1.dpi -xmlbcvrg test1_bcvrg.xml
```

The example command shown above would generate XML-formatted results similar to the following:

XML-formatted report example

```
<PROJECT name = "test1">
  <MODULE name = "D:\SAMPLE.cF90">
    <FUNCTION name="f0" freq="2">
    ...
      <BLOCK line="11" col="2">
        <INSTANCE id="1" freq="1"> </INSTANCE>
      </BLOCK>
      <BLOCK line="12" col="3">
        <INSTANCE id="1" freq="2"> </INSTANCE>
        <INSTANCE id="2" freq="1"> </INSTANCE>
      </BLOCK>
```

In the sample report, notice that one basic block is generated for the code in function f0 at the line 11, column 2 of the file sample.f90. This particular block has been executed only once. Also notice that there are two basic blocks generated for the code that starts at line 12, column 3 of file. One of these blocks, which has id = 1, has been executed two times, while the other block has been executed only once. A similar report in text format can be generated through the `-txtbcvrg` option.

Combined Exports

The code coverage tool has also the capability of exporting coverage data in the default HTML format while simultaneously generating the text- and XML-formatted reports.

Use the `-xmlbcvrgfull` and `-txtbcvrgfull` options to generate reports in all supported formats in a single run. These options export the basic-block level coverage data while simultaneously generating the HTML reports. These options generate more complete reports since they include analysis on functions that were not executed at all. However, exporting the coverage data using these options requires access to application source files and take much longer to run.

Dynamic Call Graphs

Using the `-txtdcg` option the tool can provide detailed information about the dynamic call graphs in an application. Specify an output file for the dynamic call-graph report. The resulting call graph report contains information about the percentage of static and dynamic calls (direct, indirect, and virtual) at the application, module, and function levels.

Test Prioritization Tool

The test prioritization tool, also known as the tselect tool, enables the profile-guided optimizations on all supported Intel® architectures, on Linux*, Windows*, and macOS* operating systems, to select and prioritize tests for an application based on prior execution profiles.

The tool offers a potential of significant time saving in testing and developing large-scale applications where testing is the major bottleneck.

Development often requires changing applications modules. As applications change, developers can have a difficult time retaining the quality of their functional and performance tests so they are current and on-target. The test prioritization tool lets software developers select and prioritize application tests as application profiles change.

The information about the tool is separated into the following sections:

- Features and benefits
- Requirements and syntax
- Usage model
- Tool options
- Running the tool

Features and Benefits

The test prioritization tool provides an effective testing hierarchy based on the code coverage for an application. The following list summarizes the advantages of using the tool:

- Minimizing the number of tests that are required to achieve a given overall coverage for any subset of the application. The tool defines the smallest subset of the application tests that achieve exactly the same code coverage as the entire set of tests.
- Reducing the turn-around time of testing. Instead of spending a long time on finding a possibly large number of failures, the tool enables the users to quickly find a small number of tests that expose the defects associated with the regressions caused by a change set.
- Selecting and prioritizing the tests to achieve certain level of code coverage in a minimal time based on the data of the tests' execution time.

See Understanding Profile-guided Optimization and Profile an Application topics for general information on creating the files needed to run this tool.

Test Prioritization Tool Requirements

The test prioritization tool needs the following items to work:

- The .spi file generated by Intel® compilers when compiling the application for the instrumented binaries with the `-prof-gen=srcpos` (Linux* and macOS*) or `/Qprof-gen:srcpos` (Windows*) option.
- The .dpi files generated by the profmerge tool as a result of merging the dynamic profile information .dyn files of each of the application tests. Run the profmerge tool on all .dyn files that are generated for each individual test and name the resulting .dpi in a fashion that uniquely identifies the test.
- User-generated file containing the list of tests to be prioritized. For successful instrumented code run, you should:
- Name each test .dpi file so the file names uniquely identify each test.
- Create a .dpi list file, which is a text file that contains the names of all .dpi test files.

  Each line of the .dpi list file should include one, and only one .dpi file name. The name can optionally be followed by the duration of the execution time for a corresponding test in the dd:hh:mm:ss format.

  For example: `Test1.dpi 00:00:60:35` states that Test1 lasted 0 days, 0 hours, 60 minutes and 35 seconds.

  The execution time is optional. However, if it is not provided, then the tool will not prioritize the test for minimizing execution time. It will prioritize to minimize the number of tests only.

Caution
The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code. The tool uses the following general syntax:

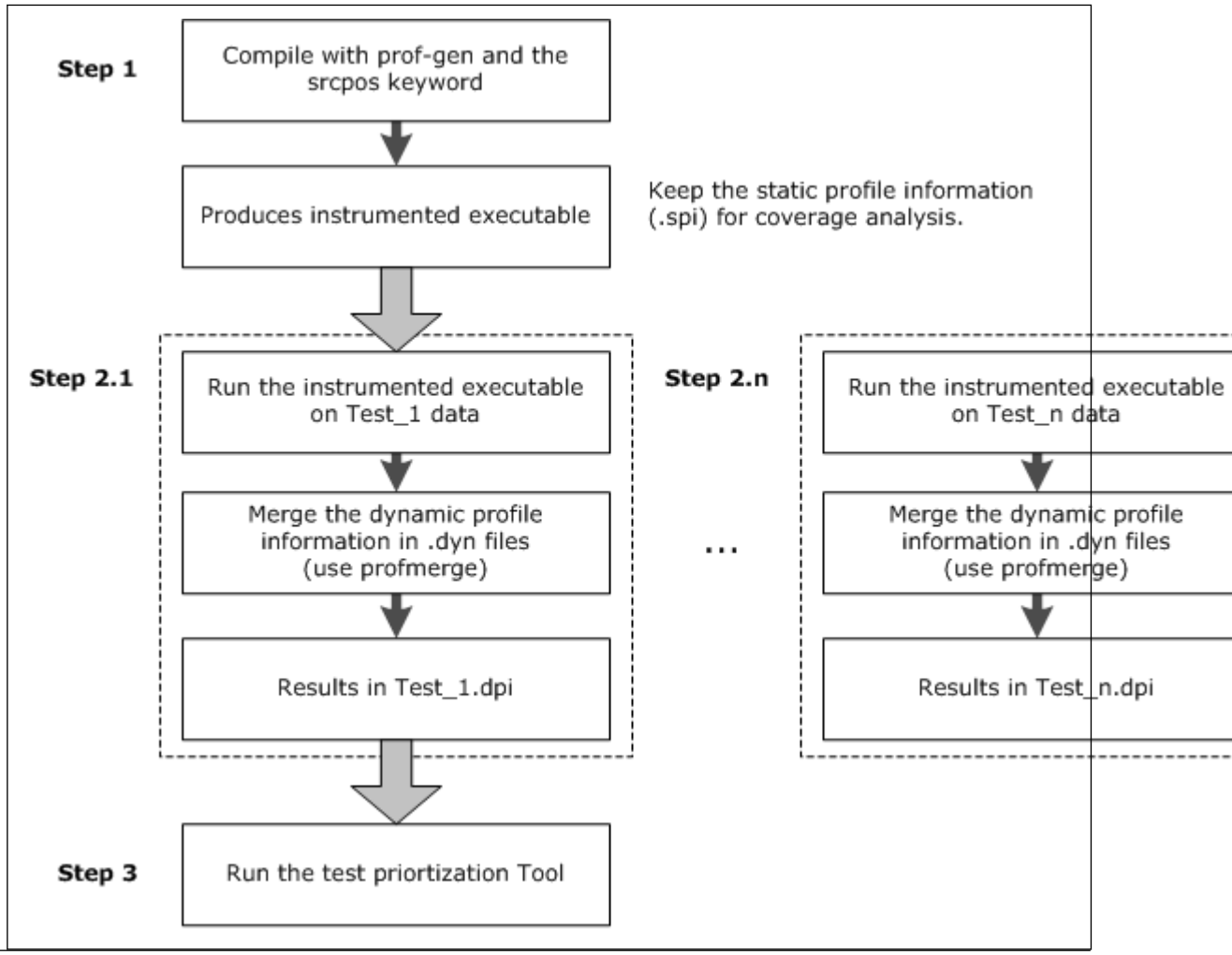
Tool Syntax
`tselect -dpi_list`*file*

`-dpi_list` is a required tool option that sets the path to the list file containing the list of the all .dpi files. All other tool commands are optional.

NOTE
Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

Usage Model

The following figure illustrates a typical test prioritization tool usage model.



Test Prioritization Tool Options

The tool uses the options that are listed in the following table:

Option	Description
`-help`	Prints tool option descriptions.
`-dpi_list`*file*	Required. Specifies the name of the file that contains the names of the dynamic profile information (`.dpi`) files. Each line of the file must contain only one .dpi file name, which can be followed by its execution time (optional). The name must uniquely identify the test.

Option	Description
`-spi`*file*	Specifies the file name of the static profile information file (.SPI). Default is `pgopti.spi`
`-o`*file*	Specifies the file name of the output report file.
`-comp`*file*	Specifies the file name that contains the list of files of interest.
`-cutoff`*value*	Instructs the tool to terminate when the cumulative block coverage reaches a preset percentage, as specified by `value`, of pre-computed total coverage. `value` must be greater than 0.0 (for example, 99.00) but not greater than 100. `value` can be set to 100.
`-nototal`	Instructs the tool to ignore the pre-compute total coverage process.
`-mintime`	Instructs the tool to minimize testing execution time. The execution time of each test must be provided on the same line of `dpi_list` file, after the test name in dd:hh:mm:ss format.
`-srcbasedir`*dir*	Specifies a different top-level project directory than was used during compiler instrumentation run with the `prof-src-root` compiler option to support relative paths to source files in place of absolute paths.
`-verbose`	Instructs the tool to generate more logging information about program progress.

Running the Tool

The following steps demonstrate one simple example for running the tool on IA-32 architectures.

1. Specify the directory by entering a command similar to the following:

Example
`set PROF_DIR=c:\myApp\prof-dir`

2. Compile the program and generate instrumented binary by issuing commands similar to the following:

Operating System	Command
Linux and macOS*	`ifort -prof-gen=srcpos myApp.f90`
Windows	`ifort /Qprof-gen:srcpos myApp.f90`

   The commands shown above compile the program and generate instrumented binary `myApp`, as well as the corresponding static profile information `pgopti.spi`.

3. Confirm that unrelated `.dyn` files are not present by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

4. Run the instrumented files by issuing a command similar to the following:

Example

```
myApp < data1
```

The command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by the `-prof-dir` step above.

5. Merge all `.dyn` file into a single file by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test1.dpi
```

The profmerge tool merges all the .dyn files into one file (`Test1.dpi`) that represents the total profile information of the application on Test1.

6. Confirm again there are no unrelated .dyn files present a second time by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

7. Run the instrumented application, and generate one or more new dynamic profile information files that have an extension .dyn in the directory specified in the `prof-dir` step above by issuing a command similar to the following:

Example

```
myApp < data2
```

8. Merge all .dyn files into a single file by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test2.dpi
```

At this step, the profmerge tool merges all the .dyn files into one file (`Test2.dpi`) that represents the total profile information of the application on Test2.

9. Confirm that there are no unrelated .dyn files present for the final time by issuing a command similar to the following:

Example

```
rm prof-dir \*.dyn
```

10. Run the instrumented application and generate one or more new dynamic profile information files that have an extension .dyn in the directory specified by `-prof-dir` by issuing a command similar to the following:

Example

```
myApp < data3
```

11. Merge all .dyn file into a single file, by issuing a command similar to the following:

Example

```
profmerge -prof_dpi Test3.dpi
```

At this step, the profmerge tool merges all the .dyn files into one file (`Test3.dpi`) that represents the total profile information of the application on Test3.

12. Create a file named `tests_list` with three lines. The first line contains `Test1.dpi`, the second line contains `Test2.dpi`, and the third line contains `Test3.dpi`.

Tool Usage Examples

When these items are available, the test prioritization tool may be launched from the command line in the `prof-dir` directory as described in the following examples.

Example 1: Minimizing the Number of Tests

The following example describes how minimize the number of test runs.

Example Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi
```

where the `-spi` option specifies the path to the .spi file.

The following sample output shows typical results.

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
 num  %RatCvrg  %BlkCvrg  %FncCvrg  Test Name @ Options
 ---  --------  --------  --------  -------------------
   1   87.50     45.65     37.50    Test3.dpi
   2   100.50    52.17     50.00    Test2.dpi
```

In this example, the results provide the following information:

- By running all three tests, you achieve 52.17% block coverage and 50.00% function coverage.
- Test3 alonecovers 45.65% of the basic blocks of the application, which is 87.50% of the total block coverage that can be achieved from all three tests.
- By adding Test2, you achieve a cumulative block coverage of 52.17% or 100% of the total block coverage of Test1, Test2, and Test3.
- Elimination of Test1 has no negative impact on the total block coverage.

Example 2: Minimizing Execution Time

Assume you have the following execution time of each test in the `tests_list` file:

Sample Output

```
Test1.dpi 00:00:60:35
Test2.dpi 00:00:10:15
Test3.dpi 00:00:30:45
```

The following command minimizes the execution time by passing the `-mintime` option:

Sample Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

The following sample output shows possible results:

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
Total execution time    =  1:41:35
 num  elaspedTime  %RatCvrg  %BlkCvrg  %FncCvrg  Test Name @ Options
 ---  -----------  --------  --------  --------  -------------------
   1   10:15        75.00     39.13     25.00    Test2.dpi
   2   41:00        100.00    52.17     50.00    Test3.dpi
```

In this case, the results indicate that running all tests sequentially would require one hour, 45 minutes, and 35 seconds, while the selected tests would achieve the same total block coverage in only 41 minutes.

The order of tests when based on minimizing time (first Test2, then Test3) may be different than when prioritization is done based on minimizing the number of tests. See Example 1 shown above: first Test3, then Test2. In Example 2, Test2 is the test that gives the highest coverage per execution time, so Test2 is picked as the first test to run.

Using Other Options

The `-cutoff` enables the tool to exit when it reaches a given level of basic block coverage. The following example demonstrates how to use the option:

Example

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

If the tool is run with the cutoff value of 85.00, as in the above example, only Test3 will be selected, as it achieves 45.65% block coverage, which corresponds to 87.50% of the total block coverage that is reached from all three tests.

The tool does an initial merging of all the profile information to figure out the total coverage that is obtained by running all the tests. The `-cutoff` enables you to skip this step. In such a case, only the absolute coverage information will be reported, as the overall coverage remains unknown.

Profmerge and Proforder Tools

Profmerge Tool

Use the profmerge tool to merge dynamic profile information (`.dyn`) files and any specified summary files (`.dpi`). The compiler executes profmerge automatically during the feedback compilation phase when you specify the `[Q]prof-use` option.

The command-line usage for profmerge is as follows:

Syntax
`profmerge [-prof_dir dir_name]`

The tool merges all `.dyn` files in the current directory, or the directory specified by `-prof_dir`, and produces a summary file: `pgopti.dpi`.

> **NOTE**
> The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example -prof_dir) instead of the hyphen used by compiler options (for example `[Q]prof-dir`) to join words. Also, on Windows* systems, the tool options are preceded by a hyphen ("-") unlike Windows* compiler options, which are preceded by a forward slash ("/").

You can use profmerge tool to merge .dyn files into a `.dpi` file without recompiling the application. You can run the instrumented executable file on multiple systems to generate `.dyn` files, and optionally use profmerge with the `-prof_dpi` option to name each summary `.dpi` file created from the multiple `.dyn` files.

Because the profmerge tool merges all the `.dyn` files that exist in the given directory, confirm that unrelated `.dyn` files are not present; otherwise, profile information will be based on invalid profile data, which can negatively impact the performance of optimized code.

Profmerge Options

The profmerge tool supports the following options:

Tool Option	Description
`-dump`	Displays profile information.
`-help`	Lists supported options.
`-nologo`	Disables version information. This option is supported on Windows* only.
`-exclude_files`*`files`*	Excludes functions from the profile if the function comes from one of the listed files. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.
`-exclude_funcs`*`functions`*	Excludes functions from the profile. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.

Tool Option	Description
`-prof_dir`*dir*	Specifies the directory from which to read `.dyn` and `.dpi` files, and write the `.dpi` file. Alternatively, you can set the environment variable`PROF_DIR`.
`-prof_dpi`*file*	Specifies the name of the `.dpi` file being generated.
`-prof_file`*file*	Merges information from file matching: dpi_file_and_dyn_tag.
`-src_old`*dir*`-src_new`*dir*	Changes the directory path stored within the `.dpi` file.
`-no_src_dir`	Uses only the file name and not the directory name when reading dyn/dpi records. If you specify `-no_src_dir`, the directory name of the source file will be ignored when deciding which profile data records correspond to a specific application routine, and the `-src-root` option is ignored.
`-src-root`*dir*	Specifies a directory path prefix for the root directory where the user's application files are stored. This option is ignored if you specify `-no_src_dir`.
`-a`*file1.dpi...fileN.dpi*	Specifies and merges available `.dpi` files.
`-verbose`	Instructs the tool to display full information during merge.
`-weighted`	Instructs the tool to apply an equal weighting (regardless of execution times) to the `.dyn` file values to normalize the data counts. This keyword is useful when the execution runs have different time durations and you want them to be treated equally.
`-gen_weight_spec` *file*	Instructs the tool to generate a text file containing a list of the `.dyn` and `.dpi` file that were merged with default weight=1/run_count. The text file is created in the directory specified by the `prof_dir` option.
`-weight_spec` *weight_spec.txt*	Instructs the profmerge tool to generate and use the text file, *weight_spec.txt,* listing individual `.dyn`/`.dpi` files or directory names along with weight values for them. When the `-weight_spec` option is used: • A new `.dpi` file is always created • Only files called out by the specification file are merged

Tool Option	Description
	• .dyn timestamps are ignored and merge always takes place The prof_dir option controls where the input/output *weight_spec.txt* is located, and the destination of the .dpi file. The -weight_spec option overrides: • Any values of -a option • Any computation from using -weighted option

Weighting the Runs

Using the -weight_spec option results in a new .dpi file. Only the files listed in the text file are merged. No files in the current directory are used unless they are included in the text file.

Relocating source files using profmerge

The Intel® Fortran Compiler uses the full path to the source file for each routine to look up the profile summary information associated with that routine. By default, this prevents you from:

- Using the profile summary file (.dpi) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

You can disable the use of directory names when reading .dyn/.dpi file records by specifying the profmerge option -no_scr_dir. This profmerge option is the same as the compiler option -no-prof-src-dir (Linux* and macOS*) and /Qprof-src-dir- (Windows*).

To enable the movement of application sources, as well as the sharing of profile summary files, you can use the profmerge option -src-root to specify a directory path prefix for the root directory where the application files are stored. Alternatively, you can specify the option pair -src_old-src_new to modify the data in an existing summary dpi file. For example:

Example: relocation command syntax

```
profmerge -prof_dir <dir1> -src_old <dir2> -src_new <dir3>
```

where <*dir1*> is the full path to dynamic information file (.dpi), <*dir2*> is the old full path to source files, and <*dir3*> is the new full path to source files. The example command (above) reads the pgopti.dpi file, in the location specified in <*dir1*>. For each routine represented in the pgopti.dpi file, whose source path begins with the <*dir2*> prefix, profmerge replaces that prefix with <*dir3*>. The pgopti.dpi file is updated with the new source path information.

You can run profmerge more than once on a given pgopti.dpi file. For example, you may need to do this if the source files are located in multiple directories:

Operating System	Command Examples
Linux* and macOS*	`profmerge -prof_dir -src_old /src/prog_1 -src_new /src/prog_2` `profmerge -prof_dir -src_old /proj_1 -src_new /proj_2`
Windows*	`profmerge -src_old "c:/program files" -src_new "e:/program files"` `profmerge -src_old c:/proj/application -src_new d:/app`

In the values specified for -src_old and -src_new, uppercase and lowercase characters are treated as identical in Windows. Likewise, forward slash (/) and backward slash (\) characters are treated as identical.

NOTE
Because the source relocation feature of profmerge modifies the `pgopti.dpi` file, consider making a backup copy of the file before performing the source relocation.

Proforder Tool

The proforder tool is used as part of the feedback compilation phase, to improve program performance. Use proforder to generate a function order list for use with the `/ORDER` linker option in Windows. The tool uses the following syntax:

Syntax
`proforder [-prof_dir dir] [-o file]`

where *dir* is the directory containing the profile files (`.dpi` and `.spi`), and *file* is the optional name of the function order list file. The default name is `proford.txt` .

NOTE
The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example `-prof_dir`) instead of the hyphen used by compiler options (for example `[Q]prof-dir`) to join words. Also, on Windows* systems, the tool options are preceded by a hyphen ("-") unlike Windows* compiler options, which are preceded by a forward slash ("/").

Proforder Options

The proforder tool supports the following options:

Tool Option	Default	Description
`-help`		Lists supported options.
`-nologo`		Disables version information. This option is supported on Windows* only.
`-omit_static`		Instructs the tool to omit static functions from function ordering.
`-prof_dir`*dir*		Specifies the directory where the `.spi` and `.dpi` file reside.
`-prof_dpi`*file*		Specifies the name of the `.dpi` file.
`-prof_file`*string*		Selects the `.dpi` and `.spi` files that include the substring value in the file name matching the values passed as string.
`-prof_spi`*file*		Specifies the name of the `.spi` file.
`-o`*file*	`proford.txt`	Specifies an alternate name for the output file.

See Also

Supported Environment Variables

Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations

Instead of doing a full multi-file interprocedural build of your application by using the compiler option `[Q]ipo`, you can obtain some of the benefits by having the compiler and linker work together to make global decisions about where to place the procedures and data in your application. These optimizations are not supported on macOS* systems.

The following table lists each optimization, the type of procedures or global data it applies to, and the operating systems and architectures that it is supported on.

Optimization	Type of Procedure or Data	Supported OS and Architectures
Function Order Lists: Specifies the order in which the linker should link the non-static routines (procedures) of your program. This optimization can improve application performance by improving code locality and reduce paging. Also see Comparison of Function Order Lists and IPO Code Layout.	`EXTERNAL` procedures and library procedures only (not other types of static procedures).	Windows: all Intel architectures Linux: not supported
Function Grouping: Specifies that the linker should place the extern and static routines (procedures) of your program into hot or cold program sections. This optimization can improve application performance by improving code locality and reduce paging. **NOTE** This option will cause functions to be placed into the linker sections named ".text.hot" and ".text.unlikely." If you are using a custom linker script, you will need to specify memory placement for these sections.	`EXTERNAL` procedures and static procedures only (not library procedures).	Linux: IA-32 and Intel 64 architectures Windows: not supported
Function Ordering: Enables ordering of static and extern routines using profile information. Specifies the order in which the linker should link the routines (procedures) of your program. This optimization can improve application performance by improving code locality and reduce paging.	`EXTERNAL` procedures and static procedures only (not library procedures)	Linux and Windows: all Intel architectures

Optimization	Type of Procedure or Data	Supported OS and Architectures
Data Ordering: Enables ordering of static global data items (data in common blocks, module variables, and variables for which the compiler applied the SAVE attribute or statement) based on profiling information. Specifies the order in which the linker should link global data of your program. This optimization can improve application performance by improving the locality of static global data, reduce paging of large data sets, and improve data cache use.	Static global data (data in common blocks, module variables, and variables for which the compiler applied the SAVE attribute or statement) only	Linux and Windows: all Intel architectures

You can only use one of the function-related ordering optimizations listed above on each application. However, you can use the Data Ordering optimization with any one of the function-related ordering optimizations listed above, such as Data Ordering with Function Ordering, or Data Ordering with Function Grouping. In this case, specify the `prof-gen` option keyword `globdata` (needed for Data Ordering) instead of `srcpos` (needed for function-related ordering).

The following sections show the commands needed to implement each of these optimizations: function order list, function grouping, function ordering, and data ordering. For all of these optimizations, omit the `[Q]ipo` or equivalent compiler option.

Generating a Function Order List (Windows)

This section provides an example of the process for generating a function order list. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata`. You would enter commands similar to the following to generate and use a function order list for your Windows application.

1. Compile your program using the `/Qprof-gen:srcpos` option. Use the `/Qprof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

   Example commands

   ```
   ifort /exe:myprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.f90 file2.f90
   ```

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

   Example commands

   ```
   myprog.exe
   ```

3. Before this step, copy all .dyn and .dpi files into the same directory. Merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file. Use the `/prof_dir` option to specify the directory location of the .dyn files.

   Example commands

   ```
   profmerge /prof_dir c:\profdata
   ```

4. Generate the function order list using the proforder tool. By default, the function order list is produced in the file `proford.txt`.

Example commands
`proforder /prof_dir c:\profdata /o myprog.txt`

5. Compile the application with the generated profile feedback by specifying the `ORDER` option to the linker. Use the `/Qprof-dir` option to specify the directory location of the profile files.

Example commands
`ifort /exe:myprog Qprof-dir c:\profdata file1.f90 file2.f90 /link /ORDER:@MYPROG.txt`

Using Function Grouping (Linux)

This section provides a general example of the process for using the function grouping optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `profdata`. You would enter commands similar to the following to use a function grouping for your Linux application.

1. Compile your program using the `-prof-gen` option. Use the `-prof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Example commands
`ifort -o myprog -prof-gen -prof-dir ./profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Example commands
`./myprog`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tools to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-func-group` option to request the function grouping as well as the `-prof-use` option to request feedback compilation. Again, use the `-prof-dir` option to specify the location of the profile files.

Example commands
`ifort /exe:myprog file1.f90 file2.f90 -prof-func-group -prof-use -prof-dir ./profdata`

NOTE On Linux, the –prof-func-group option is on by default when –prof-use is selected.

Finer grain control over the number of functions placed into the hot region can be controlled with the `-prof-hotness-threshold` compiler option, see the command line reference for more details.

Using Function Ordering

This section provides an example of the process for using the function ordering optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`, and that you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to generate and use function ordering for your application.

1. Compile your program using the `-prof-gen=srcpos` (Linux) or `/Qprof-gen:srcpos` (Windows) option. Use the `[Q]prof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`ifort -o myprog -prof-gen=srcpos -prof-dir ./profdata file1.f90 file2.f90`
Windows	`ifort /exe:myprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tools to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `[Q]prof-func-order` option to request the function ordering, as well as the `[Q]prof-use` option to request feedback compilation. Again, use the `[Q]prof-dir` option to specify the location of the profile files.

Operating System	Example commands
Linux	`ifort -o myprog -prof-dir ./profdata file1.f90 file2.f90 -prof-func-order-prof-use`
Windows	`ifort /exe:myprog /Qprof-dir c:\profdata file1.f90 file2.f90 /Qprof-func-order /Qprof-use`

Using Data Ordering

This section provides an example of the process for using the data order optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`, and that you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to use data ordering for your application.

1. Compile your program using the `-prof-gen=globdata` (Linux) or `/Qprof-gen:globdata` (Windows) option. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`ifort -o myprog -prof-gen=globdata -prof-dir ./profdata file1.f90 file2.f90`
Windows	`ifort /exe:myprog /Qprof-gen:globdata /Qprof-dir c:\profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. If you specified a location other than the current directory, change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tools to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `[Q]prof-data-order` option to request the data ordering as well as the `[Q]prof-use` option to request feedback compilation. Again, use the `[Q]prof-dir` option to specify the location of the profile files.

Operating System	Example commands
Linux	`ifort -o myprog -prof-dir ./profdata file1.f90 file2.f90 -prof-data-order-prof-use`
Windows	`ifort /exe:myprog Qprof-dir c:\profdata file1.f90 file2.f90 /Qprof-data-order/Qprof-use`

Comparison of Function Order Lists and IPO Code Layout

The Intel® compiler provides two methods of optimizing the layout of functions in the executable:

- Using a function order list
- Using the `/Qipo` (Windows) compiler option

Each method has advantages. A function order list, created with proforder, lets you optimize the layout of non-static functions (external and library functions whose names are exposed to the linker).

The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Function Order List Effects

Function Type	IPO Code Layout	Function Ordering with proforder
Static	X	No effect
Extern	X	X
Library	No effect	X

Use the following guidelines to create a function order list:

- The order list only affects the order of non-static functions.
- You must compile with `/Gy` to enable function-level linking. (This option is active if you specify either option `/O1` or `/O2`.)

Compiler Option Mapping Tool

The Intel compiler's Option Mapping Tool provides an easy method to derive equivalent options between Windows* and Linux*operating systems. For example, a Windows developer who is developing an application may want to know the Linux equivalent for the `/Oy-` option. Likewise, the Option Mapping Tool provides Windows OS equivalents for Intel compiler options supported on Linux.

NOTE
The Compiler Option Mapping Tool does not support selecting OS* X. However, because Linux and macOS* use the same options in almost all cases, specify "tlinux" if targeting macOS*.

Using the Compiler Option Mapping Tool

You can start the Option Mapping Tool from the command line by:

- Invoking the compiler and using the `/Qmap-opts` option
- Invoking the compiler and using the `-map-opts` option
- Executing the tool directly

NOTE
The Compiler Option Mapping Tool only maps compiler options on the same architecture. It will not map an option that is specific to the Intel® 64 architecture to a like option available on the IA-32 architecture.

Calling the Option Mapping Tool with the Compiler

If you use the compiler to execute the Option Mapping Tool, the following syntax applies:

```
<compiler command> <map-opts option> <compiler option(s)>
```

Example: Finding the Linux equivalent for `/Oy-`

```
ifort /Qmap-opts /Oy-
Intel(R) Compiler option mapping tool
mapping Windows options to Linux for Fortran
'-Qmap-opts' Windows option maps to
  --> '-map-opts' option on Linux
  --> '-map_opts' option on Linux
'-Oy-' Windows option maps to
  --> '-fomit-frame-pointer-' option on Linux
  --> '-fno-omit-frame-pointer' option on Linux
  --> '-fp' option on Linux
```

Example: Finding the Windows equivalent for `-fp`

```
ifort -map-opts -fp
Intel(R) Compiler option mapping tool
mapping Linux options to Windows for Fortran
'-map-opts' Linux option maps to
  --> '-Qmap-opts' option on Windows
  --> '-Qmap_opts' option on Windows
'-fp' Linux option maps to
  --> '-Oy-' option on Windows
```

NOTE
Output from the Option Mapping Tool also includes:

- Option mapping information (not shown here) for options included in the compiler configuration file
- Alternate forms of the options that are supported but may not be documented

When you call the Option Mapping Tool with the compiler, the source file is not compiled.

Calling the Option Mapping Tool Directly

Use the following syntax to execute the Option Mapping Tool directly from a command line environment where the full path to the `map_opts` executable is known (compiler `bin` directory):

```
map_opts -t<target OS> -l<language> -opts <compiler option(s)>
```

where values for:

- `<target OS> = {l|linux|w|windows}`
- `<language> = {f|fortran|c}`

Example: Finding the Linux equivalent for `/Oy-`

```
map_opts -tl -lf -opts /Oy-
Intel(R) Compiler option mapping tool
mapping Windows options to Linux for Fortran
'-Oy-' Windows option maps to
  --> '-fomit-frame-pointer-' option on Linux
  --> '-fno-omit-frame-pointer' option on Linux
  --> '-fp' option on Linux
```

Example: Finding the Windows equivalent for `-fp`

```
map_opts -tw -lf -opts -fp
Intel(R) Compiler option mapping tool
mapping Linux options to Windows for Fortran
'-fp' Linux option maps to
  --> '-Oy-' option on Windows
```

Part VIII

Compatibility and Portability

Portability Considerations Overview

This section presents topics to help you understand how language standards, operating system differences, and computing hardware influence your use of Intel® Fortran and the portability of your programs.

Your program is portable if you can implement it on one hardware-software platform and then move it to additional systems with minimum changes to the source code. Correct results on the first system should be correct on the additional systems. The number of changes you might have to make when moving your program varies significantly. You might have no changes at all (strictly portable), or enough that it is more efficient to design or implement a new program (non-portable customization). Most programs in their lifetime will need to be ported from one system to another, and this section can help you write code that makes this easy.

See Also

Understanding Fortran Language Standards Overview
Minimizing Operating System-Specific Information
Storing and Representing Data
Formatting Data for Transportability
IFPORT Portability Library

Understanding Fortran Language Standards

Understanding Fortran Language Standards Overview

A language standard specifies the form and establishes the interpretation of programs expressed in the language. Its primary purpose is to promote portability of programs across a variety of systems among vendors and users.

The vendor-user community has adopted a series of major Fortran language standards. The primary organizations that develop and publish the standards are the InterNational Committee for Information Technology Standards (INCITS) and International Standards Organization (ISO).

The major Fortran language standards are:

- Fortran 2018

  The Fortran Standards group has completed work on a revision to Fortran 2008, which is called Fortran 2018. There is a Fortran 2018 draft international standard expected to be published as the next ISO/IEC standard in early 2019, which introduces further support for interoperability with the C language, including assumed type and assumed rank arrays, C descriptors, and the RANK intrinsic.
- Fortran 2008

  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1:2010, Information technology -- Programming languages -- Fortran. This standard introduces support for submodules and coarrays, and includes various performance enhancements such as the DO CONCURRENT construct. For more information on supported Fortran 2008 language features, see the Intel® Fortran Language Reference.
- Fortran 2003

  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1:2004, Information technology -- Programming languages -- Fortran. This standard introduces extended support for floating-point exception handling, object-oriented programming, and improved interoperability with the C language. For more information on supported Fortran 2003 features, see the Intel® Fortran Language Reference.
- Fortran 95

  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1: 1997(E), Information technology -- Programming languages -- Fortran. This standard introduces certain language elements and corrections into Fortran 90. Fortran 95 includes Fortran 90 and most features of FORTRAN 77. For information about differences between Fortran 95 and Fortran 90, see the Intel® Fortran Language Reference.
- Fortran 90

  American National Standard Programming Language Fortran, ANSI X3.198-1992 and International Standards Organization, ISO/IEC 1539: 1991, Information technology -- Programming languages -- Fortran. This standard emphasizes modernization of the language by introducing new developments. For information about differences between Fortran 90 and FORTRAN 77, see the Intel® Fortran Language Reference.
- FORTRAN 77

  American National Standard Programming Language FORTRAN, ANSI X3.9-1978. This standard added new features based on vendor extensions to FORTRAN 66 and addressed problems associated with large-scale projects, such as improved control structures.
- FORTRAN 66

  American National Standard Programming Language FORTRAN, ANSI X3.9-1966. This was the first attempt to standardize the languages called FORTRAN by many vendors. The language was based heavily on IBM's FORTRAN IV language.

Although a language standard seeks to define the form and the interpretation uniquely, a standard may not cover all areas of interpretation. It may also include some ambiguities. You need to carefully craft your program in these cases to insure that you get the desired answers when producing a portable program.

See Also

Language Standards Conformance

Using Standard Features and Extensions

Standard Features

Use standard language features to achieve the greatest degree of portability for your Intel® Fortran programs. You can design a robust implementation to improve the portability of your program, or you can choose to use extensions to the standard to increase the readability, functionality, and efficiency of your programs.

You can request that the compiler warn you about program syntax that violates the standard's numbered syntax rules and constraints. While this does not insure that the program as a whole is standard-conforming, it can help to avoid many possible compatibility issues. The /stand (Windows*) or -std (Linux* and macOS*) options enable this checking, and you can specify the desire standard to check against. If you do not specify a standard, Fortran 2008 is used.

You can use the `standard-semantics` compiler option to enable all of the options that implement the current Fortran Standard behavior of the compiler where those differ from the compiler's default.

Standard Extensions

Not all extensions to the Fortran standard cause problems when porting to other platforms. Many extensions are supported on a wide range of platforms, and if a system you are porting a program to supports an extension, there is no reason to avoid using it. There is no guarantee, however, that the same feature on another system will be implemented in the same way as with Intel® Fortran. Only the Fortran standard is guaranteed to coexist uniformly on all platforms.

The Intel® Fortran Compiler supports many language extensions on multiple platforms, including Linux*, macOS*, and Microsoft Windows* operating systems. The Intel® *Fortran Language Reference Manual* identifies whether each language element is supported on other platforms.

It is a good programming practice to declare any external procedures either in an EXTERNAL statement or in a procedure interface block, for the following reasons:

- The newer Fortran standards have added many new intrinsic procedures to the language.

  Programs that conformed to earlier Fortran Standards (such as FORTRAN 77) may include non-intrinsic functions or subroutines having the same name as new Fortran Standard procedures.
- Some processors include nonstandard intrinsic procedures that might conflict with procedure names in your program.

If you do not explicitly declare the external procedures and the name duplicates an intrinsic procedure, the processor calls the intrinsic procedure, not your external routine. For more information on how the Intel® Fortran Compiler resolves name definitions, see Resolving Procedure References.

See Also

stand compiler option
standard-semantics compiler option
Resolving Procedure References

Using Compiler Optimizations

Many Fortran compilers perform code-generation optimizations to increase the speed of execution or to decrease the required amount of memory for the generated code. Although the behaviors of both the optimized and non-optimized programs fall within the language standard specification, different behaviors can occur in areas not covered by the language standard. Compiler optimization can influence floating-point numeric results.

The Intel® Fortran Compiler can perform optimizations to increase execution speed and to improve floating-point numerical consistency.

Floating-point consistency refers to obtaining results consistent with the IEEE binary floating-point standards. For more information, refer to the `/fp:consistent` option of fp-model.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

See Also

fp-model, fp compiler option

Conformance, Compatibility, and Fortran Features

Language Standards Conformance

The Fortran standard has undergone several revisions since its initial publication as FORTRAN 66 (also known as FORTRAN IV). Subsequent revisions have been FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018. Each revision has added new features; some revisions have labeled features as "deprecated" (or obsolescent) or they have removed them. Intel® Fortran continues to support deprecated and deleted features.

Intel® Fortran conforms to the Fortran 2008 standard (ISO/IEC 1539-1:2010), Fortran 2003 standard (ISO/IEC 1539-1:2004), American National Standard Fortran 95 (ANSI X3J3/96-007)[1], and American National Standard Fortran 90 (ANSI X3.198-1992).[2]

It also includes support for some features from the Fortran 2018 standard.

The Fortran Standards committee is currently answering questions of interpretation on Fortran 2008 language features. Any answers given by the committee that are related to features implemented in Intel® Fortran may result in changes in future releases of the Intel® Fortran Compiler, even if the changes produce incompatibilities with earlier releases of Intel® Fortran. The Fortran 2018 standard is in the final stages of development but subject to change. Changes to the Fortran 2018 standard may produce incompatibilities in future releases of Intel® Fortran with respect to the Fortran 2018 features included in this release.

Intel Fortran provides a number of extensions to the Fortran 2008 Standard. In the language reference, extensions (non-standard features) are displayed in this color.

Intel Fortran also includes support for programs that conform to the previous Fortran standards (ANSI X3.9-1978 and ANSI X3.0-1966), the International Standards Organization standard ISO 1539-1980 (E), the Federal Information Processing Institute standard FIPS 69-1, and the Military Standard 1753 Language Specification.

[2] This is the same as International Standards Organization standard ISO/IEC 1539-1:1997 (E).
[3] This is the same as International Standards Organization standard ISO/IEC 1539:1991 (E).

Language Compatibility

Intel® Fortran is highly source-compatible with Compaq* Visual Fortran on supported systems, and it is substantially source-compatible with DEC* Fortran 90 and VAX* FORTRAN 77.

Fortran 2018 Features

The following Fortran 2018 features are new in this release:

- Enhancements to the IMPLICIT NONE statement allow specifying that all external procedures must be declared as EXTERNAL.
- Enhancements to the GENERIC statement permit it to be used to declare generic interfaces.
- You can now specify locality for variables in a DO CONCURRENT statement.
- Enhancements to edit descriptor forms E and D, EN, ES, and G allow a field width of zero, analogous to the F edit descriptor.
- The exponent width e in a data edit descriptor can now be zero, analogous to a field width of zero.
- The RN edit descriptor now rounds to nearest as specified by Fortran 2018 and ISO/IEC/IEEE 60559:2011.
- The EX edit descriptor allows for hexadecimal format output of floating-point values, and hexadecimal format floating-point values are allowed on input.
- SIZE= can be specified for non-advancing I/O.
- The values for SIZE= and POS= in an INQUIRE statement for pending asynchronous operations have been standardized.
- The value assigned to the RECL= specifier in an INQUIRE statement now has standardized values.
- A new form of the intrinsic function CMPLX does not require the KIND= keyword if the first argument is type COMPLEX.
- The arguments to the SIGN function can be of different kinds.
- The named constants STAT_FAILED_IMAGE and STAT_UNLOCKED_FAILED_IMAGE have been defined in the intrinsic ISO_FORTRAN_ENV modules.
- The named constant kind type C_PTRDIFF_T has been added to the intrinsic module ISO_C_BINDING.
- The non-block DO statement and the arithmetic IF statement are now deleted in Fortran 2018. Intel® Fortran fully supports features deleted in the Fortran Standard.
- COMMON, EQUIVALENCE and BLOCKDATA statements are now obsolescent.
- The labeled form of a DO loop is now obsolescent.
- Specific names of intrinsic procedures are now obsolescent.
- The following atomic subroutines have been implemented: ATOMIC_ADD, ATOMIC_AND, ATOMIC_CAS, ATOMIC_FETCH_ADD, ATOMIC_FETCH_AND, ATOMIC_FETCH_OR, ATOMIC_FETCH_XOR, ATOMIC_OR, and ATOMIC_XOR.
- The following collective subroutines have been implemented: CO_BROADCAST, CO_MAX, CO_MIN, CO_REDUCE, and CO_SUM.
- The SELECT RANK construct has been implemented allowing manipulation of assumed rank dummy arguments..
- The compiler will now diagnose the use of nonstandard intrinsic procedures and modules as required by Fortran 2018.
- To comply with the latest Fortran 2018 standards, C_F_PROCPOINTER is now IMPURE.
- Transformational intrinsic functions from the intrinsic modules ISO_C_BINDING, IEEE_ARITHMETIC, and IEEE_EXCEPTIONS are now allowed in specification expressions.
- You can now specify optional argument RADIX for the IEEE_GET_ROUNDING_MODE and IEEE_SET_ROUNDING_MODE intrinsic module procedures.
- IEEE_ROUND_TYPE and IEEE_AWAY have been added to the IEEE_ARITHMETIC intrinsic module.
- The optional ROUND argument has been added to the IEEE_RINT function defined in the intrinsic module IEEE_ARITHMETIC.
- The intrinsic module IEEE_ARITHMETIC now includes the functions IEEE_FMA, IEEE_SIGNBIT, IEEE_NEXT_UP, and IEEE_NEXT_DOWN.
- The intrinsic module IEEE_EXCEPTIONS now contains a new derived type, IEEE_MODES_TYPE, which can be used to save and restore the IEEE_MODES using the IEEE_GET_MODES and the IEEE_SET_MODES intrinsic module procedures.
- SUBNORMAL is now synonymous with DENORMAL.
- The following intrinsic module procedures have been implemented:

  - IEEE_MAX_NUM, IEEE_MAX_NUM_MAG, IEEE_MIN_NUM, and IEEE_MIN_NUM_MAG
  - IEEE_QUIET_EQ, IEEE_QUIET_GE, IEEE_QUIET_GT, IEEE_QUIET_LE, IEEE_QUIET_LT, IEEE_QUIET_NE, IEEE_SIGNALING_EQ, IEEE_SIGNALING_GE, IEEE_SIGNALING_GT, IEEE_SIGNALING LE, IEEE_SIGNALING_LT, and IEEE_SIGNALING_NE
  - IEEE_SUPPORT_SUBNORMAL
- An optional STAT= specifier has been added to ATOMIC_REF and ATOMIC_DEFINE intrinsic procedures.
- Optional STAT= and ERRMSG= specifiers have been added to the MOVE_ALLOC intrinsic procedure, to image selectors, and to the CRITICAL statement and construct.
- INTEGER and LOGICAL arguments to intrinsic procedures are no longer required to be of default kind.
- The intrinsic module procedures IEEE_INT and IEEE_REAL have been implemented.
- The FAIL IMAGE statement has been implemented. It allows debugging recovery code for failed images without having to wait for an actual image failure.
- Intrinsic functions FAILED_IMAGES, IMAGE_STATUS, and STOPPED_IMAGES have been implemented, but they do not have full functionality in this release.

  The argument "team" is not yet implemented for these intrinsics. This functionality will be added in a future release.

The following Fortran 2018 features are also supported:

- The COSHAPE intrinsic function

  Returns the cobounds of a coarray argument.
- You can now add module names, OpenMP reduction identifiers, and defined I/O generic specs to PUBLIC and PRIVATE statements.
- Features to provide synchronization capabilities between images:

  The EVENT_QUERY intrinsic subroutine

  The EVENT_POST and EVENT_WAIT statements

  The derived type EVENT_TYPE in the ISO_FORTRAN_ENV intrinsic module
- Enhancements to the IMPORT statement

  The IMPORT statement can now be used in internal subprograms and BLOCK constructs to control host association. There are three new forms:

  IMPORT, ALL

  IMPORT, NONE

  IMPORT, ONLY: *import-name-list*
- Default accessibility of module entities

  A module name that is accessed via used association can appear in a PUBLIC or PRIVATE *access-id-list* . It can be used to set the default accessibility for all accessible entities from that module.
- C language interoperability

  An assumed-rank array is now permitted to be an argument to the C_SIZEOF intrinsic function.

  With the exception of C_F_POINTER, all standard procedures in the ISO_C_BINDING intrinsic module are now PURE.
- Enhancement to the NON_RECURSIVE keyword

  This keyword allows procedures to be declared as not recursive. The default in previous Fortran standards was that procedures were non-recursive unless declared RECURSIVE. Fortran 2018 changes that default. Intel© Fortran has implemented the NON_RECURSIVE keyword, but the default compilation mode for this release remains non-recursive. This will change in a future release.
- I/O enhancement

  The G0.d edit descriptor can be used to specify the output of integer, logical, and character data, using the rules for I0, L1, and A edit descriptors, respectively.
- Further Interoperability of Fortran with C, including assumed-type, assumed-rank, C descriptors, and the RANK intrinsic.
- The ERROR STOP statement is allowed in PURE procedures

For information about the Fortran standards, visit the Fortran standards technical committee website at http://j3-fortran.org/.

See Also
New Language Features
Fortran 2003 Features
Fortran 2008 Features

Fortran 2008 Features

The following Fortran 2008 features are new in this release:

- Specification expressions can now contain user-defined operators.
- Direct access user-defined I/O has been implemented.

The following Fortran 2008 features are also supported:

- The FINDLOC intrinsic function
- You can specify optional argument BACK in MAXLOC and MINLOC intrinsic functions.
- You can declare multiple type-bound procedures in a PROCEDURE list.
- Intrinsic assignment to an allocatable polymorphic variable is allowed if the variable is not a coarray. For more information, see Examples of Intrinsic Assignment to Polymorphic Variables.
- The intrinsic module functions COMPILER_OPTIONS and COMPILER_VERSION
- Subscripts and nested implied-DO limits in a data statement can be any constant expression instead of being limited to combinations of constants, implied-DO variables, and intrinsic operations.
- The intrinsic functions ACOS, ACOSH, ASIN, ASINH, ATAN, ATANH, COSH, SINH, TAN, and TANH now take arguments of type COMPLEX. The intrinsic function ATAN (Y, X) is defined as ATAN2 (Y, X).
- A pointer dummy argument with INTENT(IN) can be argument associated with a non-pointer actual argument with the TARGET attribute.
- A reference to a function that returns a data pointer is treated as a variable and is permitted in any variable-definition context .
- Allocatable components are allowed to be of recursive type, that is, the type being defined, or of a forward type, that is, a derived type that is not yet defined.
- In an ALLOCATE statement, you can specify more than one allocate object for a given SOURCE= or MOLD=. Also, if you do not specify upper and lower bounds in an ALLOCATE statement's *s-spec* list, the shape of the array is that of SOURCE=*source-expr* or MOLD=*source-expr*.
- Initialization expressions are now referred to as constant expressions.
- An EXIT statement can appear in a BLOCK, IF, ASSOCIATE, SELECT CASE, or SELECT TYPE construct.
- You can specify intrinsic types in a TYPE statement
- You can specify implied-shape array specifications for named constant arrays
- You can specify %re and %im designators for the real and imaginary parts of values of COMPLEX type
- Pointer Initialization
- Binary, octal, and hexadecimal constants are allowed for one or more arguments in intrinsic routines BGE, BGT, BLE, BLT, CMPLX, DBLE, DSHIFTL, DSHIFTR, IAND, IEOR, INT, IOR, MERGE_BITS, and REAL.
- You can specify an optional integer datatype for the *subscript-name* in the *triplet-spec* in the FORALL statement.
- An ALLOCATE statement can give a polymorphic variable the shape and type of another variable without copying the value. This is done with MOLD= replacing SOURCE=.
- Except for procedure arguments and pointer dummy arguments, a PURE function may have a dummy argument with the VALUE attribute without the explicit INTENT(IN) attribute.
- Dummy arguments of ELEMENTAL procedures may appear in specification expressions in the procedure.
- You can specify an optional RADIX argument for the intrinsic function SELECTED_REAL_KIND and the intrinsic module procedure IEEE_SELECTED_REAL_KIND.
- Submodules
- IMPURE keyword
- The EXECUTE_COMMAND_LINE subroutine
- The BLOCK construct
- The FUNCTION and SUBROUTINE keywords on the END statement are optional for internal procedures and module procedures.
- A negative value that is not zero but rounds to zero on output is displayed with a leading minus sign

- Generic resolution of procedures where one dummy argument has the ALLOCATABLE attribute and the other has the POINTER attribute without INTENT (IN), or where one is a procedure and the other is a data object.
- The ENTRY statement is an obsolescent feature.
- A source statement can begin with one or more semicolon characters.
- Coarray intrinsic routines: ATOMIC_DEFINE and ATOMIC_REF
- A polymorphic MOLD= specifier for ALLOCATE
- Coarrays (Windows* and Linux* only)
- Image control statements: SYNC ALL, SYNC IMAGES, SYNC MEMORY, CRITICAL, LOCK, and UNLOCK
- Coarray intrinsic routines: IMAGE_INDEX, LCOBOUND, NUM_IMAGES, THIS_IMAGE, and UCOBOUND
- CRITICAL construct
- Maximum array rank of 15 (Intel® Fortran allows 31 dimensions)
- G0 and G0.d format edit descriptors
- FINAL routines
- GENERIC, OPERATOR, and ASSIGNMENT overloading in type-bound procedures
- A generic interface may have the same name as a derived type
- Bounds specification and bounds remapping list on a pointer assignment
- In formatting, a * indicates an unlimited repeat count
- NEWUNIT= specifier in OPEN
- A CONTAINS section can be empty
- Attributes CODIMENSION and CONTIGUOUS
- Coarrays can be specified in ALLOCATABLE, ALLOCATE, and TARGET statements
- MOLD keyword in ALLOCATE
- DO CONCURRENT statement
- ERROR STOP statement
- Intrinsic functions BESSEL_J0, BESSEL_J1, BESSEL_JN, BESSEL_Y0, BESSEL_Y1, BESSEL_YN, BGE, BGT, BLE, BLT, DSHIFTL, DSHIFTR, ERF, ERFC, ERFC_SCALED, GAMMA, HYPOT, IALL, IANY, IPARITY, IS_CONTIGUOUS, LEADZ, LOG_GAMMA, MASKL, MASKR, MERGE_BITS, NORM2, PARITY, POPCNT, POPPAR, SHIFTA, SHIFTL, SHIFTR, STORAGE_SIZE, TRAILZ
- ISO_FORTRAN_ENV module constants ATOMIC_INT_KIND, ATOMIC_LOGICAL_KIND, CHARACTER_KINDS, INTEGER_KINDS, INT8, INT16,INT32, INT64, LOGICAL_KINDS, REAL_KINDS, REAL32, REAL64, REAL128, STAT_LOCKED, STAT_LOCKED_OTHER_IMAGE, STAT_UNLOCKED
- ISO_FORTRAN_ENV type LOCK_TYPE
- SCALAR keyword for ALLOCATED

For information about the Fortran standards, visit the Fortran standards technical committee website at http://j3-fortran.org/.

See Also

New Language Features
Fortran 2003 Features
Fortran 2018 Features

Fortran 2003 Features

Fortran 2003 is fully supported, including the following features:

- Parameterized derived types with KIND and LENGTH type parameters and the %KIND and %LEN type parameter designators
- A polymorphic dummy argument that has the attribute INTENT(OUT) becomes UNDEFINED or it will have DEFAULT INITIALIZATION applied.
- User-Defined Derived-Type I/O
- A polymorphic SOURCE= specifier for ALLOCATE
- Bounds specification and bounds remapping list on a pointer assignment
- FINAL routines
- GENERIC, OPERATOR, and ASSIGNMENT overloading in type-bound procedures
- Enumerators
- Type extension (not polymorphic)
- Type-bound procedures

- Allocatable scalar variables (not deferred-length character)
- ERRMSG keyword for ALLOCATE and DEALLOCATE
- SOURCE= keyword for ALLOCATE
- Character arguments for MAX, MIN, MAXVAL, MINVAL, MAXLOC, and MINLOC
- Intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC and IEEE_FEATURES
- ASSOCIATE construct
- DO CONCURRENT construct
- PROCEDURE declaration
- Procedure pointers
- ABSTRACT INTERFACE
- PASS and NOPASS attributes
- CONTIGUOUS attribute
- Structure constructors with component names and default initialization
- Array constructors with type and character length specifications
- I/O keywords BLANK, DECIMAL, DELIM, ENCODING, IOMSG, PAD, ROUND, SIGN, and SIZE
- Format edit descriptors DC, DP, RD, RC, RN, RP, RU, and RZ
- The COUNT_RATE argument to the SYSTEM_CLOCK intrinsic may be a REAL of any kind.
- NAMELIST I/O for internal files
- Intrinsic functions EXTENDS_TYPE_OF and SAME_TYPE_AS
- SELECT TYPE construct
- CLASS declaration
- PUBLIC types with PRIVATE components and PRIVATE types with PUBLIC components
- RECORDTYPE setting STREAM_CRLF
- A file can be opened for stream access (ACCESS='STREAM')
- Specifier POS can be specified in an INQUIRE, READ, or WRITE statement
- BIND attribute and statement
- Language binding can be specified in a FUNCTION or SUBROUTINE statement, or when defining a derived type
- IS_IOSTAT_END intrinsic function
- IS_IOSTAT_EOR intrinsic function
- INTRINSIC and NONINTRINSIC can be specified for modules in USE statements
- ASYNCHRONOUS attribute and statement
- VALUE attribute and statement
- Specifier ASYNCHRONOUS can be specified in an OPEN, INQUIRE, READ, or WRITE statement
- An ID can be specified for a pending data transfer operation
- FLUSH statement
- WAIT statement
- IMPORT statement
- NEW_LINE intrinsic function
- SELECTED_CHAR_KIND intrinsic function
- Intrinsic modules ISO_C_BINDING and ISO_FORTRAN_ENV
- Specifiers ID and PENDING can be specified in an INQUIRE statement
- User-defined operators can be renamed in USE statements
- MOVE_ALLOC intrinsic subroutine
- PROTECTED attribute and statement
- Pointer objects can have the INTENT attribute
- GET_COMMAND intrinsic
- GET_COMMAND_ARGUMENT intrinsic
- COMMAND_ARGUMENT_COUNT intrinsic
- GET_ENVIRONMENT_VARIABLE intrinsic
- Allocatable components of derived types
- Allocatable dummy arguments
- Allocatable function results
- VOLATILE attribute and statement
- Names of length up to 63 characters
- Statements up to 256 lines
- A named PARAMETER constant may be part of a complex constant
- In all I/O statements, the following numeric values can be of any kind: UNIT=, IOSTAT=
- The following OPEN numeric values can be of any kind: RECL=
- The following READ and WRITE numeric values can be of any kind: REC=, SIZE=

- The following INQUIRE numeric values can be of any kind: NEXTREC=, NUMBER=, RECL=, SIZE=
- Recursive I/O is allowed when the new I/O being started is internal I/O that does not modify any internal file other than its own
- IEEE infinities and Nans are displayed by formatted output as specified by Fortran 2003
- In an I/O format, the comma after a P edit descriptor is optional when followed by a repeat specifier
- The following intrinsics take an optional KIND= argument: ACHAR, COUNT, IACHAR, ICHAR, INDEX, LBOUND, LEN, LEN_TRIM, MAXLOC, MINLOC, SCAN, SHAPE, SIZE, UBOUND, VERIFY
- Square brackets [ ] are permitted to delimit array constructors instead of (/ /)
- The Fortran character set has been extended to contain the 8-bit ASCII characters ~ \ [ ] ` ^ { } | # @

For information about the Fortran standards, visit the Fortran standards technical committee website at http://j3-fortran.org/.

See Also

New Language Features

Fortran 2008 Features

Fortran 2018 Features

Minimizing Operating System-Specific Information

The operating system influences your program both externally and internally. For increased portability, you need to minimize the amount of operating-system-specific information required by your program. The Fortran language standards do not specify this information.

Operating-system-specific information consists of non-intrinsic extensions to the language, compiler and linker options, and possibly the graphical user interface of Windows*. Input and output operations use devices that may be system-specific and may involve a file system with system-specific record and file structures.

The operating system also governs resource management and error handling. You can depend on default resource management and error handling mechanisms or provide mechanisms of your own. For information on special library routines to help port your program from one system to another, see IFPORT Portability Library and related topics.

The minimal interaction with the operating system is for input/output (I/O) operations and usually consists of knowing the standard units preconnected for input and output. You can use default file units with the asterisk (*) unit specifier.

To increase the portability of your programs across operating systems, consider the following:

- Do not assume the use of a particular type of file system.
- Do not embed filenames or paths in the body of your program. Define them as constants at the beginning of the program or read them from input data.
- Do not assume a particular type of standard I/O device or the "size" of that device (number of rows and columns).
- Do not assume display attributes for the standard I/O device. Some environments do not support attributes such as color, underlined text, blinking text, highlighted text, inverse text, protected text, or dim text.

See Also

IFPORT Portability Library

Storing and Representing Data

The Fortran language standard specifies little about the storage of data types.

This loose specification of storage for data types results from a great diversity of computing hardware. This diversity poses problems in representing data and especially in transporting stored data among a multitude of systems. The size (as measured by the number of bits) of a storage unit (a word, usually several bytes) varies from machine to machine. In addition, the ordering of bits within bytes and bytes within words varies from one machine to another. Furthermore, binary representations of negative integers and floating-point representations of real and complex numbers take several different forms.

The simplest and most reliable means of transferring data between dissimilar systems is in *character* and not binary form. Simple programming practices ensure that your data as well as your program is portable.

See Also

Supported Native and Nonnative Numeric Formats

Data Portability

Formatting Data for Transportability

You can achieve the highest transportability of your data by formatting it as 8-bit character data. Use a standard character set, such as the ASCII standard, for encoding your character data. Although this practice is less efficient than using binary data, it will save you from shuffling and converting your data.

If you are transporting your data by means of a record-structured medium, it is best to use the Fortran sequential files. See Record Types in the reference guide for more information.

Remember also that some systems use a carriage return/linefeed pair as an end-of-record indicator, while other systems use linefeed only. There might be system-dependent values embedded within your data that complicate its transport.

Implementing a strictly portable solution requires a careful effort. Maximizing portability may also mean making compromises to the efficiency and functionality of your solution. If portability is not your highest priority, you can use some of the techniques that appear in later sections to ease your task of customizing a solution.

See Also

Methods of Specifying the Data Format
Porting Nonnative Data
Record Types
Supported Native and Nonnative Numeric Formats

Supported Native and Nonnative Numeric Formats

Data storage in different computers uses a convention of either little endian or big endian storage. The storage convention generally applies to numeric values that span multiple bytes, as follows:

Little endian storage occurs when:

- The least significant bit (LSB) value is in the byte with the lowest address.
- The most significant bit (MSB) value is in the byte with the highest address.
- The address of the numeric value is the byte containing the LSB. Subsequent bytes with higher addresses contain more significant bits.

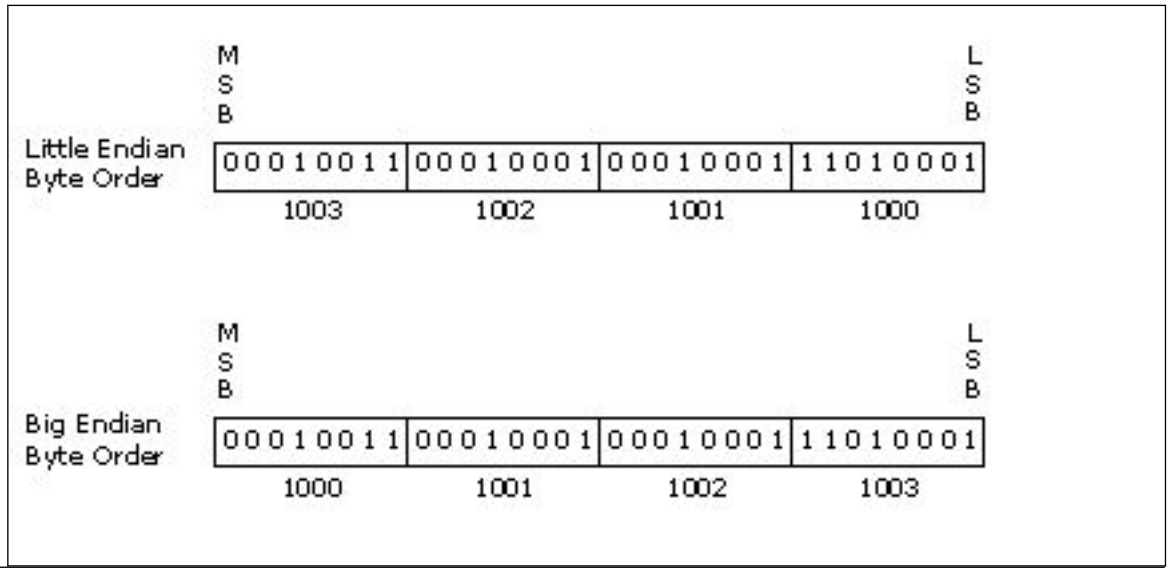
Big endian storage occurs when:

- The least significant bit (LSB) value is in the byte with the highest address.
- The most significant bit (MSB) value is in the byte with the lowest address.
- The address of the numeric value is the byte containing the MSB. Subsequent bytes with higher addresses contain less significant bits.

Intel® Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

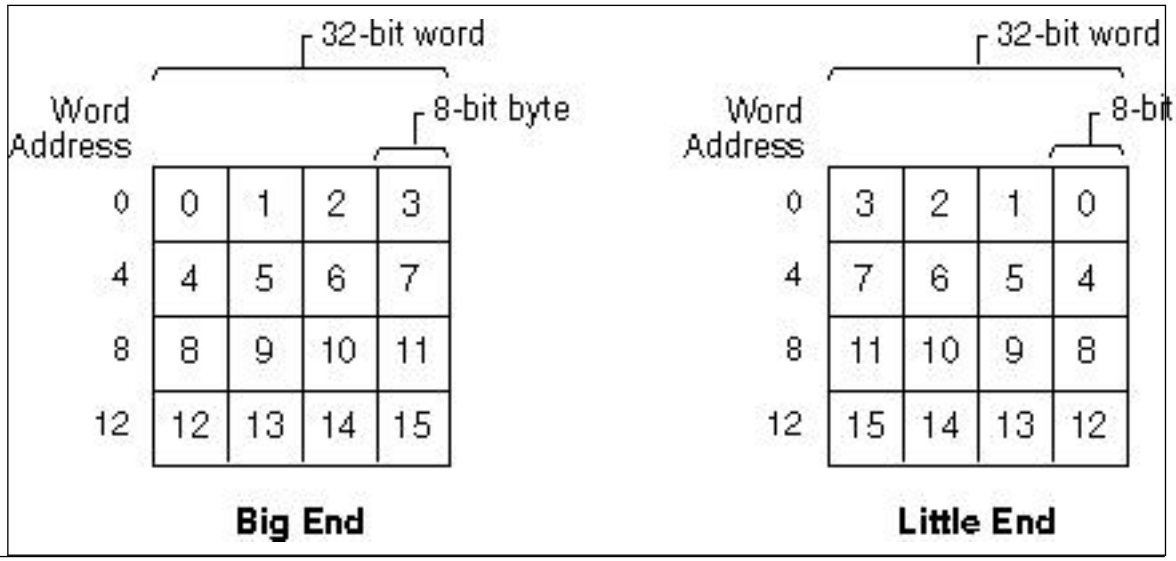
The following figures show the difference between the two byte-ordering schemes:

Little and Big Endian Storage of an INTEGER Value



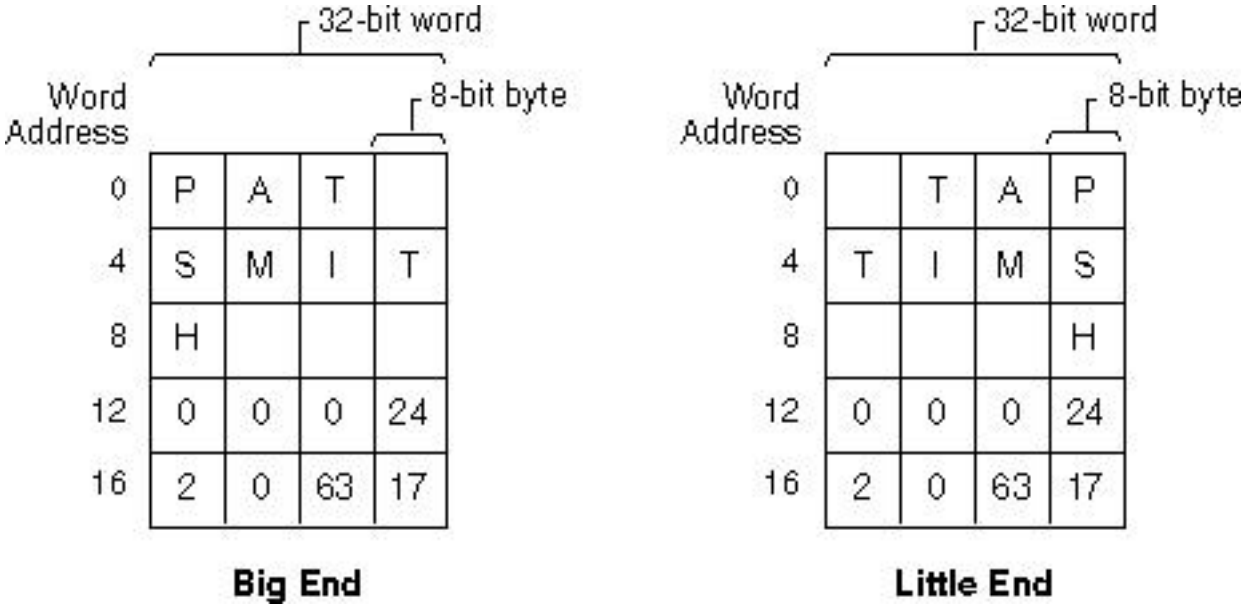
The following figure illustrates the difference between the two conventions for the case of addressing bytes within words.

Byte Order Within Words: (a) Big Endian, (b) Little Endian



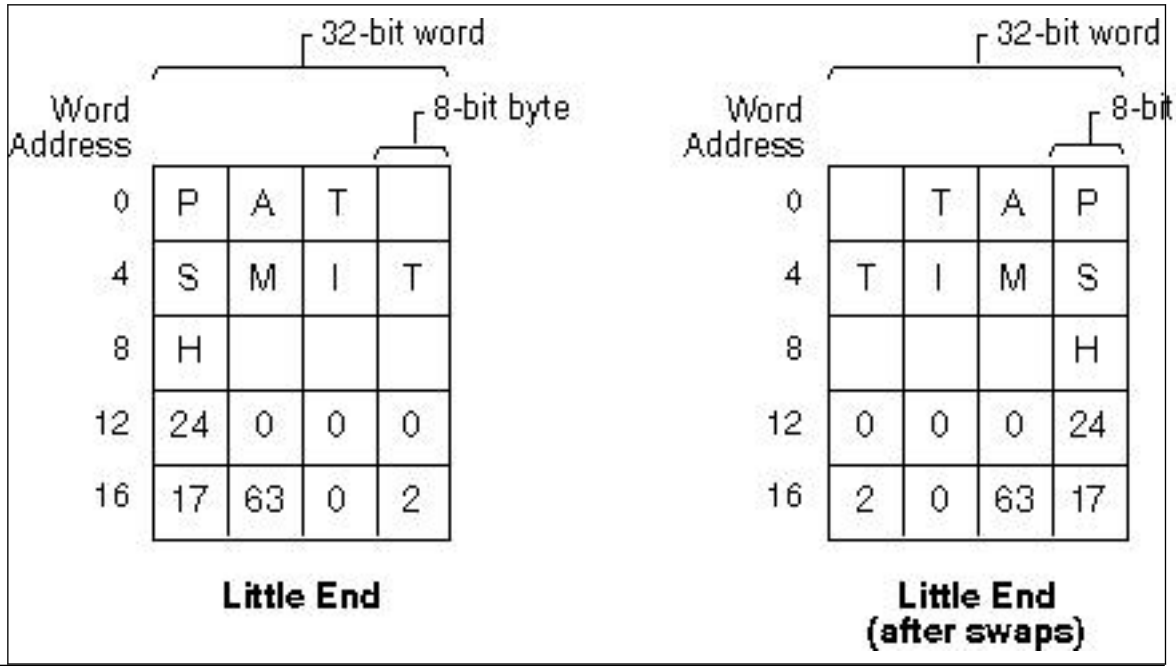
Data types stored as subcomponents (bytes stored in words) end up in different locations within corresponding words of the two conventions. The following figure illustrates the difference between the representations of several data types in the two conventions. Letters represent 8-bit character data, while numbers represent the 8-bit partial contribution to 32-bit integer data.

Character and Integer Data in Words: (a) Big Endian, (b) Little Endian



If you serially transfer bytes now from the big endian words to the little endian words (BE byte 0 to LE byte 0, BE byte 1 to LE byte 1, and so on), the left half of the figure shows how the data ends up in the little endian words. Note that data of size one byte (characters in this case) is ordered correctly, but that integer data no longer correctly represents the original binary values. The right half of the figure shows that you need to swap bytes around the middle of the word to reconstitute the correct 32-bit integer values. After swapping bytes, the two preceding figures are identical.

Data Sent from Big to Little: (a) After Transfer, (b) After Byte Swaps



You can generalize the previous example to include floating-point data types and to include multiple-word data types.

Moving unformatted data files between big endian and little endian computers requires that the data be converted.

Intel Fortran provides the capability for programs to read and write unformatted data (originally written using unformatted I/O statements) in several nonnative floating-point formats and in big endian INTEGER or floating-point format. Supported nonnative floating-point formats include Compaq* VAX* little endian floating-point formats supported by Digital* FORTRAN for OpenVMS* VAX Systems, standard IEEE big endian floating-point format found on most Sun Microsystems* systems and IBM RISC* System/6000 systems, IBM floating-point formats (associated with the IBM's System/370 and similar systems), and CRAY* floating-point formats.

Converting unformatted data instead of formatted data is generally faster and is less likely to lose precision of floating-point numbers. For unformatted files that contain record lengths before and after the data, specifying big-endian conversion will interpret the lengths as big-endian.

Caution:

Intel Fortran support of data conversion applies only to I/O list items of instrinsic type (REAL, INTEGER, and so on). I/O list items of derived type are not converted.

The native memory format includes little endian integers and little endian IEEE floating-point formats, IEEE binary32 for REAL(KIND=4) and COMPLEX(KIND=4) declarations, IEEE binary64 for REAL(KIND=8) and COMPLEX(KIND=8) declarations, and IEEE binary128 for REAL(KIND=16) and COMPLEX(KIND=16) declarations.

The keywords for supported nonnative unformatted file formats and their data types are listed in the following table:

Nonnative Numeric Formats, Keywords, and Supported Data Types

Keyword	Description
BIG_ENDIAN	Big endian integer data of the appropriate size (one, two, four, or eight bytes) and big endian IEEE floating-point formats for REAL and COMPLEX single-, double-, and extended-precision numbers. INTEGER(KIND=1) data is the same for little endian and big endian.
CRAY	Big endian integer data of the appropriate size (one, two, four, or eight bytes) and big endian CRAY proprietary floating-point format for REAL and COMPLEX single- and double-precision numbers.
FDX	Little endian integer data of the appropriate size (one, two, four, or eight bytes) and the following little endian proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE binary128 for REAL (KIND=16) and COMPLEX (KIND=16)
FGX	Little endian integer data of the appropriate size (one, two, four, or eight bytes) and the following little endian proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE binary128 for REAL (KIND=16) and COMPLEX (KIND=16)
IBM	Big endian integer data of the appropriate INTEGER size (one, two, or four bytes) and big endian IBM proprietary (System\370 and similar) floating-point format for REAL and COMPLEX single- and double-precision numbers.
LITTLE_ENDIAN	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following native little endian IEEE floating-point formats: • IEEE binary32 for REAL (KIND=4) and COMPLEX (KIND=4) • IEEE binary64 for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE binary128 for REAL (KIND=16) and COMPLEX (KIND=16) For additional information on supported ranges for these data types, see Native IEEE Floating-Point Representations.
NATIVE	No conversion occurs between memory and disk. This is the default for unformatted files.

Keyword	Description
VAXD	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8) • VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)
VAXG	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) • VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message is displayed.

See Also

Environment Variable F_UFMTENDIAN Method

Porting non-Native Data

When porting nonnative data, consider the following:

- Vendors might use different units for specifying the record length (`RECL` specifier) of unformatted files. While formatted files are specified in units of characters (bytes), unformatted files are specified in longword units for Intel® Fortran (default) and some other vendors.

  To allow you to specify RECL units (bytes or longwords) for unformatted files without source file modification, use the `assume byterecl` compiler option.
- Certain vendors apply different `OPEN` statement defaults to determine the record type. The default record type (RECORDTYPE) with Intel® Fortran depends on the values for the ACCESS and FORM specifiers for the OPEN statement.
- Certain vendors use a different identifier for the logical data types, such as hex `FF` and hex `00` instead of `01` to denote `.TRUE.` and `.FALSE.`. See the `fpscomp:logicals` compiler option.
- Source code being ported may be coded specifically for big endian use.

See Also

assume compiler option

Specifying the Data Format

Methods of Specifying the Data Format

There are a number of methods for specifying a nonnative numeric format for unformatted data:

- Set an environment variable for a specific unit number before the file is opened. The environment variable is named `FORT_CONVERT`*n*, where *n* is the unit number.
- Set an environment variable for a specific file name extension before the file is opened. The environment variable is named `FORT_CONVERT`.*ext* or `FORT_CONVERT_`*ext*, where *ext* is the file name extension (suffix).
- Set an environment variable for a set of units before the application is executed. The environment variable is named `F_UFMTENDIAN`.
- Specify the CONVERT keyword in the OPEN statement for a specific unit number.
- Compile the program with an OPTIONS statement that specifies the CONVERT=*keyword* qualifier. This method affects all unit numbers using unformatted data specified by the program.

- Compile the program with the appropriate compiler option, which affects all unit numbers that use unformatted data specified by the program. Use the `convert` compiler option.

If none of these methods are specified, the native LITTLE_ENDIAN format is assumed (no conversion occurs between disk and memory).

Any keyword listed in Supported Native and Nonnative Numeric Formats can be used with any of these methods, except for the Environment Variable F_UFMTENDIAN Method, which supports only LITTLE_ENDIAN and BIG_ENDIAN.

If you specify more than one method, the order of precedence when you open a file with unformatted data is below:

1. Check for an environment variable (`FORT_CONVERT`*n*) for the specified unit number (applies to any file opened on a particular unit).
2. Check for an environment variable (`FORT_CONVERT.`*ext* is checked before `FORT_CONVERT_`*ext*) for the specified file name extension (applies to all files opened with the specified file name extension).
3. Check for an environment variable (`F_UFMTENDIAN`) for the specified unit number (or for all units).

> **NOTE**
> This environment variable is checked only when the application starts executing.

4. Check the OPEN statement CONVERT specifier.
5. Check whether an OPTIONS statement with a CONVERT:*keyword* qualifier was present when the program was compiled.
6. Check whether the `convert` compiler option was present when the program was compiled.

See Also

convert
Environment Variable F_UFMTENDIAN Method
Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method
Environment Variable FORT_CONVERTn Method
OPEN Statement CONVERT Method
OPTIONS Statement Method
Supported Native and Nonnative Numeric Formats

Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method

Use this method to specify a non-native numeric format for each specified file name extension (suffix). Specify the numeric format at run time by setting the appropriate environment variable before an implicit or explicit `OPEN` to one or more unformatted files. Use the format FORT_CONVERT.*ext* or FORT_CONVERT_*ext* (where *ext* is the file extension or suffix). The FORT_CONVERT.*ext* environment variable is checked before FORT_CONVERT_*ext* environment variable (if *ext* is the same).

For example, assume you have a previously compiled program that reads numeric data from one file and writes to another file using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from a file with a .dat file extension and write that data in native little endian format to a file with an extension of .data. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format (IEEE binary32 and IEEE binary64) when read from `file.dat`, and then written without conversion in native little endian IEEE format to the file with a suffix of `.data`, assuming that environment variables FORT_CONVERT.DATA and FORT_CONVERT*n* (for that unit number) are not defined.

Without requiring source code modification or recompilation of a program, the following command sets the appropriate environment variables before running the program:

Example

```
// Linux*
  setenv FORT_CONVERT.DAT BIG_ENDIAN

// Windows*
  set FORT_CONVERT.DAT=BIG_ENDIAN
```

The FORT_CONVERT*n* method takes precedence over this method. When the appropriate environment variable is set when you open the file, the FORT_CONVERT.*ext* or FORT_CONVERT_*ext* environment variable is used if a FORT_CONVERT*n* environment variable is not set for the unit number.

The FORT_CONVERT*n* and the FORT_CONVERT.*ext* or FORT_CONVERT_*ext* environment variable methods take precedence over the other methods. For instance, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

You can set the appropriate environment variable using the format FORT_CONVERT.*ext* or FORT_CONVERT_*ext*. If you also use Intel® Fortran on Linux* systems, consider using the FORT_CONVERT_*ext* form, because a dot (.) cannot be used for environment variable names on certain Linux* command shells. If you do define both FORT_CONVERT.*ext* and FORT_CONVERT_*ext* for the same extension (*ext*), the file defined by FORT_CONVERT.*ext* is used.

On Windows* systems, the file name extension (suffix) is not case-sensitive. The extension must be part of the file name (not the directory path).

Environment Variable FORT_CONVERT*n* Method

You can use this method to specify a nonnative numeric format for each specified unit number. You specify the numeric format at run time by setting the appropriate environment variable before an implicit or explicit OPEN to that unit number.

When the appropriate environment variable is set when you open the file, the environment variable is always used because this method takes precedence over the other methods. For example, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

For example, assume you have a previously compiled program that reads numeric data from unit 28 and writes it to unit 29 using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from unit 28 and write that data in native little endian format to unit 29. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format when read from unit 28, and then written without conversion in native little endian IEEE format to unit 29.

Without requiring source code modification or recompilation of this program, the following command sequence sets the appropriate environment variables before running a program.

Linux:

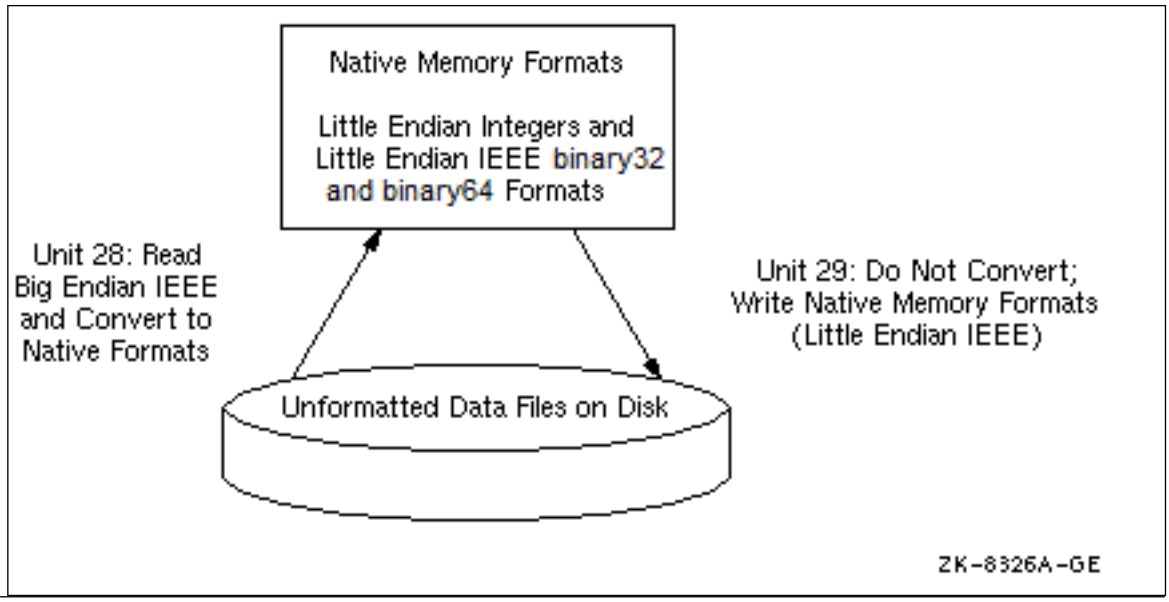
```
setenv FORT_CONVERT28 BIG_ENDIAN
setenv FORT_CONVERT29 NATIVE
```

Windows:

```
set FORT_CONVERT28=BIG_ENDIAN
set FORT_CONVERT29=NATIVE
```

The following figure shows the data formats used on disk and in memory when the program is run after the environment variables are set.

Sample Unformatted File Conversion



This method takes precedence over other methods.

Environment Variable F_UFMTENDIAN Method

This little-endian-big-endian conversion feature is intended for Fortran unformatted input/output operations. It enables the development and processing of files with little-endian and big-endian data organization.

The F_UFMTENDIAN environment variable is processed once at the beginning of program execution. Whatever it specifies for specific units or for all units continues for the rest of the execution.

Specify the numbers of the units to be used for conversion purposes by setting `F_UFMTENDIAN`. Then, the READ/WRITE statements that use these unit numbers will perform relevant conversions. Other READ/WRITE statements will work in the usual way.

General Syntax for F_UFMTENDIAN

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value:

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

where:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- `MODE` defines current format of data, represented in the files; it can be omitted.

  The keyword `little` means that the data has little endian format and will not be converted. This is the default.

  The keyword `big` means that the data has big endian format and will be converted.
- `EXCEPTION` is intended to define the list of exclusions for `MODE.EXCEPTION` keyword (`little` or `big`) defines data format in the files that are connected to the units from the `EXCEPTION` list. This value overrides `MODE` value for the units listed.

The `EXCEPTION` keyword and the colon can be omitted. The default when the keyword is omitted is `big`.

- Each list member `U` is a simple unit number or a number of units. The number of list members is limited to 64.
- `decimal` is a non-negative decimal number less than 232.

Converted data should have basic data types or arrays of basic data types. Derived data types are disabled.

Error messages may be issued during the little-endian-to-big-endian conversion. They are all fatal.

On Linux* systems, the command line for the variable setting in the shell is:

```
Sh: export F_UFMTENDIAN=MODE;EXCEPTION
```

NOTE
The environment variable value should be enclosed in quotes if the semicolon is present.

The environment variable can also have the following syntax:

```
F_UFMTENDIAN=u[,u] . . .
```

Examples

1. `F_UFMTENDIAN=big`

   All input/output operations perform conversion from `big-endian` to `little-endian` on READ and from little-endian to `big-endian` on WRITE.
2. `F_UFMTENDIAN="little;big:10,20"`

   or `F_UFMTENDIAN=big:10,20`

   or `F_UFMTENDIAN=10,20`

   The input/output operations perform big-endian to little endian conversion only on unit numbers 10 and 20.
3. `F_UFMTENDIAN="big;little:8"`

   No conversion operation occurs on unit number 8. On all other units, the input/output operations perform big-endian to little-endian conversion.
4. `F_UFMTENDIAN=10-20`

   The input/output operations perform big-endian to little-endian conversion on units 10, 11, 12 , ... 19, 20.
5. Assume you set `F_UFMTENDIAN=10,100` and run the following program.

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789

! prepare a little endian representation of data

open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

! prepare a big endian representation of data
```

```
open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

! read big endian data and operate with them on
!  little endian machine.

open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4

! Any operation with data, which have been read

! . . .
close(100)
stop
end
```

Now compare `lit.tmp` and `big.tmp` files with the help of the `od` utility:

```
> od -t x4 lit.tmp
0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034
> od -t x4 big.tmp
0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

You can see that the byte order is different in these files.

OPEN Statement CONVERT Method

Use this method to specify a nonnative numeric format for each specified unit number. This method requires an explicit file `OPEN` statement to specify the numeric format of the file for that unit number.

This method takes precedence over the `OPTIONS` statement and the compiler option `CONVERT[:]` *keyword* method, but has a lower precedence than the environment variable methods.

For example, the following source code shows how the `OPEN` statement would be coded to read unformatted VAXD numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20 (the absence of the `CONVERT` keyword or environment variables `FORT_CONVERT20`, `FORT_CONVERT.dat`, or `FORT_CONVERT_dat` indicates native little endian data for unit 20):

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
 ...
OPEN (FILE='graph3_t.dat', FORM='UNFORMATTED', UNIT=20)
```

A hard-coded `OPEN` statement `CONVERT` keyword value cannot be changed after compile time. However, to allow selection of a particular format at run time, equate the `CONVERT` keyword to a variable and provide the user with a menu that allows selection of the appropriate format (menu choice sets the variable) before the `OPEN` occurs.

You can also select a particular format at run time for a unit number by using one of the environment variable methods (Environment Variable FORT_CONVERTn Method, Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method, or Environment Variable F_UFMTENDIAN Method), which take precedence over the `OPEN` statement `CONVERT`*keyword* method.

See Also

OPEN
Environment Variable FORT_CONVERTn Method
Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method
Environment Variable F_UFMTENDIAN Method

OPTIONS Statement Method

You can only specify one numeric file format for all unformatted file unit numbers using this method unless you also use one of the environment variable methods or OPEN statement CONVERT keyword method.

You specify the numeric format at compile time and must compile all routines under the same OPTIONS statement CONVERT=*keyword* qualifier. You could use one source program and compile it using different `ifort` commands to create multiple executable programs that each read a certain format.

The environment variable methods and the OPEN statement CONVERT=*keyword* method take precedence over this method. For example, you might use the `FORT_CONVERT`*n* environment variable or OPEN CONVERT=*keyword* method to specify each unit number that will use a format other than that specified using the `ifort`*option* method.

This method takes precedence over the `ifortconvert[:]`*keyword* compiler option method.

You can use OPTIONS statements to specify the appropriate floating-point formats (in memory and in unformatted files) instead of using the corresponding `ifort` command qualifiers. For example, to use VAX F_floating, G_floating, and H_floating as the unformatted file format, specify the following OPTIONS statement:

```
OPTIONS /CONVERT=VAXG
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another format, unless you use it in combination with one of the environment variable methods or the OPEN statement CONVERT keyword method to specify a different format for a particular unit number.

For more information, see the OPTIONS statement.

See Also

OPTIONS Statement

Compiler Option -convert or /convert Method

You can only specify one numeric format for all unformatted file unit numbers using the `convert` compiler option unless you also use one (or more) of the previous methods shown in Methods of Specifying the Data Format.

You specify the numeric format at compile time and must compile all routines under the same `convert`*keyword* compiler option. You can use the same source program and compile it using different `ifort` commands (or the equivalent in the IDE) to create multiple executable programs that each read a certain format.

If you specify other methods, the other methods take precedence over using this method. For instance, you might use the environment variable or OPEN statement CONVERT specifier method to specify each unit number that will use a format different than that specified using the `convert`*keyword* compiler option method for all other unit numbers.

For example, the following command compiles program `file.for` to use VAX D_floating (and F_floating) floating-point data for all unit numbers (unless superseded by one of the other methods). Data is converted between the file format and the little endian memory format (little endian integers, IEEE binary32, IEEE binary64, and IEEE binary128 little endian IEEE* floating-point format). The created file, `vconvert.exe`, can then be run:

Example

```
// Windows*
  ifort file.for /convert:vaxd /link /out:vconvert.exe
```

Because this method affects all unformatted file unit numbers, you cannot read data in one format and write it in another file format using the `convert`*keyword* compiler option alone. You can if you use it in combination with the environment variable methods or the OPEN statement CONVERT specifier method to specify a different format for a particular unit number.

See Also

convert

Methods of Specifying the Data Format

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z