Intel(R) Fortran Compiler for Linux*
Building Applications

Document Number: 307779-002US

Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Chips, Core Inside, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Pentium Inside, skoool, Sound Mark, The Computer Inside., The Journey Inside, VTune, Xeon, Xeon Inside and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Table Of Contents

Building Applications

Introduction to Building Applications

This document provides information on how to get started with Intel® Fortran, how the compiler operates, and how to develop Intel Fortran applications.

The discussions in this document often contain content that applies generally to all supported operating systems; however, where the expected behavior is significantly different on a specific OS, the appropriate behavior is listed separately. In most cases, features and options supported for IA-32 Linux* systems are also supported on Intel-based Mac systems running Mac OS*. For more detailed information about Intel Compiler support on Mac OS, see the Intel® Compiler Release Notes.

This guide covers the following major topics:

How to build and debug applications:

- Getting Started using the Intel® Fortran Compiler
- Building Applications from the Command Line
- Using Compiler Options
- Debugging Fortran Programs
- Programming with Mixed Languages
- Error Handling Overview

Topics related to data types and data representation:

- Data Representation
- Converting Unformatted Data
- Fortran I/O

Topics related to using Fortran-supplied libraries:

- Creating and Using Libraries
- Libraries Provided by Intel Fortran
- Portability Library
- Math Libraries

How to Use This Document

This documentation assumes that you are familiar with the Fortran Standard programming language and with the Intel® processor architecture. You should also be familiar with the host computer's operating system.

Note

This document explains how information and instructions apply differently to each targeted architecture. If there is no specific indication as to the architecture, the description is applicable for all architectures.

Additional Documentation

In addition to this document, you should also have access to these documents:

- *Intel® Fortran Compiler Options reference*
- *Intel® Fortran Language Reference*
- *Intel® Fortran Libraries Reference*
- *Intel® Fortran Optimizing Applications*
- *Intel® Fortran Release Notes*

For additional technical product information, including white papers about Intel compilers, open the page associated with your product at:

http://developer.intel.com/software/products

Notation Conventions

This manual uses the following conventions.

Intel® Fortran	The name of the common compiler language supported by the Intel® Fortran Compiler products.
Intel® EM64T	The label used to indicate IA-32 systems with Intel® Extended Memory 64 Technology (Intel® EM64T).
`This type style`	Elements of syntax, reserved words, option keywords, variables, file names, and code examples are shown in a monospaced font. The text appears in lowercase unless uppercase is required.
THIS TYPE STYLE	Statements, keywords, and directives are shown in all uppercase, in a normal font. For example, "add the USE statement…"
This type style	Bold normal text shows 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 indicate a sequence of actions. For example, "Click **File>Open**" indicates that in the **File** menu, click **Open** to perform this action.
`This type style`	Bold, monospaced text indicates user input. It shows what you type as a command or input.
`This type style`	Italic, monospaced text indicates placeholders for information that you must supply. Italics are also used to introduce new terms.

[`options`]	Items inside single square brackets are optional. (In some examples, square brackets are used to show arrays.)
{`value` | `value`}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of the items unless all of the items are also enclosed in square brackets.
...	A horizontal ellipsis (three dots) following an item indicates that the item preceding the ellipsis can be repeated. In code examples, a horizontal ellipsis means that not all of the statements are shown.
Linux*	This term refers to information that is valid on all supported Linux* operating systems. An asterisk at the end of a word or name indicates it is a third-party product trademark.
Mac OS*	This term refers to information that is valid on Intel®-based systems running Mac OS*. An asterisk at the end of a word or name indicates it is a third-party product trademark.

The command to invoke the compiler on Linux* and Mac OS* systems is `ifort`.

The default compiler installation path will be used if you do not specify an installation directory when you install the product. The default installation path for most of the Fortran compilers is `/opt/intel/fc/9.1.`*`nnn`*, where *nnn* is an incremental release number. The exception is the Intel® EM64T on Intel® EM64T compiler, which uses a path of `/opt/intel/fce/9.1.`*`nnn`*. In this guide, the default installation paths will be indicated as <install-dir>.

If you specify an installation directory rather than accepting the default path, the installation process will use that directory and will install compiler files under the specified directory. Any path you specify serves as the full path and subdirectories such as `bin`, `lib`, and `include` will be created under that path. For example, if, during installation, you specify an installation directory of `/home/my_account/intel_compiler`, the `/bin` subdirectory will be installed below the named directory, as shown in the following:

```
/home/my_account/intel_compiler/bin
```

For Mac OS* systems, you cannot change the default path.

Getting Started

Getting Started Overview

See these topics:

How You Can Use the Intel® Fortran Compiler

How You Can Use the Intel® Fortran Compiler

The Intel® Fortran Compiler has the following variations:

- The Intel® Fortran Compiler for IA-32 Applications is designed for IA-32 systems. The IA-32 compilations run on any IA-32 Intel processor and produce applications that run on IA-32 or Intel® EM64T systems. This compiler can optimize code for one or more Intel® IA-32 processors, such as Pentium® M, Pentium® 4, and Intel® Xeon® processors. This compiler runs on both Linux* and Mac OS* operating systems.
- Intel® Fortran Compiler for Intel® EM64T-based Applications is designed for Intel® EM64T systems. This compiler runs on Intel® EM64T systems and produces applications that run on IA-32 or Intel® EM64T systems. This compiler runs on Linux* operating systems.
- The Intel® Fortran Compiler for Itanium®-based Applications is designed for Itanium architecture systems. This compiler runs on Itanium-based systems and produces Itanium-based applications. Itanium-based compilations can only operate on Itanium-based systems. This compiler runs on Linux* operating systems.

The command to invoke any of these compilers is `ifort`.

The Intel® Fortran Compiler has a variety of options that enable you to use the compiler features for higher performance of your application.

The Intel® Fortran Compiler enables your software to perform the best on Intel architecture-based computers. The compiler has several high-performance optimizations. Some of its features and benefits are:

What feature might you want to use?	How will this help you?
Support for Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), and Streaming SIMD Extensions 3 (SSE3)	Intel microarchitecture benefit.
Automatic vectorizer	Parallelism in your code achieved automatically.
Parallelization	Automatic generation of multithreaded code for loops. Shared memory parallel programming with OpenMP*.
Floating-point optimizations	Improved floating-point performance.
Data prefetching	Improved performance due to the accelerated data delivery.
Interprocedural optimizations	Better performance for larger applications.
Whole program optimization	Improved performance between procedures in larger applications.
Profile-guided optimization	Improved performance based on profiling the frequently used procedures.
Processor dispatch	Use of the latest Intel architecture features while maintaining object code compatibility with previous generations of Intel® Pentium® processors.

Basic Troubleshooting

The following lists some of the most basic problems you can encounter during application development and gives suggestions for troubleshooting:

- **Source code does not compile correctly.**
  The Intel Fortran Compiler conforms to the Fortran 95, Fortran 90, and Fortran 77 standards. If your source code fails to compile, check for unsupported language extensions. Typically, these produce a syntax error. The best way to resolve problems of this nature is to rewrite the source code so it conforms to the Fortran standards and does not contain unsupported extensions.
- **Program does not run produce expected results.**
  Use test scenarios that ensure the output matches your expectations. If a test fails, try compiling the files using the -O0 option, which turns off the optimizer. If the test still fails, it is likely that the source code contains a problem. If your program runs successfully with -O0, but fails with -O2 (the default), you need to determine which file or files are causing the problem. To help pinpoint where the problem lies, divide the source files into two groups (perhaps alphabetically) and

compile one group of the source files with -O2 and the other group with -O0. If the program passes the tests, you know that the problem lies in the second group of files.

- **Program runs slowly.**
  Use a tool like the VTune™ Performance Analyzer to determine where your program spends most of its time. Such an analysis will show you which lines of your program are using the most execution time. See the Optimizing Applications book for additional guidelines that will help you optimize performance and gain speed.

Controlling Compiler Diagnostic Warning and Error Messages

You can use compiler options to control the diagnostic messages issued by the compiler. For example, the `-std` compiler option turns compile time warnings on for standards violations. The `-warn [keyword]` option controls warnings issued by the compiler. It supports a wide range of values. Some of these are as follows:

`[no]alignments` -- Determines whether warnings occur for data that is not naturally aligned.

`[no]declarations` -- Determines whether warnings occur for any undeclared symbols.

`[no]errors` -- Determines whether warnings are changed to errors.

`[no]general` -- Determines whether warning messages and informational messages are issued by the compiler.

`[no]interfaces` -- Determines whether warnings about the interfaces for all called SUBROUTINEs and invoked FUNCTIONs are issued by the compiler.

`[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.

For more information, see the -warn compiler option.

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` option.)

Compilation Phases

The compiler processes Fortran language source and generates object files. You decide the input and output by setting options when you run the compiler.

When invoked, the compiler determines which compilation phases to perform based on the extension of the source filename and on the compilation options specified in the command line.

The compilation phases and the software that controls each phase are shown below:

Compilation phase	Controlling software	IA-32, Intel® EM64T, or Itanium®-based applications?
Preprocess (optional)	`fpp`	All
Compile	`fortcom`	All
Assemble (optional)	`as` or `ias`	`as` for IA-32 and Intel® EM64T applications; `ias` for Itanium-based applications
Link	`ld(1)`	All

By default, the compiler generates an object file directly without calling the assembler. However, if you need to use specific assembly input files and then link them with the rest of your project, you can use an assembler for these files.

The compiler passes object files and any unrecognized filename to the linker. The linker then determines whether the file is an object file (`.o`) or a library (`.a`) or shared library (`.so`).The compiler handles all types of input files correctly. Thus, you can use it to invoke any phase of compilation.

Preprocess Phase

Preprocessing performs such tasks as preprocessor symbol (macro) substitution, conditional compilation, and file inclusion. The compiler preprocesses files as an optional first phase of the compilation. Source files that have a filename extension of `.fpp`, `.F`, `.F90`, `.FOR`, `.FTN`, or `.FPP` are automatically preprocessed by the compiler. For example, the following command preprocesses a source file that contains standard Fortran 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.

You do not usually need to specify preprocessing for Fortran source programs. The preprocessor is necessary only if your program uses C-style preprocessing commands, such as `#if`, `#define`, and so forth.

If you choose to preprocess your source programs, you must use the preprocessor fpp, which is the preprocessor supplied with the Intel® Fortran Compiler, or the preprocessing capability of a Fortran compiler. It is recommended that you use fpp.

If you want to use another preprocessor, you must invoke it before you invoke the compiler.

fpp conforms to cpp and accepts many cpp-style directives. cpp (and thus fpp) prohibit the use of a string constant value in an `#if` expression.

You can use the Preprocessor Options on the command line to direct the operations of the preprocessor.

 **Caution**

Using a preprocessor that does not support Fortran can damage your Fortran code, especially with FORMAT statements. For example, `FORMAT (\\I4)` changes the meaning of the program because the double backslash "`\\`" indicates end-of-record with most C/C++ preprocessors.

Assemblers and Linker

The assemblers and linker you can use are summarized in this table:

Tool	Default	Provided with Intel Fortran Compiler?
Assembler for IA-32 and Intel® EM64T applications	operating system assembler, `as`	No
Assembler for Itanium®-based applications	Intel® Itanium® assembler, `ias`	Yes
Linker	System linker, `ld(1)`	No

You can specify alternate tool locations and options for preprocessing, compilation, assembly, and linking.

See also Libraries Provided by Intel Fortran.

Assemblers

For IA-32 and Intel® EM64T applications, the operating system supplies its own assembler, `as`.

For Itanium-based applications, use the Itanium assembler, `ias`. The following example compiles a Fortran file to an assembly language file, which you can modify as desired. The assembler is then used to create an object file.

1. Issue a command using the -S option to generate an assembly code file, `file.s`: **`ifort -S -c file.f`**
2. To assemble the *`file`*`.s` file, call the Itanium® assembler with this command: **`ias -Nso -p32 -o file.o file.s`**

In the above command line, the following assembler options are used:

`-Nso` suppresses the sign-on message.

`-p32` enables defining 32-bit elements as relocatable data elements. (This option is available for backward compatibility.)

`-o` *`file`*`.o` indicates the output object filename.

Linker

The compiler calls the system linker, `ld(1)`, to produce an executable file from the object files.

Default Behavior of the Intel Fortran Compiler

The compiler generates one or more output files from one or more input files. By default, it performs the following actions:

- Searches for all files, including library files, in the current directory.
- Passes options designated for linking to the linker.
- Passes user-specified libraries to the linker.
- Displays error and warning messages.
- Performs default settings and optimizations, unless these options are overridden by specific options settings.
- For IA-32 applications, uses the `-mtune=pentium4` option to optimize the code for the Intel® Pentium® 4 processor. For Itanium®-based applications, uses the `-mtune=itanium2` option to optimize the code for the Intel® Itanium® 2 processor. (This option is ignored for Intel® EM64T-based applications.)

 **Note**

On operating systems that support characters in Unicode* (multi-byte) format, the compiler will process file names containing Unicode* characters.

Input Files and Filename Extensions

The Intel Fortran Compiler interprets the type of each input file by its filename extension, such as `.a`, `.f`, `.for`, `.o`, and so on:

Filename	Interpretation	Action
`filename``.a`	Object library	Passed to `ld`.
`filename``.f` *`filename`*`.ftn` *`filename`*`.for` *`filename`*`.i`	Fortran fixed-form source	Compiled by the Intel® Fortran compiler.
`filename``.fpp` *`filename`*`.F`	Fortran fixed-form source	Must be preprocessed by the Intel Fortran preprocessor `fpp` before being compiled by the Intel

filename.FOR *filename*.FTN *filename*.FPP		Fortran compiler.
filename.F90	Fortran free-form source	Must be preprocessed by the Intel Fortran preprocessor `fpp` before being compiled by the Intel Fortran compiler.
filename.f90 *filename*.i90	Fortran free-form source	Compiled by the Intel Fortran compiler.
filename.s	Assembly file	Passed to the assembler.
filename.o	Compiled object file	Passed to `ld`.

You can use the compiler configuration file to specify default directories for input libraries.

File Specifications

A complete file specification consists of a file name optionally preceded by a pathname that specifies a directory. The pathname can be in one of two forms:

- An absolute pathname, where the directory is specified relative to the root directory. The first character is a slash (`/`). For example, the following directory and file name refer to the file named `testdata` in the `/usr/users/gdata` directory: `/usr/users/gdata/testdata`
- A relative pathname, where the specified directory is relative to the current directory. Relative pathnames do not begin with a slash (`/`). The following example uses a relative pathname from the current directory `/usr/users` to refer to the same file `testdata` in the `gdata/` subdirectory: `gdata/testdata`

Directory names and file names should not contain any operating system wildcard characters (such as `*`, `?`, and the `[ ]` construct). You can use the tilde (`~`) character as the first character in a pathname to refer to a top-level directory as in the C shell.

File names are case-sensitive and can consist of uppercase and lowercase letters. For example, the following file names represent three different files:

```
myfile.for
MYfile.for
MYFILE.for
```

Output Files

The output produced by the `ifort` command includes:

- An object file (such as `test.o`), if you specify the `-c` option on the command line. An object file is created for each source file.
- An executable file (such as `a.out`), if you omit the `-c` option.
- One or more module files (such as `datadef.mod`), if the source file contains one or more MODULE statements.

- A shareable library (such as `mylib.so` on Linux `or mylib.dylib` on Mac OS* systems), if you use the `-shared (Linux)` or `-dynamiclib` (Mac OS) option.
- Assembly files, if you use the -S option. This creates an assembly file (.s) for each source file.

You control the production of these files by specifying the appropriate options on the command line.

The compiler generates a temporary object file for each source file, unless you specify the `-c` option. The linker is then invoked to link the object files into one executable program file and the temporary object files are deleted.

If you specify the -c option, the object files are created and retained in the current working directory. You must link the object files later. You can do this by using a separate ifort command; alternatively, you can call the linker (ld) directly to link in objects, or call xild. Another possible choice is to use the archiver (ar) and xiar to create a library. For Mac OS*, you would use libtool to generate a library.

If fatal errors are encountered during compilation, or if you specify certain options such as `-c`, linking does not occur.

Note

To allow optimization across all objects in the program, use the `-ipo` option.

To specify a file name for the executable program file (other than `a.out`), use the `-o` *`output`* option, where *`output`* specifies the file name. The following command requests a file name of `prog1.out` for the source file `test1.f`:

```
ifort -o prog1.out test1.f
```

If you specify the `-c` option with the `-o` *`output`* option, you rename the object file (not the executable program file). If you specify `-c` and omit the `-o` *`output`* option, the compiler names the object files with a `.o` suffix substituted for the source file suffix.

Note

You cannot use `-c` and `-o` together with multiple source files.

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.

If the TEMP environment variable is not defined, the /tmp directory is used to store temporary files.

Controlling the Compilation Process

To customize the environment used during compilation, you can specify variables, options, and files as follows:

- Environment variables to specify paths where the compiler searches for special files such as libraries and "include" files
- Configuration files to specify the options used for every compilation and response files to specify the options and files used for individual projects

For information on using compiler options to optimize the compilation process, see Optimizing the Compilation Process Overview.

Setting and Viewing Environment Variables

You can use the SET command to view or set environment variables one at a time. You can also set environment variables by using the ifortvars.csh and ifortvars.sh files to set several at a time. The files are found in this directory: /opt/intel/fc/9.1.xxx/bin. See Running the Shell Script to Set Up the Environment Variables.

within the C Shell, 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
```

Within the Bourne* shell (sh), the Korn shell (ksh), and the bash shell, use the export command and assignment command to set the environment variable:

```
export FORT9
FORT9=/usr/users/smith/test.dat
```

To remove the association of an environment variable and its value within the Bourne* shell, the Korn shell, or the bash shell, use the unset command:

```
unset FORT9
```

Configuration File Environment Variables

By default, the compiler picks up the default configuration file (`ifort.cfg`) from the same directory where the compiler executable resides. However, if you want the compiler to use another configuration file in a different location, you can use the `IFORTCFG` environment variable to assign the directory and filename for the other configuration file.

See Also

Compile-Time Environment Variables

Run-Time Environment Variables

Running the Shell Script to Set Up the Environment Variables

Before you first invoke the compiler, you need to set the environment variables to specify locations for the various components.

The Intel Fortran Compiler installation includes a shell script that you can use to set environment variables.

Use the `source` command to execute the shell script from the command line. For example, to execute this script file for the bash shell:

```
source /opt/intel/fc/9.1.xxx/bin/ifortvars.sh
```

If you use the C shell, use the `.csh` version of this script file:

```
source /opt/intel/fc/9.1.xxx/bin/ifortvars.csh
```

If you want `ifortvars.sh` to run automatically when you start Linux*, edit your `.bash_profile` file and add the line above to the end of your file. For example:

```
# set up environment for Intel compiler
source /opt/intel/fc/9.1.xxx/bin/ifortvars.sh
```

If you compile a program without 'sourcing' `ifortvars.sh`, you will see the following error when you execute the compiled program:

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

Compatibility with Previous Versions of Intel® Fortran

This topic is written for developers who are familiar with Intel Fortran Version 7.1 or earlier versions are now using Intel Fortran Version 8.0 and later releases.

Intel® Fortran supports extensions to the ISO and ANSI standards, including a number of extensions defined by:

- Intel Fortran for various platforms
- Microsoft* Fortran PowerStation 4.0

Many language extensions associated with Microsoft* Fortran PowerStation Version 4 have been added to Intel Fortran.

Differences Between Intel Fortran Version 7.1 and Intel Fortran Version 9.1

Some differences are:

- The command name for command-line use is now `ifort`. Earlier versions of Intel Fortran used a command name of `ifc` or `efc`. For Intel Fortran, these command names will still be accepted, but in some future Intel Fortran release, only the `ifort` command name will be accepted.
- The default configuration file name is now `ifort.cfg` instead of `ifc.cfg` or `efc.cfg`.
- The predefined symbol name for the Intel Fortran compiler is __INTEL_COMPILER and it has a value of 800 for Intel Fortran Version 8.0, 810 for Intel Fortran Version 8.1, 900 for Intel Fortran Version 9.0 and 910 For Intel Fortran Version 9.1.
- The record length (RECL specifier) for unformatted files is now 32-bit words. To get the record length in bytes, use the `-assume byterecl` option.
- The backslash character (\) is not treated as an escape character for control sequences in character literals. To force the backslash to start escape sequences, use the `-assume bscc` option.
- Intel Fortran Version 8.x and later releases by default uses the integer -1 for the value of .TRUE. whereas Version 7 uses the integer 1 for the value of .TRUE.. If you use the `-fpscomp logicals` option with Version 8.x and later releases, the compiler will use the integer 1 for the value of .TRUE..
  Version 8.x and later releases always use the integer 0 as the value of .FALSE., as did Version 7.
  User-written routines in Fortran or other languages (for example, C) need to insure that they use values for .TRUE. and .FALSE. consistent with the compiler's choice.
- The random number generator used in Version 8.x and later releases is different from the random number generator used in Version 7.
  Version 8 and later releases use the random number generator based on the algorithm of Park and Miller, which is the generator used by Compaq Fortran.
  Version 7 used the Marsaglia random number generator.
  Both of these random number generators are compatible with the Fortran 90 standard.
  In addition, Version 8.x and later releases use different algorithms for the RANDOM_NUMBER and RANDOM_SEED intrinsics (compared to Version 7) and different algorithms are used for these intrinsics on IA32 and Itanium-based systems.

Documentation Information

Some documentation has been moved or changed. In particular:

- The *Intel® Fortran User's Guide, Volumes I and II* have been renamed to *Building Applications* and *Optimizing Applications*, respectively.
- Intel Fortran language information previously described in the *Intel Fortran Programmer's Reference*, including intrinsics procedures and directives, is now described in the online *Language Reference.*
- All Intel Fortran language elements and library routines are described in this online help file, allowing easy lookup of reference information.

Version 7.1 Features Not Available in Intel Fortran Version 9.1

The following Intel Fortran Version 7.1 features are not available in Intel Fortran Version 8.x or later releases:

- IMPLICIT AUTOMATIC | STATIC statements
- The Intel Fortran run-time library system's ability to work with the Itanium® processor simulator

Building Applications from the Command Line

Building Applications from the Command Line Overview

This section covers the following:

- Invoking the Intel Fortran Compiler from the Command Line
- Examples of the ifort Command
- Using Module (.mod) Files
- Searching for Include and .mod Files
- Configuration Files and Response Files
- Specifying Alternative Tools Locations and Options
- Predefined Preprocessor Symbols
- Redirecting Output to Files
- Creating, Running, and Debugging an Executable Program
- Creating Shared Libraries
- Allocating Common Blocks

Invoking the Intel® Fortran Compiler from the Command Line

You can invoke the Intel® Fortran Compiler in either of two ways:

- Using the `ifort` command
- Using the `make` command to specify a makefile

Using the ifort Command

The syntax is:

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

An *option* is specified by one or more letters preceded by a hyphen.

Some options take arguments in the form of filenames, strings, letters, or numbers. Except where otherwise noted, you can enter a space between the option and its argument(s) or you can combine them. For a complete listing of compiler options, see the Compiler Options reference.

You can specify more than one *input_file*, using a space as a delimiter. See Input Files and Filename Extensions.

Note

Options on the command line apply to all files. For example, in the following command line, the `-c` and `-nowarn` options apply to both files `x.f` and `y.f`:
`ifort -c x.f -nowarn y.f`

Using the make Command

To compile a number of files with various paths and to save this information for multiple compilations, you can use a makefile to invoke the Intel® Fortran Compiler.

To use a makefile to compile your input files, make sure that `/usr/bin` and `/usr/local/bin` are in your path.

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

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

Then you can compile as:

```
make -f yourmakefile
```

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

Examples of the ifort Command

Compiling and Linking Multiple Files

The following `ifort` command compiles the Fortran free-format source files `aaa.f90`, `bbb.f90`, and `ccc.f90`. The command invokes the `ld` linker and passes the temporary object files to the linker, which it uses to produce the executable file `a.out`:

```
ifort aaa.f90 bbb.f90 ccc.f90
```

The following `ifort` command compiles all file names that end with `.f`. as Fortran fixed-format source. The linker produces the `a.out` file:

```
ifort *.f
```

Preventing Linking

The following `ifort` command compiles, but does not link, the free-format source file `typedefs_1.f90`, which contains a MODULE TYPEDEFS_1. The command creates files `typedefs_1.mod` and `typedefs_1.o`. The object file is retained automatically. Specifying the `-c` option prevents linking:

```
ifort -c typedefs_1.f90
```

Renaming the Output File

The following `ifort` command compiles the free-format Fortran source files `circle-calc.f90` and `sub.f90` together:

```
ifort -c circle-calc.f90 sub.f90
```

The default optimization level `-O2` applies to both source files during compilation. Because the `-c` option is specified, the object files are not passed to the linker. In this case, the named output files are the object files.

Like the previous command, the following `ifort` command compiles multiple source files:

```
ifort -o circle.out circle-calc.f90 sub.f90
```

Because the `-c` option was omitted, an executable program named `circle.out` is created.

Specifying an Additional Linker Library

The following `ifort` command compiles a free-format source file `myprog.f90` using default optimization, and passes an additional library for the linker to search:

```
ifort myprog.f90 typedefs_1.o -lmylib
```

The file is processed at optimization level `-O2` and then linked with the object file `typedefs_1.o` . The `-lmylib` option instructs the linker to search in the `libmylib` library for unresolved references (in addition to the standard list of libraries the `ifort` command passes to the linker).

Using Module (.mod) Files

A module (.mod file) 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 USE statement. Typical applications of modules are the specification of global data or the specification of a derived type and its associated operations.

Some programs require modules located in multiple directories. You can use the `-I`*`dir`* option when you compile the program to locate the `.mod` files that should be included in the program.

You can use the `-module` *`path`* option to specify the directory in which to create the module files. This path is also used to locate module files. If you don't use this option, module files are created in the current directory.

You need to make sure that the module files are created before they are referenced by another program or subprogram.

Compiling Programs with Modules

If a file being compiled has one or more modules defined in it, the compiler generates one or more `.mod` files. For example, a file `a.f90` contains modules defined as follows:

```
module test
integer:: a
contains
  subroutine f()
  end subroutine
end module test
```

```
module payroll
.
.
.
end module payroll
```

This compiler command:

```
ifort -c a.f90
```

generates the following files:

- `test.mod`
- `payroll.mod`
- `a.o`

The `.mod` files contain the necessary information regarding the modules that have been defined in the program `a.f90`.

If the program does not contain a module, no `.mod` file is generated. For example, `test2.f90` does not contain any modules. This compiler command:

```
ifort -c test2.f90
```

produces just an object file, `test2.o`.

For another example, assume that `file1.f90` contains one or more modules and `file2.f90` contains one or more program units that access these modules with the USE statement. The sources can be compiled and linked by this command:

```
ifort file1.f90 file2.f90
```

Working with Multi-Directory Module Files

For an example of managing modules when the `.mod` files could be produced in different directories, assume that the program `mod_def.f90` resides in directory `/usr/yourdir/test/t`, and this program contains a module defined as follows:

```
file: mod_def.f90
module definedmod
.
.
.
end module
```

The compiler command:

```
ifort -c mod_def.f90
```

produces two files: `mod_def.o` and `definedmod.mod` in directory `/usr/yourdir/test/t`.

If you need to use the above `.mod` file in another directory, for example, in directory `/usr/yourdir/test/t2`, where the program `usemod` uses the `definedmod.mod` file, do the following:

```
file: use_mod_def.f90
program usemod
use definedmod
.
.
.
end program
```

To compile the above program, use this command:

```
ifort -c use_mod_def.f90 -I/usr/yourdir/test/t
```

where the `-Idir` option provides the compiler with the path to search and locate the `definedmod.mod` file.

Parallel Invocation with a makefile

The programs containing module definitions support parallel invocation using a makefile. Consider the following code:

```
test1.f90
module m1
.
.
.
end module
test2.f90
subroutine s2()
use m1
.
.
.
end subroutine
test3.f90
subroutine s3()
use m1
.
.
.
end subroutine
```

The makefile to compile the above code looks like this:

```
m1.mod: test1.o
test1.o: test1.f90
ifort -c test1.f90
test2.o: m1.mod test2.f90
ifort -c test2.f90
test3.o: m1.mod test3.f90
ifort -c test3.f90
```

Searching for Include and .mod 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. Directory of the source file that contains the include
2. Current working directory
3. Directories specified by the -Idir option
4. Directory specified by the -isystem option
5. Directories specified with the `FPATH` environment variable
6. 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.

A module (`.mod`) file is specified in a program by a USE statement. Module files can be located in multiple directories.

Directories are searched for `.mod` files in this order:

1. Directory of the source file that contains the USE statement
2. Directories specified by the `-module path` option

3. Current working directory
4. Directories specified by the -Idir option
5. Directories specified with the `FPATH` environment variable
6. Standard system directories

Specifying and Removing an Include File Path

You can use the `-Idir` option to indicate the location of include files and module files.

To prevent the compiler from searching the default path specified by the `FPATH` environment variable, use the `-X` option.

You can specify these options in the configuration file, `ifort.cfg`, or on the command line.

For example, to direct the compiler to search the path `/alt/include` instead of the default path, use the following command line:

```
ifort -X -I/alt/include newmain.f
```

Configuration Files and Response Files

Configuration files and *response files* are similar in that both eliminate the need to enter the same commands again and again. (Response files are also known as indirect command files.) The following describes each type of file.

Configuration Files

You can use a configuration (`.cfg`) file to:

- Decrease the time you spend entering command-line options
- Ensure consistency of often used commands

You can insert any valid command-line options into a configuration file. The compiler processes options in the configuration file in the order in which they appear followed by the command-line options that you specify when you invoke the compiler.

 **Note**

Options placed in the configuration file will be included each time you run the compiler. If you have varying option requirements for different projects, use response files.

By default, a configuration file named `ifort.cfg` is used.

This file resides in the same directory where the compiler executable resides.

However, if you want the compiler to use another configuration file in a different location, you can use the `IFORTCFG` environment variable to assign the directory and file name for the other configuration file.

Example Configuration File

An example configuration file is shown below. The pound (`#`) character indicates that the rest of the line is a comment.

```
## Example ifort.cfg file
##
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
##
## Set extended-length source lines.
-extend_source
##
## Set maximum floating-point significand precision.
-pc80
##
```

Response Files

You can use response files (also known as indirect command files) to:

- Specify options used during particular compilations for particular projects
- Save this information in individual files

Response files are invoked as an option on the command line. Options specified in a response file are inserted in the command line at the point where the response file is invoked.

Like configuration files, response files are used to:

- Decrease the time you spend entering command-line options
- Ensure consistency of often used commands

Options in a configuration file are executed every time you run the compiler. In contrast, you use response files to maintain options for individual projects.

You can place any number of options or file names on a line in the indirect command file or response file. Several files can be referenced in the same command line.

The syntax for using response files is:

```
ifort @responsefile [@responsefile2...]
```

 **Note**

An "at" sign (`@`) must precede the name of the response file on the command line.

Specifying Alternative Tool Locations and Options

The Intel® Fortran compiler lets you specify alternative tool locations and tool options to be used instead of default tools for preprocessing, compilation, assembly, and linking. You can use the `-Qlocation` and `-Qoption` options to do this.

For more information on these options, see the Compiler Options reference.

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.

Some preprocessor symbols are predefined by the compiler system and are available to compiler directives and fpp. If you want to use others, you need to specify them on the command line.

The predefined preprocessor symbols available for the Intel® Fortran compiler are described in the table below. The Default column describes whether the preprocessor symbol is enabled (on) or disabled (off) by default.

Symbol Name	Default	IA-32, Intel® EM64T, Itanium®-based?	Description
`__INTEL_COMPILER=`*n*	On, *n*=910	All	Identifies the Intel Fortran Compiler
`__INTEL_COMPILER_BUILD_DATE=`*YYYYMMDD*		All	Identifies the Intel Fortran Compiler build date
`__linux__` (Linux only) `__linux` (Linux only) `__gnu_linux__` (Linux only) `linux` (Linux only) `__unix__` (Linux only) `__unix` (Linux only) `unix` (Linux only) `__ELF__` (Linux only)		All	Defined at the start of compilation
__APPLE__ (Mac OS* only __MACH__ (Mac OS* only)		IA-32	Defined at the start of compilation

`__i386__`
`__i386`
`i386`
		IA-32	Identifies the architecture for the target hardware for which programs are being compiled

`__ia64__`
`__ia64`
`ia64` (deprecated)
		Itanium®	Identifies the architecture for the target hardware for

			which programs are being compiled
__x86_64 __x86_64__		Intel® EM64T	Identifies the architecture for the target hardware for which programs are being compiled.
_OPENMP=*n*	*n*=200505	All	Takes the form *YYYYMM*, where *YYYY* is the year and *MM* is the month of the OpenMP Fortran specification supported. This preprocessor symbol can be used in both fpp and the Fortran compiler conditional compilation. It is available only when `-openmp` is specified.
_PGO_INSTRUMENT	Off	All	Defined when `-prof_gen` is specified.
__PIC__ __pic__	Off (Linux*), On (Mac OS*)	All	Set if the code was requested to be compiled as position independent code. On Mac OS*, these symbols are always set.

Defining Preprocessor Symbols

You can use the `-D` option to define the symbol names to be used during preprocessing. This option performs the same function as the `#define` preprocessor directive.

Preprocessing replaces every occurrence of the defined symbol name with the specified value.

For more information, see the following topic:

- -D compiler option

Suppressing Preprocessor Symbols

You can use the `-U` option to suppress an automatic definition of a preprocessor symbol. This option suppresses any symbol definition currently in effect for the specified name. The `-U` option performs the same function as an `#undef` preprocessor directive.

For more information, see the following topic:

- -U compiler option

Redirecting Output to Files

For programs that display a lot of text, consider redirecting text that is usually displayed on `stdout` to a file. Displaying a lot of text will slow 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.

The following commands show how to run the program more efficiently by redirecting output to a file and then displaying the program output:

```
myprog > results.lis
more results.lis
```

Creating, Running, and Debugging an Executable Program

The example below shows a sample Fortran main program using free source form that uses a module and an external subprogram.

The function CALC_AVERAGE is contained in a separate file and depends on the module ARRAY_CALCULATOR for its interface block.

The USE statement accesses the module ARRAY_CALCULATOR. This module contains the function declaration for CALC_AVERAGE.

The 5-element array is passed to the function CALC_AVERAGE, which returns the value to the variable AVERAGE for printing.

The example is:

```
! File: main.f90
! This program calculates the average of five numbers
PROGRAM MAIN
USE ARRAY_CALCULATOR
REAL, DIMENSION(5) :: A = 0
REAL :: AVERAGE
PRINT *, 'Type five numbers: '
READ  (*,'(F10.3)') A

AVERAGE = CALC_AVERAGE(A)
PRINT *, 'Average of the five numbers is: ', AVERAGE
END PROGRAM MAIN
```

The example below shows the module referenced by the main program. This example program shows more Fortran 95/90 features, including an interface block and an assumed-shape array:

```
! File: array_calc.f90.
! Module containing various calculations on arrays.
```

```
MODULE ARRAY_CALCULATOR
INTERFACE
  FUNCTION CALC_AVERAGE(D)
    REAL :: CALC_AVERAGE
    REAL, INTENT(IN) :: D(:)
  END FUNCTION CALC_AVERAGE
END INTERFACE
! Other subprogram interfaces...
END MODULE ARRAY_CALCULATOR
```

The example below shows the function declaration CALC_AVERAGE referenced by the main program:

```
! File: calc_aver.f90.
! External function returning average of array.
FUNCTION CALC_AVERAGE(D)
REAL :: CALC_AVERAGE
REAL, INTENT(IN) :: D(:)
CALC_AVERAGE = SUM(D) / UBOUND(D, DIM = 1)
END FUNCTION CALC_AVERAGE
```

Commands to Create a Sample Program

During the early stages of program development, the sample program files shown above might be compiled separately and then linked together, using the following commands:

```
ifort -c array_calc.f90
ifort -c calc_aver.f90
ifort -c main.f90
ifort -o calc main.o array_calc.o calc_aver.o
```

In this sequence of commands:

- The `-c` option prevents linking and retains the `.o` files.
- The first command creates the files `array_calculator.mod` and `array_calc.o` (the name in the MODULE statement determines the name of module file `array_calculator.mod)`. Module files are written into the current working directory.
- The second command creates the file `calc_aver.o`.
- The third command creates the file `main.o` and uses the module file `array_calculator.mod`.
- The last command links all object files into the executable program named `calc`. To link files, use the `ifort` command instead of the `ld` command.

The order in which the file names are specified is significant. Consider the following `ifort` command:

```
ifort -o calc array_calc.f90 calc_aver.f90 main.f90
```

This command does the following:

- Compiles the file `array_calc.f90`, which contains the module definition, and creates its object file and the file `array_calculator.mod`.

- Compiles the file `calc_aver.f90`, which contains the external function CALC_AVERAGE.
- Compiles the file `main.f90` (main program). The USE statement references the module file `array_calculator.mod`.
- Uses `ld` to link the main program and all object files into an executable program file named `calc`.

Running the Sample Program

If your path definition includes the directory containing `calc`, you can run the program by simply entering its name:

```
calc
```

When running the sample program, the PRINT and READ statements in the main program result in the following dialogue between user and program:

```
Type five numbers:
55.5
4.5
3.9
9.0
5.6
Average of the five numbers is:   15.70000
```

Debugging the Sample Program

To debug a program with the debugger, compile the source files with the `-g` option to request additional symbol table information for source line debugging in the object and executable program files. The following `ifort` command also uses the `-o` option to name the executable program file `calc_debug`:

```
ifort -g -o calc_debug array_calc.f90 calc_aver.f90 main.f90
```

See also Debugging Overview and related sections.

Creating Shared Libraries

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 (Mac OS*) to create the `.so` or `.dylib` file. On Linux IA-32 systems and Intel® EM64T systems, you must also specify -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 (Mac OS*)
```

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.

The `-o` option was omitted, so the name of the shared library file is `octagon.so` (Linux) or `octagon.dylib` (Mac OS*).

You can use the `-i-static` 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 (Mac OS*) 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 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 Fortran libraries. It is easier and safer to use the ifort command. On a Mac OS* system, you would use libtool.
- The `-shared` option is required to create a shared library. On a Mac OS* system, 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 probably a good idea to look at the output of the `-dryrun` command to find the names of all the libraries used so you can 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 shared libraries, see Creating Libraries.

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, either 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, making sure that you 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 Mac OS* systems, 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.

Allocating Common Blocks

You can use the `-dyncom` (dynamic common) option to control the allocation of 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 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 the check time.

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
```

where `blk1`, `blk2`, and `blk3` are the names of the common blocks to be made dynamic.

Guidelines for Using the -dyncom Option

The following are some limitations that you should be aware of when using the `-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. This routine must be written in the C language to generate the correct routine name.

The routine prototype is:

```
void _FTN_ALLOC(void **mem, int *size, char *name);
```

where

- *mem* is the location of the base pointer of the common block which 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.


Note

You must set the `size` argument with the size, in bytes, of space you allocate. The library routine that calls `_FTN_ALLOC()` ensures that all other occurrences of this common block fit in the space you allocated. Return the size, in bytes, of the space you allocate by modifying size.

- *`name`* is the name of the common block being dynamically allocated.

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()`. If you use the nonstandard extension having the common block of the same name declared with different sizes in different routines, you might get a run-time error depending on the order in which the routines containing the common block declarations are invoked.

The Fortran run-time library contains a default version of `_FTN_ALLOC()`, which simply allocates the requested number of bytes and returns.

Using Compiler Options

Compiler Options Overview

A compiler option (also known as a switch) is an optional string of one or more alphanumeric characters preceded by a dash (-) that follows the `ifort` command on the command line. Options to the `ifort` command affect how the compiler processes a file in conjunction with the file name suffix. The simplest form of the `ifort` command is often sufficient.

Some options are on by default when you invoke the compiler.

Getting Help on Options

For help, enter `-help` on the command line, which displays brief information about all the command-line options.

The Compiler Options reference provides a complete description of each compiler option.


Note

If there are enabling and disabling versions of options on the command line, or two versions of the same option, the last one takes precedence.

Using Multiple ifort Commands

If you compile parts of your program by using multiple `ifort` commands, options that affect the execution of the program should be used consistently for all compilations, especially if data is shared or passed between procedures. For example:

- The same data alignment needs to be used for data passed or shared by module definitions (such as user-defined structures) or common blocks. Use the same version of the `-align` option for all compilations.
- The program might contain INTEGER, LOGICAL, REAL, or COMPLEX declarations without a kind parameter or size specifier that is passed or shared by module definitions or common blocks. You must consistently use the options that control the size of such numeric data declarations.

Using the OPTIONS Statement to Override Options

You can override some options specified on the command line by using the OPTIONS statement in your Fortran source program. The options specified by the OPTIONS statement affect only the program unit where the statement occurs.

Option Mapping Tool

The Intel compiler's Option Mapping Tool provides an easy method to derive equivalent options between Windows* and Linux*. If you are a Windows developer who is developing an application for Linux, you may want to know, for example, the Linux equivalent for the `/Oy-` option. Likewise, the Option Mapping Tool provides Windows equivalents for Intel compiler options supported on Linux.

Note

Mac OS* systems: The Option Mapping Tool is not available.

Using the Compiler Option Mapping Tool

You can start the Option Mapping Tool from the command line by:

- invoking the compiler and using the `-map_opts` option
- or, executing the tool directly

Note

The Compiler Option Mapping Tool only maps compiler options on the same architecture. It will not, for example, map an option that is specific to Itanium®-based systems to an option on an IA-32 system or an Intel® EM64T system.

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 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

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 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
```

Compiler Directives Related to 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.

Compiler directives and equivalent command-line compiler options are:

Compiler Directive	Equivalent Command-Line Compiler Option
DECLARE	-warn declarations
NODECLARE	-warn nodeclarations
DEFINE *symbol*	-D*name*
FIXEDFORMLINESIZE:*option*	-extend_source [*option*]
FREEFORM	-free or -nofixed
NOFREEFORM	-nofree or -fixed
INTEGER:*option*	-integer_size *option*
PACK:*option*	-align [*option*]
REAL:*option*	-real_size *option*
STRICT	-warn stderrors with -stand
NOSTRICT	-warn nostderrors

Note that the compiler directive names above are specified using the prefix `!DEC$` followed by a space. For example: `!DEC$ NOSTRICT`

Note

The prefix `!DEC$` is normally used. `!DEC$` works for both fixed-form and free-form source. You can also use these alternative prefixes for fixed-form source only: `cDEC$`, `CDEC$`,`*DEC$`, `cDIR$`, `CDIR$`, `*DIR$`, and `!MS$`.

Debugging

Debugging Overview

See these topics:

Introduction to the Intel® Debugger (idb)

Preparing Your Program for Debugging

Debugging and Optimizations

Using idb Debugger Commands and Setting Breakpoints

Summary of idb Debugger Commands

Debugging the SQUARES Example Program

Displaying Variables in the Debugger

Expressions in Debugger Commands

Debugging Mixed-Language Programs

Debugging a Program that Generates a Signal

Locating Unaligned Data

Debugging Multithread Programs

Introduction to the Intel® Debugger (idb)

The Intel® Debugger (idb) is a source-level, symbolic debugger that lets you:

- Control the execution of a program at the source line level.
- Set stops (breakpoints) at specific source lines or under various conditions.
- Change the value of variables in your program.
- Refer to program locations by their symbolic names, using the debugger's knowledge of the Intel® Fortran language to determine the proper scoping rules and how the values should be evaluated and displayed.
- Print the values of variables and set a tracepoint (trace) to notify you when the value of a variable changes. (Another term for a tracepoint is a watchpoint.)
- Perform other functions, such as examining core files, examining the call stack, or displaying registers.

The idb debugger has two modes:

- dbx (default mode)
- gdb (optional mode)

All examples in this guide are shown in dbx mode.

 **Note**

For complete information about idb, see the `idb` man page or the online *Intel® Debugger (IDB) Manual.*

Preparing Your Program for Debugging

Use the `ifort` command with certain options to create an executable program for debugging.

Debugging Options

To use the debugger, you should specify the `ifort` command and the `-g` command-line option. Traceback information and symbol table information are both necessary for debugging. If you specify `-g`, the compiler provides the symbol table and traceback information needed for symbolic debugging. (The `-notraceback` option cancels the traceback information.)

Likely uses of these options at the various stages of program development are as follows:

During early stages of program development, use the `-g` option to create unoptimized code (optimization level `-O0`). This option also might be chosen later to debug reported problems from later stages.

Traceback and symbol table information result in a larger object file. During the later stages of program development, use `-g0` or `-g1` to minimize the object file size and, as a result, the memory needed for program execution, usually with optimized code. (The `-g0` option eliminates the traceback information.)

When you have finished debugging your program, you can recompile and relink to create an optimized executable program or remove traceback and symbol table information with the `strip` command. (See `strip(1)`.)

See the `-debug keyword` option in the Compiler Options reference to learn more about settings that enhance debugging.

Starting the Debugger

To invoke the debugger, enter the debugger shell command and the name of the executable program.

The following commands create (compile and link) the executable program and invoke the interface to the debugger:

```
ifort -g -o squares squares.f90
idb squares
Linux Application Debugger for xx-bit applications, Version x.x, Build
xxxx
object file name: squares
reading symbolic information ... done
(idb)
```

In this example, the `ifort` command:

- Compiles and links the program `squares.f90`.
- Requests symbol table information needed for symbolic debugging and no optimization (`-g`).
- Names the executable file `squares` instead of `a.out` (`-o squares`).

The `idb` shell command runs the debugger, specifying the executable program `squares`.

At the debugger prompt `(idb)`, you can enter a debugger command.

See also the online *Intel ® Debugger (IDB) Manual.*

Debugging and Optimizations

This topic describes the compiler command-line options that you can use to debug your compilation and to display and check compilation errors.

The options that control debugging and optimizing are as follows:

`-O0`	Disables optimizations so you can debug your program before any optimization is attempted . Enables `-fp` option. For more information, see the following topic: • `-O0` compiler option
`-O1, -O2, -O3`	Specifies the code optimization level for applications. For more information, see the following topic: • `-O1, -O2, -O3` compiler option
`-g`	Generates symbolic debugging information and line numbers in the object code for use by the source-level debuggers. Turns off `-O2` and makes `-O0` the default unless `-O2` (or `-O1` or `-O3`) is explicitly specified in the command line together with `-g`. For more information, see the following topic: • `-g` compiler option
`-debug extended` (Linux* only)	Specifies settings that enhance debugging. To use this option, you must also specify the `-g` option. For more information, see the following topic: • `-debug extended` compiler option
`-fp` IA-32 only	Disables the use of the `ebp` register in optimizations. Directs to use the `ebp`-based stack frame for all functions. For more information, see the following topic: • `-fp` compiler option

`-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

Note

Debugging of optimized code is not fully supported on Intel platforms.

For optimized code, use the following options: `-On, -g, -debug extended, where` *n* is 1, 2, or 3. For non-optimized code, use the `-g` option.

The Use of `ebp` Register

`-fp` (IA-32 only)

Most debuggers use the `ebp` register as a stack frame pointer to produce a stack backtrace. The `-fp` option disables the use of the `ebp` register in optimizations and directs 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 turning off `-O1, -O2,` or `-O3` optimizations.

Note that using this option reduces the number of available general-purpose registers by one, and results in slightly less efficient code.

The `-traceback` Option

The `-traceback` option also forces the compiler to use `ebp` as the stack frame pointer. In addition, the `-traceback` option causes the compiler to generate extra information into the object file, which allows a symbolic stack traceback to be produced if a run-time failure occurs.

Combining Optimization and Debugging

The `-O0` option turns off all optimizations so you can debug your program before any optimization is attempted. To get debug information, use the `-g` option.

The compiler lets you generate code to support symbolic debugging while one of the `-O1`, `-O2`, or `-O3` optimization options is specified on the command line along with `-g` and `-debug extended`, which produces symbolic debug information in the object file.

Note that if you specify the `-O1`, `-O2`, or `-O3` option with the `-g` option, some of the debugging information returned may be inaccurate (a side-effect of optimization) unless the `-debug extended` option is also used.

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` 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 `-g` and `-debug extended`.

> **Note**
> When no optimization level (`-On`) is specified, the `-g` option slows program execution; this is because `-g` turns on `-O0`, which causes the slowdown. However, if, for example, both `-O2` and `-g` are specified, the code should run very nearly at the same speed as if `-g` were not specified.

Refer to the table below for the summary of the effects of using the `-g` option with the optimization options.

Options	Results
`-g`	Debugging information produced, `-O0` enabled (optimizations disabled), `-fp` enabled for IA-32-targeted compilations
`-g -O1`	Debugging information produced, `-O1` optimizations enabled.
`-g -O2`	Debugging information produced, `-O2` optimizations enabled.
`-g -O3 -fp`	Debugging information produced, `-O3` optimizations enabled, `-fp` enabled for IA-32-targeted compilations.

On Linux* systems, use the `-debug extended` option with any of the last three combinations, to improve debugging.

Using idb Debugger Commands and Setting Breakpoints

To find out what happens at critical points in your program, you need to stop execution at these points and look at the contents of program variables to see if they contain the correct values. Points at which the debugger stops program execution are called breakpoints.

To set a breakpoint, use one of the forms of the `stop` or `stopi` commands.

Using a sample program, the following debugger commands set a breakpoint at line 4, run the program, continue the program, delete the breakpoint, rerun the program, and return to the shell:

```
(idb) stop at 4
[#1: stop at "squares.f90":4 ]
(idb) run
[1] stopped at [squares:4 0x120001880]
>   4       OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
(idb) cont
```

```
Process has exited with status 0
(idb) delete 1
(idb) rerun
Process has exited with status 0
(idb) quit
%
```

In this example:

- The `stop at 4` command sets a breakpoint at line 4. To set a breakpoint at the start of a subprogram (such as `calc`), use the `stop in` command (such as `stop in calc`).
- The `run` command begins program execution and stops at the first breakpoint. The program is now active, allowing you to view the values of variables with print commands and perform related functions.
- The `cont` command resumes (continues) program execution. In addition to the `cont` command, you can also use the `step`, `next`, `run`, or `return` commands to resume execution.
- The `delete 1` command shows how to delete a previously set breakpoint (with event number 1). For instance, you might need to delete a previously set breakpoint before you use the `rerun` command.
- The `rerun` command runs the program again. Since there are no breakpoints, the program runs to completion.
- The `quit` command exits the debugger and returns to the shell.

Other Debugger Commands

Other debugger commands include the following:

- To get help on debugger commands, enter the `help` command.
- To display previously typed debugger commands, enter the `history` command.
- To examine the contents of a location, use the `print` or `dump` commands.
- To execute a shell command, use the `sh` command (followed by the desired shell command). For instance, if you cannot recall the name of a FUNCTION statement, the following `grep` shell command displays the lines containing the letters FUNCTION, allowing use of the function name (SUBSORT) in the stop in command:

```
(idb) sh grep FUNCTION data.for
INTEGER*4 FUNCTION SUBSORT (A,B)
(idb) stop in subsort
(idb)
```

See also Summary of Debugger Commands.

Summary of idb Debugger Commands

The table below lists some of the more frequently used debugging commands available in idb. Many of these commands can be abbreviated (for example, you can enter `c` instead of `cont` and `s` instead of `step`. You can use the `alias` command to get a complete list of these abbreviations and even create your own aliases.

The table shows examples of the most commonly used debugger commands. For more information, see the online *Intel ® Debugger (IDB) Manual.*

Command	Description
`catch`	Displays all signals that the debugger is currently set to catch. See also `ignore`.
`catch fpe`	Tells the debugger to catch the fpe signal (or the signal specified). This prevents the specified signal from reaching the Intel Fortran run-time library (RTL).
`catch unaligned`	Tells the debugger to catch the unaligned signal.
`cont`	Resumes (continues) execution of the program that is being debugged. Note that there is no idb command named `continue`.
`delete 2`	Removes the breakpoint or tracepoint identified by event number 2. See also `status`.
`delete all`	Removes all breakpoints and tracepoints.
`help`	Displays debugger help text.
`history 5`	Displays the last five debugger commands.
`ignore`	Displays the signals the debugger is currently set to ignore. The ignored signals are allowed to pass directly to the Intel Fortran RTL, See also `catch`.
`ignore fpe`	Tells the debugger to ignore the fpe signal (or the signal specified). This allows the specified signal to pass directly to the Intel Fortran RTL, allowing message display.
`ignore unaligned`	Tells the debugger to ignore the unaligned signal (the default).
`kill`	Terminates the program process, leaving the debugger running and its breakpoints and tracepoints intact for when the program is rerun.
`list`	Displays source program lines. To list a range of lines, add the starting line number, a comma (,), and the ending line number, such as `list 1,9`.
`print k`	Displays the value of the specified variable, such as `K`.
`printregs`	Displays all registers and their contents.
`next`	Steps one source statement but does *not* step into calls to subprograms, Compare with `step`.
`quit`	Ends the debugging session.
`run`	Runs the program being debugged. You can specify program arguments and redirection.
`rerun`	Runs the program being debugged again. You can specify program arguments and redirection.
`return [routine-name]`	Continues execution of the function until it returns to its caller. When using the `step` command, if you step into a subprogram that does not require further investigation, use the `return` command to continue execution of the current function until it returns to its caller. If you include the name of a routine with the `return` command, execution

	continues until control is returned to that routine. The *routine-name* is the name of the routine, usually named by a PROGRAM, SUBROUTINE, or FUNCTION statement. If there is no PROGRAM statement, the debugger refers to the main program with a prefix of `main$` followed by the file name.
`sh more progout.f90`	Executes the shell command `more` to display file `progout.f90`, then returns to the debugger environment.
`show thread`	Lists all threads known to the debugger.
`status`	Displays breakpoints and tracepoints with their event numbers. See also `delete`.
`step`	Steps one source statement, including stepping *into* calls of a subprogram. For Intel Fortran I/O statements, intrinsic procedures, library routines, or other subprograms, use the `next` command instead of `step` to step over the subprogram call. Compare with `next`; see also `return`.
`stop in foo`	Stops execution (breakpoint) at the beginning of routine `foo`.
`stop at 100`	Stops execution at line 100 (breakpoint) of the current source file.
`stopi at` *xxxxxxx*	Stops execution at address *xxxxxxx* of the current executable program.
`thread [`*n*`]`	Identifies or sets the current thread context.
`watch` *location*	Displays a message when the debuggee (or user program) accesses the specified memory location. For example: `watch 0x140000170`
`watch variable` *m*	Displays a message when the debuggee (or user program) accesses the variable specified by *m*.
`whatis` *symbol*	Displays the data type of the specified symbol.
`when at` *9* {*command*}	Executes a command or commands. When a specified line (such as 9) is reached, the *command* or commands are executed. For example, `when at 9 {print k}` prints the value of variable `K` when the program executes source code line 9.
`when in` *name* {*command*}	Executes a command or commands. When a procedure specified by *name* is reached, the *command* or commands are executed. For example, `when in calc_ave {print k}` prints the value of variable `K` when the program begins executing the procedure named `calc_ave`.
`where`	Displays the call stack.
`where thread all`	Displays the stack traces of all threads.

The debugger supports other special-purpose commands. For example:

- You might use the `attach` and `detach` commands for programs with very long execution times.
- The `listobj` command might be helpful when debugging programs that depend on shared libraries. The `listobj` command displays the names of executables and shared libraries currently known to the debugger.

Debugging the SQUARES Example Program

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
        K = K + 1       ! add this line
        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
   0  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:

1. 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.
2. Enters a loop in which it copies the square of each nonzero integer into another array OUTARR.
3. Prints the number of nonzero elements in the original sequence and the square of each such element.

This example assumes that the program was executed without array bounds checking (set by the `-check bounds` command-line option). When executed with array bounds checking, a run-time error message appears.

When you run SQUARES, it produces the following output, regardless of the number of nonzero elements in the data file:

```
squares
Number of nonzero elements is   0
```

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.

The following example shows how to start the debugging session and how to use the character-cell interface to idb to find the error in the sample program shown earlier. Comments keyed to the callouts at the right follow the example:

```
ifort -g -o squares squares.f90  (1)
idb squares                        (2)
Linux Application Debugger for xx-bit applications, Version x.x, Build
xxxx
object file name: squares
reading symbolic information ... done
(idb) list 1,9                          (3)
      1   PROGRAM SQUARES
      2     INTEGER INARR(20), OUTARR(20)
      3 C  ! Read the input array from the data file.
>     4      OPEN(UNIT=8, FILE='datafile.dat', STATUS='OLD')
      5      READ(8,*,END=5) N, (INARR(I), I=1,N)
      6    5 CLOSE (UNIT=8)
      7 C  ! Square all nonzero elements and store in OUTARR.
      8      K = 0
      9      DO 10 I = 1, N
(idb) stop at 8                         (4)
[#1: stop at "squares.f90":8]
(idb) run                               (5)
[1] stopped at ["squares.f90":4 0x120001a88]
>   8      K = 0
(idb) step                              (6)
stopped at [squares:9 0x120001a90]
      9        DO 10 I = 1, N
(idb) print n, k                        (7)
4 0
(idb) step                              (8)
stopped at [squares:10 0x120001ab0]]
     10       IF(INARR(I) .NE. 0)  THEN
(idb) s
stopped at [squares:11 0x1200011acc]
     11         OUTARR(K) =   INARR(I)**2
(idb) print i, k                        (9)
1 0
(idb) assign k = 1                      (10)
(idb) watch variable k                  (11)
[#2: watch variable (write) k 0x1400002c0 to 0x1400002c3 ]
(idb) cont                              (12)
Number of nonzero elements is   1
Element   1 has value     4
Process has exited with status 0
(idb) quit                              (13)
% vi squares.f90                            (14)
   .
   .
   .
10:       IF(INARR(I) .NE. 0) THEN
11:           K = K + 1
12:           OUTARR(K) = INARR(I)**2
13:       ENDIF
   .
   .
   .
% ifort -g -o squares squares.f90      (15)
```

```
% idb squares
Welcome to the idb Debugger Version x.x-xx
Reading symbolic information ...done
(idb) when at 12 {print k}                (16)
[#1: when at "squares.f90":12 { print K} ]
(idb) run                                 (17)
[1] when [squares:12 0x120001ae0]
1
[1] when [squares:12 0x120001ae0]
2
[1] when [squares:12 0x120001ae0]
3
[1] when [squares:12 0x120001ae0]
4
Number of nonzero elements is   4
Element   1 has value     9
Element   2 has value     4
Element   3 has value    25
Element   4 has value     4
Process has exited with status 0
(idb) quit                                (18)
%
```

1. On the command line, the `-g` option directs the compiler to write the symbol information associated with SQUARES into the object file for the debugger. It also disables most optimizations done by the compiler to ensure that the executable code matches the source code of the program.
2. The shell command `idb squares` runs the debugger, which displays its banner and the debugger prompt, `(idb)`. This command specifies the executable program as a file named `squares`. You can now enter debugger commands. After the `idb squares` command, execution is initially paused before the start of the main program unit (before program SQUARES, in this example).
3. The `list 1,9` command prints lines 1 through 9.
4. The command `stop at 8` sets a breakpoint (1) at line 8.
5. The `run` command begins program execution. The program stops at the first breakpoint, line 8, allowing you to examine variables `N` and `K` before program completion.
6. The `step` advances the program to line 9. The `step` command ignores source lines that do not result in executable code; also, by default, the debugger identifies the source line at which execution is paused. To avoid stepping into a subprogram, use the `next` command instead of `step`.
7. The command `print n, k` displays the current values of variables N and K. Their values are correct at this point in the execution of the program.
8. The two `step` commands continue executing the program into the loop (lines 9 to 11) that copies and squares all nonzero elements of INARR into OUTARR. Certain commands can be abbreviated. In this example, the s command is an abbreviation of the `step` command.
9. The command `print i, k` displays the current values of variables `I` and `K`. Variable `I` has the expected value, 1. But variable `K` has the value 0 instead of the expected value, 1. To fix this error, `K` should be incremented in the loop just before it is used in line 11.
10. The `assign` command assigns `K` the value 1.
11. The `watch variable k` command sets a watchpoint that is triggered every time the value of variable `K` changes. In the original version of the program, this

watchpoint is never triggered, indicating that the value of variable `K` never changes (a programming error).

12. To test the patch, the `cont` command (an abbreviation of `continue`) resumes execution from the current location. The program output shows that the patched program works properly, but only for the first array element. Because the watchpoint (`watch variable k` command) does not occur, the value of `K` did not change and there is a problem. The idb message "Process has exited with status 0" shows that the program executed to completion.
13. The `quit` command returns control to the shell so that you can correct the source file and recompile and relink the program.
14. The shell command `vi` runs a text editor and the source file is edited to add K = K + 1 after line 10, as shown. (Compiler-assigned line numbers have been added to clarify the example.)
15. The revised program is compiled and linked. The shell command `idb squares` starts the debugger, using the revised program so that its correct execution can be verified.
16. The `when at 12 {print k}` command reports the value of `K` at each iteration through the loop.
17. The `run` command starts execution.
18. The displayed values of `K` confirm that the program is running correctly.
19. The `quit` command ends the debugging session, returning control to the shell.

Displaying Variables in the Debugger

To refer to a variable, use either the uppercase or lowercase letters. For example:

```
(idb) print J
(idb) print j
```

You can enter command names in uppercase:

```
(idb) PRINT J
```

If you compiled the program with the command-line option `-names as_is` and you need to examine case-sensitive names, you can control whether idb is case-sensitive by setting the `$lang` environment variable to the name of a case-sensitive language.

Module Variables

To refer to a variable defined in a module, insert an opening quote (‘), the module name, and another opening quote (‘) before the variable name. For example, with a variable named `J` defined in a module named `modfile` (statement MODULE MODFILE), enter the following command to display its value:

```
(idb)    list 5,9
     5     USE MODFILE
     6      INTEGER*4 J
     7      CHARACTER*1 CHR
     8      J = 2**8
(idb)    print ‘MODFILE‘J
256
```

Common Block Variables

You can display the values of variables in a Fortran common block by using the debugger command `print`. Use the `whatis` command to return the type for the variable.

To display the entire common block, use the common block name.

To display a specific variable in a common block, use the variable name. For example:

```
(idb)     list 1,11
   1       PROGRAM EXAMPLE
   2
   3       INTEGER*4 INT4
   4       CHARACTER*1 CHR
   5       COMMON /COMSTRA/ INT4, CHR
   6
   7       CHR = 'L'
   8
   9       END
(idb)     print COMSTRA
COMMON
        INT4 = 0
        CHR = "L"
(idb)
(idb)     print CHR
"L"
```

If the name of a data item in a common block has the same name as the common block itself, the data item is accessed.

Derived-Type Variables

Variables in a Fortran 95/90 derived-type (TYPE statement) are represented in idb commands such as `print` or `whatis` using Fortran 95/90 syntax form.

For derived-type structures, use the derived-type variable name, a percent sign (%), and the member name. For example:

```
(idb)     list 3,11
    3  TYPE X
    4      INTEGER A(5)
    5    END TYPE X
    6
    7   TYPE (X) Z
    8
    9   Z%A = 1
   10
   11   PRINT *,Z%A
(idb)     print Z%A
(1) 1
(2) 1
(3) 1
(4) 1
(5) 1
(idb)
```

To display the entire object, use the `print` command with the object name. For example:

```
(idb)  print Z
```

Record Variables

To display the value of a field in a record structure, enter the variable name as: the record name, a delimiter (either a period (.) or a percent sign (%)), and the field name.

To view all fields in a record structure, enter the name of the record structure, such as REC (instead of REC.CHR or REC%CHR).

Pointer Variables

Intel Fortran supports two types of pointers:

- Fortran 95/90 pointers (standard-conforming)
- Integer pointers (extension to the Fortran 95/90 standards)

Fortran 95/90 Pointers

Fortran 95/90 pointers display their corresponding target data with a `print` command.

Comments keyed to the callouts at the right follow the example:

```
ifort -g point.f90
idb ./a.out
Linux Application Debugger for xx-bit applications, Version x.x, Build
xxxx
object file name: ./a.out
Reading symbolic information ...done
(idb) stop in ptr
[#1: stop in ptr ]
(idb) list 1:13
      1       program ptr
      2
      3       integer, target :: x(3)
      4       integer, pointer :: xp(:)
      5
      6       x = (/ 1, 2, 3/)
      7       xp => x
      8
      9       print *, "x = ", x
     10       print *, "xp = ", xp
     11
     12       end
(idb) run
[1] stopped at [ptr:6 0x120001838]
      6        x = (/ 1, 2, 3/)
(idb) whatis x
int x(1:3)
(idb) whatis xp                                          (1)
int xp(:)
(idb) s
```

```
stopped at [ptr:7 0x120001880]
      7       xp => x
(idb) s
stopped at [ptr:9 0x120001954]
      9       print *, "x = ", x
(idb) s
x =             1           2           3
stopped at [ptr:10 0x1200019c8]
(idb) s
 xp =             1           2           3
stopped at [point:12 0x120001ad8]
     12       end
(idb) S
 xp =             1           2           3
(idb) whatis xp                                          (2)
int xp(1:3)
(idb) print xp
(1) 1
(2) 2
(3) 3
(idb) quit
%
```

1. For the first `whatis xp` command, `xp` has not yet been assigned to point to variable `x` and is a generic pointer.
2. Since `xp` has been assigned to point to variable `x`, for the second `whatis xp` command, `xp` takes the same size, shape, and values as `x`.

Integer Pointers

Like Fortran 95/90 pointers, integer pointers (also known as Cray*-style pointers) display the target data in their corresponding source form with a print command:

```
(idb) stop at 14
[#1: stop at "dfpoint.f90":14 ]
(idb) run
[1] stopped at [dfpoint:14 0x1200017e4]
(idb) list 1,14
      1       program dfpoint
      2
      3       real i(5)
      4       pointer (p,i)
      5
      6       n = 5
      7
      8       p = malloc(sizeof(i(1))*n)
      9
     10       do j = 1,5
     11          i(j) = 10*j
     12       end do
     13
>    14       end
(idb) whatis p
float (1:5) pointer p
(idb) print p
0x140003060 = (1) 10
(2) 20
(3) 30
(4) 40
(5) 50
(idb) quit
```

```
%
```

Array Variables

For array variables, put subscripts within parentheses, as with Fortran 95/90 source statements. For example:

```
(idb) assign arrayc(1)=1
```

You can print out all elements of an array using its name. For example:

```
(idb) print arrayc
(1) 1
(2) 0
(3) 0
(idb)
```

Avoid displaying all elements of a large array. Instead, display specific array elements or array sections. For example, to print array element arrayc(2):

```
(idb) print arrayc(2)
(2) 0
```

Array Sections

An array section is a portion of an array that is an array itself. An array section can use subscript triplet notation consisting of a three parts: a starting element, an ending element, and a stride.

Consider the following array declarations:

```
INTEGER, DIMENSION(0:99)     :: arr
INTEGER, DIMENSION(0:4,0:4)  :: FiveByFive
```

Assume that each array has been initialized to have the value of the index in each position, for example, `FiveByFive(4,4) = 44, arr(43) = 43`. The following examples are array expressions that will be accepted by the debugger:

```
(idb) print arr(2)
2
(idb) print arr(0:9:2)
(0) = 0
(2) = 2
(4) = 4
(6) = 6
(8) = 8
(idb) print FiveByFive(:,3)
(0,3) = 3
(1,3) = 13
(2,3) = 23
(3,3) = 33
(4,3) = 43
(idb)
```

The only operations permissible on array sections are `whatis` and `print`.

Complex Variables

idb supports COMPLEX or COMPLEX*8, COMPLEX*16, and COMPLEX*32 variables and constants in expressions.

Consider the following Fortran program:

```
PROGRAM complextest
      COMPLEX*8 C8 /(2.0,8.0)/
      COMPLEX*16 C16 /(1.23,-4.56)/
      REAL*4     R4 /2.0/
      REAL*8     R8 /2.0/
      REAL*16    R16 /2.0/

      TYPE *, "C8=", C8
      TYPE *, "C16=", C16
    END PROGRAM
```

idb supports the display and assignment of COMPLEX variables and constants as well as basic arithmetic operators. For example:

```
Welcome to the idb Debugger Version x.x-xx
------------------
object file name: complex
Reading symbolic information ...done
(idb) stop in complextest
[#1: stop in complextest ]
(idb) run
[1] stopped at [complextest:15 0x1200017b4]
     15       TYPE *, "C8=", C8
(idb) whatis c8
complex c8
(idb) whatis c16
double complex c16
(idb) print c8
(2, 8)
(idb) print c16
(1.23, -4.56)
(idb) assign c16=(-2.3E+10,4.5e-2)
(idb) print c16
(-23000000512, 0.04500000178813934)
(idb)
```

Data Types

The table below shows the Intel Fortran data types and their equivalent built-in debugger names:

Fortran 95/90 data type declaration	Debugger equivalent
CHARACTER	character
INTEGER, INTEGER(KIND=*n*)	integer, integer*n
LOGICAL, LOGICAL (KIND=*n*)	logical, logical*n

REAL, REAL(KIND=4)	real
DOUBLE PRECISION, REAL(KIND=8)	real*8
REAL(KIND=16)	real*16
COMPLEX, COMPLEX(KIND=4)	complex
DOUBLE COMPLEX, COMPLEX(KIND=8)	double complex
COMPLEX(KIND=16), COMPLEX*32	long double complex

Expressions in Debugger Commands

Expressions in debugger commands use Fortran 95/90 source language syntax for operators and expressions.

Enclose debugger command expressions between curly braces ({ }). For example, the expression `print k` in the following statement is enclosed between curly braces ({ }):

```
(idb) when at 12 {print k}
```

Fortran Operators

The Intel Fortran operators include the following:

- Relational operators, such as less than (.LT. or <) and equal to (.EQ. or ==)
- Logical operators, such as logical conjunction (.AND.) and logical disjunction (.OR.)
- Arithmetic operators, including addition (+), subtraction (-), multiplication (*), and division (/).

For a complete list of operators, see the *Intel Fortran Language Reference Manual.*

Procedures

The idb debugger supports invocation of user-defined specific procedures using Fortran 95/90 source language syntax.

See Also

Intel Fortran Language Reference Manual

Online *Intel ® Debugger (IDB) Manual*

Debugging Mixed-Language Programs

The idb debugger lets you debug mixed-language programs. Program flow of control across subprograms written in different languages is transparent.

The debugger automatically identifies the language of the current subprogram or code segment on the basis of information embedded in the executable file. For example, if

program execution is suspended in a subprogram in Fortran, the current language is Fortran. If the debugger stops the program in a C function, the current language becomes C. The debugger uses the current language to determine the valid expression syntax and the semantics used to evaluate an expression.

The debugger sets the `$lang` environment variable to the language of the current subprogram or code segment. By manually setting the `$lang` environment variable, you can force the debugger to interpret expressions used in commands by the rules and semantics of a particular language. For example, you can check the current setting of `$lang` and change it as follows:

```
(idb) print $lang
"C++"
(idb) set $lang = "Fortran"
```

When the `$lang` environment variable is set to "Fortran", names are case-insensitive. To make names case-sensitive when the program was compiled with the `-names as_is` option, specify another language for the `$lang` environment variable, such as C, view the variable, then set the `$lang` environment variable to "Fortran."

Debugging a Program that Generates a Signal

If your program encounters a signal (exception) at run time, to make it easier to debug the program, you may want to recompile and relink with the following command-line options before debugging the cause of the signal:

- Use the `-fpe`*n* option to control the handling of floating point exceptions.
- As with other debugging tasks, use the `-g` option to generate sufficient symbol table information and debug unoptimized code.

If requested, idb will catch and handle signals before the Intel Fortran run-time library (RTL) does. You can use the idb commands `catch` and `ignore` to control whether idb catches signals or ignores them:

- When idb catches a signal, an idb message is displayed and execution stops at that statement line. The error-handling routines provided by the RTL are not called. At this point, you can examine variables and determine where in the program the signal has occurred.
- When idb ignores a signal, the signal is passed to the RTL. This allows the handling and display of run-time signal messages in the manner requested during compilation.

To obtain the appropriate run-time error message when debugging a program that generates a signal (especially one that allows program continuation), you might need to use the `ignore` command before running the program. For instance, use the following command to tell the debugger to ignore floating-point signals and pass them through to the RTL:

```
(idb) ignore fpe
```

In cases where you need to locate the part of the program causing a signal, consider using the `where` command.

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 option to generate sufficient table information and debug unoptimized code.

To determine the cause of the unaligned data when using idb, follow these steps:

1. Run the debugger, specifying the program with the unaligned data (shown as `testprog` in the following example): **`idb testprog`**
2. Before you run the program, enter the `catch unaligned` command:
   `(idb)` **`catch unaligned`**
3. Run the program:
   ```
   (idb) run
   Unaligned access pid=28413 <testprog> va=140000154
   pc=3ff80805d60
   ra=1200017e8 type=stl
   Thread received signal BUS
   stopped at [oops:13 0x120001834]
   13       end
   ```
4. Enter a `list` command to display the source code at line 12:
   ```
   (idb) list 12
   12         i4 = 1
   >  13           end
   ```
5. Enter the `where` command to find the location of the unaligned access:
   `(idb)` **`where`**
   This command generates a stack trace, which will indicate where the unaligned access occurred.
6. Use any other appropriate debugger commands needed to isolate the cause of the unaligned data, such as `up`, `list`, and `down`.
7. Repeat these steps for other areas where unaligned data is reported. Use the `rerun` command to run the program again instead of exiting the debugger and running it from the shell prompt.
8. After fixing the causes of the unaligned data, compile and link the program again.

For more information on data alignment, see the following:

Understanding Data Alignment
Setting Data Type and Alignment

Debugging Multithread Programs

Debugging Multithread Programs Overview

The debugging of multithreaded program discussed in this section 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.

 **Note**

Debugging of multithread programs is not supported on Mac OS* systems.

To debug the multithreaded programs, you can use:

- The Intel® Debugger (idb)
- The Intel Fortran Compiler debugging options and methods.
- The Intel parallelization extension routines for low-level debugging.
- The VTune(TM) Performance Analyzer to define the problematic areas.

Other best known debugging methods and tips include:

- Correct the program in a single-threaded, uni-processor environment
- Statically analyze locks
- Use a trace statement (such as the PRINT statement)
- Think in parallel, make very few assumptions
- Step through your code
- Make sense of threads and callstack information
- Identify the primary thread
- Know what thread you are debugging
- Single stepping in one thread does not mean single stepping in others
- Watch out for context switch

Debugger Limitations for Multithread Programs

The Intel® Debugger (idb) supports the debugging of programs that are executed by multiple threads. However, the debugger does not directly support the debugging of parallel decomposition directives, and therefore, there are limitations on the debugging features.

Some of the new features used in OpenMP are not yet fully supported by the debuggers, so it is important to understand how these features work to know how to debug them. The two problem areas are:

- Multiple entry points
- Shared variables

The Intel Debugger (idb) is not aware of and currently does not handle unique OpenMP features that relate to multi-threading.

Debugging Parallel Regions

The compiler implements a parallel region by enabling the code in the region and putting it into a separate, compiler-created entry point. Although this is different from outlining – the technique employed by other compilers, that is, creating a subroutine, – the same debugging technique can be applied.

Constructing an Entry-point Name

The compiler-generated parallel region entry point name is constructed with a concatenation of the following strings:

- `"__" character`
- entry point name for the original routine (for example, `_parallel`)
- `"_" character`
- line number of the parallel region
- `__par_region` for OpenMP parallel regions (`!$OMP PARALLEL`)
  `__par_loop` for OpenMP parallel loops (`!$OMP PARALLEL DO`),
  `__par_section` for OpenMP parallel sections (`!$OMP PARALLEL SECTIONS`)

- sequence number of the parallel region (for each source file, sequence number starts from zero.)

When you use routine names (for example, `padd`) and entry names (for example, `_PADD, ___PADD_6__par_loop0`), the following occurs. The Fortran Compiler, by default, first changes lower/mixed case routine names to upper case. For example, `pAdD` becomes `PADD`, and this becomes the entry name by adding one underscore. The secondary entry name change happens after that. That's why the "`__par_loop`" part of the entry name stays as lower case.

 **Note**

The debugger doesn't accept the upper case routine name "`PADD`" to set the breakpoint. Instead, it accepts the lower case routine name "`padd`".

Example 1 shows the debugging of the code with a parallel region. Example 1 is produced by this command:

```
ifort -openmp -g -O0 -S file.f90
```

Let us consider the code of subroutine `parallel` in Example 1.

Subroutine PARALLEL() source listing

```
1     subroutine parallel
2     integer id,OMP_GET_THREAD_NUM
3  !$OMP PARALLEL PRIVATE(id)
4     id = OMP_GET_THREAD_NUM()
5  !$OMP END PARALLEL
6     end
```

The parallel region is at line 3. The compiler created two entry points: `parallel_` and `___parallel_3__par_region0`. The first entry point corresponds to the subroutine `parallel()`, while the second entry point corresponds to the OpenMP parallel region at line 3.

Example 1 Debugging Code with Parallel Region

```
Machine Code Listing of the Subroutine parallel()

        .globl parallel_
parallel_:
..B1.1:                         # Preds ..B1.0
..LN1:
pushl     %ebp                                    #1.0
movl      %esp, %ebp                              #1.0
subl      $44, %esp                               #1.0
pushl     %edi                                    #1.0
... ... ... ... ... ... ... ... ... ... ... ... ...
..B1.13:                        # Preds ..B1.9
addl      $-12, %esp                             #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.2, (%esp) #6.0
movl      $0, 4(%esp)                            #6.0
movl      $_parallel__6__par_region1, 8(%esp)    #6.0
call      __kmpc_fork_call                       #6.0
# LOE
..B1.31:                        # Preds ..B1.13
addl      $12, %esp                              #6.0
# LOE
..B1.14:                        # Preds ..B1.31 ..B1.30
..LN4:
leave                                            #9.0
ret                                              #9.0
# LOE

.type parallel_,@function
.size parallel_,.-parallel_
.globl _parallel__3__par_region0
_parallel__3__par_region0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.15:                        # Preds ..B1.0
pushl     %ebp                                   #9.0
movl      %esp, %ebp                             #9.0
subl      $44, %esp                              #9.0
..LN5:
call      omp_get_thread_num_                    #4.0
# LOE eax
..B1.32:                        # Preds ..B1.15
movl      %eax, -32(%ebp)                        #4.0
# LOE
..B1.16:                        # Preds ..B1.32
movl      -32(%ebp), %eax                        #4.0
movl      %eax, -20(%ebp)                        #4.0
..LN6:
leave                                            #9.0
ret                                              #9.0
# LOE
.type _parallel__3__par_region0,@function
.size _parallel__3__par_region0,._parallel__3__par_region0
.globl _parallel__6__par_region1
_parallel__6__par_region1:
```

```
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.17:                        # Preds ..B1.0

pushl     %ebp                                          #9.0
movl      %esp, %ebp                                    #9.0
subl      $44, %esp                                     #9.0
..LN7:
call      omp_get_thread_num_                           #7.0
# LOE eax
..B1.33:                        # Preds ..B1.17
movl      %eax, -28(%ebp)                               #7.0
# LOE
..B1.18:                        # Preds ..B1.33
movl      -28(%ebp), %eax                               #7.0
movl      %eax, -16(%ebp)                               #7.0
..LN8:
leave                                                   #9.0
ret                                                     #9.0
.align    4,0x90
# mark_end;
```

Debugging the program at this level is just like debugging a program that uses POSIX threads directly. Breakpoints can be set in the threaded code just like any other routine. With the Intel® Debugger (idb) or the GNU debugger, breakpoints can be set to source-level routine names (such as parallel). Breakpoints can also be set to entry point names (such as parallel_ and _parallel__3__par_region0). Note that the compiler converted the upper case Fortran subroutine name to lower case.

Debugging Multiple Threads

When in a debugger, you can switch from one thread to another. Each thread has its own program counter so each thread can be in a different place in the code. Example 2 shows a Fortran subroutine `PADD()`. A breakpoint can be set at the entry point of an OpenMP parallel region.

Source listing of the Subroutine PADD()

```
12.       SUBROUTINE PADD(A, B, C, N)
13.       INTEGER N
14.       INTEGER A(N), B(N), C(N)
15.       INTEGER I, ID, OMP GET THREAD NUM
16. !$OMP PARALLEL DO SHARED (A, B, C, N) PRIVATE(ID)
17.       DO I = 1, N
18.         ID = OMP GET THREAD NUM()
19.         C(I) = A(I) + B(I) + ID
20.       ENDDO
21. !$OMP END PARALLEL DO
22.       END
```

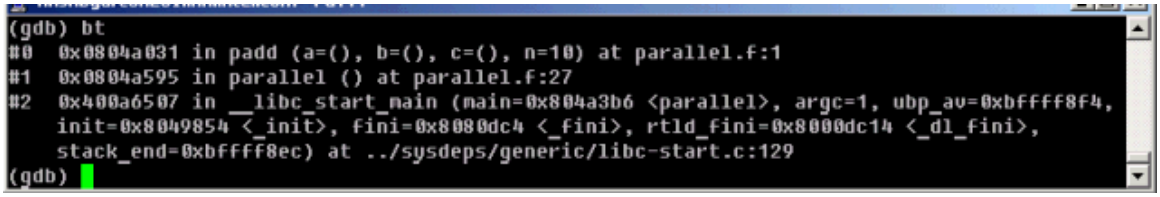
The Call Stack Dumps

The first call stack below is obtained by breaking at the entry to subroutine `PADD` using the GNU debugger. At this point, the program has not executed any OpenMP regions, and therefore has only one thread. The call stack shows a system run-time `__libc_start_main` function calling the Fortran main program `parallel`, and `parallel` calling subroutine `padd`. When the program is executed by more than one

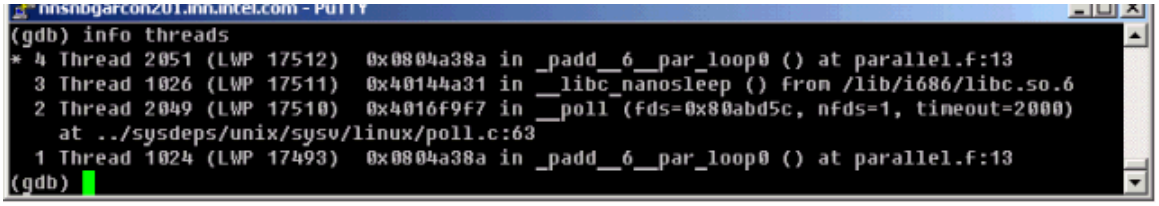
thread, you can switch from one thread to another. The second and the third call stacks are obtained by breaking at the entry to the parallel region. The call stack of master contains the complete call sequence. At the top of the call stack is `_padd__6__par_loop0`. Invocation of a threaded entry point involves a layer of Intel OpenMP library function calls (that is, functions with `__kmp` prefix). The call stack of the worker thread contains a partial call sequence that begins with a layer of Intel OpenMP library function calls.

ERRATA: The GNU debugger sometimes fails to properly unwind the call stack of the immediate caller of the Intel OpenMP library function `__kmpc_fork_call()`.

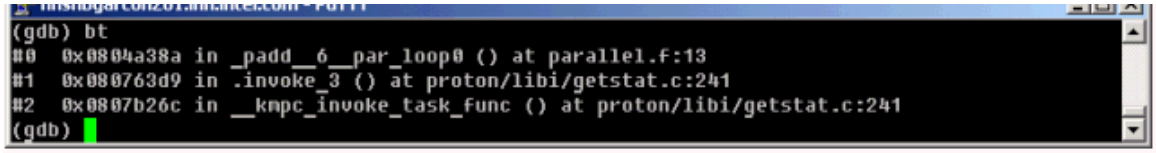
Call Stack Dump of Master Thread upon Entry to Subroutine PADD



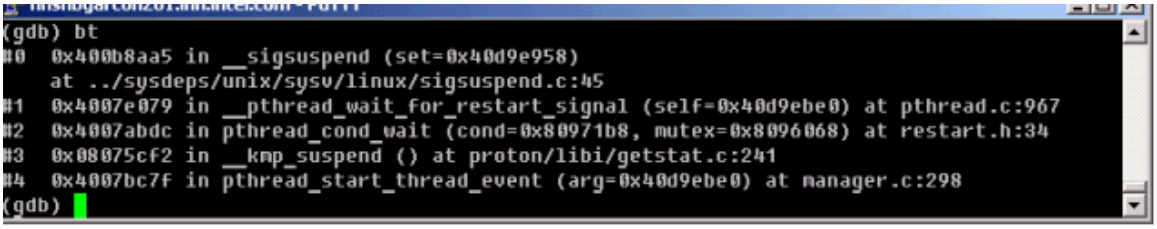
Switching from One Thread to Another



Call Stack Dump of Master Thread upon Entry to Parallel Region



Call Stack Dump of Worker Thread upon Entry to Parallel Region



Example 2 Debugging Code Using Multiple Threads with Shared Variables

```
Subroutine PADD() Machine Code Listing
      .globl padd_
padd_:
```

```
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4(n): 20 + %ebp
..B1.1:                         # Preds ..B1.0
..LN1:
pushl     %ebp                                      #1.0
... ... ... ... ... ... ... ... ... ... ... ... ...

..B1.19:                        # Preds ..B1.15
addl      $-28, %esp                                #6.0
movl      $.2.1 2 kmpc loc struct pack.1, (%esp) #6.0
movl      $4, 4(%esp)                               #6.0
movl      $ padd  6  par loop0, 8(%esp)             #6.0
movl      -196(%ebp), %eax                          #6.0
movl      %eax, 12(%esp)                            #6.0
movl      -152(%ebp), %eax                          #6.0
movl      %eax, 16(%esp)                            #6.0
movl      -112(%ebp), %eax                          #6.0
movl      %eax, 20(%esp)                            #6.0
lea       20(%ebp), %eax                            #6.0
movl      %eax, 24(%esp)                            #6.0
call        kmpc fork call                          #6.0
            # LOE
..B1.39:                        # Preds ..B1.19
addl      $28, %esp                                 #6.0
jmp         ..B1.31         # Prob 100%             #6.0
            # LOE
..B1.20:                        # Preds ..B1.30
... ... ... ... ... ... ... ... ... ... ... ... ...

call        kmpc for static init 4                  #6.0
            # LOE
..B1.40:                        # Preds ..B1.20
addl      $36, %esp                                 #6.0
            # LOE
... ... ... ... ... ... ... ... ... ... ... ... ...

..B1.26:                        # Preds ..B1.28 ..B1.21
addl      $-8, %esp                                 #6.0
movl      $.2.1 2 kmpc loc struct pack.1, (%esp) #6.0
movl      -8(%ebp), %eax                            #6.0
movl      %eax, 4(%esp)                             #6.0
call        kmpc for static fini                    #6.0
            # LOE
..B1.41:                        # Preds ..B1.26
addl      $8, %esp                                  #6.0
jmp         ..B1.31         # Prob 100%             #6.0
            # LOE
..B1.27:                        # Preds ..B1.28 ..B1.25
..LN7:
call      omp get thread num                        #8.0
             # LOE eax
..B1.42:                         # Preds ..B1.27
... ... ... ... ... ... ... ... ... ... ... ... ...

cmpl      %edx, %eax                                #10.0
jle         ..B1.27         # Prob 50%              #10.0
jmp         ..B1.26         # Prob 100%             #10.0
             # LOE
.type padd ,@function
.size padd ,.-padd
.globl  padd  6  par loop0
_padd__6__par_loop0:
```

```
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4: 20 + %ebp
# parameter 5: 24 + %ebp
# parameter 6: 28 + %ebp
..B1.30:                        # Preds ..B1.0
..LN16:
pushl     %ebp                                      #13.0
movl      %esp, %ebp                                #13.0
subl      $208, %esp                                #13.0
movl      %ebx, -4(%ebp)                            #13.0
..LN17:
movl      8(%ebp), %eax                             #6.0
movl      (%eax), %eax                              #6.0
movl      %eax, -8(%ebp)                            #6.0
movl      28(%ebp), %eax                            #6.0
..LN18:
movl      (%eax), %eax                              #7.0
movl      (%eax), %eax                              #7.0
movl      %eax, -80(%ebp)                           #7.0
movl      $1, -76(%ebp)                             #7.0
movl      -80(%ebp), %eax                           #7.0
testl     %eax, %eax                                #7.0
jg        ..B1.20       # Prob 50%                  #7.0
          # LOE
..B1.31:                      # Preds ..B1.41 ..B1.39 ..B1.38 ..B1.30
..LN19:
movl      -4(%ebp), %ebx                            #13.0
leave                                               #13.0
ret                                                 #13.0
.align    4,0x90
# mark_end;
```

Debugging Shared Variables

When a variable appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause on some block, the variable is made private to the parallel region by redeclaring it in the block. SHARED data, however, is not declared in the threaded code. Instead, it gets its declaration at the routine level. At the machine code level, these shared variables become incoming subroutine call arguments to the threaded entry points (such as `___PADD_6__par_loop0`).

In Example 2, the entry point `___PADD_6_par_loop0` has six incoming parameters. The corresponding OpenMP parallel region has four shared variables. The first two parameters (parameters 1 and 2) are reserved for the compiler's use, and each of the remaining four parameters corresponds to one shared variable. These four parameters exactly match the last four parameters to `__kmpc_fork_call()` in the machine code of `PADD`.

 **Note**

The FIRSTPRIVATE, LASTPRIVATE, and REDUCTION variables also require shared variables to get the values into or out of the parallel region.

Due to the lack of support in debuggers, the correspondence between the shared variables (in their original names) and their contents cannot be seen in the debugger at the threaded entry point level. However, you can still move to the call stack of one of the subroutines and examine the contents of the variables at that level. This technique can be used to examine the contents of shared variables. In Example 2, contents of the shared variables `A`, `B`, `C`, and `N` can be examined if you move to the call stack of `PARALLEL()`.

Data and I/O

Data Representation

Data Representation Overview

See these topics:

Intrinsic Data Types

Integer Data Representations Overview

Logical Data Representations

Native IEEE Floating-Point Representations Overview

Character Representation

Hollerith Representation

Intrinsic Data Types

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 Converting Unformatted Numeric Data.

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 example, the declaration INTEGER(4) is the same as INTEGER(KIND=4) and INTEGER*4.

Data Type	Storage	Description
BYTE INTEGER(1)	1 byte (8 bits)	A BYTE declaration is a signed integer data type equivalent to INTEGER(1).

INTEGER	See INTEGER(2), INTEGER(4), and INTEGER(8).	Signed integer, either INTEGER(2), INTEGER(4), or INTEGER(8). The size is controlled by the -integer_size *nn* compiler option. The default is -integer_size 32 (INTEGER(4)).
INTEGER(1)	1 byte (8 bits)	Signed integer value from -128 to 127.
INTEGER(2)	2 bytes (16 bits)	Signed integer value from -32,768 to 32,767.
INTEGER(4)	4 bytes (32 bits)	Signed integer value from -2,147,483,648 to 2,147,483,647.
INTEGER(8)	8 bytes (64 bits)	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)	Real floating-point values, either REAL(4), REAL(8), or REAL(16). The size is controlled by the -real_size *nn* compiler option. The default is -real_size 32 (REAL(4)).
REAL(4)	4 bytes (32 bits)	Single-precision real floating-point values in IEEE S_floating format ranging from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are denormalized (subnormal).
REAL(8) DOUBLE PRECISION	8 bytes (64 bits)	Double-precision real floating-point values in IEEE T_floating format ranging from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.94065645841246544D-324 are denormalized (subnormal).
REAL(16)	16 bytes (128 bits)	Extended-precision real floating-point values in IEEE-style X_floating format ranging from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
COMPLEX	See COMPLEX(4), COMPLEX(8) and COMPLEX(16)	Complex floating-point values 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 *nn* compiler option. The default is -real_size 32 (COMPLEX(4)).
COMPLEX(4)	8 bytes (64 bits)	Single-precision complex floating-point values in a pair of IEEE S_floating 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 denormalized (subnormal).
COMPLEX(8) DOUBLE COMPLEX	16 bytes (128 bits)	Double-precision complex floating-point values in a pair of IEEE T_floating 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 4.94065645841246544D-324 are denormalized (subnormal).
COMPLEX(16)	32 bytes (256 bits)	Extended-precision complex floating-point values in a pair of IEEE-style X_floating format parts: real and imaginary. The

		real and imaginary parts each range from 6.4751751194380251109244389582276465524996Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
LOGICAL	See LOGICAL(2), LOGICAL(4), and LOGICAL(8).	Logical value, either LOGICAL(2), LOGICAL(4), or LOGICAL(8). The size is controlled by the -integer_size *nn* compiler option. The default is -integer_size 32 (LOGICAL(4)).
LOGICAL(1)	1 byte (8 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(2)	2 bytes (16 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(4)	4 bytes (32 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(8)	8 bytes (64 bits)	Logical values .TRUE. or .FALSE.
CHARACTER	1 byte (8 bits) per character	Character data represented by character code convention. Character declarations 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	Hollerith constants.

In addition, you can define binary (bit) constants as explained in the *Language Reference*.

The following sections discuss the intrinsic data types in more detail:

- Integer Data Representations
- Logical Data Representations
- Native IEEE Floating-Point Representations
- Character Representation
- Hollerith Representation

Integer Data Representations

Integer Data Representations Overview

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)), unless the `-integer_size 16` or the `-integer_size 64` option was specified.

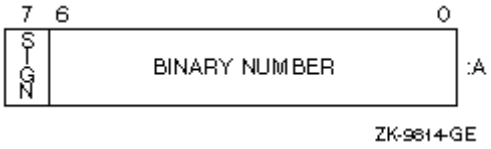
Integer data is signed with the sign bit being 0 (zero) for positive numbers and 1 for negative numbers.

The following sections discuss integer data:

- INTEGER(KIND=1) Representation
- INTEGER(KIND=2) Representation
- INTEGER(KIND=4) Representation
- INTEGER(KIND=8) Representation

INTEGER(KIND=1) Representation

INTEGER(1) values range from -128 to 127 and are stored in 1 byte, as shown below:

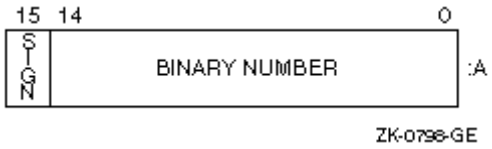


Integers are stored in a two's complement representation. For example:

```
+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:

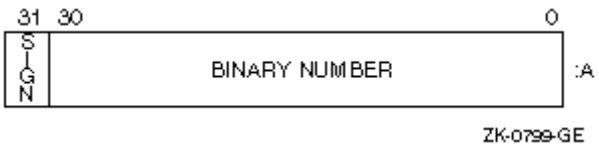


Integers are stored in a two's complement representation. For example:

```
+22 = 0016(hex)
 -7 = FFF9(hex)
```

INTEGER(KIND=4) Representation

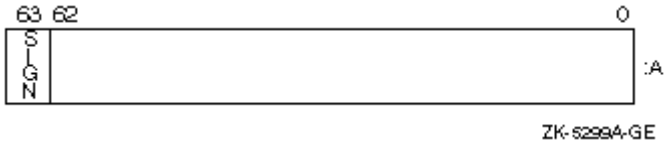
INTEGER(4) values range from -2,147,483,648 to 2,147,483,647 and are stored in 4 contiguous bytes, as shown below.



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.



Integers are stored in a two's complement representation.

Logical Data Representations

Logical data lengths 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)), unless the `-integer_size 16` or `-integer_size 64` option was specified.

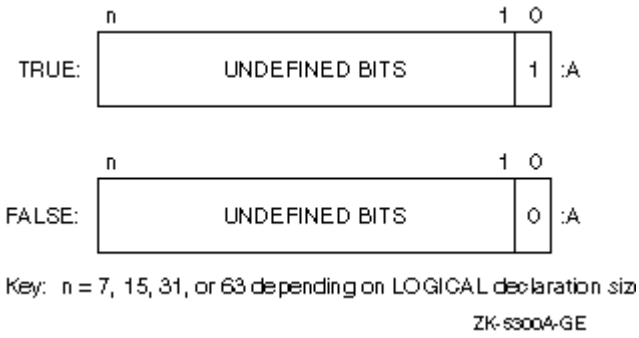
To improve performance on Intel® EM64T or Itanium®-based systems, use LOGICAL(4) (or LOGICAL(8)) rather than LOGICAL(2) or LOGICAL(1). On IA-32 systems, use LOGICAL(4) rather than LOGICAL(8), LOGICAL(2), or LOGICAL(1).

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 low-order bit determines whether the logical value is true or false. Specify `-fpscomp logical` 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 representation (when `-fpscomp nological` option was set) appears below:



Native IEEE* Floating-Point Representations

Native IEEE* Floating-Point Representations Overview

The REAL(4) (IEEE* S_floating), REAL(8) (IEEE T_floating), and REAL(16) (IEEE-style X_floating) formats are stored in standard little endian IEEE binary floating-point notation. (See IEEE Standard 754 for additional information about IEEE binary floating point notation.) COMPLEX formats use a pair of REAL values to denote the real and imaginary parts of the data.

All floating-point formats represent fractions in sign-magnitude notation, with the binary radix point to the right of the most-significant bit. Fractions are assumed to be normalized, and therefore the most-significant bit is not stored (this is called "hidden bit normalization"). This bit is assumed to be 1 unless the exponent is 0. If the exponent equals 0, then the value represented is denormalized (subnormal) or plus or minus zero.

Intrinsic REAL kinds are 4 (single precision), 8 (double precision), and 16 (extended precision), such as REAL(KIND=4) for single-precision floating-point data. Intrinsic COMPLEX kinds are also 4 (single precision), 8 (double precision), and 16 (extended precision).

To obtain the kind of a variable, use the KIND intrinsic function. You can also use a size specifier, such as REAL*4, but be aware this is an extension to the Fortran 95 standard.

If you omit certain compiler options, the default sizes for REAL and COMPLEX data declarations are as follows:

- For REAL data declarations without a kind parameter (or size specifier), the default size is REAL (KIND=4) (same as REAL*4).
- For COMPLEX data declarations without a kind parameter (or size specifier), the default data size is COMPLEX (KIND=4) (same as COMPLEX*8).

To control the size of all REAL or COMPLEX declarations without a kind parameter, use the `-real_size 64` or `-real_size 128` options; the default is `-real_size 32`.

You can explicitly declare the length of a REAL or a COMPLEX declaration using a kind parameter, or specify DOUBLE PRECISION or DOUBLE COMPLEX. To control the size of all DOUBLE PRECISION and DOUBLE COMPLEX declarations, use the `-double_size 128` option; the default is `-double_size 64`.

The following sections discuss floating-point data:

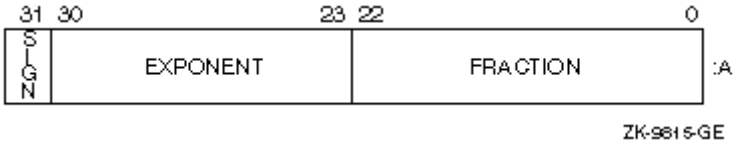
- REAL(KIND=4) (Single Precision) Representation
- REAL(KIND=8) (Double Precision) Representation
- REAL(KIND=16) (Extended Precision) Representation
- COMPLEX(KIND=4) (Single Complex) Representation
- COMPLEX(KIND=8) (Double Complex) Representation
- COMPLEX(KIND=16 (Extended Precision Complex) Representation

For information on reading or writing floating-point data other than native IEEE little endian data, see Converting Unformatted Numeric Data.

See also File fordef.for and Its Usage and Using the Floating Point Exception Handling (-fpe) Option.

REAL(KIND=4) (REAL) Representation

REAL(4) (same as REAL(KIND=4)) data occupies 4 contiguous bytes stored in IEEE S_floating format. Bits are labeled from the right, 0 through 31, as shown below.

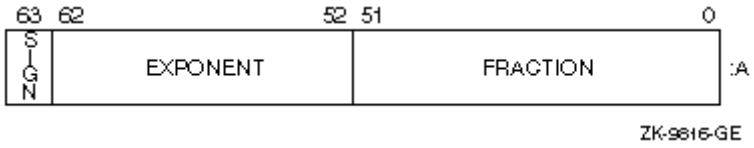


The form of REAL(4) data is sign magnitude, with bit 31 the sign bit (0 for positive numbers, 1 for negative numbers), bits 30:23 a binary exponent in biased 127 notation, and bits 22:0 a normalized 24-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range: 1.17549435E-38 (normalized) to 3.40282347E38. The IEEE denormalized (subnormal) limit is 1.40129846E-45. The precision is approximately one part in 2**23; typically 7 decimal digits.

REAL(KIND=8) (DOUBLE PRECISION) Representation

REAL(8) (same as REAL(KIND=8)) data occupies 8 contiguous bytes stored in IEEE T_floating format. Bits are labeled from the right, 0 through 63, as shown below.

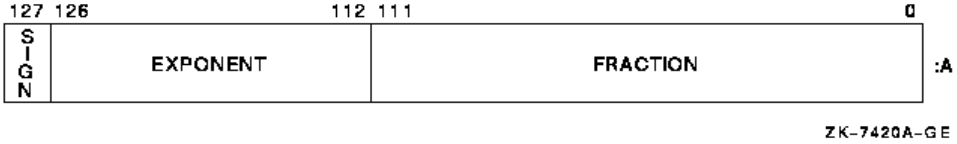


The form of REAL(8) data is sign magnitude, with bit 63 the sign bit (0 for positive numbers, 1 for negative numbers), bits 62:52 a binary exponent in biased 1023 notation, and bits 51:0 a normalized 53-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range: 2.2250738585072013D-308 (normalized) to 1.7976931348623158D308. The IEEE denormalized (subnormal) limit is 4.94065645841246544D-324. The precision is approximately one part in 2**52; typically 15 decimal digits.

REAL(KIND=16) (EXTENDED PRECISION) Representation

REAL(16) (same as REAL(KIND=16)) data occupies 16 contiguous bytes stored in IEEE-style X_floating format. Bits are labeled from the right, 0 through 127, as shown below.



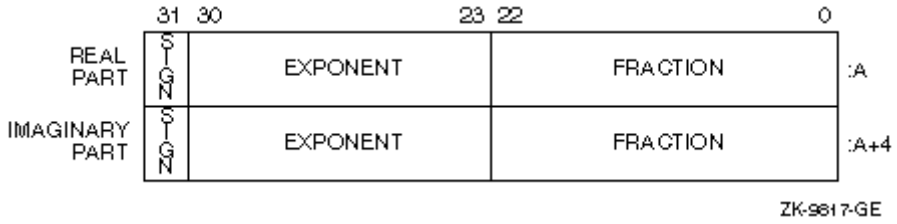
The form of REAL(16) data is sign magnitude, with bit 127 the sign bit (0 for positive numbers, 1 for negative numbers), bits 126:112 a binary exponent in biased 16383 notation, and bits 111:0 a normalized 113-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range:
6.475175119438025110924438958227646552499 6Q-4966 to
1.189731495357231765085759326628007016196477Q4932. Unlike other floating-point formats, there is little if any performance penalty from using denormalized extended-precision numbers. This is because accessing denormalized REAL (KIND=16) numbers does not result in an arithmetic trap (the extended-precision format is emulated in software). The smallest normalized number is
3.362103143112093506262677817321753Q-4932.

The precision is approximately one part in 2**112 or typically 33 decimal digits.

COMPLEX(KIND=4) (COMPLEX) Representation

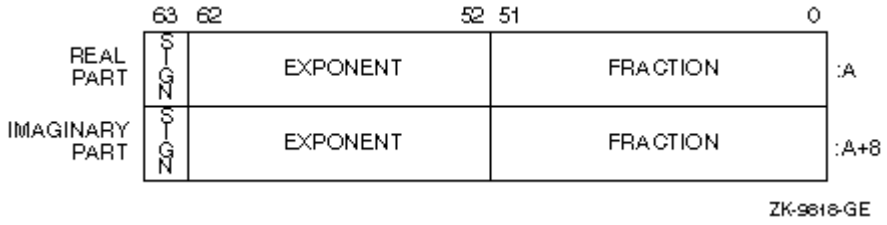
COMPLEX(4) (same as COMPLEX(KIND=4) and COMPLEX*8) data is 8 contiguous bytes containing a pair of REAL(4) values stored in IEEE S_floating format. The low-order 4 bytes contain REAL(4) data that represents the real part of the complex number. The high-order 4 bytes contain REAL(4) data that represents the imaginary part of the complex number, as shown below.



The limits and underflow characteristics for REAL(4) apply to the two separate real and imaginary parts of a COMPLEX(4) number. Like REAL(4) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

COMPLEX(KIND=8) (DOUBLE COMPLEX) Representation

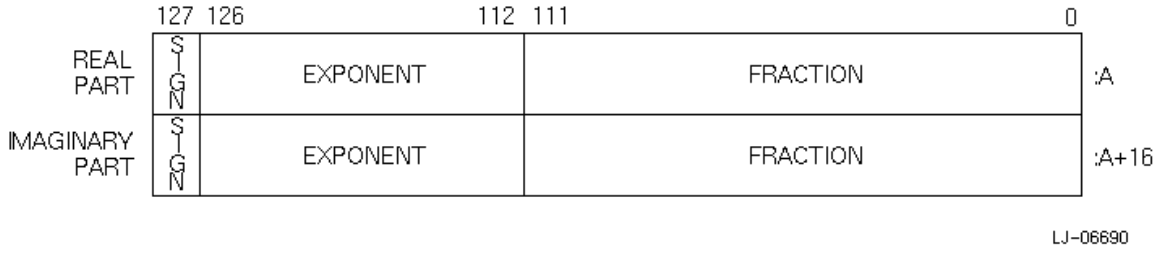
COMPLEX(8) (same as COMPLEX(KIND=8) and COMPLEX*16) data is 16 contiguous bytes containing a pair of REAL(8) values stored in IEEE T_floating format. The low-order 8 bytes contain REAL(8) data that represents the real part of the complex data. The high-order 8 bytes contain REAL(8) data that represents the imaginary part of the complex data, as shown below.



The limits and underflow characteristics for REAL(8) apply to the two separate real and imaginary parts of a COMPLEX(8) number. Like REAL(8) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

COMPLEX(KIND=16) Representation

COMPLEX(16) (same as COMPLEX(KIND=16) or COMPLEX*32) data is 32 contiguous bytes containing a pair of REAL(16) values stored in IEEE-style X_floating format. The low-order 16 bytes contain REAL(16) data that represents the real part of the complex data. The high-order 16 bytes contain REAL(8) data that represents the imaginary part of the complex data, as shown below.



The limits and underflow characteristics for REAL(16) apply to the two separate real and imaginary parts of a COMPLEX(16) number. Like REAL(16) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

File fordef.for and Its Usage

The parameter file `/opt/intel/fc/9.1.xxx/include/fordef.for` contains symbols and INTEGER*4 values corresponding to the classes of floating-point representations. Some of these classes are exceptional ones such as bit patterns that represent positive denormalized numbers.

With this file of symbols and with the FP_CLASS intrinsic function, you have the flexibility of identifying exceptional numbers so that, for example, you can replace positive and negative denormalized numbers with true zero.

The following is a simple example of identifying floating-point bit representations:

```
include 'fordef.for'
real*4 a
integer*4 class of bits
a = 57.0   ! Bit pattern is a finite number
class of bits = fp class(a)
if ( class of bits .eq. for k fp pos norm  .or.   &
class_of_bits .eq. for_k_fp_neg_norm       ) then
print *, a, ' is a non-zero and non-exceptional value'
else
print *, a, ' is zero or an exceptional value'
end if
end
```

In this example, the symbol `for_k_fp_pos_norm` from the `fordef.for` file plus the REAL*4 value 57.0 to the FP_CLASS intrinsic function results in the execution of the first print statement.

The table below explains the symbols in the `fordef.for` file and their corresponding floating-point representations.

Symbols in File fordef.for

Symbol Name	Class of Floating-Point Bit Representation
FOR_K_FP_SNAN	Signaling NaN
FOR_K_FP_QNAN	Quiet NaN
FOR_K_FP_POS_INF	Positive infinity
FOR_K_FP_NEG_INF	Negative infinity
FOR_K_FP_POS_NORM	Positive normalized finite number
FOR_K_FP_NEG_NORM	Negative normalized finite number
FOR_K_FP_POS_DENORM	Positive denormalized number
FOR_K_FP_NEG_DENORM	Negative denormalized number
FOR_K_FP_POS_ZERO	Positive zero
FOR_K_FP_NEG_ZERO	Negative zero

Another example of using file `fordef.for` and intrinsic function FP_CLASS follows. The goals of this program are to quickly read any 32-bit pattern into a REAL*4 number from an unformatted file with no exception reporting and to replace denormalized numbers with true zero:

```
include 'fordef.for'
real*4 a(100)
integer*4 class_of_bits
!     open an unformatted file as unit 1
!     ...
read (1) a
do i = 1, 100
class of bits = fp class(a(i))
if ( class_of_bits .eq. for_k_fp_pos_denorm  .or.   &
                  class of bits .eq. for k fp neg denorm       ) then
  a(i) = 0.0
end if
```

```
end do
close (1)
end
```

You can compile this program with any value of `-fpe`*n*. Intrinsic function FP_CLASS helps to find and replace denormalized numbers with zeroes before the program can attempt to perform calculations on the denormalized numbers. On the other hand, if this program did not replace denormalized numbers read from unit 1 with zeroes and the program was compiled with `-fpe0`, then the first attempted calculation on a denormalized number would result in a floating-point exception.

File `fordef.for` and intrinsic function FP_CLASS can work together to identify NaNs. A variation of the previous example would contain the symbols `for_k_fp_snan` and `for_k_fp_qnan` in the IF statement. A faster way to do this is based on the intrinsic function ISNAN. One modification of the previous example, using ISNAN, follows:

```
!     The ISNAN function does not need file fordef.for
      real*4 a(100)
!     open an unformatted file as unit 1
!     ...
      read (1) a
      do i = 1, 100
         if ( isnan (a(i)) ) then
            print *, 'Element ', i, ' contains a NaN'
         end if
      end do
      close (1)
      end
```

Using the Floating Point Exception Handling (-fpe) Option

The Fortran compiler supports several kinds of floating-point exceptions; a summary of their masked (or default) responses is given below:

- **overflow:** Overflow is signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result. The result computed is rounding mode specific:
  - Round-to-nearest (default): +/- Infinity in specified precision
  - Round-to-zero: +/- Maximum Number in specified precision
  - Round-to-+Infinity: +Infinity or –(Maximum Positive Number) in specified precision
  - Round-to--Infinity: (Maximum Positive Number) or -Infinity in specified precision

  For example, in round-to-nearest mode 1E30 * 1E30 overflows the single-precision floating-point range and results in a +Infinity; -1E30 * 1E30 results in a -Infinity.

- **divide-by-zero:** Divide-by-zero is signaled when the divisor is zero and the dividend is a finite nonzero number. The computed result is a correctly signed Infinity.
  For example, 2.0E0/+0.0 produces a divide-by-zero exception and results in a

+Infinity; -2.0E0/+0.0 produces a divide-by-zero exception and results in a –Infinity.

- **underflow:** Underflow occurs when a computed result (of an add, subtract, multiply, divide, or math function call) falls beyond the minimum range in magnitude of normalized numbers of the floating-point data type. Each floating-point type (32-, 64-, and 128-bit) has a denormalized range where very small numbers can be represented with some loss of precision. This is called gradual underflow. For example, the lower bound for normalized single-precision float-point is approximately 1E-38, while the lower bound for denormalized single-precision float-point is approximately 1E-45. Results falling below the lower bound of the denormalized range simply become zero. 1E-30 / 1E10 underflows the normalized range but not the denormalized range so the result is the denormal value 1E-40. 1E-30 / 1E30 underflows the entire range and the result is zero.
- **invalid:** Invalid occurs when operands to the basic floating-point operations or math function inputs produce an undefined (QNaN) result. Some examples include:
  - SNaN operand in any floating-point operation or math function call.
  - Division of zeroes: (+/-0.0)/(+/-0.0)
  - Sum of Infinities having different signs: Infinity + (-Infinity)
  - Difference of Infinities having the same sign: (+/-Infinity) – (+/-Infinity)
  - Product of signed Infinities with zero: (+/-Inf) * 0
  - Math Function Domain Errors: log(negative), sqrt(negative), asin(|x}>1)

The -`fpe`*n* option allows some control over the results of floating-point exceptions.

`-fpe0` restricts floating-point exceptions as follows:

- Enables the overflow, the divide-by-zero, and the invalid floating-point exceptions. The program will print an error message and abort if any of these exceptions occurs. If a floating-point underflow occurs, the result is set to zero and execution continues. This is called flush-to-zero. This option sets -IPF_fp_speculationstrict if no specific -IPF_fp_speculation option is specified. The `-fpe0` option sets `-ftz`. To get more detailed location information about where the exception occurred, use `-traceback`.

  Note
  On IA-32 and Intel® EM64T systems, explicitly setting `-fpe0` can degrade performance since the generated code stream must be synchronized after each floating-point instruction to allow for abrupt underflow fix-up.

`-fpe1` restricts only floating-point underflow:

- Floating-point overflow, floating-point divide-by-zero, and floating-point invalid produce exceptional values (NaN and signed Infinities) and execution continues. If a floating-point underflow occurs, the result is set to zero and execution continues. The `-fpe1` option sets `-ftz`.

`-fpe3`, the default, allows full floating-point exception behavior:

- Floating overflow, floating divide-by-zero, and floating invalid produce exceptional values (NaN and signed Infinities) and execution continues. Floating underflow is gradual: denormalized values are produced until the result becomes 0.

The `-fpe` option affects the Fortran main program only. The floating-point exception behavior set by the Fortran main program remains in effect throughout the execution of the entire program unless changed by the programmer. If the main program is not Fortran, the user can use the Fortran intrinsic FOR_SET_FPE to set the floating-point exception behavior.

When compiling different routines in a program separately, you should use the same value of `n` in `-fpen`.

An example follows:

```
       IMPLICIT NONE
       real*4 res_uflow, res_oflow
       real*4 res_dbyz, res_inv
       real*4 small, big, zero, scale
       small = 1.0e-30
       big   = 1.0e30
       zero  = 0.0
       scale = 1.0e-10
!      IEEE underflow condition (Underflow Raised)
       res_uflow = small * scale
       write(6,100)"Underflow: ",small, " *", scale, " = ", res_uflow
!      IEEE overflow condition (Overflow Raised)
       res_oflow = big * big
       write(6,100)"Overflow: ", big, " *", big, " = ", res_oflow
!      IEEE divide-by-zero condition (Divide by Zero Raised)
       res_dbyz = -big / zero
       write(6,100)"Div-by-zero: ", -big, " /", zero, " = ", res_dbyz
!      IEEE invalid condition (Invalid Raised)
       res_inv = zero / zero
       write(6,100)"Invalid: ", zero, " /", zero, " = ", res_inv
 100   format(A14,E8.1,A2,E8.1,A2,E10.1)
       end
```

Consider the following command line:

```
ifort fpe.f -fpe0 -g
```

The following output is produced:

```
./a.out
   Underflow:  0.1E-29 * 0.1E-09 =   0.0E+00
forrtl: error (72): floating overflow
Image        PC        Routine     Line        Source
a.out        0804A063  Unknown     Unknown  Unknown
a.out        08049E78  Unknown     Unknown  Unknown
Unknown      B746B748  Unknown     Unknown  Unknown
a.out        08049D31  Unknown     Unknown  Unknown
Aborted
```

The following command line uses `-fpe1`:

```
ifort fpe.f -fpe1 -g
```

The following output is produced:

```
./a.out
   Underflow:  0.1E-29 * 0.1E-09 =   0.0E+00
    Overflow:  0.1E+31 * 0.1E+31 = Infinity
 Div-by-zero: -0.1E+31 / 0.0E+00 = -Infinity
     Invalid:  0.0E+00 / 0.0E+00 = NaN
```

The following command line uses -`fpe3`:

```
ifort fpe.f -fpe3 -g
```

The following output is produced:

```
./a.out
   Underflow:  0.1E-29 * 0.1E-09 =   0.1E-39
    Overflow:  0.1E+31 * 0.1E+31 = Infinity
 Div-by-zero: -0.1E+31 / 0.0E+00 = -Infinity
     Invalid:  0.0E+00 / 0.0E+00 = NaN
```

Relationship between the -fpe Option and the -ftz Option

The -`ftz` option affects the results of floating underflow in the following ways:

- `-ftz` results in abrupt underflow to 0: the result of a floating underflow is set to zero and execution continues. -ftz also makes a denomal value used in a computation be treated as a zero so no floating invalid exception will occur.
- `-ftz-` results in gradual underflow to 0: the result of a floating underflow is a denormalized number or a zero.
- If the optimization level is -O2 or -O3, abrupt underflow to zero is set by default. At lower optimization levels, gradual underflow to 0 is the default.
- The `-fpe0` and `-fpe1` options set `-ftz`.

The `-ftz` option sets or resets the FTZ and the DAZ hardware flags in the MXCSR register. Setting FTZ on means that denormal results from floating-point calculations will be set to the value zero. If FTZ is off, denormal results remain as is. Setting DAZ on means that denormal values used as input to floating-point instructions will be treated as zero. If DAZ is off, denormal instruction inputs remain as is.

Itanium®-based systems have the FTZ, but not the DAZ hardware flag. Intel® EM64T-based systems have both FTZ and DAZ hardware flags. FTZ and DAZ are not supported on all IA-32 architectures.

When `-ftz` is used in combination with an SSE-enabling option (for example, `-xN`) on IA-32 systems, the compiler will insert code in the main routine to set FTZ and DAZ. When `-ftz` is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a runtime processor check (this processor check fails for non-Intel machines). Specifying `-no-ftz` will prevent the compiler from inserting any code that might set FTZ or DAZ.

On IA-32 and Intel® EM64T systems with SSE and SSE2 code, there is no performance degradation and possibly even a performance gain due to the fact that the hardware handles the denormals. The default setting of abrupt underflow affects the SSE hardware, not the instruction stream.

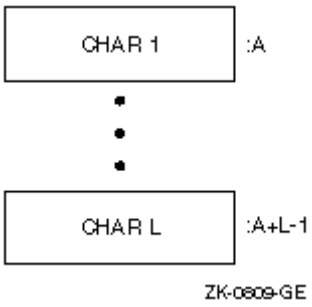
On Itanium®-based processors, gradual underflow to 0 can degrade performance. You can improve performance by using higher optimization levels to get the default abrupt underflow or explicitly setting `-ftz`.

See Also:

-fpe compiler option
-ftz compiler option

Character Representation

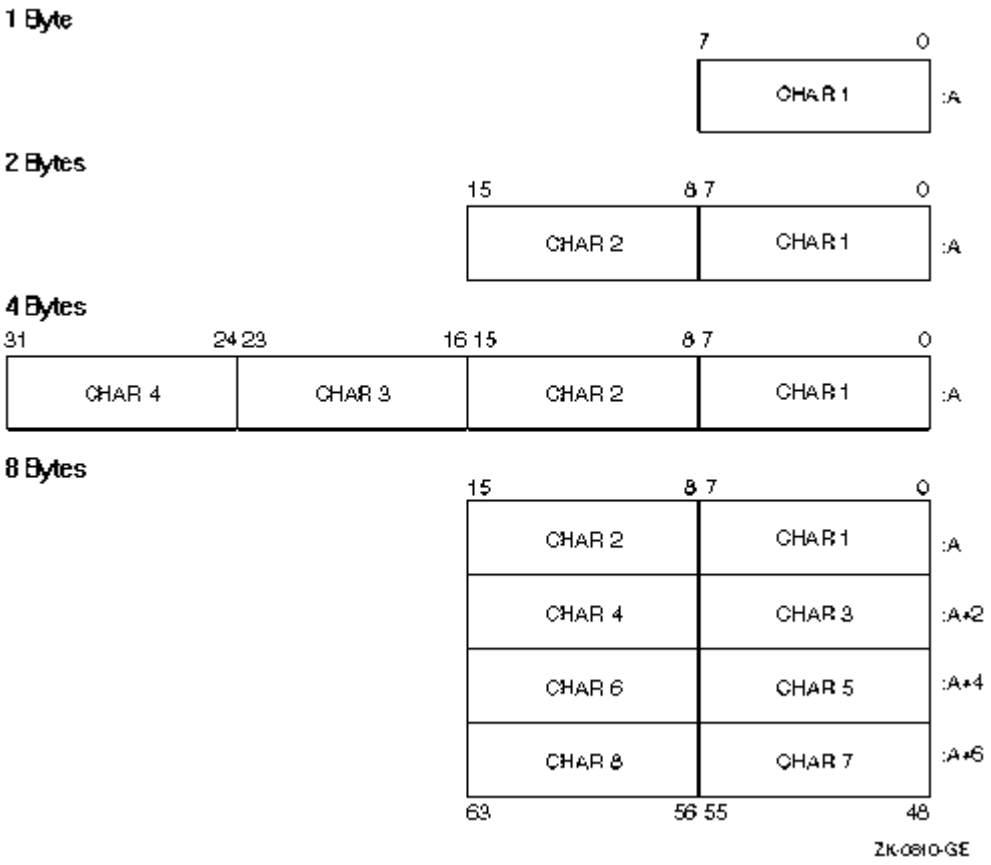
A character string is a contiguous sequence of bytes in memory, as shown below.



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. The length L of a string is in the range 1 through 2,147,483,647 (2**31-1) for IA-32 systems and in the range 1 through 9,223,372,036,854,775,807 for Intel® EM64T and Itanium®-based systems.

Hollerith Representation

Hollerith constants are stored internally, one character per byte, as shown below.



Converting Unformatted Data

Converting Unformatted Data Overview

This section describes how you can use Intel® Fortran to read and write nonnative unformatted numeric data.

See these topics:

Supported Native and Nonnative Numeric Formats

Limitations of Numeric Conversion

Methods of Specifying the Data Format: Overview

Porting Nonnative Data

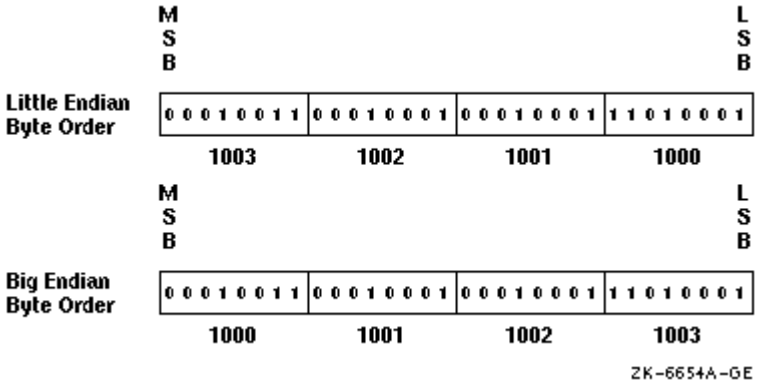
Supported Native and Nonnative Numeric Formats

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.

The following figure shows the difference between the two byte-ordering schemes:

Little and Big Endian Storage of an INTEGER Value



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 VAX* little endian floating-point formats supported by VAX FORTRAN, 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.

The native memory format includes little endian integers and little endian IEEE floating-point formats, S_floating for REAL(KIND=4) and COMPLEX(KIND=4) declarations, T_floating for REAL(KIND=8) and COMPLEX(KIND=8) declarations, and IEEE style X_floating 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 Format Keywords and Supported Data Types

Keyword	Description
BIG_ENDIAN	Big endian integer data of the appropriate INTEGER size (one, two, four, or eight bytes) and big endian IEEE floating-point formats for REAL and COMPLEX single- and 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 INTEGER 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	Native little endian integers of the appropriate INTEGER 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 style X_float for REAL (KIND=16) and COMPLEX (KIND=16)
FGX	Native little endian integers of the appropriate INTEGER 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 style X_float 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: • S_float for REAL (KIND=4) and COMPLEX (KIND=4) • T_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)

NATIVE	No conversion occurs between memory and disk. This is the default for unformatted files.
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

Limitations of Numeric Conversion

The Intel Fortran floating-point conversion solution is not expected to fulfill all floating-point conversion needs.

For instance, data fields in record structure variables (specified in a STRUCTURE statement) and data components of derived types (TYPE statement) are not converted. When they are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified. With EQUIVALENCE statements, the data type of the variable named in the I/O statement is used.

If a program reads an I/O record containing multiple format floating-point fields into a single variable (such as an array) instead of their respective variables, the fields will not be converted. When they are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified.

Methods of Specifying the Data Format

Methods of Specifying the Data Format: Overview

There are six methods of specifying a nonnative numeric format for unformatted data:

- Setting 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. See Environment Variable FORT_CONVERTn Method.
- Setting 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). See Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method.
- Setting an environment variable for a set of units before any files are opened. The environment variable is named F_UFMTENDIAN. See Environment Variable F_UFMTENDIAN Method.
- Specifying the CONVERT keyword in the OPEN statement for a specific unit number. See OPEN Statement CONVERT Method.
- Compiling 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. See OPTIONS Statement Method.
- Compiling the program with the command line `-convert` *`keyword`* option, which affects all unit numbers that use unformatted data specified by the program. See Compiler Option -convert Method.

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 to:

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).
4. Check the OPEN statement CONVERT qualifier.
5. Check whether an OPTIONS statement with a CONVERT=*keyword* qualifier was present when the program was compiled.
6. Check whether the compiler option `-convert` *`keyword`* was present when the program was compiled.

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 instance, you might use this method to specify that all files with a specific extension will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

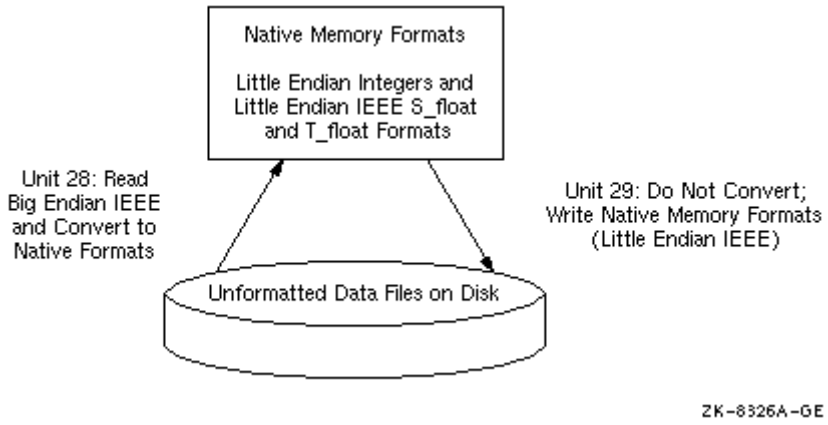
For example, assume you have a previously compiled program that reads numeric data from unit 28 and writes it to unit 29 using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from unit 28 and write that data in native little endian format to 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 the program (`/usr/users/leslie/convieee`):

```
setenv FORT_CONVERT28 BIG_ENDIAN
setenv FORT_CONVERT29 NATIVE
/usr/users/leslie/convieee
```

The following figure shows the data formats used on disk and in memory when the example file (`/usr/users/leslie/convieee`) is run after the environment variables are set.

Sample Unformatted File Conversion



This method takes precedence over other methods.

Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method

You can use this method to specify a nonnative numeric format for each specified file name extension (suffix). You 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. You can 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 extension and write that data in native little endian format to a file with a extension of .data. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format (S_float and T_float) 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 this program, the following command sequence sets the appropriate environment variables before running the program:

```
setenv FORT_CONVERT.DAT BIG_ENDIAN
/usr/users/proj2/cvbigend
```

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*. 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.

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.

Little-Big Endian Conversion Environment Variable

In order to use the little-endian-big-endian conversion feature, specify the numbers of the units to be used for conversion purposes by setting the `F_UFMTENDIAN` environment variable. Then, the READ/WRITE statements that use these unit numbers will perform relevant conversions. Other READ/WRITE statements will work in the usual way.

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value. The variable has the following syntax:

```
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 the current format of the data to be processed on all of the units in the program; it can be omitted.
  The keyword `little` means that the data is in little-endian format and will not be converted. This is the default.
  The keyword `big` means that the data is in big-endian format and will be converted.
- `EXCEPTION` defines the list of units that are the exception to `MODE`. The `EXCEPTION` keyword `little` or `big` defines the data format for the units in the `EXCEPTION` list. This value overrides the `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 range of units. The number of list members is limited to 64.
- `decimal` is a non-negative decimal number less than 2^{32}.

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.

Another Possible Environment Variable Setting

The environment variable can also have the following syntax:

```
F_UFMTENDIAN=u[,u] . . .
```

The command line for the variable setting in the shell is:

```
Sh: export  F_UFMTENDIAN=u[,u] . . .
```

Usage 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`

In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

3. `F_UFMTENDIAN="big;little:8"`

In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

4. `F_UFMTENDIAN=10-20`

Define 10, 11, 12 … 19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

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

C  prepare little endian representation of data

open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

C  prepare big endian representation of data

open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

C  read big endian data and operate with them on
C  little endian machine

open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4

C    Any operation with data, which have been read
```

```
C    . . .
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

You can 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`, `FORT_CONVERT_dat`, or `F_UFMTENDIAN` 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 (FORT_CONVERT*n*, FORT_CONVERT.*ext*, FORT_CONVERT_*ext*, or F_UFMTENDIAN), which take precedence over the OPEN statement CONVERT *keyword* 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 the 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.

The environment variable methods and the OPEN statement CONVERT method take precedence over this method. For instance, you might use the environment variable FORT_CONVERTn method or OPEN statement CONVERT method to specify each unit number that will use a format other than that specified using the `ifort` compiler option method.

This method takes precedence over the compiler option method.

You can use OPTIONS statements to specify the appropriate numeric formats in unformatted files instead of using the corresponding compiler command qualifiers. For example, to use VAX F_floating and G_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.

Compiler Option -convert Method

You can only specify one numeric format for all unformatted file unit numbers using the compiler option `-convert` method unless you also use one (or more) of the previous methods.

You specify the numeric format at compile time and must compile all routines under the same `-convert` *`keyword`* compiler option. You could use the same source program and compile it using different compiler commands to create multiple executable programs that each read a certain format.

If you specify other methods, they take precedence over this method. For instance, you might use the environment variable or OPEN statement CONVERT keyword 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, S_float and T_float little endian IEEE floating-point format). The created file, `vconvert.exe`, can then be run:

```
ifort file.for -o vconvert.exe -convert vaxd
```

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 method alone. You can if you use it in combination with 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 following topic:

- -convert compiler option

Porting Nonnative Data

Keep this information in mind when porting nonnative data:

- When porting source code along with the unformatted data, 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 the RECL units (bytes or longwords) for unformatted files without source file modification, use the -assume byterecl compiler option.

  The Fortran 95 standard (American National Standard Fortran 95, ANSI X3J3/96-007, and International Standards Organization standard ISO/IEC 1539-1:1997, states: "If the file is being connected for unformatted input/output, the length is measured in processor-dependent units."

- 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 instead of 01 to denote "true."
- Source code being ported may be coded specifically for big endian use.

Fortran I/O

Fortran I/O Overview

This section contains the following topics:

Logical I/O Units

Types of I/O Statements

Forms of I/O Statements

Files and File Characteristics Overview

Accessing and Assigning Files

Default Pathnames and File Names

Using Preconnected Standard I/O Files

Opening Files: OPEN Statement

Obtaining File Information: INQUIRE Statement

Closing a File: CLOSE Statement

Record Operations Overview

User-Supplied OPEN Procedures: USEROPEN Specifier

Format of Record Types

Microsoft* Fortran PowerStation* Compatible Files

In addition, *Optimizing Applications* contains guidelines for improving I/O performance. For more information, see Improving I/O Performance.

Logical I/O Units

Every file, internal or external, is associated with a logical device, sometimes referred to as a logical unit. You identify the logical device associated with a file by a 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 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 six 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, 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_PRINT 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 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	Description
Asterisk (*)	Always represents the keyboard and screen (unless the appropriate environment variable is defined, such as FOR_READ).
0	Initially represents the screen (unless FORT0 is defined)
5	Initially represents the keyboard (unless FORT5 is defined)
6	Initially represents the screen (unless FORT6 is defined)

The asterisk (*) specifier is the only unit specifier that cannot be reconnected to another file, and attempting to close this unit causes a compile-time error. Units 0, 5, and 6, however, can be connected to any file with the OPEN statement. If you close unit 0, 5, or

6, it is automatically reconnected to its preconnected device the next time an I/O statement attempts to use that unit.

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* (except when the compiler option `-fpscomp filesfromcmd` 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 a = 0.1, 6.3, 0.1
           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
```

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.

Internal Files

The unit specifier associated with an internal file is a character string or character array. There are two types of internal files:

- An internal file that is a character variable, character array element, or noncharacter array element that is exactly one record, which is the same length as the variable, array element, or noncharacter array element.
- An internal file that is a character array, a character derived type, or a noncharacter array that is a sequence of elements, each of which is a record. The order of records is the same as the order of array elements or type elements, and the record length is the length of one array element or the length of the derived-type element.

Follow these rules when using internal files:

- Use only formatted I/O, including I/O formatted with a format specification and list-directed I/O. (List-directed I/O is treated as sequential formatted I/O.) Namelist formatting is not allowed.
- 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 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, 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. That is, 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')
```

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	
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 a 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` (same as PRINT).
PRINT	Transfers data from internal storage to `stdout`. Unlike WRITE, PRINT only provides formatted sequential output and does not specify a unit number.

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 the main record I/O statements, by category, 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
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.

Files and File Characteristics

Files and File Characteristics Overview

See these topics:

File Organization

Internal Files and Scratch Files

Record Types

Record Overhead

Record Length

File Organization

File organization refers to the way records are physically arranged on a storage device.

Intel Fortran supports two kinds of file organization:

- Sequential
- Relative

The default file organization is always ORGANIZATION= 'SEQUENTIAL' for an OPEN statement. The organization of a file is specified by means of the ORGANIZATION specifier in the 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 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.

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 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 (direct access). (Detecting deleted records is only available if you specified the `-vms` option when the program was 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 two other types of files that are not file organizations:

- Internal files
- 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 and internal storage (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 in (and later deleted from) the directory specified in the OPEN statement DEFAULTFILE (if specified).

Record Types

Record type refers to whether records stored 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.

You can use any of the record types with sequential files. Relative files require the fixed-length record type.

When creating a new file or opening an existing file, specify one of the record types described below.

See also Format of Record Types,

Fixed-Length Record Type

Records in the file must be the same length.

You must specify the record length (RECL) when the file is opened.

See also Fixed-Length Records.

Variable-Length Record Type

Records in the file can vary in length.

Record length information is stored as control bytes at the beginning and end of each record.

See also Variable-Length Records.

Segmented Record Type

This pertains to a single logical record containing one or more unformatted records of varying length, which can only be used for unformatted sequential access.

Avoid the segmented record type when the application requires that the same file be used for programs written in languages other than Fortran and for non-Intel platforms.

See also Segmented Records.

Stream Record Type

A stream file is not grouped into records and uses no record delimiters.

Stream files contain character or binary data that is read or written to the extent of the sizes of the variables specified. Specify CARRIAGECONTROL= ' NONE ' for stream files.

See also Stream Files.

Stream_LF and Stream_CR Record Type

Records are of varying length where the line feed (LF) or the carriage return (CR) character serve as record delimiters (LF for Stream_LF files and CR for Stream_CR files).

Stream_LF files must not contain embedded LF characters or use CARRIAGECONTROL= ' LIST ' . Instead, specify CARRIAGECONTROL= ' NONE ' . Stream_CR files must not contain embedded CR characters. The Stream_LF record type is the usual record type for text files.

See also Stream_LF and Stream_CR Records.

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, and Stream_LF 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, 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 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.

Record Overhead

Record overhead 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.

The various record types each require a different number of bytes for record overhead, as described in the table below:

Record Type	File Organization	Record Overhead
Fixed-length	Sequential	None.
Fixed-length	Relative	None if the `-vms` option was omitted. One byte if the `-vms` option was specified.
Variable-length	Sequential	Eight bytes per record.
Segmented	Sequential	Four bytes per record. One additional padding byte (space) is added if the specified record size is an odd number.
Stream	Sequential	None required.
Stream_CR	Sequential	One byte per record.
Stream_LF	Sequential	One byte per record.

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` 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.

Accessing and Assigning Files

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.

Assigning Files to Logical Units

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

For example:

```
OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
```

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`).

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 directory `/usr/users/smith`.

Default Pathnames and File Names

Intel Fortran provides the following possible ways of specifying all or part of a file specification (directory and file name), such as `/usr/proj/testdata`:

- 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 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.

Using Preconnected Standard I/O Files

If you do not use an OPEN statement to open logical unit 5, 6, or 0 and do not set the appropriate environment variable (`FORT`*n*), Intel Fortran, at run time, implicitly opens (preconnected) units 5, 6, and 0 and associates them with the following operating system standard I/O files:

Unit	Environment Variable	Equivalent Linux* Standard I/O File
5	`FORT5`	Standard input, `stdin`
6	`FORT6`	Standard output, `stdout`
0	`FORT0`	Standard error, `stderr`

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 `<`).

Opening Files: OPEN Statement

To open a file, you should use a preconnected file (such as for terminal output) or explicitly open files with an OPEN statement. Although you can also implicitly open a file,

this prevents you from using the OPEN statement to 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 in the *Intel Fortran Language Reference Manual*) 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.
- 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 or sequential).
- 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.

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 block physical I/O buffer 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.

Coding 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 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 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 a File: CLOSE Statement

Usually, any external file opened should be closed by the same program before it completes. The CLOSE statement 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.

There usually is no need to close preconnected units. Internal files are neither opened nor closed.

Record Operations

Record Operations Overview

See these topics:

Record I/O Statement Specifiers

Record Access

File Sharing

Specifying the Initial Record Position

Advancing and Nonadvancing Record I/O

Record Transfer

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.

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 instance, you can:

- 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
```

Limitations of Record Access by File Organization and Record Type

You can use both 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
Relative file organization		
Fixed	Yes	Yes

Note

Direct access and relative files require that the file resides on a disk device.

File Sharing

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
```

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.

User-Supplied OPEN Procedures: USEROPEN Specifier

You can use the USEROPEN specifier in an Intel Fortran 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 establish special context that changes the effect of subsequent Intel Fortran I/O statements.

The Intel Fortran 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 the file (or pipe) and return the file descriptor of the file when it returns control to the RTL.

When opening the file, the called function usually specifies options different from those provided by a normal OPEN statement.

You can obtain the file descriptor from the Intel Fortran RTL for a specific unit number with the `getfd` 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.

Syntax and Behavior of the USEROPEN Specifier

The USEROPEN specifier for the OPEN statement has the form:

```
USEROPEN = function-name
```

function-name represents the name of an external function. In the calling program, the function must be declared in an EXTERNAL statement. For example, the following Intel Fortran code might 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 OPEN statement, the `uopen_` function receives control. The function opens the file, may perform other operations, and subsequently returns control (with the file descriptor) to the RTL.

If the USEROPEN function is written in C, declare it as a C function that returns a 4-byte integer (`int`) result to contain the file descriptor. For example:

```
int   uopen_ (                (1)
   char  *file_name,        (2)
   int   *open_flags,       (3)
   int   *create_mode,      (4)
```

```
int   *lun,              (5)
int   file_length);      (6)
```

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. The first argument is the pathname (includes the file name) to be opened.
3. The open flags are described in the header file `/usr/include/sys/file.h` or `open(2)`.
4. The create mode (protection needed when creating a file) is described in `open(2)`.
5. The fourth argument is the logical unit number.
6. The fifth (last) argument is the pathname length (hidden length argument of the pathname).

Of the arguments, the `open` system call (see `open(2)`) requires the passed pathname, the open flags (that 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 length of the pathname is also passed.

When creating a new file, the create system call might be used in place of open (see `create(2)`). You can usually use other appropriate 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.

If the USEROPEN function is written in Fortran, declare it as a FUNCTION with an INTEGER (KIND=4) result, perhaps with an interface block. In any case, the called function must return the file descriptor as a 4-byte integer to the RTL.

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.

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.

System calls and library routines that 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.

Example USEROPEN Program and Function

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 the default Fortran compiler options are used, the external name is passed using lowercase letters with an appended trailing underscore (_). In the preceding example, the external function UOPEN would be known as `uopen_` to the linker and must be declared in C as `uopen_`.

Compiling and Linking the C and Intel Fortran Programs

Use the icc 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 to create the file `a.out` file, as follows:

```
icc -c uopen_.c
ifort ex1.f uopen_.o
```

Source Code for the C Function and Header File

The following example shows the C language function called `uopen_` and its associated header file.

```
/*

** File: uopen.h -- header file for uopen_.c
*/

#ifndef UOPEN
#define UOPEN 1
/*
**
**    Function Prototypes
**
*/
int   uopen_ (
   char  *file_name,    /* access read: name of the file to open. */
   int   *open_flags,   /* access read: READ/WRITE, see file.h or
open(2)*/
   int   *create_mode,  /* access read: set if new file (to be
created).*/
   int   *lun,          /* access read: logical unit file opened on.*/
   int  file_length);   /* access read: number of characters in
file_name*/

#endif

/* End of file uopen.h */

/*
```

```
** File: uopen_.c
*/

/*
** This routine opens a file using data passed by Intel Fortran RTL.
**
**  INCLUDE FILES
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include "uopen.h"/* Include file for this module */

int uopen_ (file_name, open_flags, create_mode, lun, file_length)

/*
** Open a file using the parameters passed by the calling Intel
**   Fortran program.
**
**  Formal Parameters:
*/

char  *file_name;    /* access read: name of the file to open. */
int   *open_flags;   /* access read: READ/WRITE, see file.h */
int   *create_mode;  /* access read: set if new file (to be created). */
int   *lun;          /* access read: logical unit number file opened on.
*/
int   file_length;   /* access read: number of characters in file_name.
*/

/*
**  Function Value/Completion Code
**
** Whatever is returned by open is immediately returned to the
** Fortran OPEN.  The returned value is the following:
**    value >= 0 is a valid fd.
**    value <  0 is an error.
**
**  Modify open flags (logical OR) to specify the file be opened for
**  write access only, with records appended at the end (such as
**  writing to a shared log file).
*/

{
    int    result ;           /* Function result value */

    *open_flags =
          O_CREAT  |
          O_WRONLY |
          O_APPEND;

   result = open (file_name, *open_flags, *create_mode) ;

    return (result) ;         /* return file descriptor or error */

    }/* End of routine uopen_ */

   /* End of file uopen_.c */
```

Source Code for the Calling Intel Fortran Program

The following example shows the Fortran program that calls the `uopen_ C` function and then performs I/O.

```
!
!  Program EX1 opens a file using USEROPEN and writes records to it.
!  It closes and re-opens the file (without USEROPEN) and reads 10
records.
PROGRAM EX1

  EXTERNAL    UOPEN         ! The USEROPEN function.
  INTEGER     ERRNUM, CTR, I

1   FORMAT (I)
  ERRNUM = 0
  WRITE (6,*) 'EX1. Access data using formatted I/O.'
  WRITE (6,*) 'EX1. Open file with USEROPEN and put some data in it.'

 OPEN (UNIT=1, FILE='ex1.dat', STATUS='NEW', USEROPEN=UOPEN,
ERR=9,IOSTAT=errnum)
  DO CTR=1,10
   WRITE (1,1) CTR
  END DO
  WRITE (6,*) 'EX1. Close and re-open without USEROPEN.'
  CLOSE (UNIT=1)
  OPEN (UNIT=1, FILE='ex1.dat', STATUS='OLD', FORM='FORMATTED', ERR=99,
IOSTAT=errnum)
  WRITE (6,*) 'EX1. Read and display what is in file.'
  DO CTR=1,10
     READ (1,1) i
     WRITE (6,*) i
  END DO
  WRITE (6,*) 'EX1.  Successful if 10 records shown.'
  CLOSE (UNIT=1,STATUS='DELETE')
  STOP
9 WRITE (6,*) 'EX1.  Error on USEROPEN is ', errnum
  STOP
99 WRITE (6,*) 'EX1.  Error on 2nd open is ', errnum
END PROGRAM EX1
```

Format of Record Types

Fixed-Length Records

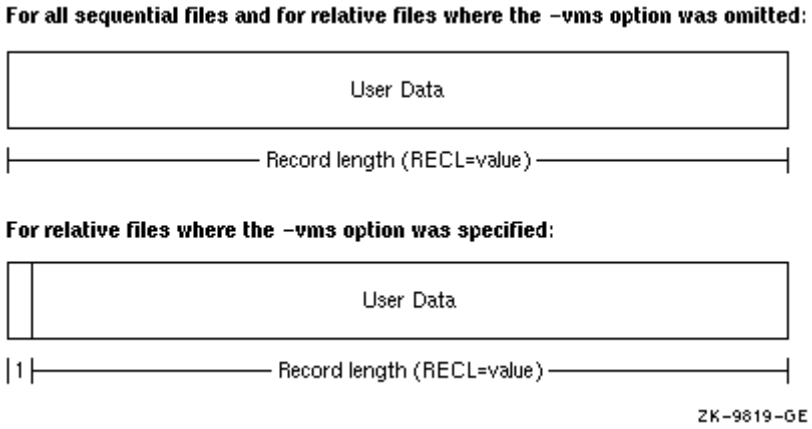
When you specify fixed-length records, all records in the file contain the same number of bytes. When you open a file that is to contain fixed-length records, you must specify the record size by using the RECL specifier. A sequentially organized file opened for direct access must contain fixed-length records, to allow the record position in the file to be computed correctly.

For relative files, the layout and overhead of fixed-length records depend on whether or not the program accessing the file was compiled with the `-vms` option.

For relative files where the `-vms` option was omitted, each record has no control information.

For relative files where the `-vms` option was specified, each record has one byte of control information at the beginning of the record.

The figure below shows the record layout of fixed-length records:



Variable-Length Records

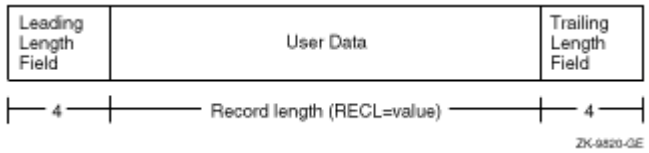
Variable-length records can contain any number of bytes up to a specified maximum record length, and apply only to sequential files.

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.

Variable-Length Records Less Than 2 Gigabytes

The figure below shows the record layout of variable-length records that are less than 2 gigabytes:



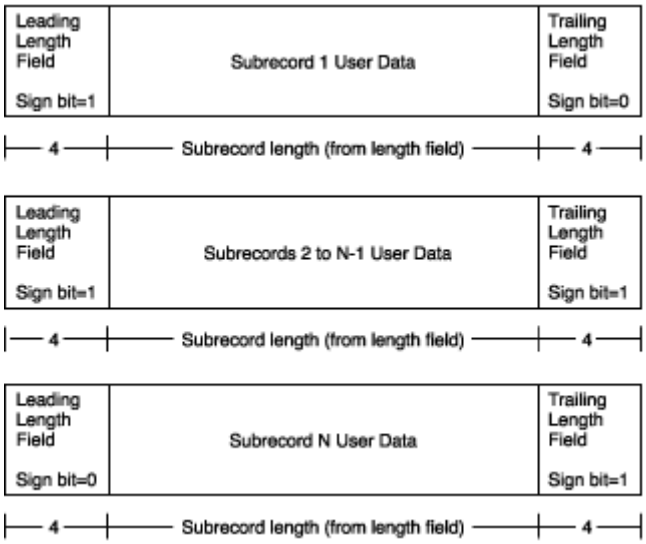
Variable-Length Records Greater Than 2 Gigabytes

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.

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 figure below 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

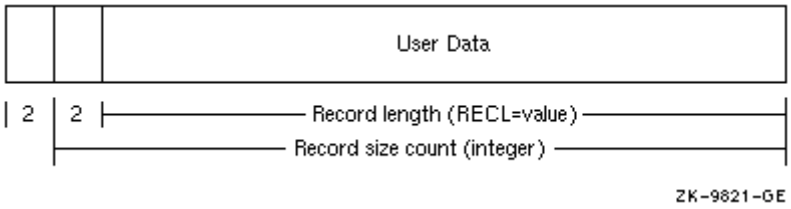
A *segmented record* is a single *logical record* consisting of one or more variable-length, unformatted records in a sequentially organized disk file. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

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.

As shown in the figure below, 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 size count (includes the two bytes used by the segment identifier), followed by a 2-byte integer segment identifier that identifies this segment as one of the following:

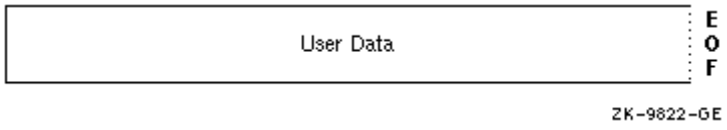
Identifier Value	Segment Identified
0	One of the segments between the first and last segments
1	First segment
2	Last segment
3	Only segment

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.

Stream File

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 sizes of the variables specified on the input or output statement.

The figure below shows the layout of a Stream file:



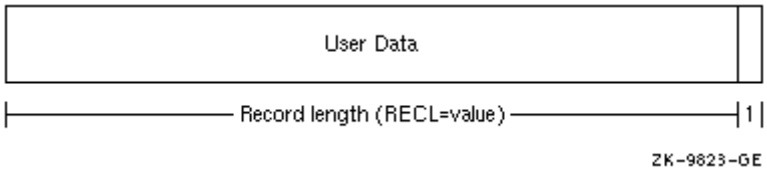
Stream_CR and Stream_LF Records

A Stream_CR or Stream_LF record is a variable-length record 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 a different 1-byte record terminator:

- Stream_CR files use only a carriage-return as the terminator, so Stream_CR files must not contain embedded carriage-return characters.
- Stream_LF files use only a line-feed (new line) as the terminator, so Stream_LF files must not contain embedded line-feed (new line) characters. This is the usual operating system text file record type.

The figure below shows the layout of Stream_CR and Stream_LF records:



Microsoft* Fortran PowerStation* Compatible Files

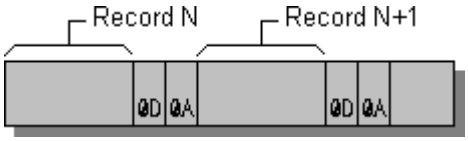
When using the `-fpscomp` options for Microsoft* Fortran PowerStation* compatibility, the following types of files are possible:

- Formatted Sequential
- Formatted Direct
- Unformatted Sequential
- Unformatted 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.

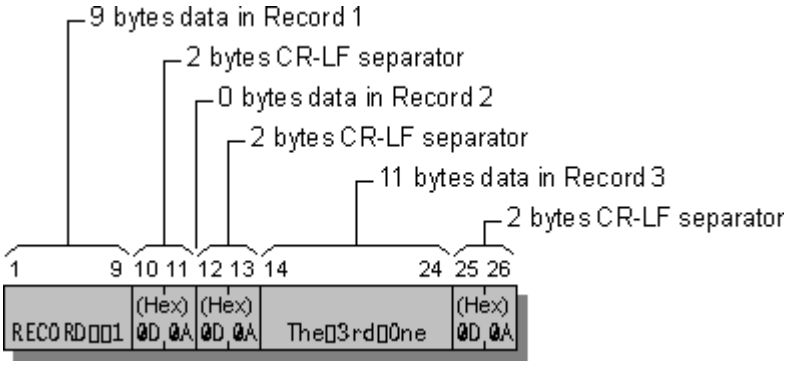
Formatted Records in a Formatted Sequential File



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.

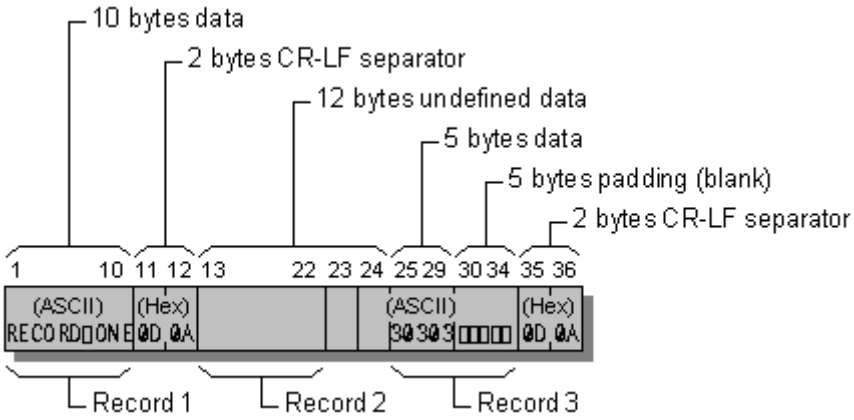
You can override the default blank padding on input by setting PAD='NO' in the OPEN statement for the file. If PAD='NO', the input record must contain the amount of data indicated by the input list and format specification. Otherwise, an error occurs. PAD='NO' has no effect on output.

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

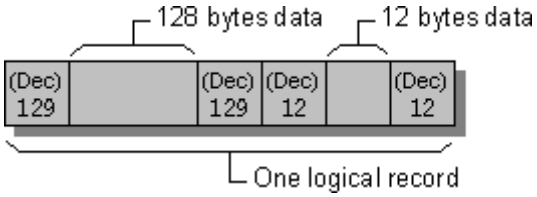
Unformatted sequential files are organized slightly differently on different platforms. This section describes unformatted sequential files created by Intel Fortran when the `-fpscomp` option (such as `-fpscomp ioformat`) was specified. If you are accessing files from another platform that organizes them differently, see Converting Unformatted Data Overview.

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

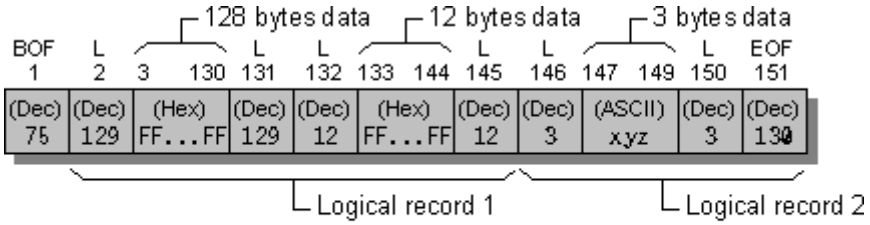


The first and last bytes in an unformatted sequential file are reserved; the first contains a value of 75, and the last holds a value of 130. Fortran uses these bytes for error checking and end-of-file references.

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



BOF Beginning-of-file byte (75 decimal)
L Physical-block-length byte (0 <= L <= 129)
EOF End-of-file byte (130 decimal)

Unformatted Direct Files

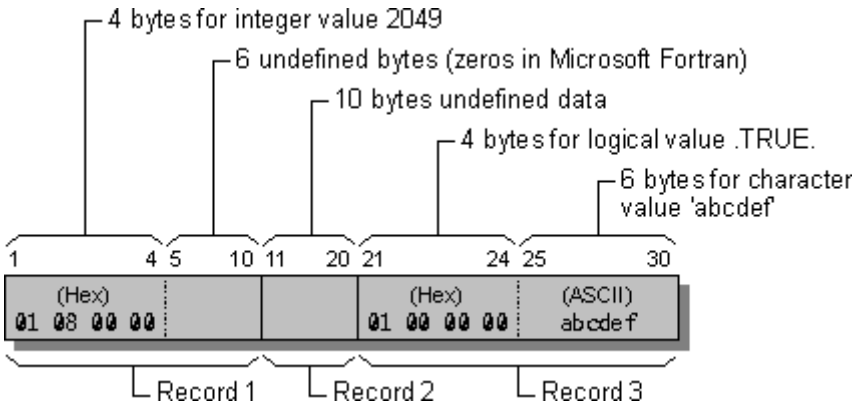
An unformatted direct file is a series of unformatted records. You can write or read the records in any order you choose. All records have the same length, given by the RECL specifier in an OPEN statement. No delimiting bytes separate records or otherwise indicate record structure.

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



Programming with Mixed Languages

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. This capability allows you to:

- Call existing code that is written in another language
- Use procedures that may be difficult to implement in a particular language
- Gain advantages in processing speeds

Mixed-language programming is possible between Intel® Fortran and other languages. Although other languages (such as assembly language) are discussed, the primary focus of this section is programming using Intel Fortran and C/C++ . Mixed language programming between these two languages is relatively straightforward because each language implements functions, subroutines, and procedures in approximately the same way.

Summary of Mixed-Language Issues

Mixed-language programming involves a call from a routine written in one language to a function, procedure, or subroutine written in another language. For example, a Fortran main program may need to execute a specific task that you want to program separately in an assembly-language procedure, or you may need to call an existing shared library or system procedure.

Programming with Fortran and C/C++ Considerations

A summary of major Fortran and C/C++ mixed-language issues follows:

- Fortran and C implement functions and routines differently. For example, a C main program could call an external void function, which is actually implemented as a Fortran subroutine:

  Language Equivalents for Calls to Routines

Language	Call with Return Value	Call with No Return Value
Fortran	FUNCTION	SUBROUTINE
C and C++	function	(void) function

- Generally, Fortran/C programs are mixed to allow one to use existing code written in the other language. Either Fortran or C can call the other, so the main routine can be in either language. If Fortran is not the main routine, the `-nofor_main` compiler option must be specified on the command line.
- Fortran adds an underscore to external names; C does not.
- Fortran changes the case of external names to lowercase; C leaves them in their original case.
- Fortran passes numeric data by reference; C passes by value.

  Note

  You can override some default Fortran behavior by using ATTRIBUTES and ALIAS. ATTRIBUTES C causes Fortran to act like C in external names and the passing of numeric data. ALIAS causes Fortran to use external names in their original case.

- 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 the following data types, Fortran adds a hidden first argument to contain function return values: COMPLEX, REAL*16, and CHARACTER.
- The -fexceptions option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.

Programming with Fortran and Assembly-Language Considerations

A summary of Fortran/assembly language issues follows:

- Assembly-language routines can be small and can execute quickly because they do not require initialization as do high-level languages like Fortran and C.
- Assembly-language routines allow access to hardware instructions unavailable to the high-level language user. In a Fortran/assembly-language program, compiling the main routine in Fortran gives the assembly code access to Fortran high-level procedures and library functions, yet allows freedom to tune the assembly-language routines for maximum speed and efficiency. The main program can also be an assembly-language program.

Other Mixed-Language Programming Considerations

There are other important differences in languages; 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. The remainder of this section provides an explanation of the techniques you can use to reconcile differences between Fortran and other languages.

Adjusting calling conventions, adjusting naming conventions and writing interface procedures are discussed in the following topics:

- Adjusting Calling Conventions in Mixed-Language Programming
- Adjusting Naming Conventions in Mixed-Language Programming
- Prototyping a Procedure in Fortran

After establishing a consistent interface between mixed-language procedures, you then need to reconcile any differences in the treatment of individual data types (strings, arrays, and so on). This is discussed in Exchanging and Accessing Data in Mixed-Language Programming. You also need to be concerned with data types, because each language handles them differently. This is discussed in Handling Data Types in Mixed-Language Programming. Finally, you may need to debug a mixed language programs, as detailed in Debugging Mixed-Language Programs.

Note

This section uses the term "routine" in a generic way, to refer to functions, subroutines, and procedures from different languages.

Calling Subprograms from the Main Program

Calls from the Main Program

The Intel® Fortran main program can call Intel Fortran subprograms, including subprograms in static and shared libraries.

For mixed-language applications, the Intel Fortran main program can call subprograms written in C/C++ if the appropriate calling conventions are used (see Calling C Procedures from a Fortran program).

Intel Fortran subprograms can be called by C/C++ main programs

Calls to the Subprogram

You can use subprograms in static libraries if the main program is written in Intel Fortran or C/C++.

You can use subprograms in shared libraries in mixed-language applications if the main program is written in Intel Fortran or C/C++.

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:

```
icc -c cfunc.c
ifort -o myprog myprog.for cfunc.o
```

The `icc` (for Intel C++) command compiles `cfunc.c`. The `-c` option specifies that the linker is not called. This command creates `cfunc.o`. The `ifort` command compiles `myprog.for` and links `cfunc.o` with the object file created from `myprog.for` to create `myprog`.

You can use the `-cxxlib[-mode]` compiler option to instruct the Fortran driver to add the C++ libraries to the link command. The *mode* argument specifies which C++ run-time libraries to use. By default, C++ libraries are not linked with Fortran applications.

You can 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. By default, mixed Fortran/C++ applications abort in the Fortran code if a C++ exception is thrown.

If your C/C++ program calls an Intel Fortran subprogram, specify the option `-nofor_main` on the ifort command line:

```
icc -c  cmain.c
ifort -nofor_main cmain.o fsub.f90
```

Using Modules in Fortran/C Mixed-Language Programming

Modules are the simplest way to exchange large groups of variables with C, because Intel Fortran modules are directly accessible from C/C++.

The following example declares a module in Fortran, then accesses its data from C:

Fortran code

```
! F90 Module definition
    MODULE EXAMP
       REAL A(3)
       INTEGER I1, I2
       CHARACTER(80) LINE
       TYPE MYDATA
          SEQUENCE
          INTEGER N
          CHARACTER(30) INFO
       END TYPE MYDATA
    END MODULE EXAMP
```

C code

```
  \* C code accessing module data *\
extern float examp_mp_a[3];
extern int examp_mp_i1, examp_mp_i2;
extern char examp_mp_line[80];
extern struct {
           int n;
           char info[30];
} examp_mp_mydata;
```

When the C++ code resides in a `.cpp` file, C++ semantics are applied to external names, often resulting in linker errors. In this case, use the extern "C" syntax (see C/C++ Naming Conventions):

```
\* C code accessing module data in .cpp file*\
extern "C" float examp_mp_a[3];
extern "C" int examp_mp_i1, examp_mp_i2;
extern "C" char examp_mp_line[80];
extern "C" struct {
           int n;
           char info[30];
} examp_mp_mydata;
```

You can define an interface to a C routine in a module, then use it like you would an interface to a Fortran routine. The C code is:

```
 // C procedure
void pythagoras (float a, float b, float *c)
{
    *c = (float) sqrt(a*a + b*b);
}
```

Using the same example when the C++ code resides in a `.cpp` file, use the extern "C" syntax (see C/C++ Naming Conventions):

```
// C procedure
extern "C" void pythagoras (float a, float b, float *c)
{
    *c = (float) sqrt(a*a + b*b);
}
```

The Fortran code to define the module CPROC:

```
 ! Fortran 95/90 Module including procedure
    MODULE CPROC
       INTERFACE
          SUBROUTINE PYTHAGORAS (a, b, res)
           !DEC$ ATTRIBUTES C :: PYTHAGORAS
           !DEC$ ATTRIBUTES REFERENCE :: res
! res is passed by REFERENCE because its individual attribute
! overrides the subroutine's C attribute
             REAL a, b, res
! a and b have the VALUE attribute by default because
! the subroutine has the C attribute
          END SUBROUTINE
       END INTERFACE
    END MODULE
```

The Fortran code to call this routine using the module CPROC:

```
 ! Fortran 95/90 Module including procedure
    USE CPROC
       CALL PYTHAGORAS (3.0, 4.0, X)
       TYPE *,X
    END
```

Calling C Procedures from an Intel® Fortran Program

Naming Conventions

By default, the Fortran compiler converts function and subprogram names to lower case. The C compiler never performs case conversion. A C procedure called from a Fortran

program must, therefore, be named using the appropriate case. For example, consider the following calls:

`CALL PROCNAME()`	The C procedure must be named `PROCNAME`.
`X=FNNAME()`	The C procedure must be named `FNNAME`

In the first call, any value returned by `PROCNAME` is ignored. In the second call to a function, `FNNAME` must return a value.

Passing Arguments Between Fortran and C Procedures

By default, Fortran subprograms pass arguments by reference; that is, they pass a pointer to each actual argument rather than the value of the argument. C programs, however, 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.

Adjusting Calling Conventions in Mixed-Language Programming

Adjusting Calling Conventions in Mixed-Language Programming Overview

The calling convention determines how a program makes a call to a routine, how the arguments are passed, and how the routines are named.

A calling convention includes:

- Naming conventions
  - Is lowercase or uppercase significant or not significant?
  - Are external names altered?
- Argument passing protocol
  - Are arguments passed by value or by reference?
  - What are the equivalent data types and data structures among languages?

In a single-language program, calling conventions are nearly always correct, because there is one default for all routines and because header files or Fortran module files with interface blocks enforce consistency between the caller and the called routine.

In a mixed-language program, different languages cannot share the same header files. If you link Fortran and C routines that use different calling conventions, the error is not apparent until the bad call is made at run time. During execution, the bad call causes indeterminate results and/or a fatal error. The error, caused by memory or stack corruption due to calling errors, often occurs in a seemingly arbitrary place in the program.

The discussion of calling conventions between languages applies only to external procedures. You cannot call internal procedures from outside the program unit that contains them.

A calling convention affects programming in four ways:

1. The caller routine uses a calling convention to determine the order in which to pass arguments to another routine; the called routine uses a calling convention to determine the order in which to receive the arguments passed to it. In Fortran, you can specify these conventions in a mixed-language interface with the INTERFACE statement or in a data or function declaration. C/C++ and Fortran both pass arguments in order from left to right.
2. The caller routine and the called routine use a calling convention to select the option of passing a variable number of arguments.
3. The caller routine and the called routine use a calling convention to pass arguments by value (values passed) or by reference (addresses passed). Individual Fortran arguments can also be designated with ATTRIBUTES option VALUE or REFERENCE.
4. The caller routine and the called routine use a calling convention to establish naming conventions for procedure names. You can establish any procedure name you want, regardless of its Fortran name, with the ALIAS directive (or ATTRIBUTES option ALIAS). This is useful because C is case-sensitive, while Fortran is not.

See also ATTRIBUTES Properties and Calling Conventions.

ATTRIBUTES Properties and Calling Conventions

The ATTRIBUTES properties (also known as options) C, REFERENCE, VALUE, and VARYING all affect the calling convention of routines. You can specify the:

- C, REFERENCE, and VARYING properties for an entire routine
- 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 property is used, the default changes to passing almost all data except arrays by value. However, in addition to the calling-convention property C, you can specify argument properties VALUE and REFERENCE (to pass arguments by value or by reference), regardless of the calling convention property. Arrays can only be passed by reference.

Different Fortran calling conventions can be specified by declaring the Fortran procedure to have certain attributes. Assume this example:

```
INTERFACE
SUBROUTINE MY_SUB (I)
!DEC$ ATTRIBUTES C, ALIAS:'My_Sub_' :: MY_SUB ! IA-32 systems
INTEGER I
END SUBROUTINE MY_SUB
END INTERFACE
```

This code declares a subroutine named MY_SUB with the C property and the external name My_Sub_ set with the ALIAS property.

For another example, the following declaration assumes the subroutine is called with the C calling convention:

```
SUBROUTINE CALLED_FROM_C (A)
!DEC$ ATTRIBUTES C :: CALLED_FROM_C
INTEGER A
.
.
.
```

The following table summarizes the effect of the most common Fortran calling-convention directives:

Calling Conventions for ATTRIBUTES Properties

Argument	Default	C	C, REFERENCE
Scalar	Reference	Value	Reference
Scalar [value]	Value	Value	Value
Scalar [reference]	Reference	Reference	Reference
String	Reference, either Len:End or Len:Mixed	String(1:1)	Reference, either Len:End or Len:Mixed
String [value]	Error	String(1:1)	String(1:1)
String [reference]	Reference, either No Len or Len:Mixed	Reference, No Len	Reference, No Len
Array	Reference	Reference	Reference
Array [value]	Error	Error	Error
Array [reference]	Reference	Reference	Reference
Derived Type	Reference	Value, size dependent	Reference
Derived Type [value]	Value, size dependent	Value, size dependent	Value, size dependent
Derived Type [reference]	Reference	Reference	Reference
F90 Pointer	Descriptor	Descriptor	Descriptor
F90 Pointer [value]	Error	Error	Error
F90 Pointer [reference]	Descriptor	Descriptor	Descriptor

The procedure name is all lowercase for all the calling conventions.

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 for IA-32 based systems and to the next 8-byte boundary for Intel® EM64T and Itanium-based systems.
Reference	On IA-32 systems, the 4-byte argument address is pushed on the stack. On Intel® EM64T and Itanium®-based systems, 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` 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` 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` 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` 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 IA-32 systems, the 4-byte address of the array descriptor. On Intel® EM64T and Itanium-based systems, the 8-byte address of the array descriptor.
Size dependent	Derived-type arguments specified by value are passed as follows: On IA-32 systems: • 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 of more than 8 bytes provide value semantics by passing a temporary storage address by reference. On Intel® EM64T and Itanium-based systems: • Arguments from 1 to 8 bytes are passed by value. • Arguments of more than 8 bytes provide value semantics by

	passing a temporary storage address by reference.

The following table shows another Fortran ATTRIBUTES property that matches another language calling convention:

Other Language Calling Convention	Matching ATTRIBUTES Property
C/C++ cdecl (default)	C

The ALIAS property can be used with any other Fortran calling-convention property to preserve mixed-case names. You can also use the DECORATE property in combination with the ALIAS property to specify that the external name specified in ALIAS should have the correct prefix and postfix decorations for the calling mechanism in effect.

For information on naming conventions, see Adjusting Naming Conventions in Mixed-Language Programming Overview.

Adjusting Naming Conventions in Mixed-Language Programming

Adjusting Naming Conventions in Mixed-Language Programming Overview

The ATTRIBUTES option C determines naming conventions as well as calling conventions.

Calling conventions specify how arguments are moved and stored; naming conventions specify how symbol names are altered when placed in a `.o` file. Names are an issue for external data symbols shared among parts of the same program as well as among external routines. Symbol names (such as the name of a subroutine) identify a memory location that must be consistent among all calling routines.

Parameter names (names given in a procedure definition to variables that are passed to it) are never affected.

Names are altered because of case sensitivity (in C), lack of case sensitivity (in Fortran), name decoration (in C++), or other issues. If naming conventions are not reconciled, the program cannot successfully link and you will receive an "unresolved external" error.

See these topics:

C/C++ Naming Conventions

Naming Conventions for Fortran, C, and C++

Reconciling the Case of Names

Fortran Module Names and ATTRIBUTES

C/C++ Naming Conventions

C and C++ preserve case sensitivity in their symbol tables while Fortran by default does not, a difference that requires attention. Fortunately, you can use the Fortran directive ATTRIBUTES ALIAS option to resolve discrepancies between names, to preserve mixed-case names, or to override the automatic conversion of names to all lowercase by Fortran.

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 (created when you select C/C++ file from the integrated development environment), 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 ); }
```

Note that, by default, names are lowercase and numbers are passed by reference.

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.

Use of extern "C" has some 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.

Procedure Names in Fortran, C, and C++

The following table summarizes how Fortran, C, and C++ handle procedure names:

Language	Attributes	Name Translated As	Case of Name in .o File
Fortran	*c*DEC$ ATTRIBUTES C	*name*	All lowercase
Fortran	default	*name_*	All lowercase
C	`cdecl` (default)	*name*	Mixed case preserved
C	`__stdcall`	*name*	Mixed case preserved
C++	Default	*name@@decoration*	Mixed case preserved

Reconciling the Case of Names

The following summarizes how to reconcile names between languages:

- All-lowercase names

  If the name of the routine appears as all lowercase in C, then naming conventions are automatically correct. Any case can be used in the Fortran source code, including mixed case, since the name is changed to all lowercase.

- Mixed-case names

  If the name of a routine appears as mixed-case in C and you need to preserve the case, use the Fortran ATTRIBUTES ALIAS option.

  To use the ALIAS option, place the name in quotation marks exactly as it is to appear in the `.o` file.
  The following is an example for referring to the C function `My_Proc`:
  ```
  !DEC$ ATTRIBUTES ALIAS:'My_Proc_' :: My_Proc
  ```

Fortran Module Names and ATTRIBUTES

Fortran module entities (data and procedures) have external names that differ from other external entities. Module names use the convention:

```
modulename_mp_entity_
```

`modulename` is the name of the module and *`entity`* is the name of the module procedure or module data contained within *`modulename`*. `_mp_` is the separator between the module and entity names and is always lowercase.

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 `.o` file:

```
mymod_mp_a_
mymod_mp_b_
```

Compiler options can affect the naming of module data and procedures.

Note

ATTRIBUTES properties do not affect the module name.

The following table shows how each ATTRIBUTES property affects the subroutine in the previous example module.

Effect of ATTRIBUTES Options on Fortran Module Names

ATTRIBUTES Property Given to Routine 'b'	Procedure Name in .o file
None	`mymod_mp_b_`
C	`mymod_mp_b`
ALIAS	Overrides all others, name as given in the alias
VARYING	No effect on name

You can write code to call Fortran modules or access module data from other languages. As with other naming and calling conventions, the module name must match between the two languages. Generally, this means using the C convention in Fortran, and if defining a module in another language, using the ALIAS property to match the name within Fortran. For examples, see Using Modules in Fortran/C Mixed-Language Programming.

Prototyping a Procedure in Fortran

You define a prototype (interface block) in your Fortran source code to tell the Fortran compiler which language conventions you want to use for an external reference. The interface block is introduced by the INTERFACE statement. See "Program Units and Procedures" in the *Language Reference* for a description of the INTERFACE statement.

The general form for the INTERFACE statement is:

```
INTERFACE
routine statement
[routine ATTRIBUTE options]
[argument ATTRIBUTE options]
formal argument declarations
END routine name
END INTERFACE
```

The *routine statement* defines either a FUNCTION or a SUBROUTINE, where the choice depends on whether a value is returned or not, respectively. The optional *routine ATTRIBUTE options* (such as C) determine the calling, naming, and argument-passing conventions for the routine in the prototype statement. The optional *argument ATTRIBUTE options* (such as VALUE and REFERENCE) are properties attached to individual arguments. The *formal argument declarations* are Fortran data type declarations. Note that the same INTERFACE block can specify more than one procedure.

For example, suppose you are calling a C function that has the following prototype:

```
extern void My_Proc (int i);
```

The Fortran call to this function should be declared with the following INTERFACE block:

```
INTERFACE
 SUBROUTINE my_Proc (I)
   !DEC$ ATTRIBUTES C, ALIAS:'My_Proc' :: my_Proc
   INTEGER I
 END SUBROUTINE my_Proc
END INTERFACE
```

Note that, except in the ALIAS string, the case of `My_Proc` in the Fortran program does not matter.

Exchanging and Accessing Data in Mixed-Language Programming

Exchanging and Accessing Data in Mixed-Language Programming Overview

You can use several approaches to sharing data between mixed-language routines, which can be used within the individual languages as well.

Generally, if you have a large number of parameters to work with or you have a large variety of parameter types, you should consider using modules or external data declarations. This is true when using any given language, and to an even greater extent when using mixed languages.

See also Using Modules in Fortran/C Mixed-Language Programming.

See these topics:

Passing Arguments in Mixed-Language Programming

Using Common External Data in Mixed-Language Programming

Passing Arguments in Mixed-Language Programming

You can pass data between Fortran, C, and C++ through calling 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 IA-32 systems, Fortran, C, and C++ use 4-byte addresses. On Intel® EM64T and Itanium® - based systems, 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).

If the ATTRIBUTES C option is used, the default changes to passing all data by value except arrays. If the procedure has the REFERENCE option as well as the C option, all arguments by default are passed by reference.

In Fortran, in addition to establishing argument passing with the calling-convention option of C, you can specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference. In mixed-language programming, it is a good idea to specify the passing technique explicitly rather than relying on defaults.

Examples of passing by reference and value for C follow. All are interfaces to the example Fortran subroutine `TESTPROC` below. The definition of `TESTPROC` declares how each argument is passed. The REFERENCE option is not strictly necessary in this example, but using it makes the argument's passing convention conspicuous.

```
SUBROUTINE TESTPROC( VALPARM, REFPARM )
   !DEC$ ATTRIBUTES VALUE :: VALPARM
   !DEC$ ATTRIBUTES REFERENCE :: REFPARM
   INTEGER VALPARM
   INTEGER REFPARM
END SUBROUTINE
```

In C and C++ all arguments are passed by value, except arrays, which are passed by reference to the address of the first member of the array. Unlike Fortran, C and C++ do not have calling-convention directives to affect the way individual arguments are passed. To pass non-array C data by reference, you must pass a pointer to it. To pass a C array by value, you must declare it as a member of a structure and pass the structure. The following C declaration sets up a call to the example Fortran `testproc` subroutine:

```
extern void testproc_( int ValParm, int *RefParm );
```

The following table summarizes how to pass arguments by reference and value. An array name in C is equated to its starting address because arrays are normally passed

by reference. You can assign the REFERENCE property to a procedure, as well as to individual arguments.

Passing Arguments by Reference and Value

Language	ATTRIBUTE	Argument Type	To Pass by Reference	To Pass by Value
Fortran	Default	Scalars and derived types	Default	VALUE option
	C option	Scalars and derived types	REFERENCE option	Default
	Default	Arrays	Default	Cannot pass by value
	C option	Arrays	Default	Cannot pass by value
C/C++		Non-arrays	Pointer argument_name	Default
		Arrays	Default	Struct {type} array_name

This table does not describe argument passing of strings and Fortran 95/90 pointer arguments in Intel Fortran, which are constructed differently than other arguments. By default, Fortran passes strings by reference along with the string length. String length placement depends on whether the compiler option `-mixed_str_len_arg` (immediately after the address of the beginning of the string) or `-nomixed_str_len_arg` (after all arguments) is set.

Fortran 95/90 array pointers and assumed-shape arrays are passed by passing the address of the array descriptor.

For a discussion of the effect of attributes on passing Fortran 95/90 pointers and strings, see Handling Fortran 90 Pointers and Allocatable Arrays and Handling Character Strings.

Using Common External Data in Mixed-Language Programming

Common external data structures include Fortran common blocks, and C structures and variables that have been declared global or external. All of these data specifications create external variables, which are variables available to routines outside the routine that defines them.

External variables are case-sensitive, so the cases must be matched between different languages, as discussed in the section on naming conventions. Common external data exchange is described in the following sections:

- Using Global Variables
- Using Fortran Common Blocks and C Structures

Using Global Variables in Mixed-Language Programming

A variable can be shared between Fortran and C by declaring it as global (or COMMON) in one language and accessing it as an external variable in the other language. In Fortran programs, variables must be passed as arguments.

In Fortran, a variable can access a global parameter by using the EXTERN option for ATTRIBUTES. For example:

```
   !DEC$ ATTRIBUTES C, EXTERN :: idata
   INTEGER idata (20)
```

EXTERN tells the compiler that the variable is actually defined and declared global in another source file. If Fortran declares a variable external with EXTERN, the language it shares the variable with must declare the variable global.

In C, a variable is declared global with the statement:

```
 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
 COMMON /PI/ PI ! Common Block and variable have the same name
```

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, not the name of a variable within a common block. Thus, you cannot use blank common to make data accessible between C and Fortran. In the preceding example, the common block and the variable have the same name, which helps keep track of the variable between the two languages. Obviously, if a common block contains more than one variable they cannot all have the common block name. (See Using Fortran Common Blocks and C Structures.)

Using Fortran Common Blocks and C Structures

To reference C structures from Fortran common blocks and vice versa, you must take into account the way the 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, so that data is packed.

Note

If the `-align commons` or `-align decommons` compiler option is specified, C structures need to be padded accordingly.

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. Use of common blocks for mixed-language data exchange is discussed in the following sections:

- Accessing Common Blocks and C Structures Directly
- Passing the Address of a Common Block

Accessing Common Blocks and C Structures Directly

You can access Fortran common blocks directly from C by defining an external C structure with the appropriate fields, and making sure that alignment and padding between Fortran and C are compatible. The C and ALIAS ATTRIBUTES options can be used with a common block to allow mixed-case names.

As an example, suppose your Fortran code has a common block named `Really`, as shown:

```
  !DEC$ ATTRIBUTES ALIAS:'Really' :: Really
  REAL(4) x, y, z(6)
  REAL(8) ydbl
  COMMON / Really / x, y, z(6), ydbl
```

You can access this data structure from your C code with the following external data structure:

```
 extern struct {
  float x, y, z[6];
  double ydbl;
} Really;
```

You can also access C structures from Fortran by creating common blocks that correspond to those structures. This is the reverse case from that just described. However, the implementation is the same because after common blocks and structures have been defined and given a common address (name), and assuming the alignment in memory has been dealt with, both languages share the same memory locations for the variables.

Passing the Address of a Common Block

To pass the address of a common block, simply pass the address of the first variable in the block, that is, pass the first variable by reference. The receiving C or C++ routine should expect to receive a structure by reference.

In the following example, the C function `initcb` receives the address of a common block with the first variable named `n`, which it considers to be a pointer to a structure with three fields:

Fortran source code:

```
!
  INTERFACE
     SUBROUTINE initcb (BLOCK)
       !DEC$ ATTRIBUTES C :: initcb
       !DEC$ ATTRIBUTES REFERENCE :: BLOCK
       INTEGER BLOCK
     END SUBROUTINE
  END INTERFACE
!
  INTEGER n
  REAL(8) x, y
  COMMON /CBLOCK/n, x, y
    .  .  .
  CALL initcb( n )
```

C source code:

```
//
 struct block_type
{
  int n;
  double x;
  double y;
};
 //
void initcb( struct block_type *block_hed )
 {
   block_hed->n = 1;
   block_hed->x = 10.0;
   block_hed->y = 20.0;
}
```

Handling Data Types in Mixed-Language Programming

Handling Data Types in Mixed-Language Programming Overview

Even when you have reconciled calling conventions, naming conventions, and methods of data exchange, you must still be concerned with data types, because each language handles them differently.

The following table lists the equivalent data types between Fortran and C:

Equivalent Data Types

Fortran Data Type	C Data Type
INTEGER(1)	`char`
INTEGER(2)	`short`
INTEGER(4)	`int, long`
INTEGER(8)	`_int64`
REAL(4)	`float`
REAL(8)	`double`
REAL(16)	`---`
CHARACTER(1)	`unsigned char`
CHARACTER*(*)	See Handling Character Strings
COMPLEX(4)	`struct complex4 { float real, imag; };`
COMPLEX(8)	`struct complex8 { double real, imag; };`
COMPLEX(16)	`---`
All LOGICAL types	Use integer types for C

See these topics:

Handling Numeric, Complex, and Logical Data Types

Handling Fortran Array Pointers and Allocatable Arrays

Handling Intel Fortran Pointers

Handling Arrays and Fortran Array Descriptors

Handling Character Strings

Handling User-Defined Types

Handling Numeric, Complex, and Logical Data Types

Normally, passing numeric data does not present a problem. If a C program passes an unsigned data type to a Fortran routine, the routine can accept the argument as the equivalent signed data type, but you should be careful that the range of the signed type is not exceeded.

The table of Equivalent Data Types summarizes equivalent numeric data types for Fortran and C/C++.

C and C++ do not directly implement the Fortran types COMPLEX(4), COMPLEX(8), and COMPLEX(16). However, you can write structures that are equivalent. The type COMPLEX(4) has two fields, both of which are 4-byte floating-point numbers; the first contains the real-number component, and the second contains the imaginary-number component. The type COMPLEX is equivalent to the type COMPLEX(4). The types COMPLEX(8) and COMPLEX(16) are similar except that each field contains an 8-byte or 16-byte floating-point number respectively.

 **Note**

Fortran functions of type COMPLEX place a hidden COMPLEX argument at the beginning of the argument list. C functions that implement such a call from Fortran must declare this hidden argument explicitly, and use it to return a value. The C return type should be void.

Following are the C/C++ structure definitions for the Fortran COMPLEX types:

```
struct complex4 {
   float real, imag;
};
struct complex8 {
   double real, imag;
};
```

A Fortran LOGICAL(2) is stored as a 2-byte indicator value (0=false, and the `-fpscomp [no]logicals` compiler option determines how true values are handled). A Fortran LOGICAL(4) is stored as a 4-byte indicator value, and LOGICAL(1) is stored as a single byte. The type LOGICAL is the same as LOGICAL(4), which is equivalent to type `int` in C.

You can use a variable of type LOGICAL in an argument list, module, common block, or global variable in Fortran and type `int` in C for the same argument. Type LOGICAL(4) is recommended instead of the shorter variants for use in common blocks.

The Intel C++ class type has the same layout as the corresponding C `struct` type, unless the class defines virtual functions or has base classes. Classes that lack those features can be passed in the same way as C structures.

Returning Complex Type Data

If a Fortran program expects a procedure to return a COMPLEX (KIND=4, 8, or 16) value, the Fortran compiler adds an additional argument to the beginning of the called procedure argument list. This additional argument is a pointer to the location where the called procedure must store its result.

The example below shows the Fortran code for returning a complex data type procedure called `WBAT` and the corresponding C routine.

Example of Returning Complex Data Types from C to Fortran

Fortran code:

```
COMPLEX BAT, WBAT
REAL X, Y
BAT = WBAT ( X, Y )
```

Corresponding C routine:

```
struct _mycomplex { float real; float imag; };
typedef struct _mycomplex _single_complex;

void wbat_ (_single_complex *location, float *x, float *y){
*location->real = *x;
*location->imag = *y;
return;
}
```

In the above example, the following restrictions and behaviors apply:

- The argument `location` does not appear in the Fortran call; it is added by the compiler.
- The C subroutine must copy the result's real and imaginary parts correctly into `location`.
- The called procedure is type `void`.

If the function returned a DOUBLE COMPLEX value, the type `float` would be replaced by the type `double` in the definition of `location` in `WBAT`.

Handling Fortran Array Pointers and Allocatable Arrays

The following affects how Fortran 95/90 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. This is true for array pointers and all arrays, not just allocatable arrays. 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.

When a Fortran 95/90 array pointer or array is passed to another language, either its descriptor or its base address can be passed.

The following shows how allocatable arrays and Fortran 95/90 array pointers are passed with different attributes in effect:

- If the property of the array pointer or array is not included or is REFERENCE, it is passed by descriptor, regardless of the property of the passing procedure (None; C; or C, REFERENCE).
- If the property of the array pointer or array is VALUE, an error is returned, regardless of the property of the passing procedure.

Note that the VALUE option cannot be used with descriptor-based arrays.

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, 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 Arrays and Fortran Array Descriptors.

Fortran 95/90 pointers that point to scalar data contain the address of the data and are not passed by descriptor.

Handling Integer Pointers

Integer pointers (also known as Cray*-style pointers) are not the same as Fortran 90 pointers, but are instead like C pointers. Integer pointers are 4-byte INTEGER quantities on IA-32 systems, and 8-byte INTEGER quantities on Intel®EM64T and Itanium®-based systems.

Passing Integer Pointers

When passing an integer pointer to a routine written in another language:

- The argument should be declared in the non-Fortran routine as a pointer of the appropriate data type.
- The argument passed from the Fortran routine should be the integer pointer name, not the pointee name.

For example:

```
! Fortran main program.
INTERFACE
  SUBROUTINE Ptr_Sub (p)
 !DEC$ ATTRIBUTES C, ALIAS:'Ptr_Sub' :: Ptr_Sub
     INTEGER (KIND=INT_PTR_KIND()) p
  END SUBROUTINE Ptr_Sub
END INTERFACE
REAL A(10), VAR(10)
POINTER (p, VAR) ! VAR is the pointee
                 ! p is the integer pointer
p = LOC(A)
```

```
CALL Ptr_Sub (p)
WRITE(*,*) 'A(4) = ', A(4)
END
!
//C subprogram
void Ptr_Sub (float *p)
{
 p[3] = 23.5;
}
```

On Intel® EM64T and Itanium-based systems, the declaration for `p` in the INTERFACE block is equivalent to INTEGER(8) p and on IA-32 systems, it is equivalent to INTEGER (4) p.

When the main Fortran program and C function are built and executed, the following output appears:

```
A(4) = 23.50000
```

Receiving Pointers

When receiving a pointer from a routine written in another language:

- The argument should be declared in the non-Fortran routine as a pointer of the appropriate data type and passed as usual.
- The argument received by the Fortran routine should be declared as an integer pointer name, and the POINTER statement should associate it with a pointee of the appropriate data type (matching the data type of the passing routine). When inside the Fortran routine, use the pointee to set and access what the pointer points to. The pointer must be passed by value.

For example:

```
! Fortran subroutine.
SUBROUTINE Iptr_Sub (p)
!DEC$ ATTRIBUTES C, ALIAS:'Iptr_Sub' :: Iptr_Sub
  integer VAR(10)
  POINTER (p, VAR)
  OPEN (8, FILE='STAT.DAT')
  READ (8, *) VAR(4) ! Read from file and store the
                     ! fourth element of VAR
END SUBROUTINE Iptr_Sub
!
//C main program
extern void Iptr_Sub(int *p);
main ( void ) {   int a[10];
Iptr_Sub (&a[0]);
printf("a[3] = %i\n", a[3]);
}
```

When the main C program and Fortran subroutine are built and executed, the following output appears if the `STAT.DAT` file contains 4:

```
   a[3] = 4
```

Handling Arrays and Fortran Array Descriptors

Fortran 95/90 allows arrays to be passed as array elements, as array subsections, or as whole arrays referenced by array name. Within Fortran, array elements are ordered in column-major order, meaning the subscripts of the lowest dimensions vary first.

When using arrays between Fortran and another language, differences in element indexing and ordering must be taken into account. You must reference the array elements individually and keep track of them. Fortran and C vary in the way that array elements are indexed. Array indexing is a source-level consideration and involves no difference in the underlying data.

Fortran and C arrays differ in two ways:

- The value of the lower array bound is different. By default, Fortran indexes the first element of an array as 1. C and C++ index it as 0. Fortran subscripts should therefore be one higher. (Fortran also provides the option of specifying another integer lower bound.)
- In arrays of more than one dimension, Fortran varies the left-most index the fastest, while C varies the right-most index the fastest. These are sometimes called column-major order and row-major order, respectively.

In C, the first four elements of an array declared as X[3][3] are:

```
X[0][0] X[0][1] X[0][2] X[1][0]
```

In Fortran, the first four elements are:

```
X(1,1) X(2,1) X(3,1) X(1,2)
```

The order of indexing extends to any number of dimensions you declare. For example, the C declaration:

```
int arr1[2][10][15][20];
```

is equivalent to the Fortran declaration:

```
INTEGER arr1( 20, 15, 10, 2 )
```

The constants used in a C array declaration represent extents, not upper bounds as they do in other languages. Therefore, the last element in the C array declared as int arr[5][5] is arr[4][4], not arr[5][5].

The following table shows equivalencies for array declarations.

Equivalent Array Declarations for Different Languages

Language	Array Declaration	Array Reference from Fortran
Fortran	DIMENSION x(i, k) -or- *type* x(i, k)	x(i, k)
C/C++	*type* x[k] [i]	x(i -1, k -1)

Intel Fortran Array Descriptor Format

For cases where Fortran 95/90 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), Intel Fortran generates a descriptor for the following types of array arguments:

- Pointers to arrays (array pointers)
- Assumed-shape arrays
- Allocatable arrays

Certain data structure 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 a non-Fortran language (such as C), use an *implicit interface* to allow 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 95/90 pointer with any piece of memory, organized in any way desired (so long as it is "rectangular" in terms of array bounds). You can also pass Fortran 95/90 pointers to other languages, such as C, and have the other language correctly interpret the descriptor to obtain the information it needs.

However, using array descriptors can increase the opportunity for errors and the corresponding code is not portable. In particular, be aware of the following:

- 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. For example, the current Intel Fortran array descriptor format differs from the array descriptor format for Intel Fortran 7.0.
- The array descriptor format may change in the future.

The components of the current Intel Fortran array descriptor on IA-32 systems are as follows:

- The first longword (bytes 0 to 3) contains the base address. The base address plus the offset defines the first memory location (start) of the array.
- The second longword (bytes 4 to 7) contains the size of a single element of the array.
- The third longword (bytes 8 to 11) contains the offset. The offset is added to the base address to define the start of the array.
- The fourth longword (bytes 12 to 15) contains the low-order bit set if the array has been defined (storage allocated). Other bits may also be set by the compiler within this longword, for example, to indicate a contiguous array.
- The fifth longword (bytes 16 to 19) contains the number of dimensions (rank) of the array.
- The sixth longword (bytes 20 to 23) is reserved.
- The remaining longwords (bytes 24 up to 107) contain information about each dimension (up to seven). Each dimension is described by three additional longwords:
    - 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 longwords for a total of nine longwords (6 + 3*1) and ends at byte 35. An array of rank seven is described in a total of 27 longwords (6 + 3*7) and ends at byte 107.

For example, 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:

- The first longword (bytes 0 to 3) contain the base address (assigned at run-time).
- The second longword (bytes 4 to 7) is set to 4 (size of a single element).
- The third longword (bytes 8 to 11) contain the offset (assigned at run-time).
- The fourth longword (bytes 12 to 15) contains 1 (low bit is set).

- The fifth longword (bytes 16 to 19) contains 2 (rank).
- The sixth longword is reserved.
- The seventh, eighth, and ninth longwords (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, the tenth, eleventh, and twelfth longwords (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 Intel® EM64T and Itanium-based systems is identical to that on IA-32 systems, except that all fields are 8-bytes long, instead of 4-bytes.

Handling Character Strings

By default, Intel Fortran passes a hidden length argument for strings. The hidden length argument consists of an unsigned 4-byte integer (IA-32 systems) or unsigned 8-byte integer (Intel® EM64T and Itanium®-based systems), always passed by value, added to the end of the argument list. You can alter the default way strings are passed by using attributes.

The following table shows the effect of various attributes on passed strings:

Effect of ATTRIBUTES Properties on Character Strings Passed as Arguments

Argument	Default	C	C, REFERENCE
String	Passed by reference, along with length	First character converted to INTEGER(4) and passed by value	Passed by reference, along with length
String with VALUE option	Error	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value
String with REFERENCE option	Passed by reference, possibly along with length	Passed by reference, no length	Passed by reference, no length

The table above highlights the following points:

- Character strings without the VALUE or REFERENCE attribute that are passed to routines declared with ATTRIBUTES C are not passed by reference. Instead, only the first character is passed and it is passed by value.
- Character strings with the VALUE option passed to routines declared with ATTRIBUTES C are not passed by reference. Instead, only the value of the first character is passed.
- For string arguments with default ATTRIBUTES, ATTRIBUTES C, or REFERENCE:
    - When `-nomixed_str_len_arg` is set, the length of the string is pushed (by value) on the stack after all of the other arguments.
    - When `-mixed_str_len_arg` is set, the length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
- For string arguments passed by reference with default ATTRIBUTES:
    - When `-nomixed_str_len_arg` is set, the length of the string is not available to the called procedure.
    - When `-mixed_str_len_arg` is set, the length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.

Since all strings in C are pointers, C expects strings to be passed by reference, without a string length. In addition, C strings are null-terminated while Fortran strings are not. There are two basic ways to pass strings between Fortran and C: convert Fortran strings to C strings, or write C routines to accept Fortran strings.

To convert a Fortran string to C, choose a combination of attributes that passes the string by reference without length, and null terminate your strings. For example:

```
INTERFACE
   SUBROUTINE Pass_Str (string)
     !DEC$ ATTRIBUTES C, ALIAS:'Pass_Str' :: Pass_Str
     CHARACTER*(*) string
     !DEC$ ATTRIBUTES REFERENCE :: string
   END SUBROUTINE
END INTERFACE
CHARACTER(40) forstring
DATA forstring /'This is a null-terminated string.'C/
```

The following example shows the extension of using the null-terminator for the string in the Fortran DATA statement (see "C Strings" in the *Intel® Fortran Language Reference*):

```
DATA forstring /'This is a null-terminated string.'C/
```

The C interface is:

```
void Pass_Str (char *string)
```

To get your C routines to accept Fortran strings, C must account for the length argument passed along with the string address. For example:

```
! Fortran code
INTERFACE
SUBROUTINE Pass_Str (string)
CHARACTER*(*) string
END INTERFACE
```

The C routine must expect two arguments:

```
  void pass_str (char *string, unsigned int length_arg )
```

This interface handles the hidden-length argument, but you must still reconcile C strings that are null-terminated and Fortran strings that are not. In addition, if the data assigned to the Fortran string is less than the declared length, the Fortran string will be blank padded.

Rather than trying to handle these string differences in your C routines, the best approach in Fortran/C mixed programming is to adopt C string behavior whenever possible.

Fortran functions that return a character string using the syntax `CHARACTER*(*)` place a hidden string argument and the length of the string at the beginning of the argument list.

C functions that implement such a Fortran function call must declare this hidden string argument explicitly and use it to return a value. The C return type should be void. However, you are more likely to avoid errors by not using character-string return functions. Use subroutines or place the strings into modules or global variables whenever possible.

Returning Character Data Types

If a Fortran program expects a function to return data of type `CHARACTER`, the Fortran compiler adds two additional arguments to the beginning of the called procedure's argument list:

- The first argument is a pointer to the location where the called procedure should store the result.
- The second is the maximum number of characters that must be returned, padded with white spaces if necessary.

The called routine must copy its result through the address specified in the first argument. Example that follows shows the Fortran code for a return character function called `MAKECHARS` and corresponding C routine.

Example of Returning Character Types from C to Fortran

```
Fortran code
CHARACTER*10 CHARS, MAKECHARS
DOUBLE PRECISION X, Y
CHARS = MAKECHARS( X, Y )
```

```
Corresponding C Routine
void makechars ( result, length, x, y );
```

```
char *result;
int length;
double *x, *y;
{
...program text, producing returnvalue...
for (i = 0; i < length; i++ ) {
result[i] = returnvalue[i];
}
}
```

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`.

Handling User-Defined Types

Fortran 95/90 supports *user-defined types* (data structures similar to C structures). User-defined types can be passed in modules and common blocks just as other data types, but the other language must know the type's structure.

For example:

Fortran Code:

```
    TYPE LOTTA_DATA
        SEQUENCE
        REAL A
        INTEGER B
        CHARACTER(30) INFO
        COMPLEX CX
        CHARACTER(80) MOREINFO
     END TYPE LOTTA_DATA
     TYPE (LOTTA_DATA) D1, D2
     COMMON /T_BLOCK/ D1, D2
```

In the Fortran code above, the SEQUENCE statement preserves the storage order of the derived-type definition.

C Code:

```
/* C code accessing D1 and D2 */
extern struct {
   struct {
      float a;
```

```
       int b;
       char info[30];
       struct {
          float real, imag;
          } cx;
       char moreinfo[80];
    } d1, d2;
} t_block;
```

Error Handling

Error Handling Overview

See these topics:

Run-Time Library Default Error Processing

Handling Run-Time Errors

Locating Run-Time Errors

Signal Handling

Overriding the Default Run-Time Library Exception Handler

Obtaining Traceback Information with TRACEBACKQQ

Run-Time Library Default Error Processing

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.

A default action is defined for each error recognized by the Fortran RTL. The default actions described throughout this chapter occur unless overridden by explicit error-processing methods.

The way in which the Fortran RTL 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. The following compiler options are related to handling errors and exceptions:

- The `-check bounds` option generates extra code to catch certain conditions.
- The `-check noformat` and `-check nooutput_conversion` options reduce the severity level of the associated run-time error to allow program continuation.
- The `-fpe`*n* options control the handling and reporting of floating-point arithmetic exceptions at run time.
- The `-warn` *xxxx* and `-nowarn` options control compile-time warning messages, which in some circumstances can help determine the cause of a run-time error.
- The -fexceptions option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.

Run-Time Message Format

When errors occur during execution (run time) of a program, the Fortran RTL issues diagnostic messages. These run-time messages have the following format:

```
forrtl: severity (nnn): message-text
```

where:

- `forrtl` identifies the source as the Intel Fortran RTL.
- *severity* identifies the severity level: `severe`, `error`, `warning`, or `info`.
- *nnn* identifies the message number; also the IOSTAT value for I/O statements.
- *message-text* explains the event that caused the message.

The severity levels are described in order of greatest to least severity:

- A `severe` message 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.
- An `error` message should be corrected. The program might continue execution, but the output from this execution may be incorrect.

- A `warning` message should be investigated. The program continues execution, but output from this execution may be incorrect.
- An `info` message is for informational purposes only. The program continues.

For severe errors, stack trace information is produced by default, unless the environment variable `FOR_DISABLE_STACK_TRACE` is set. If the command-line option `-traceback` is set, the stack trace information contains program counters set to symbolic information. Otherwise, the information contains merely 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.

If `FOR_DISABLE_STACK_TRACE` is set, no stack trace information is produced.

See the following example of stack trace information. The 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

> ifort -O0 -fpe0 -traceback ovf.f90 -o ovf.exe
> ovf.exe

x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34    (1)
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

> setenv FOR_DISABLE_STACK_TRACE true
> ovf.exe

x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34
forrtl: error (72): floating overflow              (2)
Abort
```

The following information corresponds to the numbers at the right of the example:

(1) Stack trace information when the traceback information is present.

(2) No stack trace information, because the `FOR_DISABLE_STACK_TRACE` environment variable is set.

Values Returned to the Shell at Program Termination

An Intel Fortran program can terminate in a number of ways:

- The program runs to normal completion. A value of zero is returned to the shell.
- The program stops with a STOP statement. If an integer value *stop-code* is specified, a status equal to the *stop-code* is returned to the shell; otherwise, a value of zero is returned.
- The program stops with a STOP or a PAUSE statement. A value of zero is returned to the shell.
- The program stops because of a signal that is caught but does not allow the program to continue. A value of 1 is returned to the shell.
- The program stops because of a severe run-time error. The error number for that run-time error is returned to the shell.
- The program stops with a CALL EXIT statement. The value passed to EXIT is returned to the shell.
- The program stops with a CALL ABORT statement. A value of 134 is returned to the shell.

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 `DECFORT_DUMP_FLAG` environment variable to any of the common TRUE values (Y, y, Yes, yEs, True, and so forth) to cause severe errors to create a `core` file. For instance, the following C shell command sets the `DECFORT_DUMP_FLAG` environment variable:

```
setenv decfort_dump_flag 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 don't 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.

Handling Run-Time 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 ERR, EOR, and END branch specifiers in I/O statements.
- 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).
- 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.

These error-processing methods are complementary; you can use any or all of them within the same program to obtain Intel Fortran run-time and Linux* system error codes.

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 nonadvancing 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 an error-handling routine.

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 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 nonadvancing 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='FILNAM', 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

You can use the IOSTAT specifier to continue program execution after an I/O error and to return information about I/O operations. Certain errors are not returned in IOSTAT.

The IOSTAT specifier can supplement or replace the END, EOR, and ERR branch transfers. Execution of an I/O statement containing the IOSTAT specifier suppresses the display of an error message and defines the specified integer variable, array element, or scalar field reference as one of the following:

- A value of -2 if an end-of-record condition occurs with nonadvancing reads.
- A value of -1 if an end-of-file condition occurs.
- A value of 0 for normal completion (not an error condition, end-of-file, or end-of-record condition).
- 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 Run-Time Error Messages.

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.

You can include `/opt/intel/fc/9.1xxx/include/for_iosdef.for` in your program to obtain symbolic definitions for the values of IOSTAT.

The following example uses the IOSTAT specifier and the `for_iosdef.for` file to handle an OPEN statement error (in the FILE specifier):

```
CHARACTER(LEN=40) :: FILNM
INCLUDE 'for_iosdef.for'
DO I=1,4
 FILNM = ''
 WRITE (6,*)  'Type file name '
 READ (5,*) FILNM
 OPEN (UNIT=1, FILE=FILNM, STATUS='OLD', IOSTAT=IERR, ERR=100)
 WRITE (6,*) 'Opening file: ', FILNM
!       (process the input file)
 CLOSE (UNIT=1)
 STOP
100  IF (IERR .EQ. FOR$IOS_FILNOTFOU) THEN
   WRITE (6,*) 'File: ', FILNM, ' does not exist '
 ELSE IF (IERR .EQ. FOR$IOS_FILNAMSPE) THEN
    WRITE (6,*) 'File: ', FILNM, ' was bad, enter new file name'
```

```
 ELSE          PRINT *, 'Unrecoverable error, code =', IERR
   STOP
 END IF
END DO
WRITE (6,*) 'File not found. Type ls to find file and run again'
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*).

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.

To locate the cause of errors, use the various compiler options to isolate programming errors at compile time and run-time or use the debugger to locate the cause of exceptions:

- The `-check keyword` options generate extra code to catch certain conditions at run time. For example, you can specify the `-check bounds` option to generate code to perform 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 `-check uninit` option checks for uninitialized variables and, if a variable is read before written, will call a run-time error routine. The `-check noformat` and `-check nooutput_conversion` options reduce the severity level of the associated run-time error to allow program continuation. For more information on these options, refer to the Compiler Options reference.
- The `-ftrapuv` option is useful in detecting uninitialized variables. It sets 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, which are not properly initialized by the application, are likely to cause run-time errors that can help you detect coding errors.
- The `-traceback` option allows program counter to source file line correlation, which simplifies the task of locating the cause of severe run-time errors. Without `-traceback`, 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.
- The `-fpe` option controls the handling of floating-point arithmetic exceptions (IEEE arithmetic) at run time. For example, if you specified `-fpe3`, exceptions related to exceptional IEEE values are not reported and your application may generate exceptional IEEE values, which later in your application may cause an exception or unexpected values. If you recompile the application at `-fpe0`, any exceptional IEEE values generated will cause the program to terminate and report an error message earlier.

Signal Handling

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, 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
```

Obtaining Traceback Information with TRACEBACKQQ

You can obtain traceback information in your application by calling the TRACEBACKQQ routine.

TRACEBACKQQ allows an application to initiate a stack trace. You can use this routine to report application detected errors, use it for debugging, and so on. It uses the standard stack trace support in the Intel Fortran run-time system to produce the same output that the run-time system produces for unhandled errors and exceptions (severe error message). The TRACEBACKQQ subroutine generates a stack trace showing the program call stack as it was leading up to the point of the call to TRACEBACKQQ.

The error message string normally included from the run-time support is replaced with the user-supplied message text or omitted if no user 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 support.

In the most simple case, a user 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.

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.

Creating and Using Libraries

Creating and Using Libraries Overview

See these topics:

Creating Libraries

Libraries Provided by Intel Fortran

Portability Library Overview

Math Libraries Overview

Creating Libraries

Libraries are simply an indexed collection of object files that are included as needed in a linked program. Combining object files into a library makes it easy to distribute your code without disclosing the source. It also reduces the number of command-line entries needed to compile your project.

Static Libraries

Executables generated using static libraries are no different than executables generated from individual source or object files. Static libraries are not required at runtime, so you do not need to include them when you distribute your executable. At compile time, linking to a static library is generally faster than linking to individual source files.

To build a static library:

1. Use -c option and, optionally, the -fpic option to generate object files from the source files:
   ifort -fpic -c my_source1.f90 my_source2.f90 my_source3.f90
2. Use the GNU ar tool to create the library file from the object files:
   ar 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
```

Shared Libraries

Shared libraries, also referred to as dynamic libraries or Dynamic Shared Objects (DSO), 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.

On Linux* IA-32 systems and Intel® EM64T systems, you must specify option fpic for the compilation of each object file you want to include in the shared library.

To build a shared library:

1. Use the `-fpic` and `-c` options to generate object files from the source files:
   ifort -fpic -c my_source1.f90 my_source2.f90 my_source3.f90
2. Use the `-shared` option (for Linux* systems) or the `-dynamiclib` option (for Mac OS* systems) to create the library file from the object files:
   Linux: **ifort -shared my_lib.so my_source1.o my_source2.o my_source3.o**
   Mac OS: **ifort -dynamiclib my_lib.dylib my_source1.o my_source2.o my_source3.o**
3. Compile and link your project with your new library:
   Linux: **ifort main.f90 my_lib.so**
   Mac OS: **ifort main.f90 my_lib.dylib**

See also Creating Shared Libraries.

Libraries Provided by Intel® Fortran

Intel® Fortran provides different types of libraries, such as static or shared, single-threaded or multi-threaded, for certain libraries.

The table below shows the libraries provided by the compiler:

File	Description

`crtxi.o` `crtxn.o`	C initialization support; Linux* only.
`for_main.o`	main routine for Fortran programs
`icrt.internal.map` `icrt.link`	C link support; Linux* only

`libcprts.a`
`libcprts.so`
`libcprts.so.5` (IA-32 and Intel® EM64T compilers)
`libcprts.so.6` (Itanium® compiler)
	C++ standard language library; Linux* only

`libcxa.a`
`libcxa.so`
`libcxa.so.5` (IA-32 and Intel® EM64T compilers)
`libcxa.so.6` (Itanium® compiler)
	C++ language library indicating I/O data location; Linux* only

`libcxaguard.a` `libcxaguard.so` (`.dylib` for Mac OS*) `libcxaguard.so.5` (IA-32 and Intel® EM64T Linux compilers) `libcxaguard.so.6` (Itanium® compiler)	Used for interoperability with the `-cxxlib-gcc` option.
`libguide.a` `libguide.so` (`.dylib` for Mac OS*)	OpenMP* static library for the parallelizer tool
`libguide_stats.a` `libguide_stats.so` (`.dylib` for Mac OS*)	Support for parallelizer tool with performance and profile information
`libifcore.a` `libifcore.so` (`.dylib` for Mac OS*) `libifcore.so.5` (IA-32 and Intel® EM64T Linux compilers) `libifcore.so.6` (Itanium® compiler)	Intel-specific Fortran run-time library
`libifcore_pic.a` `libifcoremt_pic.a`	Intel-specific Fortran static libraries; Linux* only. These support position independent code and allow creation of shared libraries linked to Intel-specific Fortran ***static*** run-time libraries, instead of shared run-time libraries.
`libifcoremt.a` `libifcoremt.so` (`.dylib` for Mac OS*) `libifcoremt.so.5` (IA-32 and Intel® EM64T Linux compilers) `libifcoremt.so.6` (Itanium® compiler)	Multithreaded Intel-specific Fortran run-time library

`libifport.a` `libifport.so` (`.dylib` for Mac OS*) `libifport.so.5` (IA-32 and Intel® EM64T Linux compilers) `libifport.so.6` (Itanium® compiler) `libifportmt.dylib` (Mac OS* only)	Portability and POSIX support
`libimf.a` `libimf.so` (`.dylib` for Mac OS*)	Math library

`libirc.a`
`libirc_s.a`
`libirc.dylib (Mac OS*)`
	Intel-specific library (optimizations)

`libompstub.a`	Library that resolves references to OMP subroutines when OMP is not in use
`libsvml.a` libsvml.dylib (Mac OS*)	Short vector math library (Note: For Intel® EM64T-based systems, this library is linked against by default. For IA-32 systems, linking against this library is dependent on optimization switches.)
`libunwind.a` `libunwind.so` (`.dylib` for Mac OS*) `libunwind.so.5` (IA-32 and Intel® EM64T Linux compilers) `libunwind.so.6` (Itanium® compiler)	Unwind support

Portability Library

Portability Library Overview

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.a`. 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.

See these topics:

Using the Portability Library libifport.a

Using the Portability Library libifport.a

You can use the portability library `libifport.a` in one of two ways:

- Add the statement USE IFPORT to your program.
- Call portability routines using the correct parameters and return value.

`libifport.a` is passed to the linker by default during linking. To prevent `libifport.a` from being passed to the linker, specify the `-fpscomp nolibs` option.

Using the IFPORT mod file provides interface blocks and parameter definitions for the routines, as well as compiler verification of calls.

Some routines in this library can be called with different sets of arguments, and sometimes even as a function instead of a subroutine. In these cases, the arguments and calling mechanism determine the meaning of the routine. The IFPORT mod file contains generic interface blocks that give procedure definitions for these routines.

Fortran 95/90 contains intrinsic procedures for many of the functions provided by the portability routines. The portability routines are extensions to the Fortran 95 standard. When writing new code, use Fortran 95/90 intrinsic procedures whenever possible (for portability and performance reasons).

Portability Routines

This section describes some of the portability routines and how to use them.

For a complete list of the routines, see the table of Portability Routines in the Overview chapter of the *Intel Fortran Libraries Reference*.

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* 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.

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 95/90. 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 95/90 includes many bit operation routines. These routines are listed in Chapter 9 of the *Language Reference*, in table 9-2, under Category Bit.

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 95/90. 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 parameter 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 95/90 includes date and time intrinsic subroutines.

Error Handling Routines

Error handling routines detect and report errors.

IERRNO error codes are analogous to `errno` on UNIX 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 90 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 these routines.

Standard Fortran 90 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.

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. See the table of Portability Routines in the Overview chapter of the *Intel Fortran Libraries Reference*.

Math Libraries

`libimf.a` is the math library provided by Intel and `libm.a` is the math library provided with gcc*.

Both of these libraries are linked in by default on IA-32, Intel® EM64T, and Itanium® -based compilers. Both libraries are linked in because there are math functions supported by the GNU math library that are not 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.

`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.

libimf.a on the IA-32 Compiler

For the IA-32 compiler, `libimf.a` contains both generic math routines and versions of the math routines optimized for special use with the Intel Pentium® 4 and Intel® Xeon® processors.

libimf.a on the Itanium-Based Compiler

For the Itanium-based compiler, `libimf.a` is optimized for use with the Itanium architecture. The compiler provides inlined versions of math library primitives and schedules the generated code with surrounding instructions. This can improve the performance of typical floating-point applications.

libimf.a on the Intel® EM64T-Based Compiler

For the Intel EM64T-based compiler, libimf.a contains both generic math routines and versions of the math routines optimized for special use with the Intel® EM64T processor.

Reference Information

Compile-Time Environment Variables

The compile-time environment variables are:

- `FPATH`
  The path for include and module files.
- GCCROOT
  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
  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.
- GXX_ROOT
  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.
- `IFORTCFG`
  The configuration file to use instead of the default configuration file.
- INTEL_LICENSE_FILE
  The location of the product license file.
- `LD_LIBRARY_PATH` (Linux*)
  The path for shared (.so) library files.
- DYLD_LIBRARY_PATH (Mac OS*)

  The path for dynamic libraries.

- `PATH`

  The path for compiler executable files.

- `TMP, TMPDIR, TEMP`
  Specifies the directory in which to store temporary files. See Temporary Files Created by the Compiler or Linker.

Run-Time Environment Variables

The Intel® Fortran run-time system recognizes several environment variables. These variables can be used to customize run-time diagnostic error reporting, for example.

The run-time environment variables are:

- `decfort_dump_flag`
  If this variable is set to Y or y, a core dump will be taken when any severe Intel Fortran run-time error occurs. If the program is executing under a debugger, a signal will be raised, which will allow you to trace back to where the error was detected.
- `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.

- `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 `stdin` (standard input). If `FOR_ACCEPT` is defined (as a file name optionally containing a path), the specified file would be read.
- `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.
  The setting of `FOR_DIAGNOSTIC_LOG_FILE` is independent of `FOR_DISABLE_DIAGNOSTIC_DISPLAY`, so you can disable the screen display of information but still capture the error information in a file. The text string you assign for the file name is used literally, so you must specify the full name. If the file open fails, no error is reported and the run-time system continues diagnostic processing.
- `FOR_DISABLE_DIAGNOSTIC_DISPLAY`
  Disables the display of all error information.
  This variable is helpful if you just want to test the error status of your program and do not want the Fortran run-time system to display any information about an abnormal program termination.
- `FOR_DISABLE_STACK_TRACE`
  This variable disables the call stack trace information that follows the displayed severe error message text.
  The Fortran run-time error message is displayed whether or not `FOR_DISABLE_STACK_TRACE` is set to true.
- `FOR_IGNORE_EXCEPTIONS`
  This variable disables the default run-time exception handling, for example, to allow 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.
- `FOR_NOERROR_DIALOGS`
  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.
- `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 `stdout` (standard output). If `FOR_PRINT` is defined (as a filename 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 `stdin`

(standard input). If FOR_READ is defined (as a filename 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 stdout (standard output). If FOR_TYPE is defined (as a filename optionally containing a path), the specified file would be written to.
- FORT_BUFFERED
  Lets you request that buffered I/O should be used at run time for output of all Fortran I/O units, except those with output to the terminal. This provides a run-time mechanism to support the -assume buffered_io compiler option.
- FORT_CONVERT*n*
  Lets you specify the data format for an unformatted file associated with a particular unit number (*n*). See Methods of Specifying the Data Format: Overview and Environment Variable FORT_CONVERTn Method.
- FORT_CONVERT.*ext* and FORT_CONVERT_*ext*
  Lets you specify the data format for unformatted files with a particular file extension suffix (ext). See Methods of Specifying the Data Format: Overview and Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method.
- FORT*n*
  Lets you specify 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 filesfromcmd was not specified. Preconnected files attached to units 0, 5, and 6 are by default associated with system standard I/O files.
- NLSPATH
  The path for the Intel Fortran run-time error message catalog.
- TBK_ENABLE_VERBOSE_STACK_TRACE
  This variable 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, followed by the PC, routine name, line number, and source file are given.
  The verbose output, if selected, will provide (in addition to the information in brief output) the exception context record 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 (but 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.
  The variable FOR_ENABLE_VERBOSE_STACK_TRACE is also recognized for compatibility with Compaq* Fortran.
- TBK_FULL_SRC_FILE_SPEC
  This variable displays complete file name information for traceback output, including the path.
  By default, the traceback output displays only the file name and extension in the source file field. You must set this variable to display more.

The variable FOR_FULL_SRC_FILE_SPEC is also recognized for compatibility with Compaq* Fortran.

- TMP, TMPDIR, and TEMP
  Specifies an alternate working directory where temporary files are created. See Temporary Files Created by the Compiler or Linker.

Key IA-32 and Intel® EM64T Compiler Files Summary

The following table shows files that are installed for use by the IA-32 and Intel® EM64T compilers in /opt/intel/fc/9.1.xxx/bin:

File	Description
codecov	Executable for the Code-coverage tool
fortcom	Executable used by the compiler
fpp	Fortran Preprocessor
ifc	For compatibility with previous releases
ifcbin	For compatibility with previous releases
ifc.cfg	For compatibility with previous releases
ifort	Intel Fortran Compiler Version 8 and later
ifortbin	Executable used by the compiler
ifort.cfg	Configuration file
ifortvars.csh	Setup file for C shell
ifortvars.sh	Setup file for bash shell
map_opts	Utility used for option translation
profmerge	Utility used for Profile Guided Optimizations
profrun	Used for Profile Guided Optimizations
profrun.bin	Used for Profile Guided Optimizations
proforder	Utility used for Profile Guided Optimizations
tselect	Test-prioritization tool
uninstall.sh	Uninstall utility
xiar	Tool used for Interprocedural Optimizations
xilibtool	Tool used for Interprocedural Optimizations (Mac OS*)
xild	Tool used for Interprocedural Optimizations

For a list of the files installed in /lib, see Libraries Provided by Intel Fortran.

Key Itanium®-Based Compiler Files Summary

The following table shows files that are installed for use by the Itanium® -based compiler in /opt/intel/fc/9.1.xxx/bin:

File	Description
codecov	Executable for the Code-coverage tool
efc	For compatibility with previous releases

efc.cfg	For compatibility with previous releases
efcbin	For compatibility with previous releases
fortcom	Executable used by the compiler
fpp	Fortran Preprocessor
ias	Intel assembler
ifort	Intel Fortran Compiler
ifort.cfg	Configuration file
ifortbin	Executable used by the compiler
ifortvars.csh	Setup file for C shell
ifortvars.sh	Setup file for bash shell
map_opts	Utility used for option translation
profdcg	Utility used for Profile Guided Optimizations
profmerge	Utility used for Profile Guided Optimizations
profrun	Used for Profile Guided Optimizations
profrun.bin	Used for Profile Guided Optimizations
proforder	Utility used for Profile Guided Optimizations
tselect	Test-prioritization tool
uninstall.sh	Uninstall utility
xiar	Tool used for Interprocedural Optimizations
xild	Tool used for Interprocedural Optimizations

For a list of files installed in /lib, see Libraries Provided by Intel Fortran.

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	7
Array construction nesting	20
Array elements per dimension	9,223,372,036,854,775,807 = 2**31-1 on IA-32 systems; 2**63-1 on Intel® EM64T and Itanium-based systems; plus limited by current memory configuration
Constants: character and Hollerith	7198 characters

Constants: characters read in list-directed I/O	2048 characters
Continuation lines - free form	Depends on line complexity and the number of lexical tokens allowed.
Continuation lines - fixed form	Depends on line complexity and the number of lexical tokens allowed.
Data and I/O implied DO nesting	7
DO and block IF statement nesting (combined)	256
DO loop index variable	9,223,372,036,854,775,807 = 2**63-1
Format group nesting	8
Format statement length	2048 characters
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	7
Nesting of input/output implied DOs	7
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 digits in a numeric constant	Limited only by memory constraints
Parentheses nesting in expressions	Limited only by memory constraints
Structure nesting	30
Symbolic name length	63 characters

See the product Release Notes for more information on memory limits for large data objects.

Hexadecimal-Binary-Octal-Decimal Conversions

The following table lists hexadecimal, binary, octal, and decimal conversion:

Hex Number	Binary Number	Octal Number	Decimal Number
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3

4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
A	1010	12	10
B	1011	13	11
C	1100	14	12
D	1101	15	13
E	1110	16	14
F	1111	17	15

Run-Time Error Messages

The table below lists the errors processed by the Intel Fortran 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:

```
/opt/intel/fc/9.1.xxx/include/for_iosdef.f
```

As described in the table, the severity of the message determines which of the following occurs: program execution continues with `info` and `warning`, the results might be incorrect with `error`, and program execution stops (unless a recovery method is specified) with `severe`. 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.

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
None [1]	`info: Fortran error message number is` *nnn* The Intel Fortran message catalog file was not found on this system. This error has no condition symbol.

None [1]	`warning: Could not open message catalog: ifcore_msg.cat` The Intel Fortran message catalog file was not found on this system. This error has no condition symbol.
None [1]	`info: Check environment variable NLSPATH and protection of pathname/ifcore_msg.cat` The Intel Fortran message catalog file was not found. This error has no condition symbol.
None [1]	`Insufficient memory to open Fortran RTL catalog: message 41` The Intel Fortran message catalog file could not be opened because of insufficient virtual memory. To overcome this problem, increase the per-process data limit by using the `limit` (C shell) or `ulimit` (Bourne* and Korn and bash shells) commands before running the program again. For more information, see error 41. This error has no condition symbol.
1 [1]	`severe (1): Not a Fortran-specific error` FOR$IOS_NOTFORSPE. An error in the user program or in the RTL was not a Intel Fortran-specific error and was not reportable through any other Intel Fortran run-time messages. If you call ERRSNS, an error of this kind returns a value of 1 (for more information on the ERRSNS subroutine, see the *Intel Fortran Language Reference Manual*).
8	`severe (8): Internal consistency check failure` FOR$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$IOS_PERACCFIL. Check the mode (protection) of the specified file. Make sure the correct file was being accessed. Change the protection, specified file, or process used before rerunning program.
10	`severe (10): Cannot overwrite existing file` FOR$IOS_CAOVEEXI. 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	`info (11) [1]: 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. • An attempt was made to specify a substring of a noncharacter 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: • A 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$IOS_TOOMANREC. An attempt was made to do one of the following: • Read or write more than one record with an ENCODE or DECODE statement. • Write more records than existed.
28	`severe (28): CLOSE error` FOR$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$IOS_MIXFILACC. An attempt was made to use any of the following combinations: • Formatted and unformatted operations on the same unit • An invalid combination of access modes on a unit, such as direct and sequential • A Intel Fortran RTL I/O statement on a logical unit that was opened by a program coded in another language
32	`severe (32): Invalid logical unit number` FOR$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$IOS_ENDFILERR. One of the following conditions occurred: • The file was not a sequential organization file with variable-length records. • The file was not opened for sequential or append access. • An unformatted file did not contain segmented records. • The Intel Fortran RTL I/O system detected an error during execution of an ENDFILE statement.
34	`severe (34): Unit already open` FOR$IOS_UNIALROPE. A DEFINE FILE statement specified a logical unit that was already opened.
35	`severe (35): Segmented record format error` FOR$IOS_SEGRECFOR. An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE= ' FIXED ' or ' VARIABLE ' in effect, or was created by a program written in a language other than Fortran.
36	`severe (36): Attempt to access non-existent record` FOR$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 in a relative file.
37	`severe (37): Inconsistent record length` FOR$IOS_INCRECLEN. An attempt was made to open a direct access file without specifying a record length.
38	`severe (38): Error during write` FOR$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$IOS_INSVIRMEM. The Intel Fortran RTL attempted to exceed its available virtual memory while dynamically allocating space. To overcome this problem, increase the per-process data limit by using the `limit` (C shell) or `ulimit` (Bourne* and Korn and bash shell) commands before you run this program again. To determine whether the maximum per-process data size is already allocated, check the value of the *maxdsiz* parameter in the sysconfigtab or system configuration file. If necessary, increase its value. Changes to do not take effect until the system has been rebooted (you do not need to rebuild the kernel if you modify sysconfigtab). Before you try to run this program again, wait until the new system resources take effect.
42	`severe (42): No such device` FOR$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= ' APPEND ' with STATUS= ' REPLACE ' • ACCESS= ' DIRECT ' or ' KEYED ' with POSITION= ' APPEND ' , ' ASIS ' , or ' REWIND '
47	`severe (47): Write to READONLY file` 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$IOS_FMTSYN. Check the statement containing *xx*, a character substring from the format string, for a format syntax error. For information about FORMAT statements, see the *Intel Fortran Language Reference Manual*.
59	`severe (59): List-directed I/O syntax error` FOR$IOS_LISIO_SYN [2]. 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.
60	`severe (60): Infinite format loop` FOR$IOS_INFFORLOO. The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.
61	`severe or info` [3] `(61): Format/variable-type mismatch` FOR$IOS_FORVARMIS [2]. 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).
62	`severe (62): Syntax error in format` FOR$IOS_SYNERRFOR. A syntax error was encountered while the RTL was processing a format stored in an array or character variable.
63	`error or info` [3] `(63): Output conversion error` FOR$IOS_OUTCONERR [2]. 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.
64	`severe (64): Input conversion error` FOR$IOS_INPCONERR [2]. During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable was set to zero.
65	`error (65): Floating invalid` FOR$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, when requesting 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$IOS_OUTSTAOVE. An output statement attempted to transfer more data than would fit in the maximum record size.
67	`severe (67): Input statement requires too much data` FOR$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 specifier value of `' NO '` .
68	`severe (68): Variable format expression value error` FOR$IOS_VFEVALERR [2]. The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of 1 was assumed, except for a P edit descriptor, for which a value of zero was assumed.
69 [1]	`error (69): Process interrupted (SIGINT)` FOR$IOS_SIGINT. The process received the signal SIGINT. Determine source of this interrupt signal (described in `signal(3)`).
70 [1]	`severe (70): Integer overflow` FOR$IOS_INTOVF. During an arithmetic operation, an integer value exceeded byte, word, or longword range. The result of the operation was the correct low-order part. Consider specifying a larger integer data size (modify source program or, for an INTEGER declaration, possibly use the `f90` option `-integer_size` *nn*).
71 [1]	`severe (71): Integer divide by zero` FOR$IOS_INTDIV. During an integer arithmetic operation, an attempt was made to divide by zero. The result of the operation was set to the dividend, which is equivalent to division by 1.
72 [1]	`error (72): Floating overflow` FOR$IOS_FLTOVF. During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type.
73 [1]	`error (73): Floating divide by zero` FOR$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.

75 [1]	`error (75): Floating point exception` FOR$IOS_SIGFPE. A floating-point exception occurred. Core dump file created. Possible causes include: • Division by zero • Overflow • Invalid operation, such as subtraction of infinite values, multiplication of zero by infinity (without signs), division of zero by zero or infinity by infinity • Conversion of floating-point to fixed-point format when an overflow prevents conversion
76 [1]	`error (76): IOT trap signal` FOR$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$IOS_SUBRNG. An array reference was detected outside the declared array bounds.
78 [1]	`error (78): Process killed (SIGTERM)` FOR$IOS_SIGTERM. The process received the signal SIGTERM. Determine source of this software termination signal (described in `signal(3)`).
79 [1]	`error (79): Process quit (SIGQUIT)` FOR$IOS_SIGQUIT. The process received the signal SIGQUIT. Core dump file created. Determine source of this quit signal (described in `signal(3)`).
95 [1]	`info (95): Floating-point conversion failed` FOR$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: • Whether the correct file was specified. • Whether the record layout matches the format Intel Fortran is expecting. • The ranges for the data being used.

	• Whether the correct nonnative floating-point data format was specified.
96	`info (96): F_UFMTENDIAN environment variable was ignored:erroneous syntax` FOR$IOS_UFMTENDIAN. Syntax for specifying whether little endian or big endian conversion is performed for a given Fortran unit was incorrect. Even though the program will run, the results might not be correct if you do not change the value of F_UFMTENDIAN. For correct syntax, see Environment Variable F_UFMTENDIAN Method.
108	`severe (108): Cannot stat file` FOR$IOS_CANSTAFIL. Attempted stat operation on the indicated file failed. Make sure correct file and unit were specified.
120	`severe (120): Operation requires seek ability` FOR$IOS_OPEREQSEE. Attempted an operation on a file that requires the ability to perform seek operations on that file. Make sure the correct unit, directory path, and file were specified.
134	No associated message Program was terminated internally through abort( ).
138 [1]	`severe (138): Array index out of bounds (SIGILL)` FOR$IOS_BRK_RANGE. Break exception generated a SIGTRAP signal (described in `signal(3)`). Core dump file created. The cause is an array subscript that is outside the dimensioned boundaries of that array. Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the core dump file to determine the source code in error.
139 [1]	`severe (139): Array index out of bounds for index` *nn* `(SIGILL)` FOR$IOS_BRK_RANGE2. Break exception generated a SIGTRAP signal (described in `signal(3)`). Core dump file created. The cause is an array subscript that is outside the dimensioned boundaries of the array index *n*. Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the core dump file to determine the source code in error.
140 [1]	`severe (140): Floating inexact` FOR$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$IOS_ROPRAND. The Intel Fortran RTL encountered a reserved operand. Please report the problem to Intel..
145 [1]	`severe (145): Assertion error` FOR$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$IOS_STRLENERR. During a string operation, an integer value appears in a context where the value of the integer is outside the permissible string length range. Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the `core` file to determine the source code causing the error.
149 [1]	`severe (149): Substring error` FOR$IOS_SUBSTRERR. An array subscript is outside the dimensioned boundaries of an array. Either recompile with the `-check bounds` option (perhaps with the `decfort_dump_flag` environment variable set) or examine the `core` file to determine the source code causing the error.
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.
152 [1]	`severe (152): Unresolved contention for Intel Fortran RTL global resource` FOR$IOS_RESACQFAI. Failed to acquire a 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$IOS_INVDEALLOC. A Fortran-90 allocatable array or pointer must already be allocated when you attempt to deallocate it. You must allocate the array or pointer before it can again be deallocated.
173 [1]	`severe (173): A pointer passed to DEALLOCATE points to an array that cannot be deallocated` FOR$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 can be validly passed to DEALLOCATE.
174 [1]	`severe (174): SIGSEGV, `*`message-text`* FOR$IOS_SIGSEGV. One of two possible messages occurs for this error number: • `severe (174): SIGSEGV, segmentation fault occurred` This message indicates that the program attempted an invalid memory reference. Check the program for possible errors. • `severe (174): SIGSEGV, possible program stack overflow occurred` The following explanatory text also appears: `Program requirements exceed current stacksize resource limit.`
175 [1]	`severe (175): DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8` FOR$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 8 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 10 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$IOS_SHORTZONEARG. The number of characters associated with the ZONE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 5 characters in length. Verify that the DATE and TIME arguments also meet their minimum lengths.
178 [1]	`severe(178): Divide by zero` FOR$IOS_DIV. A floating-point or integer divide-by-zero exception occurred.
179 [1,4]	`severe(179): Cannot allocate array---overflow on array size calculation` FOR$IOS_ARRSIZEOVF. An attempt to dynamically allocate storage for an array failed because the required storage size exceeds addressable memory.
256	`severe (256): Unformatted I/O to unit open for formatted transfers` FOR$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$IOS_FMTIO_UNF. Attempted formatted I/O (such as list-directed or namelist I/O) to a unit where the OPEN statement indicated the file was unformatted (FORM specifier). Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains formatted data, specify FORM= ' FORMATTED ' in the OPEN statement. Otherwise, if appropriate, use unformatted I/O.
264	`severe (264): operation requires file to be on disk or tape` FOR$IOS_OPERREQDIS. Attempted to use a BACKSPACE statement on such devices as a terminal or pipe.
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 `abort` to terminate the program.
268 [1]	`severe (268): End of record during read` FOR$IOS_ENDRECDUR. An end-of-record condition was encountered during execution of a nonadvancing I/O READ statement that did not specify the EOR branch specifier.
297 [1]	`info (297):` *nn* `floating invalid traps` FOR$IOS_FLOINVEXC. The total number of floating-point invalid data traps encountered during program execution was *nn*. This message appears at program completion.
298 [1]	`info (298):` *nn*`floating overflow traps` FOR$IOS_FLOOVFEXC. The total number of floating-point overflow traps encountered during program execution was *nn*. This message appears at program completion.
299 [1]	`info (299):` *nn*`floating divide-by-zero traps` FOR$IOS_FLODIV0EXC. The total number of floating-point divide-by-zero traps encountered during program execution was *nn*. This 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 message appears at program completion.

Footnotes:

1 Identifies errors not returned by IOSTAT.

2 The ERR transfer is taken after completion of the I/O statement for error numbers 59, 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.

3 For errors 61 and 63, the severity depends on the `-check` options used during compilation.

4 Identifies errors that can be returned by STAT in an ALLOCATE statement.

Index

D

E

S