Intel® Fortran Building Applications

Document number: 304970-005US

Disclaimer and Legal Information

Table Of Contents

Building Applications

Overview: Building Applications

Welcome to the Intel® Fortran Compiler.

This Building Applications document explains how to use the Intel® Compiler to build applications on Linux*, Windows*, and Mac OS* X operating systems. Intel® Fortran provides you with a variety of alternatives for building applications. Depending on your needs, you can build your source code into several types of programs and libraries using the command line. Additionally, on Windows systems, you can use the Microsoft Visual Studio* integrated development environment (IDE) and on Mac OS X operating systems, you can use the Xcode* IDE to build your application.

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 general, the compiler features and options supported on Linux OS using IA-32 architecture or Intel® 64 architecture are also supported on Intel®-based systems running Mac OS X. For more detailed information about support for specific operating systems, refer to the appropriate option in the Compiler Options reference or the Release Notes.

Getting Started

Choosing Your Development Environment

Depending on your operating system, you can build programs from the command line and/or from an IDE such as Microsoft Visual Studio* (Windows* OS) or Xcode* (Mac OS* X).

An IDE offers a number of ways to simplify the task of compiling and linking programs. It provides a default text editor. You can also use your favorite text editor outside the integrated development environment.

Because software development is an iterative process, it is important to be able to move quickly and efficiently to various locations in your source code. If you use an IDE to build your programs, you can display both the description of the error message and the relevant source code directly from the displayed error messages.

When you build programs from the command line, you may have more control of the build tools. If you choose to, you can customize how your program is built by your selection of compiler and linker options. Compiler options are described in the Compiler Options Reference.

See Also

Invoking the Intel® Fortran Compiler
Using the Compiler and Linker from the Command Line

Invoking the Intel® Fortran Compiler

The command to invoke the compiler is `ifort`.

Requirements Before Using the Command Line

On Linux* and Mac OS* X operating systems, you need to set some environment variables to specify locations for the various components prior to using the command line. The Intel Fortran Compiler installation includes a shell script that you can run to set environment variables. For more information, see Using the ifortvars File to Specify Location of Components.

On Windows* operating systems, you typically do not need to set any environment variables prior to using the command line. Each of the Intel® Fortran Compiler variations has its own

Intel Compiler command-line window, available from the Intel Fortran program folder. This window has the appropriate environment variables already set for the command-line environment.

Using the ifort Command from the Command Line

Use the `ifort` command either on a command line or in a makefile to invoke the Intel Fortran compiler. The syntax is:

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

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 Understanding Input File Extensions.

For more information on ifort syntax, see Syntax for the ifort Command.

Note

For Windows and Mac OS* X systems, you can also use the compiler from within the integrated development environment.

You can use the command-line window to invoke the Intel Fortran Compiler in a number of ways, detailed below.

Using makefiles from the Command Line

Use makefiles to specify a number of files with various paths and to save this information for multiple compilations. For more information on using makefiles, see Using Makefiles to Compile Your Application.

Using the devenv command from the Command Line (Windows only)

Use `devenv` to set various options for the integrated development environment (IDE) as well as build, clean, and debug projects from the command line. For more information on the `devenv` command, see the `devenv` description in the Microsoft Visual Studio* documentation.

Using a .bat file from the Command Line

Use a .bat file to consistently execute the compiler with a desired set of options. This spares you retyping the command each time you need to recompile.

Default Tools
The default tools are summarized in the table below.

Tool	Default	Provided with Intel® Fortran Compiler?

Assembler for IA-32 architecture-based applications and Intel® 64 architecture-based applications	MASM* (Windows OS)	No
	operating system assembler, `as` (Linux OS and Mac OS X)	No
Assembler for IA-64 architecture-based applications	`ias`	Yes
Linker	Microsoft* linker (Windows OS)	No
	System linker, `ld(1)` (Linux OS and Mac OS X)	No

You can specify alternative tools and locations for preprocessing, compilation, assembly, and linking.

Assembler

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.

IA-32 architecture-based applications

Use any 32-bit assembler. For Windows, you can use the Microsoft Macro Assembler* (MASM), version 6.15 or higher, to link assembly language files with the object files generated by the compiler.

Intel® 64 architecture-based applications

For Windows systems, use the MASM provided on the Microsoft SDK. For Linux OS and Mac OS X systems, use the operating system assembler, `as`.

IA-64 architecture-based applications

Use the 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.

Use the `-S` (Linux) or `/asmfile:file.asm` (Windows) option to generate an assembly code file.

- The following command line on Linux OS and Mac OS X generates the assembly code file, `file.s`:
  ifort -S -c *file*.f
- The following command line on Windows OS generates the assembly code for `file.asm`:
  ifort /asmfile:file /c *file*.f

To assemble the file just produced, call the IA-64 architecture assembler.

- The following is the Linux OS command line:
  ias -Nso -p32 -o *file*.o file.s
- The following is the Windows OS command line:
  ias /Nso /p32 /o*file*.obj *file*.asm
  where the following assembler options are used:
  - Nso suppresses the sign-on message
  - p32 enables defining 32-bit elements as relocatable data elements; kept for backward compatibility
  - The file specified by the `o` option indicates the output object file name

The above `ias` command generates an object file, which you can link with the object file of the project.

Linker

On Linux OS and Mac OS X, the compiler calls the system linker, `ld(1)`, to produce an executable file from the object file.

On Windows OS, the compiler calls the Microsoft linker, `link`, to produce an executable file from the object files. The linker searches the path specified in the environment variable `LIB` to find any library files.

Specifying Alternative Tools and Locations

The Intel® Fortran Compiler lets you specify alternatives to default tools and locations for preprocessing, compilation, assembly, and linking. In addition, you can invoke options specific to the alternate tools on the command line. This functionality is provided by the `-Qlocation` or `/Qlocation` and `-Qoption` or `/Qoption` options.

For more information see the following topics:

- Qlocation compiler option
- Qoption compiler option

Compilation Phases

When invoked, the compiler driver 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 table that follows shows the compilation phases and the software that controls each phase.

Phase	Software	Architecture (IA-32, Intel® 64, or IA-64)
Preprocess (optional)	`fpp`	All
Compile	`fortcom`	All
Assemble	`IAS` or `MASM` (Windows OS)	IAS for IA-64 architecture based applications; Microsoft Macro Assembler* (MASM) can be used for IA-32 architecture based applications. See Default Tools for more information.

	as or `ias` (Linux OS)	`as` for IA-32 architecture-based applications and Intel® 64 architecture-based applications; `ias` for IA-64 architecture-based applications
Link	`LINK` (Windows OS)	All
	`ld(1)` (Linux OS and Mac OS X)	All

By default, the compiler driver performs the compile and link phases to produce the executable file.

The compiler driver passes object files and any unrecognized filename to the linker. The linker then determines whether the file is an object file or a library. For Linux OS and Mac OS X, the linker can also determine whether the file is a shared library (.so).

The compiler driver handles all types of input files correctly. Therefore, it can be used to invoke any phase of compilation.

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.

Compiling and Linking for Optimization

By default, all Fortran source files are separately compiled into individual object files.

If you want to allow full interprocedural optimizations to occur, you must use the `-ipo` (Linux OS and Mac OS X) or `/Qipo` (Windows OS) option.

By default, compilation is done with `-O2` (Linux OS and Mac OS X) or `/O2` (Windows). If you want to see if your code will benefit from some added optimizations, use `O3`. These aggressive optimizations may or may not improve your code speed.

For complete information about optimization, see Compiler Optimizations Overview in *Optimizing Applications*.

Compiling and Linking Multithread Programs

To build a multithread application that uses the Fortran run-time libraries, specify the `-threads` (Linux* OS and Mac OS* X) or `/threads` (Windows* OS) compiler option from the command line. For Windows systems, you can use also use the Microsoft integrated development environment (IDE), as described later in this topic.

You must also link with the correct library files.

The following applies to Linux OS and Mac OS X:

To create statically linked multithread programs, link with the static library named libifcoremt.a. To use shared libraries, link your application with libifcoremd.so (Linux OS) or libifcoremd.dylib (Mac OS X).

The following applies to Windows OS:

To create statically linked multithread programs, link with the re-entrant support library LIBIFCOREMT.LIB. To use shared libraries, use the shared LIBIFCOREMD.DLL library, which also re-entrant, and is referenced by linking your application with the LIBIFCOREMD.LIB import library.

Programs built with LIBIFCOREMT.LIB do not share Fortran run-time library code or data with any dynamic-link libraries they call. You must link with LIBIFCOREMD.LIB if you plan to call a DLL.

Additional Notes for Windows OS:

- The `/threads` compiler option is automatically set when you specify a multithread application in the visual development environment.
- Specify the compiler options `/libs=dll` and /threads if you are using both multithreaded code and DLLs. You can use the `/libs=dll` and `/threads` options only with Fortran Console projects, not QuickWin applications.

To compile and link your multithread program from the command line:

1. Make sure your IA32ROOT or IA64ROOT (Linux OS and Mac OS X) or LIB (Windows) environment variable points to the directory containing your library files.
2. Compile and link the program with the `-threads` (Linux OS and Mac OS X) or `/threads` (Windows) compiler option.
   For example:

`ifort -threads mythread.f90` (Linux OS and Mac OS X)

`ifort /threads mythread.f90` (Windows OS)

To compile and link your multithread program using the IDE (Windows OS):

1. Create a new project by clicking **File > New > Project**.
2. Click **Intel Fortran Projects** in the left pane (as shown above) to display the Intel Fortran project types. Choose the project type.
3. Add the file containing the source code to the project.
4. From the **Project** menu, select **Properties**.
   The **Property Pages** dialog box appears.
5. Choose the **Fortran** folder, **Libraries** category, and set the Runtime Library to **Multithreaded** or **Multithread DLL** (or their debug equivalents).
6. Create the executable file by choosing **Build Solution** from the **Build** menu.

What the Compiler Does by Default

By default, the compiler driver generates executable file(s) of the input file(s) and performs the following actions:

- Displays certain diagnostic messages, warnings, and errors.
- Performs default settings and optimizations, unless these options are overridden by specific options settings.
- Searches for source files in the current directory or in the directory path explicitly specified before a file name (for example, looks in "src" when the directory path is `src\test.f90`).
- Searches for include and module files in:
  - The directory path explicitly specified before a file name (for example, looks in "src" when the including source is specified as `src\test.f90`)
  - The current directory
  - The directory specified by using the `-module path` (Linux* OS and Mac OS* X) or `/module:path` (Windows* OS) option (for all module files)

    - The directory specified by using the `-Idir` ((Linux OS and Mac OS X) or `/Idir` (Windows OS) option (for module files referenced in USE statements and include files referenced in INCLUDE statements.)
    - For Windows OS, the include path specified by the INCLUDE environment variable (for all include or module files)
    - Any directory explicitly specified in any INCLUDE within an included file
- Passes options designated for linking as well as user-defined libraries to the linker. The linker searches for any library files in directories specified by the LIB variable, if they are not found in the current directory.

For unspecified options, the compiler uses default settings or takes no action.

You may want to use the `-assume keyword` (Linux OS and Mac OS X) or `/assume:keyword` (Windows OS) option to instruct the compiler to make certain assumptions. For example, `-assume buffered_io` tells the compiler to accumulate records in a buffer. For more information and the complete list of supported keywords, see the assume option reference page.

Note

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

Generating Listing and Map Files
Compiler-generated assembler output listings and linker-generated map files can help you understand the effects of compiler optimizations and see how your application is laid out in memory. They may also help you interpret the information provided in a stack trace at the time of an error.

How to Generate Assembler Output

When compiling from the command line, specify the `-S` (Linux* OS and Mac OS* X) or `/asmattr` option with one of its keyword (Windows* OS):

```
ifort -S file.f90 (Linux OS and Mac OS X)
ifort file.f90 /asmattr:all (Windows OS)
```

On Linux OS and Mac OS X, the resulting assembly file name has a .s suffix. On Windows OS, the resulting assembly file name has an .asm suffix.

Additionally, on Windows OS, you can use the Visual Studio integrated development environment:

1. Select **Project>Properties**.
2. Click `the` Fortran tab.
3. In the **Output Files** category, change the **Assembler Output** settings according to your needs. You can choose from a number of options such as **No Listing**, **Assembly-only Listing**, and **Assembly, Machine Code and Source**.

How to Generate a Link Map (.map) File

When compiling from the command line, specify the -Xlinker and `-M` options (Linux OS and Mac OS X) or the `/map` (Windows) option:

```
ifort file.f90 -Xlinker -M  (Linux OS and Mac OS X)
ifort file.f90 /map (Windows)
```

Additionally, on Windows systems, you can use the Visual Studio integrated development environment:

1. Select **Project>Properties**.
2. Click the **Linker** tab.
3. In the **Debug** category, select **Generate Map File**.

Saving Compiler Information in your Executable

If you want to save information about the compiler in your executable, use the `-sox` (Linux* OS and Mac OS* X) or `/Qsox` (Windows* OS) option. When you use this option, the following information is saved:

- compiler version number
- compiler options that were used to produce the executable

On Linux OS and Mac OS X:

To view the information stored in string format in the object file, use the following command:

```
strings -a a.out|grep comment:
```

On Windows OS:

To view the linker directives stored in string format in the object file, use the following command:

```
link /dump /directives filename.obj
```

The `-?comment` linker directive displays the compiler version information.

To search your executable for compiler information, use the following command:

```
findstr "Compiler" filename.exe
```

This searches for any strings that have the substring "Compiler" in them.

Building Applications from the Command Line

Using the Compiler and Linker from the Command Line

The `ifort` command is used to compile and link your programs from the command line.

You can either compile and link your projects in one step with the `ifort` command, or compile them with `ifort` and then link them as a separate step.

In most cases, you will use a single `ifort` command to invoke the compiler and linker.

You can use the `ifort` command in either of two windows:

- Your own terminal window, in which you have set the appropriate environment variables by executing the file called `ifortvars.sh` or `ifortvars.csh` (Linux* OS and Mac OS* X) or `ifortvars.bat` (Windows* OS). This file sets the environment variables such as PATH. By default, the `ifortvars` file is installed in the `\bin` directory

for your compiler. For more information, see Using the ifortvars File to Specify Location of Components.

- On Windows operating systems, the supplied Fortran command-line window in the Intel® Fortran program folder, in which the appropriate environment variables in `ifortvars.bat` are preset.

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

The driver program does the following:

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

You can also use `ld` (Linux OS and Mac OS X) or `link` (Windows OS) to build libraries of object modules. These commands provide syntax instructions at the command line if you request it with the `/?` or `/help` option.

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

Because the driver calls other software components, error messages may be returned by these other components. For instance, the linker may return a message if it cannot resolve a global reference. The `-watch` (Linux OS and Mac OS X) or `/watch` (Windows OS) command-line option can help clarify which component is generating the error.

Note

Windows systems support characters in Unicode* (multibyte) format; the compiler will process file names containing Unicode* characters.

Syntax for the ifort Command

Use the syntax below to invoke the Intel® Fortran Compiler from the command line:

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

An *option* is specified by one or more letters preceded by a hyphen (-) for Linux and Mac OS* X operating systems and a slash (/) for the Windows* operating system. (You can use a hyphen (-) instead of a slash for Windows OS, but this is not the preferred method.)

The following rules apply:

- 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. When a file is not in your path or working directory, specify the directory path before the file name. The filename extension specifies the type of file.

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`

- You cannot combine options with a single slash or hyphen, but must specify the slash or hyphen for each option specified. For example, this is correct: /1 /c
  But this is not: /1c
- Some compiler options are case-sensitive. For example, c and C are two different options.
- Options can take arguments in the form of filenames, strings, letters, and numbers. If a string includes spaces, it must be enclosed in quotation marks.
- All compiler options must precede the `-Xlinker` (Linux OS and Mac OS X) or `/link` (Windows OS) options. Options that appear following -Xlinker or /link are passed directly to the linker.
- Unless you specify certain options, the command line will both compile and link the files you specify. To compile without linking, specify the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) option.
- You can abbreviate some option names, entering as many characters as are needed to uniquely identify the option.
- Compiler options remain in effect for the whole compilation unless overridden by a compiler directive.
- On Windows OS, certain options accept one or more keyword arguments following the option name. To specify multiple keywords, you typically specify the option multiple times. However, some options allow you to use comma-separated keywords. For example:

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

You can use an equals sign (=) instead of the colon:

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

Examples of the ifort Command

This topic provides some examples of valid `ifort` commands. It also shows various ways to compile and link source files.

Compiling and Linking a Single Source File

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

```
ifort x.for
```

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

```
ifort x.for -o myprog.out (Linux OS and Mac OS X)
ifort x.for /exe:myprog.exe (Windows OS)
```

Compiling, but not Linking, a Source File

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

```
ifort -c x.for (Linux OS and Mac OS X)
ifort x.for /c (Windows OS)
```

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

```
ifort x.o (Linux OS and Mac OS X)
ifort x.obj (Windows )S)
```

Compiling and Linking Multiple Fortran Source Files

The following command compiles `a.for, b.for,` and `c.for`. It creates three temporary object files, then links the object files into an executable file named `a.out` (on Linux OS and Mac OS X) or `a.exe` (Windows OS):

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

When you use modules and compile multiple files, compile the source files that define modules before the files that reference the modules (in USE statements).

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

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

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:

Linux OS and Mac OS* X example:

1. **ifort -c array_calc.f90**
2. **ifort -c calc_aver.f90**
3. **ifort -c main.f90**
4. **ifort -o calc main.o array_calc.o calc_aver.o**

Windows* example:

1. **ifort /c array_calc.f90**
2. **ifort /c calc_aver.f90**
3. **ifort /c main.f90**
4. **ifort /exe:calc main.obj array_calc.obj calc_aver.obj**

In this sequence of commands:

Line **1**: The `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) option prevents linking and retains the object files. This command creates the module file `array_calculator.mod` and the object file `array_calc.o` (Linux OS and Mac OS X) or array_calc.obj (Windows OS). Note that the name in the MODULE statement determines the name of module file `array_calculator.mod`. Module files are written into the current working directory.

Line **2**: This command creates the object file `calc_aver.o` (Linux OS and Mac OS X) or `calc_aver.obj` (Windows OS).

Line **3**: This command creates the object file `main.o` (Linux OS and Mac OS X) or `main.out` (Windows OS)and uses the module file `array_calculator.mod`.

Line **4**: This command links all object files into the executable program named `calc`. To link files, use the `ifort` command instead of the `ld` command.

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` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) option to request additional symbol table information for source line debugging in the object and executable program files.

The following `ifort` command lines for Linux OS and Mac OS X systems use the `-o` option to name the executable program file `calc_debug`. The Mac OS X command line also uses the `-save-temps` option, which specifies that the object files should be saved; otherwise, they will be deleted by default.

```
ifort -g -o calc_debug array_calc.f90 calc_aver.f90 main.f90 (Linux)
ifort -g -save-temps -o calc_debug array_calc.f90 calc_aver.f90 main.f90 (Mac OS X)
```

The Windows OS equivalent of this command is the following:

```
ifort /debug:full /exe:calc_debug array_calc.f90 calc_aver.f90 main.f90
```

See also Debugging Fortran Programs and related sections.

Redirecting Command-Line Output to Files

When using the command line, you can redirect standard output and standard error to a file. This avoids displaying a lot of text, which 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.

Linux* OS and Mac OS* X:

The following example applies to Linux* OS and Mac OS* X.

To run the program more efficiently, redirect output to a file and then display the program output:

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

Windows* OS:

The following examples apply to Windows OS.

To place standard output into file `one.out` and standard error into file `two.out`, use the `ifort` command as follows:

```
ifort filenames /options 1>one.out 2>two.out
```

You can also use a short-cut form (omit the 1):

```
ifort filenames /options >one.out 2>two.out
```

To place standard output and standard error into a single file `both.out`, enter the `ifort` command as follows:

```
ifort filenames /options 1>both.out 2>&1
```

You can also use a short-cut form (omit the 1):

```
ifort filenames /options >both.out 2>&1
```

Using Makefiles to Compile Your Application

To specify a number of files with various paths and to save this information for multiple compilations, you can use a makefile.

On Linux OS and Mac OS X:

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.

On Windows OS:

To use a makefile to compile your input files, use the `nmake` command. For example, if your project is `your_project.mak`:

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

The arguments of this `nmake` command are as follows:

`/f`	A particular makefile.
`your_project.mak`	A makefile you want to use to generate object and executable files.
`FPP`	The compiler-invoking command you want to use. The name of this macro might be different for your makefile. This command invokes the preprocessor.
`LINK32`	The linker you want to use.

The `nmake` command creates object files (`.obj`) and executable files (`.exe`) specified in `your_project.mak` file.

.

Specifying Memory Models to use with Systems Based on Intel® 64 Architecture

The following applies to Linux* operating systems only.

Applications designed to take advantage of Intel® 64 architecture can be built with one of three memory models:

- small (-mcmodel=small)
  This causes code and data to be restricted to the first 2GB of address space so that all accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.
- medium (-mcmodel=medium)
  This causes code to be restricted to the first 2GB; however, there is no restriction on data. Code can be addressed with IP-relative addressing, but access of data must use absolute addressing.
- large (-mcmodel=large)
  There are no restrictions on code or data; access to both code and data uses absolute addressing.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. This can affect code size and performance. (IP-relative addressing is somewhat faster.)

Additional Notes on Memory Models and on Large Data Objects

- When you specify the medium or large memory models, you must also specify the –shared-intel compiler option to ensure that the correct dynamic versions of the Intel run-time libraries are used.
- When you build shared objects (.so), Position-Independent Code (PIC) is specified (that is, -fpic is added by the compiler driver) so that a single .so can support all three memory models. However, code that is to be placed in a static library, or linked statically, must be built with the proper memory model specified. Note that there is a performance impact to specifying the medium or large memory models.
- The use of the memory model (medium, large) option and the -shared-intel option is required as a by-product of the code models stipulated in the 64-bit Application Binary Interface (ABI), which is written specifically for processors with the 64-bit memory extensions. Both the compiler and the GNU linker (ld) are responsible for generating the proper code and necessary relocations on this platform according to the chosen memory model.
- The 2GB restriction on Intel® 64 architecture involves not only arrays greater than 2GB, but also COMMON blocks and local data with a total size greater than 2GB. The Compiler Options reference contains additional discussion of the supported memory models and offers details about the 2GB restrictions for each model. See the -mcmodel options page.
- If, during linking, you fail to use the appropriate memory model and dynamic library options, an error message in this format occurs:

```
<some lib.a library>(some .o): In Function <function>:
 : relocation truncated to fit: R_X86_64_PC32 <some symbol>
```

Allocating Common Blocks

Use the `-dyncom` (Linux OS and Mac OS X) or `/Qdyncom` (Windows OS) option to dynamically allocate common blocks at run time.

This option designates a common block to be dynamic. The space for its data is allocated at run time rather than 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` (Linux OS and Mac OS X)

`ifort /Qdyncom"BLK1,BLK2,BLK3" test.f` (Windows OS)

where `BLK1`, `BLK2`, and `BLK3` are the names of the common blocks to be made dynamic.

Guidelines for Using the /Qdyncom Option

The following are some limitations that you should be aware of when using the `-dyncom` (Linux OS and Mac OS X) or `/Qdyncom` (Windows OS) 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.

For more information, see the following topic:

- `-dyncom` compiler option

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 return the size in bytes of the 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.

Running Fortran Applications from the Command Line

If you run a program from the command line, the operating system searches directories listed in the PATH environment variable to find the executable file you have requested.

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

Multithreaded Programs

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

Using the /fpscomp:filesfromcmd Option

If you specify the `-fpscomp filefromcmd` (Linux OS and Mac OS X) or `/fpscomp:filesfromcmd` (Windows OS) option, the command line that executes the program can also include additional filenames to satisfy OPEN statements in your program in which the filename field (FILE specifier) has been left blank. The first filename on the command line is used for the first such OPEN statement executed, the second filename for the second OPEN statement, and so on. (In the Visual Studio IDE, you can provide these filenames using **Project>Properties.** Choose the Debugging category and enter the filenames in the Command Arguments text box.)

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

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

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

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

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

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

Instead of using the `-fpscomp filesfromcmd` or `/fpscomp:filesfromcmd` option, you can:

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

For more information, see the following topic:

- `fpscomp` compiler option

Input and Output Files

Understanding Input File Extensions

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

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

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

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

  - The compiler assumes that files with an extension of .f90 or .i90 are free-form source files.
  - The compiler assumes that files with an extension of .f, .for, .ftn, or .i are fixed-form (or tab-form) files.

- Files passed to the linker: `.a, .lib, .obj, .o, .exe, .res, .rbj, .def, .dll`

The most common file extensions and their interpretations are:

Filename	Interpretation	Action
filename`.a` (Linux* OS and Mac OS* X) *filename*`.lib` (Windows* OS)	Object library	Passed to the linker.
filename`.f` *filename*`.for` *filename*`.ftn` *filename*`.i`	Fortran fixed-form source	Compiled by the Intel® Fortran compiler.
filename`.fpp` and, on Linux, filenames with the following uppercase extensions: `.FPP, .F, .FOR, .FTN`	Fortran fixed-form source	Automatically preprocessed by the Intel Fortran preprocessor `fpp`; then compiled by the Intel Fortran compiler.
filename`.f90` *filename*`.i90`	Fortran free-form source	Compiled by the Intel Fortran compiler.
filename`.F90` (Linux OS and Mac OS X)	Fortran free-form source	Automatically preprocessed by the Intel Fortran preprocessor `fpp`; then compiled by the Intel Fortran compiler.
filename`.s` (Linux OS and Mac OS X) *filename*`.asm` (Windows)	Assembly file	Passed to the assembler.
filename`.o` (Linux OS and Mac OS X)	Compiled object	Passed to the linker.

filename.obj (Windows OS) file

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

Producing Output Files

The output produced by the ifort command includes:

- An object file, if you specify the -c (Linux OS and Mac OS X) or /c (Windows OS) option on the command line. An object file is created for each source file.
- An executable file, if you omit the -c or /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 OS, mylib.dylib on Mac OS X, or mylib.dll on Windows OS), if you use the -shared (Linux), -dynamiclib (Mac OS X) or /libs:dll (Windows OS) option.
- Assembly files, if you use the -S (Linux OS and Mac OS X) or /S (Windows OS) option. This creates an assembly file for each source file.

You control the production of output files by specifying the appropriate compiler options on the command line or using the appropriate properties in the integrated development environment for Windows OS and Mac OS X.

For instance, if you do not specify the -c or /c option, the compiler generates a temporary object file for each source file. It then invokes the linker to link the object files into one executable program file and causes the temporary object files to be deleted.

If you specify the -c or /c option, 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 for Linux OS and Mac OS X or link for Windows OS) directly to link in objects. On Linux OS and Mac OS X systems, you can also call xild or use the archiver (ar) and xiar to create a library. For Mac OS X, you would use libtool to generate a library.

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

The output files include the following:

Output File	Extension	How Created on the Command Line
Object file	.o (Linux OS and Mac OS X) .obj (Windows OS)	Created automatically.
Executable file	.out (Linux OS and Mac OS X) .exe (Windows OS)	Do not specify -c or /c.
Shareable library file	.so (Linux OS) .dylib (Mac OS X)	Specify -shared (Linux OS), -dynamiclib (Mac OS X) or /libs:dll (Windows OS) and do not specify -c or /c .

	.dll (Windows OS)	
Module file	.mod	Created if a source file being compiled defines a Fortran module (MODULE statement).
Assembly file	.s (Linux OS and Mac OS* X) .asm (Windows OS)	Created if you specify the `S` option. An assembly file for each source file is created.

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 the default) use the `-o` *`output`* (Linux OS and Mac OS X) or `/exe:`*`output`* (Windows OS) option, where *`output`* specifies the file name.

Note

You cannot use the `c` and `o` options 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.

For Windows* OS, if the TEMP environment variable is not defined, the current working directory is used to store temporary files. For Linux* OS and Mac OS* X, if the TEMP environment variable is not defined, the `/tmp` directory is used to store temporary files.

Temporary files are usually deleted. Use the `-save-temps` (Linux OS and Mac OS X) or `/Qsave-temps` (Windows OS) compiler option to save temporary files created by the compiler in the current working directory. This option only saves intermediate files that are normally created during compilation.

For performance reasons, use a local drive (rather than a network drive) to contain temporary files.

To view the file name and directory where each temporary file is created, use the `-watch all` (Linux OS and Mac OS X) or `/watch:all` (Windows OS) option.

To create object files in your current working directory, use the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) option.

Any object files that you specify on the command line are retained.

Setting Environment Variables

Using the ifortvars File to Specify Location of Components

Before you first invoke the compiler, you need to be sure certain environment variables are set. These environment variables define the location of the various compiler-related components.

The Intel Fortran Compiler installation includes a file that you can run to set environment variables.

- On Linux* OS and Mac OS* X, the file is a shell script called `ifortvars.sh` or `ifortvars.csh`.
- On Windows* OS, the file is a batch file called `ifortvars.bat`.

The following information is operating system-dependent.

Linux OS and Mac OS X:

Set the environment variables before using the compiler. You can use the `source` command to execute the shell script, `ifortvars.sh` or `ifortvars.csh`, from the command line to set them.

The script takes an architecture argument:

- ia32: Compiler and libraries for IA-32 architecture only
- intel64: Compiler and libraries for Intel® 64 architecture only
- ia64: Compiler and libraries for IA-64 architectures only (Linux OS)

For example, to execute this script file for the bash shell:

```
source /opt/intel/Compiler/11.0/package_id/bin/ifortvars.sh ia32
```

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

```
source /opt/intel/Compiler/11.0/package_id/bin/ifortvars.csh ia32
```

If you want `ifortvars.sh` to run automatically when you start Linux OS or Mac OS X, you can 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/11.0/package_id/bin/ifortvars.sh ia32
```

If you compile a program without ensuring the proper environment variables are set, you will see an error similar to the following 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
```

Windows OS:

Under normal circumstances, you do not need to run the ifortvars.bat batch file. The Fortran command-line window sets these variables for you automatically. To activate this command-line window, select **Fortran Build Environment for applications...** available from the Intel Fortran program folder.

Note

You will need to run the batch file if you open a command-line window without using the provided **Build Environment for applications...** menu item in the Intel Fortran program folder or if you want to use the compiler from a script of your own.

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

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

The batch file takes an architecture argument:

- ia32: Compiler and libraries for IA-32 architecture only
- ia32_intel64: Compiler running on IA-32 architecture that generates code for Intel® 64 architecture; uses Intel® 64 architecture libraries
- ia32_ia64: Compiler running on IA-32 architecture that generates code for IA-64 architecture; uses IA-64 architecture libraries
- intel64: Compiler and libraries for Intel® 64 architecture only
- ia64: Compiler and libraries for IA-64 architectures only

Setting Compile-Time Environment Variables

The following table shows the compile-time environment variables that affect the Intel® Fortran Compiler:

Environment Variable	Description
IFORTCFG	Specifies a configuration file that the compiler should use instead of the default configuration file. By default, the compiler uses the default configuration file (`ifort.cfg`) from the same directory where the compiler executable resides. **Note:** On Windows* operating systems, this environment variable cannot be set from the IDE.
INTEL_LICENSE_FILE	Specifies the location of the product license file.
PATH	Specifies the directory path for the compiler executable files.
TMP, TMPDIR, TEMP	Specifies the directory in which to store temporary files. See Temporary Files Created by the Compiler or Linker. **Note:** On Windows operating systems, this environment variable cannot be set from the IDE.
FPATH (Linux* OS and Mac OS* X)	The path for include and module files.
GCCROOT (Linux OS and Mac OS X)	Specifies the location of the gcc binaries. Set this variable only when the compiler cannot locate the gcc binaries when using the -gcc-name option.

GXX_INCLUDE (Linux OS and Mac OS X)	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 (Linux OS and Mac OS X)	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.
LIBRARY_PATH (Linux OS and Mac OS X)	The path for libraries to be used during the link phase.
LD_LIBRARY_PATH (Linux OS)	The path for shared (.so) library files.
DYLD_LIBRARY_PATH (Mac OS X)	The path for dynamic libraries.
INCLUDE (Windows OS)	Specifies the directory path for the include files (files included by an INCLUDE statement, #include files, RC INCLUDE files, and module files referenced by a USE statement).
LIB (Windows OS)	Specifies the directory path for .LIB (library) files, which the linker links in. If the LIB environment variable is not set, the linker looks for .LIB files in the current directory.

Additionally, there are a number of run-time environment variables that you can set. For a list of environment variables recognized at run time and information on setting and viewing environment variables, see Setting Run-Time Environment Variables.

You can use the `SET` command at the command prompt to set environment variables. Depending on your operating system, there are additional ways to set environment variables.

Setting Environment Variables (Linux OS and Mac OS X)

You can set environment variables by using the `ifortvars.csh and ifortvars.sh` files to set several at a time. The files are found in the product's `bin` directory. See Using the ifortvars File to Specify Location of Components.

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
```

Setting Environment Variables (Windows OS)

Certain environment variables specifying path, library, and include directories can be defined in the IDE on a per user basis using **Tools>Options...** from the menu bar. For more information, see Specifying Path, Library, and Include Directories.

Additionally, you can set the environment variables needed by Intel Fortran using the `ifortvars.bat` file. See Using the ifortvars File to Specify Location of Components.

Note

If you specify `devenv/useenv` on the command line to start the IDE, the IDE uses the PATH, INCLUDE, and LIB environment variables defined for that command line when performing a build. It uses these values instead of the values defined using **Tool>Options**.

For more information on the `devenv` command, see the `devenv` description in the Microsoft Visual Studio* documentation.

During installation, the Intel Fortran compiler may modify certain system-wide environment variables, depending on your installation choices. (For more information, see the install.htm file.)

To view or change these environment variable settings, do the following:

On Windows* 2000, Windows NT* 4, or Windows XP* operating systems:

1. Log into an account with Administrator privilege.
2. Open the Control panel (**Start>Settings>Control panel**).
3. Click **System**.
4. On Windows 2000 and Windows XP systems: Click the **Advanced** tab and then click the **Environment Variables** button. On Windows NT 4 systems: Click the **Environment** tab.
5. View or change the displayed environment variables.
6. To have any environment variable changes take effect immediately, click **Apply**.
7. Click **OK**.

Note

Changing system-wide environment variables affects command line builds (those done without IDE involvement), but not builds done through the IDE. IDE builds are managed by the environment variables set in the IDE Using **Tools>Options**. An exception to this is an IDE build (`devenv`) done from the command line that specifies the `/useenv` option. In this case, the IDE uses the PATH, INCLUDE, and LIB environment variables defined for that command line.

You can set an environment variable from within a program by calling the SETENVQQ routine. For example:

```
USE IFPORT
 LOGICAL(4) success
 success = SETENVQQ("PATH=c:\mydir\tmp")
 success = &
 SETENVQQ("LIB=c:\mylib\bessel.lib;c:\math\difq.lib")
```

Setting Run-Time Environment Variables

The Intel® Fortran run-time system recognizes a number of environment variables. These variables can be used to customize run-time diagnostic error reporting, allow program continuation under certain conditions, disable the display of certain dialog boxes under certain conditions, and allow just-in-time debugging. For a list of run-time environment variables used by OpenMP*, see OpenMP Environment Variables in *Optimizing Applications*.

For information on setting compile time environment variables, see Setting Compile-Time Environment Variables.

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.
- `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_FMT_TERMINATOR`
  This variable specifies the numbers of the units to have a specific record terminator. See Record Types.
- `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 `CONIN$` (standard input). If `FOR_ACCEPT` is defined (as a file name optionally containing a path), the specified file would be read.
- `FOR_DEBUGGER_IS_PRESENT`
  This variable tells the Fortran run-time library that your program is executing under a debugger. If set to True, it generates debug exceptions whenever severe or continuous errors are detected. Under normal conditions you don't need to set this variable on Windows systems, as this information can be extracted from the operating system. On Linux* OS and Mac OS* X, you will need to set this variable if you want debug exceptions. Setting this variable to True when your program is *not* executing under a debugger will cause unpredictable behavior.
- `FOR_DEFAULT_PRINT_DEVICE` (Windows* OS only)
  This variable lets you specify the print device other than the default print device PRN (LPT1) for files closed (CLOSE statement) with the DISPOSE='PRINT' specifier. To specify a different print device for the file associated with the CLOSE statement DISPOSE='PRINT' specifier, set `FOR_DEFAULT_PRINT_DEVICE` to any legal DOS print device before executing the program.

- FOR_DIAGNOSTIC_LOG_FILE
  If this variable is set to the name of a file, diagnostic output is written to the specified file.
  The Fortran run-time system attempts to open that file (append output) and write the error information (ASCII text) to the file.
  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.
  See also Locating Run-Time Errors and Using Traceback Information.
- FOR_DISABLE_DIAGNOSTIC_DISPLAY
  This variable 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.
  See also Using Traceback Information.
- 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. If the program is executing under a debugger, the automatic output of the stack trace information by the Fortran library will be disabled to reduce noise. You should use the debugger's stack trace facility if you want to view the stack trace.
  See also Locating Run-Time Errors and Using Traceback Information.
- 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 CONOUT$ (standard output). If FOR_PRINT is defined (as a file name optionally containing a path), the specified file would be written to.
- FOR_READ
  A READ statement that uses an asterisk (*) in place of a unit number does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the FOR_READ environment variable. If FOR_READ is not defined, the code READ `(*,f) iolist` or READ `f,iolist` reads from CONIN$ (standard input). If

FOR_READ is defined (as a file name optionally containing a path), the specified file would be read.

- FOR_TYPE
  The TYPE statement does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the FOR_TYPE environment variable. If FOR_TYPE is not defined, the code TYPE `f,iolist` writes to CONOUT$ (standard output). If FOR_TYPE is defined (as a file name 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` (Linux OS and Mac OS X) or `/assume:buffered_io` (Windows OS) compiler option.
- FORT_CONVERT*n*
  Lets you specify the data format for an unformatted file associated with a particular unit number (*n*), as described in Methods of Specifying the Data Format.
- FORT_CONVERT.*ext* and FORT_CONVERT_*ext*
  Lets you specify the data format for unformatted files with a particular file extension suffix (ext), as described in Methods of Specifying the Data Format.
- FORT_FMT_RECL
  Lets you specify the default record length (normally 132 bytes) for formatted files.
- FORT_UFMT_RECL
  Lets you specify the default record length (normally 2040 bytes) for unformatted files.
- 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` (Linux OS and Mac OS X) or `/fpscomp:filesfromcmd` (Windows OS) was not specified. Preconnected files attached to units 0, 5, and 6 are by default associated with system standard I/O files.
- NLSPATH (Linux OS and Mac OS X only)
  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 containing 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* Visual Fortran.
  See also Using Traceback Information.
- TBK_FULL_SRC_FILE_SPEC
  By default, the traceback output displays only the file name and extension in the source file field. To display complete file name information including the path, set the environment variable TBK_FULL_SRC_FILE_SPEC to true.

The variable `FOR_FULL_SRC_FILE_SPEC` is also recognized for compatibility with Compaq* Visual Fortran.
See also Using Traceback Information.

- `TMP`, `TMPDIR`, and `TEMP`
  Specifies an alternate working directory where temporary files are created. See Temporary Files Created by the Compiler or Linker.

Setting Environment Variables within a Program

You can set a run-time environment variable from within a program by calling the SETENVQQ routine. For example:

```
program ENVVAR
use ifport
LOGICAL(4) res
! Add other data declarations here
! call SETENVQQ as a function
res=SETENVQQ("FOR_IGNORE_EXCEPTIONS=T")
```

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 (-) (Linux* OS and Mac OS* X) or a forward slash (/) (Windows*OS).

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

Depending on your operating system, compiler options are typically specified in the following ways:

- On the compiler command line
- In the IDE, either Xcode (Mac OS X) or Microsoft Visual Studio* (Windows OS)

Most compiler options perform their work at compile time, although a few apply to the generation of extra code used at run time.

For more information about compiler options, see the Compiler Options reference.

For information on the option mapping tool, which shows equivalent options between Windows and Linux OS, see the Option Mapping Tool.

Getting Help on Options

For help, enter `-help` [*category*] (Linux OS and Mac OS X) or `/help` [*category*] (Windows) on the command line, which displays brief information about all the command-line options or a specific category of compiler options.

The Compiler Options reference provides a complete description of each compiler option, including the `-help` 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` (Linux OS and Mac OS X) or `/align` (Windows) 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.

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

Note

The Compiler Option Mapping Tool does not run on Mac OS* X.

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 the IA-64 architecture to a like option available on the IA-32 architecture or Intel® 64 architecture.

Calling the Option Mapping Tool with the Compiler

If you use the compiler to execute the Option Mapping Tool, the following syntax applies:

```
<compiler command> <map-opts option> <compiler option(s)>
```

Example: Finding the Windows OS equivalent for `-fp`

```
ifort -map-opts -fp
Intel(R) Compiler option mapping tool

mapping Linux OS options to Windows OS for Fortran

'-map-opts' Linux OS option maps to
  --> '-Qmap-opts' option on Windows OS
  --> '-Qmap_opts' option on Windows OS

'-fp' Linux OS option maps to
  --> '-Oy-' option on Windows OS
```

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 OS options to Windows OS for Fortran

'-fp' Linux OS option maps to
  --> '-Oy-' option on Windows OS
```

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` (Linux* OS and Mac OS* X) `/warn:declarations` or `/4Yd` (Windows* OS)
NODECLARE	`-warn nodeclarations` (Linux OS and Mac OS X) `/warn:nodeclarations` or `/4Nd` (Windows OS)

DEFINE *symbol*	`-Dname` (Linux OS and Mac OS X) `/define:symbol` or `/Dname` (Windows OS)
FIXEDFORMLINESIZE:*option*	`-extend_source [option]` (Linux OS and Mac OS X) /extend_source[:*n*] or /4L*n* (Windows OS)
FREEFORM	`-free` or `-nofixed` (Linux OS and Mac OS X) `/free` or `/nofixed` or `/4Yf` (Windows OS)
NOFREEFORM	`-nofree` or `-fixed` (Linux OS and Mac OS X) `/nofree` or `/fixed` or `/4Nf` (Windows OS)
INTEGER:*option*	`-integer_size` *option* (Linux OS and Mac OS X) `/integer_size:`*option* or `/4I`*option* (Windows OS)
OBJCOMMENT	`/libdir:user` (Windows OS)
OPTIMIZE [: *n*]	-O (Linux OS and Mac OS X) or /O (Windows OS) n is 0, 1, 2, or 3 for opt levels -O0 through -O3. If n is omitted, default is 2.
NOOPTIMIZE	-O0 (Linux OS and Mac OS X) or /Od (Windows OS)
PACK:*option*	`-align [`*option*`]` (Linux OS and Mac OS X) `/align[:`*n*`]` or `/Zp`*n* (Windows OS)
REAL:*option*	`-real_size` *option* (Linux OS and Mac OS X) `/real_size:`*option* or `/4R`*option* (Windows)
STRICT	`-warn stderrors` with `-stand` (Linux OS and Mac OS X) `/warn:stderrors with /stand:f90` or `/4Ys` (Windows)
NOSTRICT	`-warn nostderrors` (Linux OS and Mac OS X) `/warn:nostderrors` or `/4Ns` (Windows OS)

Note

For Windows OS, the compiler directive names above are specified using the prefix `!DEC$` followed by a space; for example: `!DEC$ NOSTRICT`. The prefix `!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$, and !MS$`.

For more information on compiler directives, see Directive Enhanced Compilation.

Preprocessing

Using the fpp Preprocessor

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

The Fortran preprocessor, fpp, is provided as part of the Intel® Fortran product. When you use a preprocessor for Intel Fortran source files, the generated output files are used as input source files by the Compiler.

Preprocessing performs such tasks as preprocessor symbol (macro) substitution, conditional compilation, and file inclusion. Intel Fortran predefined symbols are described in Predefined Preprocessor Symbols.

The Compiler Options reference provides syntactical information on fpp. Additionally, it contains a list of fpp options that are available when fpp is in effect.

Automatic Preprocessing by the Compiler

By default, the preprocessor is not run on files before compilation. However, the Intel Fortran compiler automatically calls fpp when compiling source files that have a filename extension of `.fpp`, and, on Linux* OS and Mac OS* X, file extensions of `.F`, `.F90`, `.FOR`, `.FTN`, or `.FPP`. For example, the following command preprocesses a source file that contains fpp preprocessor directives, then passes the preprocessed file to the compiler and linker:

```
ifort source.fpp
```

If you want to preprocess files that have other Fortran extensions than those listed, you have to explicitly specify the preprocessor with the `-fpp` compiler option.

The fpp preprocessor can process both free- and fixed-form Fortran source files. By default, filenames with the suffix of .F, .f, .for, or .fpp are assumed to be in fixed form. Filenames with a suffix of .F90 or .f90 (or any other suffix not specifically mentioned here) are assumed to be free form. You can use the `-free` (Linux OS and Mac OS X) or /free (Windows* OS) option to specify free form and the `-fixed` (Linux OS and Mac OS X) or `/fixed` (Windows OS) option to explicitly specify fixed form.

The fpp preprocessor recognizes tab format in a source line in fixed form.

Running fpp to Preprocess Files

You can explicitly run fpp in these ways:

- On the `ifort` command line, use the `ifort` command with the `-fpp` (Linux OS and Mac OS X) or `/fpp` (Windows OS) option. By default, the specified files are then compiled and linked. To retain the intermediate (.i or .i90) file, specify the `-save-temps` (Linux OS and Mac OS X) or `/Qsave-temps` (Windows OS) option.
- On the command line, use the fpp command. In this case, the compiler is not invoked. When using the fpp command line, you need to specify the input file and the intermediate (.i or .i90) output file. For more information, type `fpp -help` (Linux OS and Mac OS X) or fpp `/help` (Windows OS) on the command line.
- In the Microsoft Visual Studio* IDE, set the Preprocess Source File option to Yes in the **Fortran Preprocessor Option** Category. To retain the intermediate files, add `/Qsave-temps` to Additional Options in the **Fortran Command Line** Category.

fpp has some of the capabilities of the ANSI C preprocessor and supports a similar set of directives. Directives must begin in column 1 of any Fortran source files. Preprocessor directives are not part of the Fortran language and not subject to the rules for Fortran statements. Syntax for directives is based on that of the C preprocessor.

The following lists some common cpp features that are supported by fpp; it also shows common cpp features that are not supported.

Supported cpp features: **Unsupported cpp features:**

#define, #undef, #ifdef, #ifndef, #if, #elif, #else, #endif, #include, #error, #warning, #line	#pragma, #ident
# (stringsize) and ## (concatenation) operators	spaces or tab characters preceding the initial "#" character # followed by empty line
! as negation operator	\ backslash-newline

Unlike cpp, fpp does not merge continued lines into a single line when possible.

You do not usually need to specify preprocessing for Fortran source programs unless your program uses fpp preprocessing commands, such as those listed above.

Caution

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

fpp Source Files

A source file can contain fpp tokens in the form of :

- fpp directive names. For more information on directives, see Using fpp Directives.
- symbolic names including Fortran keywords. fpp permits the same characters in names as Fortran. For more information on symbolic names, see Using Predefined Preprocessor Symbols.
- constants. Integer, real, double, and quadruple precision real, binary, octal, hexadecimal (including alternate notation), character, and Hollerith constants are allowed.
- special characters, space, tab and newline characters
- comments, including:
  - Fortran language comments. A fixed form source line containing one of the symbols C, c, *, d, or D in the first position is considered a comment line. The ! symbol is interpreted as the beginning of a comment extending to the end of the line except when the "!" occurs within a constant-expression in a #if or #elif directive. Within such comments, macro expansions are not performed, but they can be switched on by `-f-com=no`.
  - fpp comments between / and */. They are excluded from the output and macro expansions are not performed within these symbols. fpp comments can be nested: for each /* there must be a corresponding */. fpp comments are useful for excluding from the compilation large portions of source instead of commenting every line with a Fortran comment symbol.
  - C++ -like line comments which begin with // (double-slash).

A string that is a token can occupy several lines, but only if its input includes continued line characters using the Fortran continuation character &. fpp will not merge such lines into one line.

Identifiers are always placed on one line by fpp. For example, if an input identifier occupies several lines, it will be merged by fpp into one line.

fpp Output

Output consists of a modified copy of the input, plus lines of the form:

```
#line_number  file_name
```

These are inserted to indicate the original source line number and filename of the output line that follows. Use the fpp option `-P` (Linux OS and Mac OS X) or `/P` (Windows OS) to disable the generation of these lines.

Diagnostics

There are three kinds of fpp diagnostic messages:

- warnings: preprocessing of source code is continued and the fpp return value is 0
- errors: fpp continues preprocessing but sets the return value a nonzero value which is the number of errors
- fatal errors: fpp stops preprocessing and returns a nonzero return value.

The messages produced by fpp are intended to be self-explanatory. The line number and filename where the error occurred are displayed along with the diagnostic on stderr.

Using fpp Directives

All fpp directives start with the number sign (#) as the first character on a line. White space (blank or tab characters) can appear after the initial "#" for indentation.

fpp directives (beginning with the # symbol in the first position of lines) can be placed anywhere in a source code, in particular before a Fortran continuation line. However, fpp directives within a macro call may not be divided among several lines by means of continuation symbols.

fpp directives can be grouped according to their purpose.

Directives for string substitution

fpp contains directives that result in substitutions in a user's program:

Directive	Result
`__FILE__`	replace this string with the input file name (a character string literal)
`__LINE__`	replace this string with the current line number in the input file (an integer constant)
`__DATE__`	replace this string with the date that fpp processed the input file (a character string literal in the form `Mmm dd yyyy`)

__TIME__ replace this string with the time that fpp processed the input file (a character string literal in the form hh:mm:ss)

Directive for inclusion of external files

There are two forms of file inclusion:

#include "*filename*"

#include <*filename*>

This directive reads in the contents of the named file into this location in the source. The lines read in from the file are processed by fpp just as if they were part of the current file.

When the <*filename*> notation is used, *filename* is only searched for in the standard "include" directories. See the -I option and also the -Y option for more detail. No additional tokens are allowed on the directive line after the final '"' or ">".

Files are searched for in the following order:

- for #include "*filename*":
  - in the directory in which the source file resides
  - in the directories specified by the -I or -Y option
  - in the default directory
- for #include <*filename*>:
  - in the directories specified by the -I or -Y option
  - in the default directory

Directive for line control

This directive takes the following form:

#*line-number* "*filename*"

This directive generates line control information for the Fortran compiler. *line-number* is an integer constant that is the line number of the next line. "*filename*" is the name of the file containing the line. If "*filename*" is not given, the current filename is assumed.

Directive for fpp variable and macro definitions

The #define directive, used to define both simple string variables and more complicated macros, takes the two forms.

The first form is the definition of an fpp variable:

#define *name token-string*

In the above, occurrences of *name* in the source file will be substituted with *token-string*.

The second form is the definition of an fpp macro.

#define *name*(*argument* [, *argument*] ...) *token-string*

In the above, occurrences of the macro *name* followed by the comma-separated list of actual arguments within parentheses are replaced by *token-string* with each occurrence of each *argument* in *token-string* replaced by the token sequence representing the corresponding "actual" argument in the macro call.

An error is produced if the number of macro call arguments is not the same as the number of arguments in the corresponding macro definition. For example, consider this macro definition:

```
#define INTSUB(m, n, o) call mysub(m, n, o)
```

Any use of the macro INTSUB must have three arguments. In macro definitions, spaces between the macro name and the open parenthesis "(" are prohibited to prevent the directive from being interpreted as an fpp variable definition with the rest of the line beginning with the open parenthesis "(" being interpreted as its token-string.

An fpp variable or macro definition can be of any length and is limited only by the newline symbol. It can be defined in multiple lines by continuing it to the next line with the insertion of "\". The occurrence of a newline without a macro-continuation signifies the end of the macro definition.

Example:

```
#define long_macro_name(x,\
   y) x*y
```

The scope of a definition begins from the #define and encloses all the source lines (and source lines from #include files) to the end of the current file, except for:

- files included by Fortran INCLUDE statements
- fpp and Fortran comments
- Fortran IMPLICIT statements that specify a single letter
- Fortran FORMAT statements
- numeric, typeless, and character constants

Directive for undefining a macro

This directive takes the following form:

```
#undef name
```

This directive removes any definition for *name* (produced by -D options, #define directives, or by default). No additional tokens are permitted on the directive line after *name*. If *name* has not been defined earlier, then the #undef directive has no effect.

Directive for macro expansion

If, during expansion of a macro, the column width of a line exceeds column 72 (for fixed format) or column 132 (for free format), fpp inserts appropriate Fortran continuation lines.

For fixed format, there is a limit on macro expansions in label fields (positions 1-5):

- a macro call (together with possible arguments) should not extend beyond column 5
- a macro call whose name begins with one of the Fortran comment symbols is considered to be part of a comment
- a macro expansion may produce text extending beyond column 5. In this case, a warning will be issued

In fixed format, when the fpp `-Xw` option has been specified, an ambiguity may occur if a macro call occurs in a statement position and a macro name begins or coincides with a Fortran keyword. For example, consider the following:

```
#define callp(x)   call f(x)
  call p(0)
```

fpp cannot determine how to interpret the "call p" token sequence above. It could be considered as a macro name. The current implementation does the following:

- the longer identifier is chosen (callp in this case)
- from this identifier the longest macro name or keyword is extracted
- if a macro name has been extracted a macro expansion is performed. If the name begins with some keyword, fpp issues an appropriate warning
- the rest of the identifier is considered as a whole identifier

In the previous example, the macro expansion is performed and the following warning is produced:

```
warning: possibly incorrect substitution of macro callp
```

This situation appears only when preprocessing a fixed format source code and when the space symbol is not interpreted as a token delimiter.

In the following case, a macro name coincides with a beginning of a keyword:

```
 #define INT  INTEGER*8
              INTEGER k
```

The INTEGER keyword will be found earlier than the INT macro name. There will be no warning when preprocessing such a macro definition.

Directives for conditional selection of source text

There are three forms of conditional selection of source text.

Form 1:

```
      #if condition_1
                  block_1
            #elif condition_2
                  block_2
            #elif ...
            #else
                   block_n
            #endif
```

Form 2:

```
       #ifdef name
                  block_1
            #elif condition
                  block_2
            #elif ...
            #else
```

```
            block_n
        #endif
```

Form 3:

```
    #ifndef name
            block_1
        #elif condition
            block_2
        #elif ...
        #else
            block_n
        #endif
```

The `elif` and `else` parts are optional in all three forms. There may be more than one `elif` part in each form.

Conditional expressions

c*ondition_1*, *condition_2*, etc. are logical expressions involving fpp constants, macros, and intrinsic functions. The following items are permitted:

- C language operations: <, >, ==, !=, >=, <=, +, -, /, *, %, <<, >>, &, ~, |, &&, || They are interpreted by fpp in accordance to the C language semantics (this facility is provided for compatibility with "old" Fortran programs using cpp)
- Fortran language operations: .AND., .OR., .NEQV., .XOR., .EQV., .NOT., .GT., .LT., .LE., .GE., .NE., .EQ., ** (power).
- Fortran logical constants: .TRUE. , .FALSE.
- the fpp intrinsic function `"defined": defined(name)` or `defined name` which returns .TRUE. if name is defined as an fpp variable or a macro or returns .FALSE. if the name is not defined

  `#ifdef` is a shorthand for `#if defined(name)` and `#ifndef` is a shorthand for `#if .not. defined(name)`.

Only these items, integer constants, and names can be used within a constant-expression. A name that has not been defined with the `-D` option, a `#define` directive, or by default, has a value of 0. The C operation `!=` (not equal) can be used in `#if` or `#elif` directive, but not in the `#define` directive, where the symbol `!` is considered as the Fortran comment symbol by default.

Conditional constructs

The following table summarizes conditional constructs.

Construct	Result
`#if` *`condition`*	Subsequent lines up to the matching `#else`, `#elif`, or `#endif` directive appear in the output only if *`condition`* evaluates to .TRUE. .
`#ifdef` *`name`*	Subsequent lines up to the matching `#else`, `#elif`, or `#endif` appear in the output only if `name` has been defined, either by a `#define` directive or by the `-D` option, with no intervening `#undef` directive. No additional tokens are permitted on the directive line after *`name`*.

`#ifndef` *`name`*
: Subsequent lines up to the matching `#else`, `#elif`, or `#endif` appear in the output only if *`name`* has not been defined, or if its definition has been removed with an `#undef` directive. No additional tokens are permitted on the directive line after *`name`*.

`#elif` *`condition`*
: Subsequent lines up to the matching `#else`, `#elif`, or `#endif` appear in the output only if all of the following occur:

  - The condition in the preceding `#if` directive evaluates to .FALSE. or the name in the preceding `#ifdef` directive is not defined, or the name in the preceding `#ifndef` directive is defined.
  - The conditions in all of the preceding `#elif` directives evaluate to .FALSE.
  - The condition in the current `#elif` evaluates to .TRUE.

  Any condition allowed in an `#if` directive is allowed in an `#elif` directive. Any number of `#elif` directives may appear between an `#if`, `#ifdef`, or `#ifndef` directive and a matching `#else` or `#endif` directive.

`#else`
: Subsequent lines up to the matching `#endif` appear in the output only if all of the following occur:

  - The condition in the preceding `#if` directive evaluates to .FALSE. or the name in the preceding `#ifdef` directive is not defined, or the name in the preceding `#ifndef` directive is defined.
  - The conditions in all of the preceding `#elif` directives evaluate to .FALSE.

`#endif`
: End a section of lines begun by one of the conditional directives `#if`, `#ifdef`, or `#ifndef`. Each such directive must have a matching `#endif`.

Using Predefined Preprocessor Symbols

Preprocessor symbols (macros) let you substitute values in a program before it is compiled. The substitution is performed in the preprocessing phase.

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

You can use the `-D` (Linux* OS and Mac OS* X) or `/D` (Windows* OS) option to define the symbol names to be used during preprocessing. This option performs the same function as the `#define` preprocessor directive.

For more information, see the following topic:

- D compiler option

Preprocessing with fpp replaces every occurrence of the defined symbol name with the specified value. Preprocessing compiler directives only allow IF and IF DEFINED.

If you want to disable symbol replacement (also known as macro expansion) during the preprocessor step, you can specify the following: `-fpp:"-macro=no"` (Linux OS and Mac OS X or `/fpp:"/macro=no"` (Windows OS).

Disabling preprocessor symbol replacement is useful for running fpp to perform conditional compilation (using #ifdef, etc.) without replacement.

You can use the `-U` (Linux OS and Mac OS X) or `/U` (Windows OS) option to suppress an automatic definition of a preprocessor symbol. This option suppresses any symbol definition currently in effect for the specified name. This option performs the same function as an `#undef` preprocessor directive.

For more information, see the following topic:

- U compiler option

Windows OS:

The following information applies to Windows operating systems.

In addition to specifying preprocessor symbols on the command line, you can also specify them in the Visual Studio* integrated development environment (IDE). To do this, select **Project>Properties** and use the **Preprocessor Definitions** item in the **General Options** or **Preprocessor Options** category.

The following preprocessor symbols are available:

Predefined Symbol Name and Value	Conditions When this Symbol is Defined
`__INTEL_COMPILER=1100`	Identifies the Intel Fortran compiler
`__INTEL_COMPILER_BUILD_DATE=`*YYYYMMDD*	Identifies the Intel Fortran compiler build date
`_DLL=1`	Only if `/libs:dll`, `/MDs`, `/MD`, `/dll`, or `/LD` is specified, but not when `/libs:static` is specified
`_MT=1`	Only if `/threads` or `/MT` is specified
`_M_IX86=`*n*`00`	Only for systems based on IA-32 architecture; *n* is the number specified for /G (for example, `_M_IX86=700` for /G7)
`_M_IA64=64`*n*`00`	Only for systems based on IA-64 architecture; *n* is the number specified for /G (for example, `_M_IA64=64200` for /G1)
`_M_X64`	Only for systems based on Intel® 64 architecture. For use in conditionalizing applications for the Intel® 64 platform.
`_M_AMD64`	Only for systems based on Intel® 64 architecture. This symbol is set by default.
`_OPENMP=200805`	Valid when OpenMP processing has been requested (that is, when `/Qopenmp` is specified) Takes the form *YYYYMM* where *YYYY* is the year

	and *MM* is the month of the OpenMP Fortran specification supported. This symbol can be used in both fpp and the Fortran compiler conditional compilations.
_PGO_INSTRUMENT	Defined if /Qprof_gen is specified
_WIN32	Always defined
_WIN64	Only for systems based on Intel® 64 architecture and systems based on IA-64 architecture
_VF_VER=1100	Valid when Compaq* Visual Fortran-compatible compile commands (df or f90) are used

When using the non-native IA-64 architecture based compiler, platform-specific symbols are set for the target platform of the executable, not for the system in use.

Linux OS and Mac OS X:

The following information applies to Linux OS and Mac OS X systems.

Symbol Name	Default	Architecture (IA-32, Intel® 64, IA-64)	Description
__INTEL_COMPILER=*n*	On, *n*=1100	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 X only) __MACH__ (Mac OS X 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__ (Linux only) __ia64 (Linux only)		IA-64	Identifies the architecture for the target hardware for which programs are being compiled
__x86_64 __x86_64__		Intel® 64	Identifies the architecture for the target hardware for which

			programs are being compiled.
_OPENMP=*n*	*n*=200805	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 OS), On (Mac OS X)	All	Set if the code was requested to be compiled as position independent code. On Mac OS X, these symbols are always set.

Using Configuration Files and Response Files

Configuration Files and Response Files Overview

Configuration files and response files let you enter command-line options in a file. Both types of files provide the following benefits:

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

See these topics:

- Using Configuration Files
- Using Response Files

Using Configuration Files

Configuration files are automatically processed every time you run the compiler. You can insert any valid command-line options into the configuration file. The compiler processes options in the configuration file in the order in which they appear, followed by the command-line options that you specify when you invoke the compiler.

Note

Options in the configuration file are executed every 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 configuration file.

Sample Configuration Files

Examples that follow illustrate sample configuration files. The pound (#) character indicates that the rest of the line is a comment.

Linux* OS and Mac OS* X Example:

```
## 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
##
```

Windows* OS Example:

```
## Sample ifort.cfg file
## Define preprocessor macro MY_PROJECT
/DMY_PROJECT

## Set extended-length source lines.
/extend_source
##
## Set maximum floating-point significand precision.
/Qpc80
##

## Additional directories to be searched for include
## files, before the default.

/Ic:\project\include

## Use the static, multithreaded run-time library.

/MT
```

Using Response Files

You can use response files to:

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

Unlike configuration files, which are automatically processed every time you run the compiler, response files must be invoked as an option on the command line. If you create a response file but do not specify it on the command line, it will not be invoked.

Options specified in a response file are inserted in the command line at the point where the response file is invoked.

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

The syntax for using response files is:

ifort *@responsefile1* [*@responsefile2* ...]

Note

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

Debugging

Debugging Fortran Programs
Depending on your operating system and your architecture platform, several debuggers may be available to you.

You can use the debugger provided by your operating system. On Linux* OS and Mac OS* X, this debugger is gdb. On Windows* OS, the debugger is the Microsoft integrated debugger.

On Linux OS and Mac OS X systems, you can also use the Intel® Debugger to debug Intel® Fortran programs.

For more information on IDB, see the Intel Debugger online documentation.

Note

On Linux OS and Mac OS X systems, use of the IDB debugger is recommended.

Preparing Your Program for Debugging
This section describes preparing your program for debugging.

Preparing for Debugging using the Command Line

To prepare your program for debugging when using the command line (ifort command):

1. Correct any compilation and linker errors.
2. In a command window, (such as the Fortran command window available from the Intel Fortran program folder), compile and link the program with full debug information and no optimization:
   `ifort -g file.f90` (Linux OS and Mac OS X)
   `ifort /debug:full file.f90` (Windows OS)

On Linux OS and Mac OS X, specify the `-g` compiler option to create unoptimized code and provide the symbol table and traceback information needed for symbolic debugging. (The `-notraceback` option cancels the traceback information.)

On Windows OS, specify the `/debug:full` compiler option to produce full debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking.

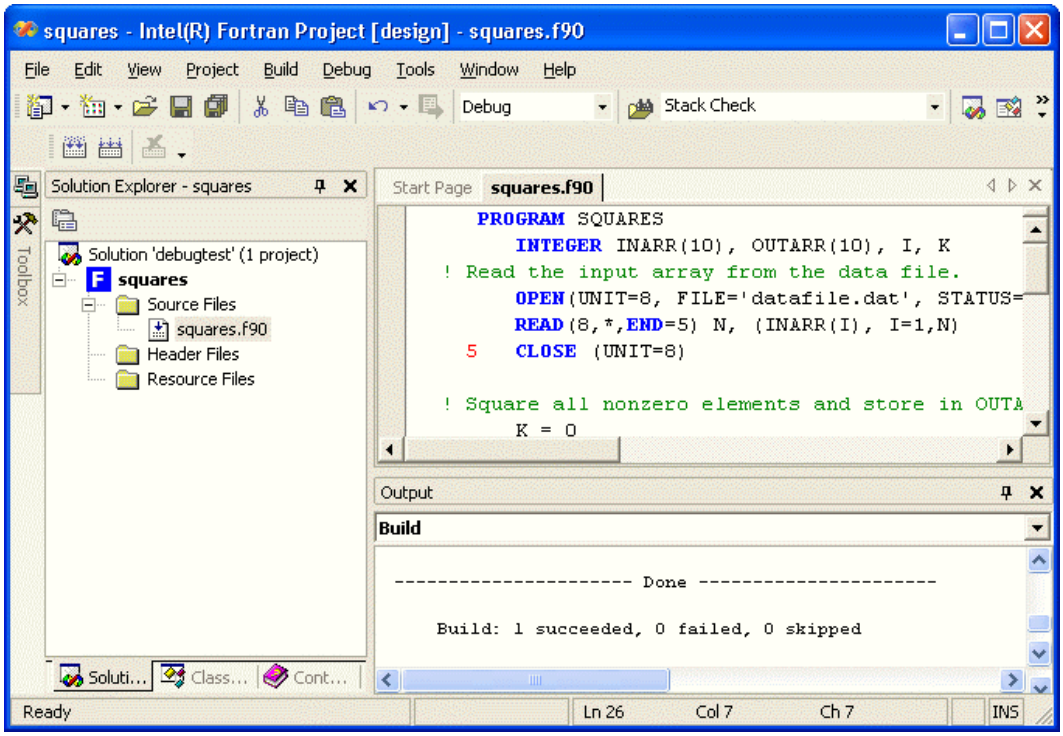
Preparing for Debugging using Microsoft Visual Studio*

The following applies to Windows* operating systems only.

To prepare your program for debugging when using the integrated development environment (IDE):

1. Start the IDE (select the appropriate version of **Microsoft Visual Studio** in the program folder).

2. Open the appropriate solution (using the **Solution** menu, either **Open Solution** or **Recent Projects**).
3. Open the Solution Explorer View.
4. To view the source file to be debugged, double-click on the file name. The screen resembles the following:



5. In the **Build** menu, select **Configuration Manager** and select the **Debug** configuration.
6. To check your project settings for compiling and linking, select the project name in the Solution Explorer. Now, in the **Project** menu, select **Properties**, then click the Fortran folder in the left pane. Similarly, to check the debug options set for your project (such as command arguments or working directory), click the Debugging folder in the Property Pages dialog box.
7. To build your application, select **Build>Build Solution**.
8. Eliminate any compiler diagnostic messages using the text editor to resolve problems detected in the source code and recompile if needed.
9. Set breakpoints in the source file and debug the program.

Locating Unaligned Data

Unaligned data can slow program execution. You should determine the cause of the unaligned data, fix the source code (if necessary), and recompile and relink the program.

If your program encounters unaligned data at run time, to make it easier to debug the program, you should recompile and relink with the `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) option to generate sufficient table information and debug unoptimized code.

For more information on data alignment, see the following:

Understanding Data Alignment

Setting Data Type and Alignment

Debugging a Program that Encounters a Signal or Exception
If your program encounters a signal (exception) at run time, you may want to recompile and relink with certain command-line options before debugging the cause. The following will make it easier to debug the program:

- Use the `-fpe`*n* (Linux OS and Mac OS X) or `/fpe:n` (Windows OS) option to control the handling of floating point exceptions.
- As with other debugging tasks, use the `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) compiler option to generate sufficient symbol table information and debug unoptimized code.

Debugging an Exception in the Microsoft Debugger

The following applies to Windows* operating systems.

You can request that the program always stop when a certain type of exception occurs. Certain exceptions are caught by default by the Intel Visual Fortran run-time library, so your program stops in the run-time library code. In most cases, you want the program to stop in your program's source code instead .

To change how an exception is handled in the Microsoft debugger:

1. In the **Debug** menu, select **Exceptions**.
2. View the displayed exceptions.
3. Select **Windows Exceptions**. Select each type of exception to be changed and change its handling using the radio buttons.
4. Start program execution using **Start** in the **Debug** menu.
5. When the exception occurs, you can now view the source line being executed, examine current variable values, execute the next instruction, and so on to help you better understand that part of your program.
6. After you locate the error and correct the program, consider whether you want to reset the appropriate type of exception to "Use Parent Setting" before you debug the program again.

For machine exceptions, you can use the just-in-time debugging feature to debug your programs as they run outside of the visual development environment. To do this, set the following items:

- In **Tools>Options**, select **Native** in the **Debugging Just-In Time** category.
- Set the FOR_IGNORE_EXCEPTIONS environment variable to TRUE.

Debugging and Optimizations
This topic describes the relationship between various command-line options that control debugging and optimizing.

Whenever you enable debugging with `-g` (Linux* OS and Mac OS* X) or `/debug:full` (Windows* OS), you disable optimizations. You can override this behavior by explicitly specifying compiler options for optimizations on the command line.

The following summarizes commonly used options for debugging and for optimization.

`-O0` (Linux OS and Mac OS X) or `/Od`	Disables optimizations so you can debug your program before any optimization is attempted. This is the default behavior when

(Windows* OS)	debugging. On Linux OS and Mac OS X, `-fno-omit-frame-pointer` is set if option -O0 (or -g) is specified. For more information, see the following topic: • `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS) compiler option
`-O1` or `/O1` `-O2` or `/O2` `-O3` or `/O3`	Specifies the code optimization level for applications. If you use any of these options, it is recommended that you use `-debug extended` when debugging. For more information, see the following topic: • `-O1, -O2, -O3` (Linux OS and Mac OS X) or `/O` (Windows OS) compiler option
`-g` or `/debug:full`	Generates symbolic debugging information and line numbers in the object code for use by the source-level debuggers. Turns off `-O2` (Linux OS and Mac OS X) or `/O2` (Windows) and makes `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS) the default. The exception to this is if `-O2, -O1` or `-O3` (Linux OS and Mac OS X) or `/O2, /O1` or `/O3` (Windows OS) is explicitly specified in the command line. For more information, see the following topic: • `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) compiler option
`-debug extended` (Linux OS and Mac OS X)	Specifies settings that enhance debugging. For more information, see the following topic: • `-debug extended` (Linux OS and Mac OS X)
`-fp` or `/Oy-` (IA-32 architecture only)	Disables the ebp register in optimizations and sets the ebp register to be used as the frame pointer. For more information, see the following topic: • `-fp` (Linux OS and Mac OS X) or `/Oy` (Windows OS) compiler option
`-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS)	Causes the compiler to generate extra information in the object file, which allows a symbolic stack traceback. For more information, see the following topic: • `-traceback` compiler option

Combining Optimization and Debugging

The compiler lets you generate code to support symbolic debugging when one of the `O1`, `O2`, or `O3` optimization options is specified on the command line along with `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS); this produces symbolic debug information in the object file.

Note that if you specify an `O1`, `O2`, or `O3` option with the -g or `/debug:full` option, some of the debugging information returned may be inaccurate as a side-effect of optimization. To counter this on Linux OS and Mac OS X, you should also specify the `-debug extended` option.

It is best to make your optimization and/or debugging choices explicit:

- If you need to debug your program excluding any optimization effect, use the `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS) option, which turns off all the optimizations.
- If you need to debug your program with optimizations enabled, then you can specify the `O1`, `O2`, or `O3` option on the command line along with `debug extended`.

Note

When no optimization level is specified, the `-g` or `/debug:full` option slows program execution; this is because this option turns on `-O0` or `/Od`, which causes the slowdown. However, if, for example, both `-O2` (Linux OS and Mac OS X) or `/O2` (Windows OS) and `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) are specified, the code should not experience much of a slowdown.

Refer to the table below for the summary of the effects of using the `-g` or `/debug:full` option with the optimization options.

These options	Produce these results
`-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS)	Debugging information produced, `-O0 or /Od` enabled (meaning optimizations are disabled). For Linux OS and Mac OS X, `-fp` is also enabled for compilations targeted for IA-32 architecture.
`-g -O1` (Linux OS and Mac OS X) or `/debug:full /O1` (Windows*)	Debugging information produced, O1 optimizations enabled.
`-g -O2` (Linux OS and Mac OS X) or `/debug:full /O2` (Windows OS)	Debugging information produced, O2 optimizations enabled.
`-g -O2` (Linux OS and Mac OS X) or `/debug:full /O2 /Oy-` (Windows OS)	Debugging information produced, O2 optimizations enabled; for Windows OS using IA-32 architecture, `/Oy` disabled.
`-g -O3 -fp` (Linux OS and Mac OS X) or `/debug:full /O3` (Windows OS)	Debugging information produced, O3 optimizations enabled; for Linux OS, `-fp` enabled for compilations targeted for IA-32 architecture.

Note

Even the use of `debug extended` with optimized programs may not allow you to examine all variables or to set breaks at all lines, due to code movement or removal during the optimization process

Debugging Multithreaded Programs

The debugging of multithreaded program discussed in this topic applies to both the OpenMP* Fortran API and the Intel Fortran parallel compiler directives. When a program uses parallel decomposition directives, you must take into consideration that the bug might be caused either by an incorrect program statement or it might be caused by an incorrect parallel decomposition directive. In either case, the program to be debugged can be executed by multiple threads simultaneously.

To determine the correctness of and debug multithreaded programs, you can use the following:

- For Linux OS and Mac OS X systems, Intel® Debugger (IDB) or GDB
- For Windows operating systems, Microsoft Visual Studio* Debugger
- Intel Fortran Compiler debugging options and methods; in particular, -d-lines (Linux OS and Mac OS X) or /d-lines (Windows OS)

Data and I/O

Data Representation

Data Representation Overview

Intel® Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte). For information on using nonnative big endian and VAX* floating-point formats, see Supported Native and Nonnative Numeric Formats.

The symbol :A in any figure specifies the address of the byte containing bit 0, which is the starting address of the represented data element.

The following table lists the intrinsic data types used by Intel® Fortran, the storage required, and valid ranges. For information on declaring Fortran intrinsic data types, see Type Declaration Statements. For example, the declaration INTEGER(4) is the same as INTEGER(KIND=4) and INTEGER*4.

Fortran Data Types and Storage

Data Type	Storage	Description
BYTE 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	Signed integer, either INTEGER(2), INTEGER(4), or INTEGER(8). The size is controlled by the `-integer-size` (Linux* OS and Mac OS* X) or `/integer-size` (Windows* OS) compiler option.

	INTEGER(8)	
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` (Linux OS and Mac OS X) or `/real-size` (Windows) compiler option.
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)	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` (Linux OS and Mac OS X) or `/real-size` (Windows OS) compiler option.
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 6.475175119438025110924438958227646552499 6Q-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` (Linux OS and Mac OS X) or `/integer-size` (Windows OS) compiler option.
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.

See these topics:

- Integer Data Representations
- Logical Data Representations
- Character Representation
- Hollerith Representation

Integer Data Representations

Integer Data Representations Overview

The Fortran numeric environment is flexible, which helps make Fortran a strong language for intensive numerical calculations. The Fortran standard purposely leaves the precision of

numeric quantities and the method of rounding numeric results unspecified. This allows Fortran to operate efficiently for diverse applications on diverse systems.

The effect of math computations on integers is straightforward:

- Integers of KIND=1 consist of a maximum positive integer (127), a minimum negative integer (-128), and all integers between them including zero.
- Integers of KIND=2 consist of a maximum positive integer (32,767), a minimum negative integer (-32,768) , and all integers between them including zero.
- Integers of KIND=4 consist of a maximum positive integer (2,147,483,647), a minimum negative integer (-2,147,483,648), and all integers between them including zero.
- Integers of KIND=8 consist of a maximum positive integer (9,223,372,036,854,775,807), a minimum negative integer (-9,223,372,036,854,775,808), and all integers between them including zero.

Operations on integers usually result in other integers within this range. Integer computations that produce values too large or too small to be represented in the desired KIND result in the loss of precision. One arithmetic rule to remember is that integer division results in truncation (for example, 8/3 evaluates to 2).

Integer data lengths can be 1, 2, 4, or 8 bytes in length.

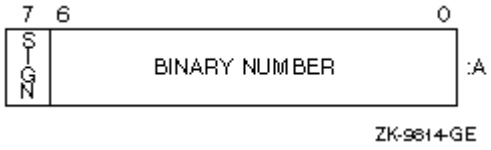
The default data size used for an INTEGER data declaration is INTEGER(4) (same as INTEGER(KIND=4), unless the `-integer-size 16` (Linux* OS and Mac OS* X) or `/integer-size:16` (Windows*OS) or the `-integer-size 64` (Linux OS and Mac OS X) or `/integer-size:64` (Windows OS) option was specified.

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

INTEGER(KIND=1) Representation

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

INTEGER(1) Data Representation



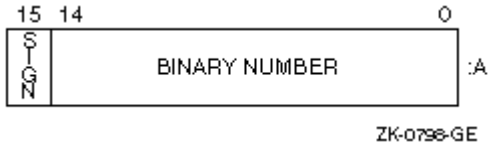
Integers are stored in a two's complement representation. For example:

```
+22 = 16(hex)
 -7 = F9(hex)
```

INTEGER(KIND=2) Representation

INTEGER(2) values range from -32,768 to 32,767 and are stored in 2 contiguous bytes, as shown below:

INTEGER(2) Data Representation



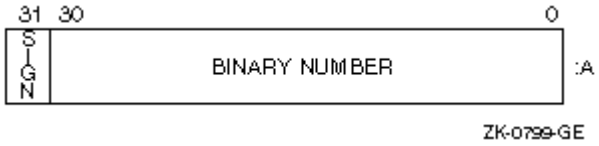
Integers are stored in a two's complement representation. For example:

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

INTEGER(KIND=4) Representation

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

INTEGER(4) Data Representation

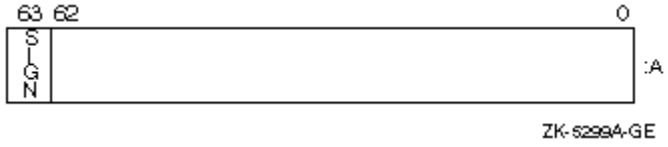


Integers are stored in a two's complement representation.

INTEGER(KIND=8) Representation

INTEGER(8) values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and are stored in 8 contiguous bytes, as shown below.

INTEGER(8) Data Representation



Integers are stored in a two's complement representation.

Logical Data Representations

Logical data 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 `-integer-size 16` or `-integer-size 64` (Linux OS and Mac OS X) or `/integer-size:16` or `/integer-size:64` (Windows OS) was specified.

To improve performance on systems using Intel® 64 architecture and IA-64 architecture, use LOGICAL(4) (or LOGICAL(8)) rather than LOGICAL(2) or LOGICAL(1). On systems using IA-32 architecture, use LOGICAL(4) rather than LOGICAL(8), LOGICAL(2), or LOGICAL(1).

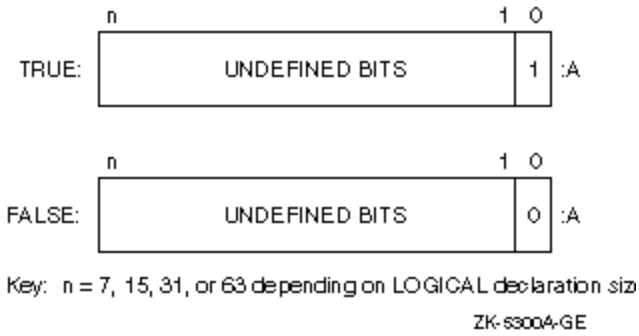
LOGICAL(KIND=1) values are stored in 1 byte. In addition to having logical values .TRUE. and .FALSE., LOGICAL(1) data can also have values in the range -128 to 127. Logical variables can also be interpreted as integer data.

In addition to LOGICAL(1), logical values can also be stored in 2 (LOGICAL(2)), 4 (LOGICAL(4)), or 8 (LOGICAL(8)) contiguous bytes, starting on an arbitrary byte boundary.

If the `-fpscomp nological` (Linux OS and Mac OS X) or `/fpscomp:nological` (Windows OS) compiler option is set (the default), the low-order bit determines whether the logical value is true or false. Specify `logical` instead of `nological` 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 `nological` is specified) appears below.

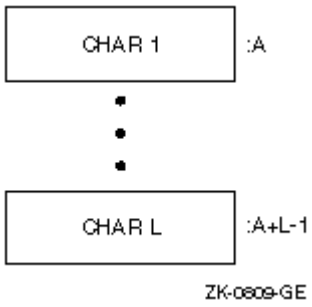
LOGICAL(1), LOGICAL(2), LOGICAL(4), and LOGICAL(8) Data Representations



Character Representation

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

CHARACTER Data Representation

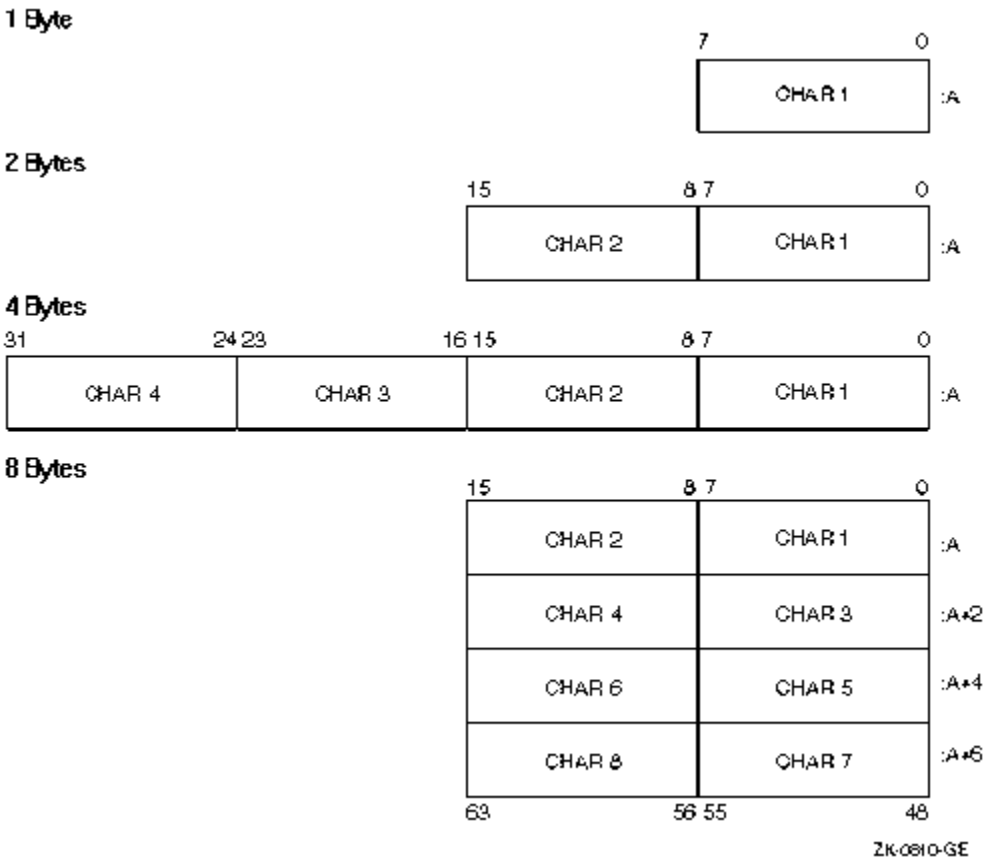


A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. For Windows* OS, the length L of a string is in the range 1 through 2,147,483,647 (2**31-1) . For Linux* OS, the length L of a string is in the range 1 through 2,147,483,647 (2**31-1) for systems based on IA-32 architecture and in the range 1 through 9,223,372,036,854,775,807 (2**63-1) for systems based on Intel® 64 architecture and systems based on IA-64 architecture.

Hollerith Representation

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

Hollerith Data Representation



Converting Unformatted 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.

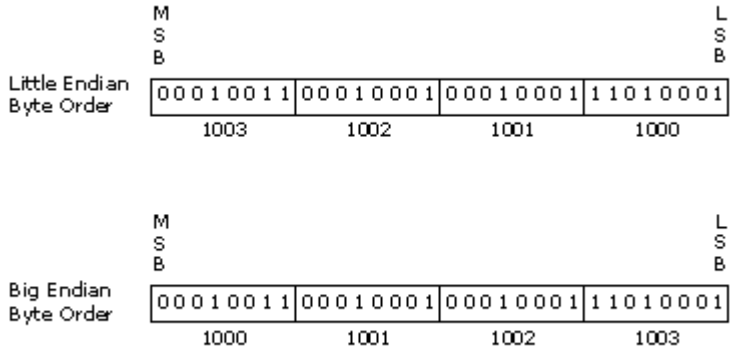
Big endian storage occurs when:

- 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.

Intel® Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

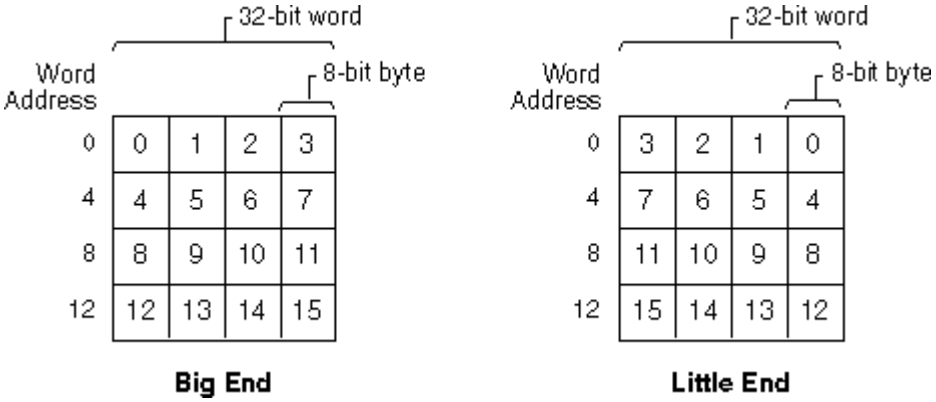
The following figures show the difference between the two byte-ordering schemes:

Little and Big Endian Storage of an INTEGER Value



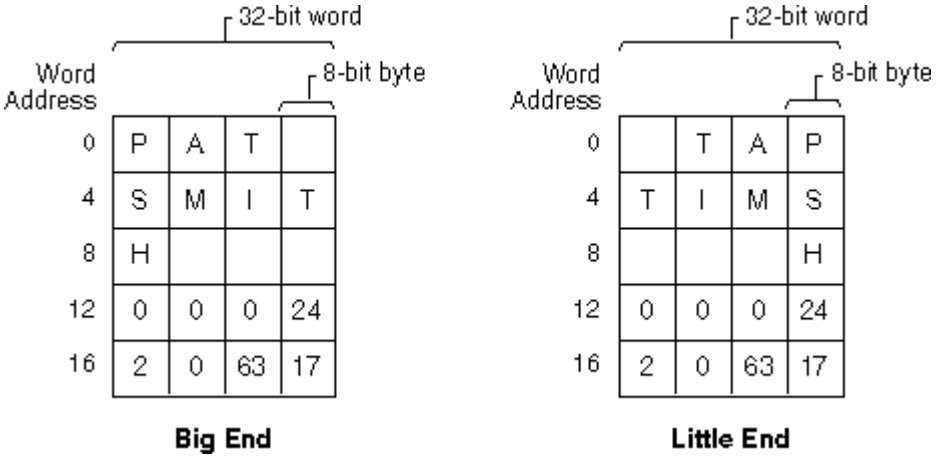
The following figure illustrates the difference between the two conventions for the case of addressing bytes within words.

Byte Order Within Words: (a) Big Endian, (b) Little Endian



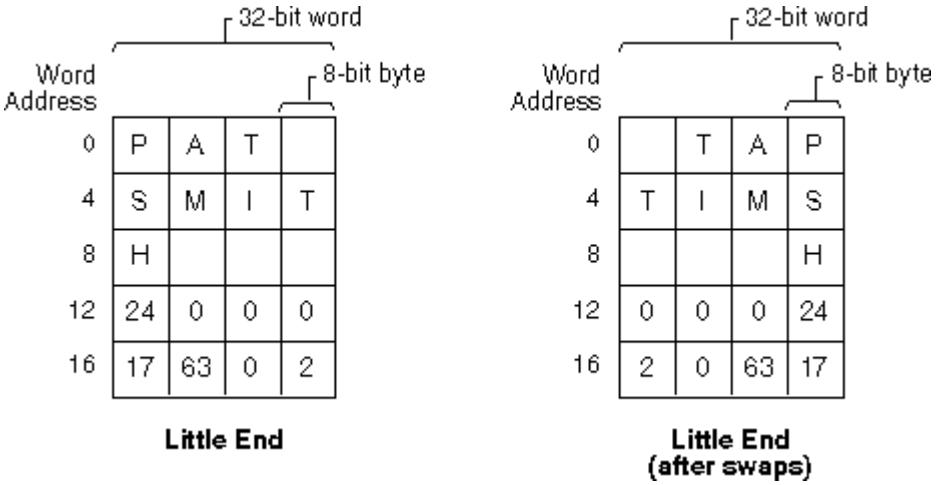
Data types stored as subcomponents (bytes stored in words) end up in different locations within corresponding words of the two conventions. The following figure illustrates the difference between the representation of several data types in the two conventions. Letters represent 8-bit character data, while numbers represent the 8-bit partial contribution to 32-bit integer data.

Character and Integer Data in Words: (a) Big Endian, (b) Little Endian



If you serially transfer bytes now from the big endian words to the little endian words (BE byte 0 to LE byte 0, BE byte 1 to LE byte 1, ...), the left half of the figure shows how the data ends up in the little endian words. Note that data of size one byte (characters in this case) is ordered correctly, but that integer data no longer correctly represents the original binary values. The right half of the figure shows that you need to swap bytes around the middle of the word to reconstitute the correct 32-bit integer values. After swapping bytes, the two preceding figures are identical.

Data Sent from Big to Little: (a) After Transfer, (b) After Byte Swaps



You can generalize the previous example to include floating-point data types and to include multiple-word data types.

Moving unformatted data files between big endian and little endian computers requires that the data be converted.

Intel Fortran provides the capability for programs to read and write unformatted data (originally written using unformatted I/O statements) in several nonnative floating-point formats and in big endian INTEGER or floating-point format. Supported nonnative floating-point formats include Compaq* VAX* little endian floating-point formats supported by Digital* FORTRAN for OpenVMS* VAX Systems, standard IEEE big endian floating-point format found on most Sun Microsystems* systems and IBM RISC* System/6000 systems, IBM floating-point formats (associated with the IBM's System/370 and similar systems), and CRAY* floating-point formats.

Converting unformatted data instead of formatted data is generally faster and is less likely to lose precision of floating-point numbers.

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 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 Formats, Keywords, and Supported Data Types

Keyword	Description
BIG_ENDIAN	Big endian integer data of the appropriate size (one, two, four, or eight bytes) and big endian IEEE floating-point formats for REAL and

COMPLEX single- 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 size (one, two, four, or eight bytes) and big endian CRAY proprietary floating-point format for REAL and COMPLEX single- and double-precision numbers.

FDX — Little endian integer data of the appropriate size (one, two, four, or eight bytes) and the following little endian proprietary floating-point formats:

- VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4)
- VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8)
- IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)

FGX — Little endian integer data of the appropriate size (one, two, four, or eight bytes) and 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)

For additional information on supported ranges for these data types, see Native IEEE Floating-Point Representations.

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

VAXD — Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats:

- VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4)
- VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8)
- VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)

VAXG — Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats:

- VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4)
- VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8)
- VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message is displayed.

See also:

Environment Variable F_UFMTENDIAN Method

Porting 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` (Linux OS and Mac OS X) or `/assume:byterecl` (Windows) compiler option. Alternatively, for Windows OS, you can specify **Use Bytes as RECL=Unit for Unformatted Files** in the **Data Options** property page.

  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.

Specifying the Data Format

Methods of Specifying the Data Format

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

- 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 the application is executed. 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 appropriate compiler option, which affects all unit numbers that use unformatted data specified by the program. Use the `-convert` *`keyword`* option for Linux* OS and Mac OS* X or, for Windows* OS, the command-line `/convert:`*`keyword`* option or the IDE equivalent.

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). Note: this environment variable is checked only when the application starts executing.
4. Check the OPEN statement CONVERT specifier.
5. Check whether an OPTIONS statement with a `CONVERT=`*`keyword`* qualifier was present when the program was compiled.
6. Check whether the compiler option `-convert` *`keyword`* (Linux OS and Mac OS X) or `/convert:`*`keyword`* (Windows OS) was present when the program was compiled.

Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method

You can use this method to specify a non-native 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 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 a program, the following command sets the appropriate environment variables before running the program:

Linux:

```
setenv FORT_CONVERT.DAT BIG_ENDIAN
```

Windows:

```
set FORT_CONVERT.DAT=BIG_ENDIAN
```

The FORT_CONVERT*n* method takes precedence over this method. When the appropriate environment variable is set when you open the file, the FORT_CONVERT.*ext* or FORT_CONVERT_*ext* environment variable is used if a FORT_CONVERT*n* environment variable is not set for the unit number.

The FORT_CONVERT*n* and the FORT_CONVERT.*ext* or FORT_CONVERT_*ext* environment variable methods take precedence over the other methods. For instance, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

You can set the appropriate environment variable using the format FORT_CONVERT.*ext* or FORT_CONVERT_*ext*. If you also use Intel Fortran on Linux* systems, consider using the FORT_CONVERT_*ext* form, because a dot (.) cannot be used for environment variable names on certain Linux command shells. If you do define both FORT_CONVERT.*ext* and FORT_CONVERT_*ext* for the same extension (*ext*), the file defined by FORT_CONVERT.*ext* is used.

On Windows systems, the file name extension (suffix) is not case-sensitive. The extension must be part of the file name (not the directory path).

Environment Variable FORT_CONVERT*n* Method

You can use this method to specify a non-native 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 a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

For example, assume you have a previously compiled program that reads numeric data from unit 28 and writes it to unit 29 using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from unit 28 and write that data in native little endian format to unit 29. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format when read from unit 28, and then written without conversion in native little endian IEEE format to unit 29.

Without requiring source code modification or recompilation of this program, the following command sequence sets the appropriate environment variables before running a program.

Linux:

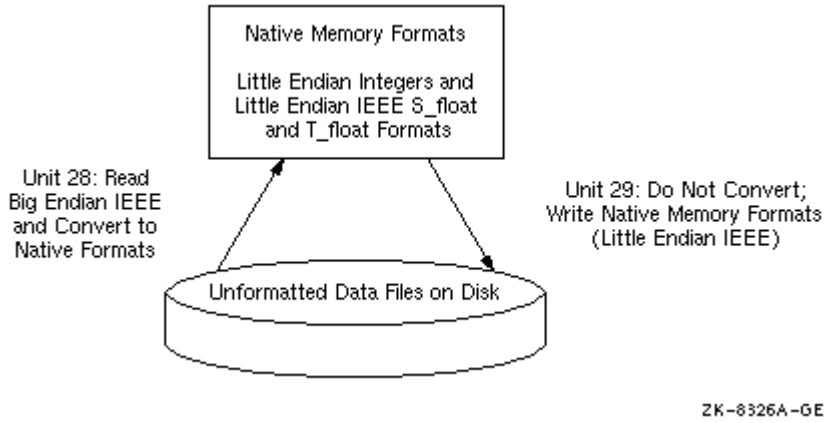
```
setenv FORT_CONVERT28 BIG_ENDIAN
setenv FORT_CONVERT29 NATIVE
```

Windows:

```
set FORT_CONVERT28=BIG_ENDIAN
set FORT_CONVERT29=NATIVE
```

The following figure shows the data formats used on disk and in memory when the program is run after the environment variables are set.

Sample Unformatted File Conversion



This method takes precedence over other methods.

Environment Variable F_UFMTENDIAN Method

This little-endian-big-endian conversion feature is intended for Fortran unformatted input/output operations. It enables the development and processing of files with little-endian and big-endian data organization.

The F_UFMTENDIAN environment variable is processed once at the beginning of program execution. Whatever it specifies for specific units or for all units continues for the rest of the execution.

Specify the numbers of the units to be used for conversion purposes by setting F_UFMTENDIAN. Then, the READ/WRITE statements that use these unit numbers will perform relevant conversions. Other READ/WRITE statements will work in the usual way.

General Syntax for F_UFMTENDIAN

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

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

where:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- MODE defines current format of data, represented in the files; it can be omitted.
  The keyword little means that the data has little endian format and will not be converted. This is the default.
  The keyword big means that the data has big endian format and will be converted.
- EXCEPTION is intended to define the list of exclusions for MODE. EXCEPTION keyword (little or big) defines data format in the files that are connected to the units from the EXCEPTION list. This value overrides MODE value for the units listed. The EXCEPTION keyword and the colon can be omitted. The default when the keyword is omitted is big.
- Each list member U is a simple unit number or a number of units. The number of list members is limited to 64.
- decimal is a non-negative decimal number less than 2^{32}.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Error messages may be issued during the little-endian-to-big-endian conversion. They are all fatal.

On Linux*systems, the command line for the variable setting in the shell is:

```
Sh: export F_UFMTENDIAN=MODE;EXCEPTION
```

Note

The environment variable value should be enclosed in quotes if the semicolon is present.

The environment variable can also have the following syntax:

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

Examples

1. F_UFMTENDIAN=big

   All input/output operations perform conversion from big-endian to little-endian on READ and from little-endian to big-endian on WRITE.

2. F_UFMTENDIAN="little;big:10,20"
   or F_UFMTENDIAN=big:10,20
   or F_UFMTENDIAN=10,20

   The input/output operations perform big-endian to little endian conversion only on unit numbers 10 and 20.

3. F_UFMTENDIAN="big;little:8"

   No conversion operation occurs on unit number 8. On all other units, the input/output operations perform big-endian to little-endian conversion.

4. F_UFMTENDIAN=10-20

   The input/output operations perform big-endian to little-endian conversion on units 10, 11, 12 &ldots; 19, 20.

5. Assume you set `F_UFMTENDIAN=10,100` and run the following program.

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789

C  prepare a little endian representation of data

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

C  prepare a 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 non-native 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` (Linux OS and Mac OS X) or `/convert:`*keyword* (Windows OS) method, but has a lower precedence than the environment variable methods.

For example, the following source code shows how the OPEN statement would be coded to read unformatted VAXD numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20 (the absence of the CONVERT keyword or environment variables `FORT_CONVERT20`, `FORT_CONVERT.dat`, or `FORT_CONVERT_dat` indicates native little endian data for unit 20):

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
  .
  .
  .
OPEN (FILE='graph3_t.dat', FORM='UNFORMATTED', UNIT=20)
```

A hard-coded OPEN statement CONVERT keyword value cannot be changed after compile time. However, to allow selection of a particular format at run time, equate the CONVERT keyword to a variable and provide the user with a menu that allows selection of the appropriate format (menu choice sets the variable) before the OPEN occurs.

You can also select a particular format at run time for a unit number by using one of the environment variable methods (FORT_CONVERTn, FORT_CONVERT.ext or 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 OPEN statement `CONVERT` *`keyword`* method.

You specify the numeric format at compile time and must compile all routines under the same OPTIONS statement /CONVERT=*keyword* qualifier. You could use one source program and compile it using different ifort commands to create multiple executable programs that each read a certain format.

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

This method takes precedence over the `ifort -convert keyword` (Linux OS and Mac OS X) or `/convert:`*`keyword`* (Windows OS) compiler option method.

You can use OPTIONS statements to specify the appropriate floating-point formats (in memory and in unformatted files) instead of using the corresponding `ifort` command qualifiers. For example, to use VAX F_floating, G_floating, and X_floating as the unformatted file format, specify the following OPTIONS statement:

```
OPTIONS /CONVERT=VAXG
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another format, unless you use it in combination with one of the environment variable methods or the OPEN statement CONVERT keyword method to specify a different format for a particular unit number.

For more information, see the OPTIONS statement.

Compiler Option -convert or /convert Method

You can only specify one numeric format for all unformatted file unit numbers using the compiler option `-convert` (Linux OS and Mac OS X) or `/convert` (Windows OS) 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` (Linux OS and Mac OS X) or `/convert:keyword` (Windows OS) compiler option. You can use the same source program and compile it using different `ifort` commands (or the equivalent in the IDE) to create multiple executable programs that each read a certain format.

If you specify other methods, 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` (Linux OS and Mac OS X) or `/convert:keyword` (Windows OS) 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_floating, T_floating, and X_floating little endian IEEE* floating-point format). The created file, `vconvert.exe`, can then be run:

Linux OS and Mac OS X:

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

Windows OS:

```
ifort file.for /convert:vaxd /link /out:vconvert.exe
```

Because this method affects all unformatted file unit numbers, you cannot read data in one format and write it in another file format using the `-convert keyword` (Linux OS and Mac OS X) or `/convert:keyword` (Windows OS) 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

Fortran I/O

Devices and Files Overview

In Fortran's I/O system, data is stored and transferred among files. All I/O data sources and destinations are considered files.

Devices such as the screen, keyboard and printer are *external files*, as are data files stored on a device such as a disk.

Variables in memory can also act as a file on a disk, and are typically used to convert ASCII representations of numbers to binary form. When variables are used in this way, they are called *internal files*.

Topics covered in this section include the following:

- Logical Devices
- Types and Forms of I/O Statements
- File Organization, File Access and File Structure
- Internal Files and Scratch Files
- File Records, including Record Types, Length, Access, and Transfer
- Using I/O Statements such as OPEN, INQUIRE, and CLOSE

For information on techniques you can use to improve I/O performance, see Improving I/O Performance.

Logical Devices

Every file, internal or external, is associated with a logical device. 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 either a number you assign with the OPEN statement, a number preconnected as a unit specifier to a device, or 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 (UNIT=5), unless the FOR_READ environment variable is defined.
- WRITE statements that contain only an I/O list and format specifier, which write to standard output, unless the FOR_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 predefined parameters FOR_K_PRINT_UNITNO, FOR_K_TYPE_UNITNO, FOR_K_ACCEPT_UNITNO, and FOR_K_READ_UNITNO may not be in that range. For more information, see the *Language Reference*.)

The following example connects the external file `UNDAMP.DAT` to unit 10 and writes to it:

```
OPEN (UNIT = 10, FILE = 'UNDAMP.DAT')
WRITE (10, '(A18,\)') ' Undamped Motion:'
```

The asterisk (*) unit specifier specifies the keyboard when reading and the screen when writing. The following example uses the asterisk specifier to write to the screen:

```
WRITE (*, '(1X, A30,\)') ' Write this to the screen.'
```

Intel Fortran has four units preconnected to external files (devices), as shown in the following table.

External Unit Specifier	Environment Variable	Description
Asterisk (*)	None	Always represents the keyboard and screen (unless the appropriate environment variable is defined, such as FOR_READ).
0	FORT0	Initially represents the screen (unless FORT0 is explicitly defined)
5	FORT5	Initially represents the keyboard (unless FORT5 is explicitly defined)
6	FORT6	Initially represents the screen (unless FORT6 is explicitly defined)

The asterisk (*) specifier is the only unit specifier that cannot be reconnected to another file, 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.

Intel® Fortran does not support buffering to `stdout` or `stdin`. All I/O to units * and 6 use line buffering. Therefore, C and Fortran output to stdout should work well as long as the C code is not performing buffering. In the case of buffering, the C code will have to flush the buffers with each write. For more information on `stdout` and `stdin`, see Assigning Files to Logical Units.

You can change these preconnected files by doing one of the following:

- Using an OPEN statement to open unit 5, 6, or 0. When you explicitly OPEN a file for unit 5, 6, or 0, the OPEN statement keywords specify the file-related information to be used instead of the preconnected standard I/O file.
- Setting the appropriate environment variable (`FORT`*n*) to redirect I/O to an external file.

To redirect input or output from the standard preconnected files at run time, you can set the appropriate environment variable or use the appropriate shell redirection character in a pipe (such as `>` or `<`).

When you omit the file name in the OPEN statement or use an implicit OPEN, you can define the environment variable `FORT`*n* to specify the file name for a particular unit number *n*. An exception to this is when the compiler option `-fpscomp filesfromcmd` (Linux* OS and Mac OS* X) or `/fpscomp:filesfromcmd` (Windows) 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
```

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 has 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 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 of the variable, before the first record.

With internal files, you can use the formatting capabilities of the I/O system to convert values between external character representations and Fortran internal memory representations. 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 the record at the current record position in a relative file as deleted (direct access only).
ENDFILE	Writes an end-of-file marker after the current record (sequential access only).
FIND	Changes the record position in a direct access file. Provided for compatibility with compilers older than FORTRAN-77.
REWIND	Sets the record position to the beginning of the file (sequential access only).
Record input	

READ	Transfers data from an external file record or an internal file to internal storage.
ACCEPT	Reads input from `stdin`. Unlike READ, ACCEPT only provides formatted sequential input and does not specify a unit number.

Record output

WRITE	Transfers data from internal storage to an external file record or to an internal file.
REWRITE	Transfers data from internal storage to an external file record at the current record position (direct access relative files only).
TYPE	Writes record output to `stdout`.
PRINT	Transfers data from internal storage to `stdout`. Unlike WRITE, PRINT only provides formatted sequential output and does not specify a unit number.
FLUSH	Flushes the contents of an external unit's buffer to its associated file.

In addition to the READ, WRITE, REWRITE, TYPE, and PRINT statements, other I/O record-related statements are limited to a specific file organization. For instance:

- The DELETE statement only applies to relative files. (Detecting deleted records is only available if the `-vms` option was specified when the program was compiled.)
- The BACKSPACE statement only applies to sequential files open for sequential access.
- The REWIND statement only applies to sequential files open for sequential access and to direct access files.
- The ENDFILE statement only applies to certain types of sequential files open for sequential access and to direct access files.

The file-related statements (OPEN, INQUIRE, and CLOSE) apply to any relative or sequential file.

Forms of I/O Statements

Each type of record I/O statement can be coded in a variety of forms. The form you select depends on the nature of your data and how you want it treated. When opening a file, specify the form using the FORM specifier.

The following are the forms of I/O statements:

- *Formatted I/O statements* contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.
- *List-directed* and *namelist I/O statements* are similar to formatted statements in function. However, they use different mechanisms to control the translation of data: formatted I/O statements use explicit format specifiers, and list-directed and namelist I/O statements use data types.

- *Unformatted I/O statements* do not contain format specifiers and therefore do not translate the data being transferred (important when writing data that will be read later).

Formatted, list-directed, and namelist I/O forms require translation of data from internal (binary) form within a program to external (readable character) form in the records. Consider using unformatted I/O for the following reasons:

- Unformatted data avoids the translation process, so I/O tends to be faster.
- Unformatted data avoids the loss of precision in floating-point numbers when the output data will subsequently be used as input data.
- Unformatted data conserves file storage space (stored in binary form).

To write data to a file using formatted, list-directed, or namelist I/O statements, specify FORM= 'FORMATTED' when opening the file. To write data to a file using unformatted I/O statements, specify FORM= 'UNFORMATTED' when opening the file.

Data written using formatted, list-directed, or namelist I/O statements is referred to as formatted data; data written using unformatted I/O statements is referred to as unformatted data.

When reading data from a file, you should use the same I/O statement form that was used to write the data to the file. For instance, if data was written to a file with a formatted I/O statement, you should read data from that file with a formatted I/O statement.

Although I/O statement form is usually the same for reading and writing data in a file, a program can read a file containing unformatted data (using unformatted input) and write it to a separate file containing formatted data (using formatted output). Similarly, a program can read a file containing formatted data and write it to a different file containing unformatted data.

You can access records in any sequential or relative file using sequential access. For relative files and certain (fixed-length) sequential files, you can also access records using direct access.

The table below shows 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
External file, stream access	

Formatted	READ, WRITE
List-directed	READ, WRITE
Namelist	READ, WRITE
Unformatted	READ, WRITE
Internal file	
Formatted	READ, WRITE
List-directed	READ, WRITE
Unformatted	None

Note

You can use the REWRITE statement only for relative files, using direct access.

Assigning Files to Logical Units

Most I/O operations involve a disk file, keyboard, or screen display. Other devices can also be used:

- Sockets can be read from or written to if a USEROPEN routine (usually written in C) is used to open the socket.
- Pipes opened for read and write access block (wait until data is available) if you issue a READ to an empty pipe.
- Pipes opened for read-only access return EOF if you issue a READ to an empty pipe.

You can access the terminal screen or keyboard by using preconnected files listed in Logical Devices.

You can choose to assign files to logical units by using one of the following methods:

- Using default values, such as a preconnected unit
- Supplying a file name (and possibly a directory) in an OPEN statement
- Using environment variables

Using Default Values

In the following example, the PRINT statement is associated with a preconnected unit (`stdout`) by default.

```
PRINT *,100
```

The READ statement associates the logical unit 7 with the file `fort.7` (because the FILE specifier was omitted) by default:

```
OPEN (UNIT=7,STATUS='NEW')
READ (7,100)
```

Supplying a File Name in an OPEN Statement

The FILE specifier in an OPEN statement typically specifies only a file name (such as `filnam`) or contains both a directory and file name (such as `/usr/proj/filnam`).

For example:

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

The DEFAULTFILE specifier in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).

Implied OPEN

Performing an implied OPEN means that the FILE and DEFAULTFILE specifier values are not specified and an environment variable is used, if present. Thus, if you used an implied OPEN, or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Using Environment Variables

You can use shell commands to set the appropriate environment variable to a value that indicates a directory (if needed) and a file name to associate a unit with an external file.

Intel Fortran recognizes environment variables for each logical I/O unit number in the form of `FORT`*`n`*, where *`n`* is the logical I/O unit number. If a file name is not specified in the OPEN statement and the corresponding `FORT`*`n`* environment variable is not set for that unit number, Intel Fortran generates a file name in the form `fort.`*`n`* , where *`n`* is the logical unit number.

Implied Intel Fortran Logical Unit Numbers

The ACCEPT, PRINT, and TYPE statements, and the use of an asterisk (*) in place of a unit number in READ and WRITE statements, do not include an explicit logical unit number.

Each of these Fortran statements uses an implicit internal logical unit number and environment variable. Each environment variable is in turn associated by default with one of the Fortran file names that are associated with standard I/O files. The table below shows these relationships:

Intel Fortran statement	Environment variable	Standard I/O file name
READ (*,f) iolist	FOR_READ	stdin
READ f,iolist	FOR_READ	stdin
ACCEPT f,iolist	FOR_ACCEPT	stdin
WRITE (*,f) iolist	FOR_PRINT	stdout
PRINT f,iolist	FOR_PRINT	stdout
TYPE f,iolist	FOR_TYPE	stdout
WRITE(0,f) iolist	FORT0	stderr
READ(5,f) iolist	FORT5	stdin

WRITE(6,f) iolist	FORT6	stdout

You can change the file associated with these Intel Fortran environment variables, as you would any other environment variable, by means of the environment variable assignment command. For example:

```
setenv FOR_READ /usr/users/smith/test.dat
```

After executing the preceding command, the environment variable for the READ statement using an asterisk refers to file `test.dat` in the specified directory.

Note

The association between the logical unit number and the physical file can occur at run-time. Instead of changing the logical unit numbers specified in the source program, you can change this association at run time to match the needs of the program and the available resources. For example, before running the program, a script file can set the appropriate environment variable or allow the terminal user to type a directory path, file name, or both.

File Organization

File organization refers to the way records are physically arranged on a storage device. This topic describes the two main types of file organization.

Related topics describe the following:

- *Record type* refers to whether records in a file are all the same length, are of varying length, or use other conventions to define where one record ends and another begins. For more information on record types, see Record Types.
- *Record access* refers to the method used to read records from or write records to a file, regardless of its organization. The way a file is organized does not necessarily imply the way in which the records within that file will be accessed. For more information on record access, see File Access and File Structure and Record Access.

Types of File Organization

Fortran supports two types of file organizations:

- Sequential
- Relative

The organization of a file is specified by means of the ORGANIZATION keyword in the OPEN statement.

The default file organization is always ORGANIZATION= 'SEQUENTIAL' for an OPEN statement.

You can store sequential files on magnetic tape or disk devices, and can use other peripheral devices, such as terminals, pipes, and line printers as sequential files.

You must store relative files on a disk device.

Sequential File Organization

A sequentially organized file consists of records arranged in the sequence in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, and so on). As a result, records can be added only at the end of the file. Attempting to add records at some place other than the end of the file will result in the file begin truncated at the end of the record just written.

Sequential files are usually read sequentially, starting with the first record in the file. Sequential files with a fixed-length record type that are stored on disk can also be accessed by relative record number (direct access).

Relative File Organization

Within a relative file are numbered positions, called *cells*. These cells are of fixed equal length and are consecutively numbered from 1 to *n*, where 1 is the first cell, and *n* is the last available cell in the file. Each cell either contains a single record or is empty. Records in a relative file are accessed according to cell number. A cell number is a record's relative record number; its location relative to the beginning of the file. By specifying relative record numbers, you can directly retrieve, add, or delete records regardless of their locations. (Detecting deleted records is only available if you specified the `-vms` (Linux OS and Mac OS X) or `/vms` (Windows OS) 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 supports two other types of files:

- 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).

File Access and File Structure

Fortran supports three methods of *file access*:

- Sequential
- Direct
- Stream

Fortran supports three kinds of *file structure*:

- Formatted
- Unformatted
- Binary

Sequential-access and direct-access files can have any of the three file structures. Stream-access files can have a file structure of formatted or unformatted.

Choosing a File Access and File Structure

Each kind of file has advantages and the best choice depends on the application you are developing:

- Formatted Files

  You create a formatted file by opening it with the FORM='FORMATTED' option, or by omitting the FORM parameter when creating a sequential file. The records of a formatted file are stored as ASCII characters; numbers that would otherwise be stored in binary form are converted to ASCII format. Each record ends with the ASCII carriage return (CR) and/or line feed (LF) characters.

  If you need to view a data file's contents, use a formatted file. You can load a formatted file into a text editor and read its contents directly, that is, the numbers

would look like numbers and the strings like character strings, whereas an unformatted or binary file looks like a set of hexadecimal characters.

- Unformatted Files

  You create an unformatted file by opening it with the FORM='UNFORMATTED' option, or by omitting the FORM parameter when creating a direct-access file. An unformatted file is a series of records composed of physical blocks. Each record contains a sequence of values stored in a representation that is close to that used in program memory. Little conversion is required during input/output.

  The lack of formatting makes these files quicker to access and more compact than files that store the same information in a formatted form. However, if the files contain numbers, you will not be able to read them with a text editor.

- Binary Files

  You create a binary file by specifying FORM='BINARY'. Binary files are similar to unformatted files, except binary files have no internal record format associated with them.

- Sequential-Access Files

  Data in sequential files must be accessed in order, one record after the other (unless you change your position in the file with the REWIND or BACKSPACE statements). Some methods of I/O are possible only with sequential files, including nonadvancing I/O, list-directed I/O, and namelist I/O. Internal files also must be sequential files. You must use sequential access for files associated with sequential devices.

  A sequential device is a physical storage device that does not allow explicit motion (other than reading or writing). The keyboard, screen, and printer are all sequential devices.

- Direct-Access Files

  Data in direct-access files can be read or written to in any order. Records are numbered sequentially, starting with record number 1. All records have the length specified by the RECL option in the OPEN statement. Data in direct files is accessed by specifying the record you want within the file. If you need random access I/O, use direct-access files. A common example of a random-access application is a database.

- Stream-Access Files
  Stream access I/O is a method of accessing a file without reference to a record structure. With stream access, a file is seen as a continuous sequence of bytes and is addressed by a positive integer starting from 1.
  To enable stream access, specify ACCESS='STREAM' in the OPEN statement for the file. You can use the STREAM= specifier in the INQUIRE statement to determine if STREAM is listed in the set of possible access methods for the file. A value of YES, NO, or UNKNOWN is returned.
  A file enabled for stream access is positioned by file storage units (normally bytes) starting at position 1. To determine the current position, use the POS= specifier in an

INQUIRE statement for the unit. You can indicate a required position in a read or write statement with a POS= specifier.
Stream access can be either formatted or unformatted.
When connected for formatted stream access, an external file has the following characteristics:

- The first file storage unit in the file is at position 1. The relationship between positions of successive file storage units is processor dependent; not all positive integers need to correspond to valid positions.
- Some file storage units may contain record markers; this imposes a record structure on the file in addition to its stream structure. If there is no record marker at the end of the file, the final record is incomplete. Writing an empty record with no record marker has no effect.

When connected for unformatted stream access, an external file has the following characteristics:

- The first file storage unit in the file is at position 1. The position of each subsequent file storage unit is one greater than that of the preceding file storage unit.
- If it is possible to position the file, the file storage units do not need to be read or written in order of their position. For example, you may be able to write the file storage unit at position 2, even though the file storage unit at position 1 has not been written.
- Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and a READ statement for this connection is allowed.
- You cannot use BACKSPACE in an unformatted stream.

File Records

Files may be composed of records. Each record is one entry in the file. It can be a line from a terminal or a logical record on a magnetic tape or disk file. All records within one file are of the same type.

In Fortran, the number of bytes transferred to a record must be less than or equal to the record length. One record is transferred for each unformatted READ or WRITE statement. A formatted READ or WRITE statement can transfer more than one record using the slash (***/***) edit descriptor.

For binary files, a single READ or WRITE statement reads or writes as many records as needed to accommodate the number of bytes being transferred. On output, incomplete formatted records are padded with spaces. Incomplete unformatted and binary records are padded with undefined bytes (zeros).

Record Types

An I/O record is a collection of data items, called fields, that are logically related and are processed as a unit. The record type refers to the convention for storing fields in records.

The record type of the data within a file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

A number of record types are available, as shown in the following table. The table also lists the record overhead. This refers to bytes associated with each record that are used internally by the file system and are not available when a record is read or written. Knowing the record overhead helps when estimating the storage requirements for an application. Although the overhead bytes exist on the storage media, do not include them when specifying the record length with the RECL specifier in an OPEN statement.

Record Type	Available File Organizations and Portability Considerations	Record Overhead
Fixed-length	Relative or sequential file organizations.	None for sequential or for relative if the `-vms` (Linux OS and Mac OS X) or `/vms` (Windows OS) option was omitted. One byte for relative if the `vms` option was specified.
Variable-length	Sequential file organization only. The variable-length record type is generally the most portable record type across multi-vendor platforms.	Eight bytes per record.
Segmented	Sequential file organization only and only for unformatted sequential access. The segmented record type is unique to Intel Fortran and should not be used for portability with programs written in languages other than Fortran or for places where Intel Fortran is not used. However, because the segmented record type is unique to Intel Fortran products, formatted data in segmented files can be ported across Intel Fortran platforms.	Four bytes per record. One additional padding byte (space) is added if the specified record size is an odd number.
Stream (uses no record terminator)	Sequential file organization only.	None required.
Stream_CR (uses CR as record terminator)	Sequential file organization only.	One byte per record.
Stream_LF (uses LF as record terminator)	Sequential file organization only.	One byte per record.
Stream_CRLF (uses CR and LF as record	Sequential file organization only.	Two bytes per record.

terminator)

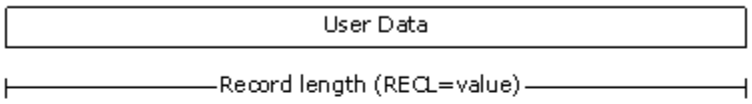
Fixed-Length Records

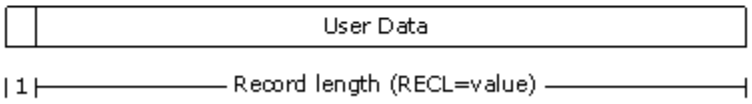
When you specify fixed-length records, you are specifying that all records in the file contain the same number of bytes. When you open a file that is to contain fixed-length records, you must specify the record size using the RECL keyword. A sequentially organized opened file for direct access must contain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

For relative files, the layout and overhead of fixed-length records depends upon whether the program accessing the file was compiled using the `-vms` (Linux OS and Mac OS X) or `/vms` (Windows OS) option:

- For relative files where the `vms` option was omitted (the default), 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 following figures show the record layout of fixed-length records. The first is for all sequential and relative files where the `vms` option was omitted. The second is for relative files where the `vms` option was specified.





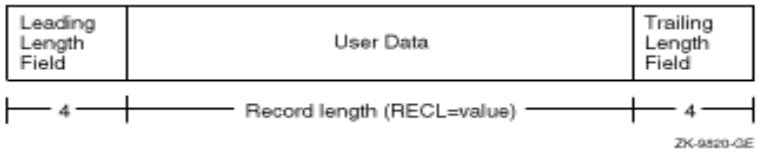
Variable-Length Records

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

Variable-length records are prefixed and suffixed by 4 bytes of control information containing length fields. The trailing length field allows a BACKSPACE request to skip back over records efficiently. The 4-byte integer value stored in each length field indicates the number of data bytes (excluding overhead bytes) in that particular variable-length record.

The character count field of a variable-length record is available when you read the record by issuing a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes should be in an I/O list.

The record layout of variable-length records that are less than 2 gigabytes is shown below:

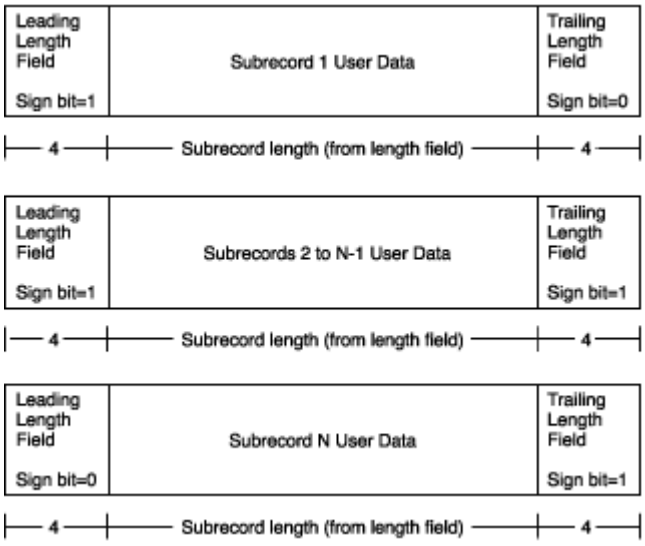


For a record length greater than 2,147,483,639 bytes, the record is divided into *subrecords*. The subrecord can be of any length from 1 to 2,147,483,639, inclusive.

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 record layout of variable-length records that are greater than 2 gigabytes is shown below:



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

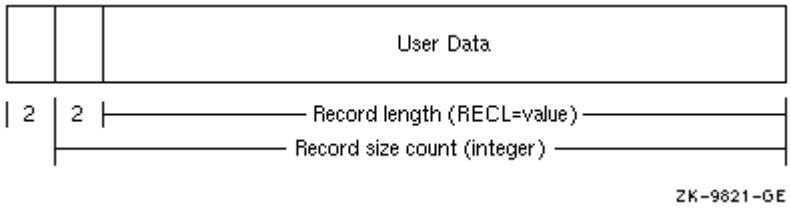
A segmented record is a single logical record consisting of one or more variable-length, unformatted records in a sequentially organized disk file. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

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 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 length count (includes the 2 bytes used by the segment identifier), followed by a 2-byte integer segment identifier that identifies this segment as one of the following:

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

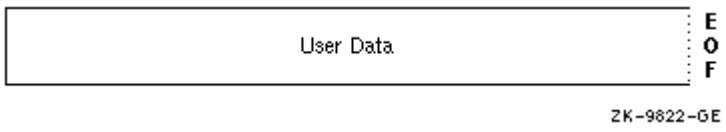
If the specified record length is an odd number, the user data will be padded with a single blank (one byte), but this extra byte is not added to the 2-byte integer record size count.

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

Stream File Data

A stream file is not grouped into records and contains no control information. Stream files are used with CARRIAGECONTROL='NONE' and contain character or binary data that is read or written only to the extent of the variables specified on the input or output statement.

The layout of a stream file appears below.



Stream_CR, Stream_LF and Stream_CRLF Records

Stream_CR, Stream_LF, and Stream_CRLF records are variable-length records whose length is indicated by explicit record terminators embedded in the data, not by a count. These

terminators are automatically added when you write records to a stream-type file and are removed when you read records.

Each variety uses either a different 1-byte or 2-byte record terminator:

- Stream_CR files 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 on Linux* OS and Mac OS* X systems.
- Stream_CRLF files use a carriage return/line-feed (new line) pair as the terminator, so Stream_CRLF files must not contain embedded carriage returns or line-feed (new line) characters. This is the usual operating system text file record type on Windows* systems.

Guidelines for Choosing a Record Type

Before you choose a record type, consider whether your application will use formatted or unformatted data. If you are using formatted data, you can choose any record type except segmented. If you are using unformatted data, avoid the Stream, Stream_CR, Stream_LF and Stream_CRLF record types.

The segmented record type can only be used for unformatted sequential access with sequential files. You should not use segmented records for files that are read by programs written in languages other than Intel Fortran.

The Stream, Stream_CR, Stream_LF, Stream_CRLF and segmented record types can be used only with sequential files.

The default record type (RECORDTYPE) depends on the values for the ACCESS and FORM specifiers for the OPEN statement. (The RECORDTYPE= specifier is ignored for stream access.)

The record type of the file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

An I/O record is a collection of fields (data items) that are logically related and are usually processed as a unit.

Unless you specify nonadvancing I/O (ADVANCE specifier), each Intel Fortran I/O statement transfers at least one record.

Specifying the Line Terminator for Formatted Files

Use the FOR_FMT_TERMINATOR environment variable to specify the line terminator value used for Fortran formatted files with no explicit RECORDTYPE= specifier.

The FOR_FMT_TERMINATOR environment variable is processed once at the beginning of program execution. Whatever it specifies for specific units continues for the rest of the execution.

You can specify the numbers of the units to have a specific record terminator. The READ/WRITE statements that use these unit numbers will now use the specified record terminators. Other READ/WRITE statements will work in the usual way.

A RECORDTYPE=specifier on an OPEN statement overrides the value set by FOR_FMT_TERMINATOR. The FOR_FMT_TERMINATOR value is ignored for ACCESS='STREAM' files.

General Syntax for FOR_FMT_TERMINATOR

The syntax for this environment variable is as follows:

```
FOR_FMT_TERMINATOR=MODE[:ULIST][;MODE[:ULIST]]
```

where:

MODE=CR | LF | CRLF

ULIST = U | ULIST,U

U = decimal | decimal - decimal

- MODE specifies the record terminator to be used. The keyword CR means to terminate records with a carriage return. The keyword LF means to terminate records with a line feed; this is the default on Linux OS and Mac OS X systems. The keyword CRLF means to terminate records with a carriage return/line feed pair; this is the default on Windows systems.
- Each list member U is a simple unit number or a number of units as a range. The number of list members is limited to 64.
- decimal is a non-negative decimal number less than 232.

No spaces are allowed inside the FOR_FMT_TERMINATOR value.

On Linux* systems, the command line for the variable setting in the shell is:

```
Sh: export FOR_FMT_TERMINATOR=MODE:ULIST
```

Note

The environment variable value should be enclosed in quotes if the semicolon is present.

Example:

The following specifies that all input/output operations on unit numbers 10, 11, and 12 have records terminated with a carriage return/line feed pair:

FOR_FMT_TERMINATOR=CRLF:10-12

Record Length

Use the RECL specifier to specify the record length.

The units used for specifying record length depend on the form of the data:

- Formatted files (FORM= 'FORMATTED'): Specify the record length in bytes.
- Unformatted files (FORM= 'UNFORMATTED'): Specify the record length in 4-byte units, unless you specify the -assume byterecl (Linux OS and Mac OS X) or /assume:byterecl (Windows OS) option to request 1-byte units.

For all but variable-length sequential records on 64-bit addressable systems, the maximum record length is 2.147 billion bytes (2,147,483,647 minus the bytes for record overhead). For variable-length sequential records on 64-bit addressable systems, the theoretical maximum record length is about 17,000 gigabytes. When considering very large record sizes, also consider limiting factors such as system virtual memory.

Note

The RECL specifier is ignored for stream access.

Record Access

Record access refers to how records will be read from or written to a file, regardless of the file's organization. Record access is specified each time you open a file; it can be different each time. The type of record access permitted is determined by the combination of file organization and record type.

For 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
```

Stream Access

Stream access transfers bytes from records sequentially until a record terminator is found or a specified number of bytes have been read or written. Formatted stream records are terminated with a new line character; unformatted stream data contains no record terminators. Bytes can be read from or written to a file by byte position, where the first byte of the file is position 1. An example follows:

```
OPEN (UNIT=12, ACCESS='STREAM')
READ (12) I, J, K              ! start at the first byte of the file
READ (12, POS=200) X, Y       ! then read staring at byte 200
READ (12) A, B            ! then read starting where the previous READ stopped
```

The POS= specifier on INQUIRE can be used to determine the current byte position in the file.

Note

The RECORDTYPE= specifier is ignored for stream access.

Limitations of Record Access by File Organization and Record Type

You can use sequential and direct access modes on sequential and relative files. However, direct access to a sequential organization file can only be done if the file resides on disk and contains fixed-length records.

The table below summarizes the types of access permitted for the various combinations of file organizations and record types.

Record Type	Sequential Access?	Direct Access?
Sequential file organization		
Fixed	Yes	Yes
Variable	Yes	No
Segmented	Yes	No
Stream	Yes	No
Stream_CR	Yes	No
Stream_LF	Yes	No
Stream_CRLF	Yes	No
Relative file organization		
Fixed	Yes	Yes

Note

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

Record Transfer

I/O statements transfer all data as records. The amount of data that a record can contain depends on the following circumstances:

- With formatted I/O (except for fixed-length records), the number of items in the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.
- With namelist and list-directed output, the items listed in the NAMELIST statement or I/O statement list (in conjunction with the NAMELIST or list-directed formatting rules) determine the amount of data to be transferred.
- With unformatted I/O (except for fixed-length records), the I/O statement alone specifies the amount of data to be transferred.
- When you specify fixed-length records (RECORDTYPE= 'FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding).

Typically, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for a single I/O statement to transfer data from or to more than one record, depending on the form of I/O used.

Input Record Transfer

When using advancing I/O, if an input statement specifies fewer data fields (less data) than the record contains, the remaining fields are ignored.

If an input statement specifies more data fields than the record contains, one of the following occurs:

- For formatted input using advancing I/O, if the file was opened with PAD='YES', additional fields are read as spaces. If the file is opened with PAD='NO', an error occurs (the input statement should not specify more data fields than the record contains).
- For formatted input using nonadvancing I/O (ADVANCE='NO'), an end-of-record (EOR) condition is returned. If the file was opened with PAD='YES', additional fields are read as spaces.
- For list-directed input, another record is read.
- For NAMELIST input, another record is read.
- For unformatted input, an error occurs.

Output Record Transfer

If an output statement specifies fewer data fields than the record contains (less data than required to fill a record), the following occurs:

- With fixed-length records (RECORDTYPE= 'FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding) in the form of spaces (for a formatted record) or zeros (for an unformatted record).

- With other record types, the fields present are written and those omitted are not written (might result in a short record).

If the output statement specifies more data than the record can contain, an error occurs, as follows:

- With formatted or unformatted output using fixed-length records, if the items in the output statement and its associated format specifier result in a number of bytes that exceeds the maximum record length (RECL), an error occurs.
- With formatted or unformatted output not using fixed-length records, if the items in the output statement and its associated format specifier result in a number of bytes that exceeds the maximum record length (RECL), the Intel Fortran RTL attempts to increase the RECL value and write the longer record. To obtain the RECL value, use an INQUIRE statement.
- For list-directed output and namelist output, if the data specified exceeds the maximum record length (RECL), another record is written.

Specifying Default Pathnames and File Names

Intel Fortran provides a number of ways of specifying all or part of a file specification (directory and file name). The following list uses the Linux* pathname `/usr/proj/testdata` as an example:

- The FILE specifier in an OPEN statement typically specifies only a file name (such as `testdata`) or contains both a directory and file name (such as `/usr/proj/testdata`).
- The DEFAULTFILE specifier in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).
- If you used an implied OPEN or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Examples of Applying Default Pathnames and File Names

For example, for an implied OPEN of unit number 3, Intel Fortran will check the environment variable FORT3. If the environment variable FORT3 is set, its value is used. If it is not set, the system supplies the file name `fort.3`.

In the following table, assume the current directory is `/usr/smith` and the I/O uses unit 1, as in the statement READ (1,100).

OPEN FILE value	OPEN DEFAULTFILE value	FORT1 environment variable value	Resulting pathname
not specified	not specified	not specified	`/usr/smith/fort.1`
not specified	not specified	`test.dat`	`/usr/smith/test.dat`
not specified	not checked	`/usr/tmp/t.dat`	`/usr/tmp/t.dat`
not specified	`/tmp`	not specified	`/tmp/fort.1`

not specified	/tmp	testdata	/tmp/testdata
not specified	/usr	lib/testdata	/usr/lib/testdata
file.dat	/usr/group	not checked	/usr/group/file.dat
/tmp/file.dat	not checked	not checked	/tmp/file.dat
file.dat	not specified	not checked	/usr/smith/file.dat

When the resulting file pathname begins with a tilde character (~), C-shell-style pathname substitution is used (regardless of what shell is being used), such as a top-level directory (below the root). For additional information on tilde pathname substitution, see csh(1).

Rules for Applying Default Pathnames and File Names

Intel Fortran determines the file name and the directory path based on certain rules. It determines a file name string as follows:

- If the FILE specifier is present, its value is used.
- If the FILE specifier is not present, Intel Fortran examines the corresponding environment variable. If the corresponding environment variable is set, that value is used. If the corresponding environment variable is not set, a file name in the form fort.*n* is used.

Once Intel Fortran determines the resulting file name string, it determines the directory (which optionally precedes the file name) as follows:

- If the resulting file name string contains an absolute pathname, it is used and the DEFAULTFILE specifier, environment variable, and current directory values are ignored.
- If the resulting file name string does not contain an absolute pathname, Intel Fortran examines the DEFAULTFILE specifier and current directory value: If the corresponding environment variable is set and specifies an absolute pathname, that value is used. Otherwise, the DEFAULTFILE specifier value, if present, is used. If the DEFAULTFILE specifier is not present, Intel Fortran uses the current directory as an absolute pathname.

Opening Files: OPEN Statement

To open a file, you can use a preconnected file (such as for terminal output) or can open a file with an OPEN statement. The OPEN statement lets you specify the file connection characteristics and other information.

OPEN Statement Specifiers

The OPEN statement connects a unit number with an external file and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers. Once you open a file, you should close it before opening it again unless it is a preconnected file.

If you open a unit number that was opened previously (without being closed), one of the following occurs:

- If you specify a file specification that does not match the one specified for the original open, the Intel Fortran run-time system closes the original file and then opens the second file. This resets the current record position for the second file.
- If you specify a file specification that matches the one specified for the original open, the file is reconnected without the internal equivalent of the CLOSE and OPEN. This lets you change one or more OPEN statement run-time specifiers while maintaining the record position context.

You can use the INQUIRE statement to obtain information about whether or not a file is opened by your program.

Especially when creating a new file using the OPEN statement, examine the defaults (see the description of the OPEN statement 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, sequential, or stream).
- SHARED sets file locking for shared access. Indicates that other users can access the same file.

- NOSHARED sets file locking for exclusive access. Indicates that other users who use file locking mechanisms cannot access the same file.
- SHARE specifies shared or exclusive access; for example, SHARE='DENYNONE' or SHARE='DENYRW'.
- POSITION indicates whether to position the file at the beginning of file, before the end-of-file record, or leave it as is (unchanged).
- ACTION or READONLY indicates whether statements will be used to only read records, only write records, or both read and write records.
- MAXREC specifies the maximum record number for direct access.
- ASSOCIATEVARIABLE specifies the variable containing the next record number for direct access.
- ASYNCHRONOUS specifies whether input/output should be performed asynchronously.

Specifiers for Record Transfer Characteristics

These specifiers identify record transfer characteristics:

- BLANK indicates whether to ignore blanks in numeric fields.
- DELIM specifies the delimiter character for character constants in list-directed or namelist output.
- PAD, when reading formatted records, indicates whether padding characters should be added if the item list and format specification require more data than the record contains.
- BUFFERED indicates whether buffered or non-buffered I/O should be used.
- BLOCKSIZE specifies the 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.

Specifying File Locations in an OPEN Statement

You can use the FILE and DEFAULTFILE specifiers of the OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. (The *Language Reference Manual* describes the OPEN statement in greater detail.)

For example:

```
OPEN (UNIT=4, FILE='/usr/users/smith/test.dat', STATUS='OLD')
```

The file `test.dat` in directory `/usr/users/smith` is opened on logical unit 4. No defaults are applied, because both the directory and file name were specified. The value of the FILE specifier can be a character constant, variable, or expression.

In the following interactive example, the user supplies the file name and the DEFAULTFILE specifier supplies the default values for the full pathname string. The file to be opened is located in `/usr/users/smith` and is concatenated with the file name typed by the user into the variable DOC:

```
CHARACTER(LEN=9) DOC
WRITE (6,*)  'Type file name '
READ (5,*) DOC
OPEN (UNIT=2, FILE=DOC, DEFAULTFILE='/usr/users/smith',STATUS='OLD')
```

A slash (backslash on Windows systems) is appended to the end of the default file string if it does not have one.

Obtaining File Information: INQUIRE Statement

The INQUIRE statement returns information about a file and has the following forms:

- Inquiry by unit
- Inquiry by file name
- Inquiry by output item list
- Inquiry by directory

Inquiry by Unit

An inquiry by unit is usually done for an opened (connected) file. An inquiry by unit causes the Intel Fortran RTL to check whether the specified unit is connected or not. One of the following occurs, depending on whether the unit is connected or not:

If the unit is connected:

- The EXIST and OPENED specifier variables indicate a true value.
- The pathname and file name are returned in the NAME specifier variable (if the file is named).
- Other information requested on the previously connected file is returned.
- Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement.
- The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

If the unit is not connected:

- The OPENED specifier indicates a false value.
- The unit NUMBER specifier variable is returned as a value of -1.
- Any other information returned will be undefined or default values for the various specifiers.

For example, the following INQUIRE statement shows whether unit 3 has a file connected (OPENED specifier) in logical variable I_OPENED, the name (case-sensitive) in character

variable I_NAME, and whether the file is opened for READ, WRITE, or READWRITE access in character variable I_ACTION:

```
INQUIRE (3, OPENED=I_OPENED, NAME=I_NAME, ACTION=I_ACTION)
```

Inquiry by File Name

An inquiry by name causes the Intel Fortran RTL to scan its list of open files for a matching file name. One of the following occurs, depending on whether a match occurs or not:

If a match occurs:

- The EXIST and OPENED specifier variables indicate a true value.
- The pathname and file name are returned in the NAME specifier variable.
- The UNIT number is returned in the NUMBER specifier variable.
- Other information requested on the previously connected file is returned.
- Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement.
- The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

If no match occurs:

- The OPENED specifier variable indicates a false value.
- The unit NUMBER specifier variable is returned as a value of -1.
- The EXIST specifier variable indicates (true or false) whether the named file exists on the device or not.
- If the file does exist, the NAME specifier variable contains the pathname and file name.
- Any other information returned will be default values for the various specifiers, based on any information specified when calling INQUIRE.

The following INQUIRE statement returns whether the file named `log_file` is connected in logical variable I_OPEN, whether the file exists in logical variable I_EXIST, and the unit number in integer variable I_NUMBER:

```
INQUIRE (FILE='log_file', OPENED=I_OPEN, EXIST=I_EXIST, NUMBER=I_NUMBER)
```

Inquiry by Output Item List

Unlike inquiry by unit or inquiry by name, inquiry by output item list does not attempt to access any external file. It returns the length of a record for a list of variables that would be used for unformatted WRITE, READ, and REWRITE statements. The following INQUIRE statement returns the maximum record length of the variable list in variable I_RECLENGTH. This variable is then used to specify the RECL value in the OPEN statement:

```
INQUIRE (IOLENGTH=I_RECLENGTH) A, B, H
OPEN (FILE='test.dat', FORM='UNFORMATTED', RECL=I_RECLENGTH, UNIT=9)
```

For an unformatted file, the IOLENGTH value is returned using 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

Inquiry by Directory

An inquiry by directory verifies that a directory exists.

If the directory exists:

- The EXIST specifier variable indicates a true value.
- The full directory pathname is returned in the DIRSPEC specifier variable.

If the directory does not exist:

- The EXIST specified variable indicates a false value.
- The value of the DIRSPEC specifier variable is unchanged.

For example, the following INQUIRE statement returns the full directory pathname:

```
LOGICAL        ::L_EXISTS
CHARACTER (255)::C_DIRSPEC
INQUIRE (DIRECTORY=".", DIRSPEC=C_DIRSPEC, EXIST=L_EXISTS)
```

The following INQUIRE statement verifies that a directory does not exist:

```
INQUIRE (DIRECTORY="I-DO-NOT-EXIST",
EXIST=L_EXISTS)
```

Closing Files: CLOSE Statement

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

After you open a file or use a preconnected file, you can use the following statements:

- READ, WRITE, ACCEPT, and PRINT to perform record I/O.
- BACKSPACE, ENDFILE, and REWIND to set record position within the file.
- DELETE, REWRITE, TYPE, and FIND to perform various operations.

The record I/O statement must use the appropriate record I/O form (formatted, list-directed, namelist, or unformatted).

You can use the following specifiers with the READ and WRITE record I/O statements:

- UNIT specifies the unit number to or from which input or output will occur.
- END specifies a label to branch to if end-of-file occurs; only applies to input statements on sequential files.
- ERR specifies a label to branch to if an error occurs.
- IOSTAT specifies an integer variable to contain the error number if an error occurs.
- FMT specifies a label of a FORMAT statement or character data specifying a FORMAT.
- NML specifies the name of a NAMELIST.
- REC specifies a record number for direct access.

When using nonadvancing I/O, use the ADVANCE, EOR, and SIZE specifiers.

When using the REWRITE statement, you can use the UNIT, FMT, ERR, and IOSTAT specifiers.

File Sharing on Linux* OS and Mac OS* X Systems

Depending on the value specified by the ACTION (or READONLY) specifier in the OPEN statement, the file will be opened by your program for reading, writing, or both reading and writing records. This simply checks that the program itself executes the type of statements intended.

File locking mechanisms allow users to enable or restrict access to a particular file when that file is being accessed by another process.

Intel® Fortran file locking features provide three file access modes:

- Implicit Shared mode, which occurs when no mode is specified. This is also called No Locking.
- Explicit Shared mode, when all cooperating processes have access to a file. This mode is set in the OPEN statement by the SHARED specifier or the SHARE='DENYNONE' specifier.
- Exclusive mode, when only one process has access to a file. This mode is set in the OPEN statement by the NOSHARED specifier or the SHARE='DENYRW' specifier.

The file locking mechanism looks for explicit setting of the corresponding specifier in the OPEN statement. Otherwise, the Fortran run time does not perform any setting or checking for file locking and the process can access the file regardless of the fact that other processes have already opened or locked the file.

Example 1: Implicit Shared Mode (No Locking)

Process 1 opens the file without a specifier, resulting in no locking.
Process 2 now tries to open the file:

- It gains access regardless of the mode it is using.

Example 2: Explicit Shared Mode

Process 1 opens the file with Explicit Shared mode.
Process 2 now tries to open the file:

- If process 2 opens the file with Explicit Shared mode or Implicit Shared (No Locking) mode, it gets access to the file.
- If process 2 opens the file with Exclusive mode, it receives an error.

Example 3: Exclusive Mode

Process 1 opens the file with Exclusive mode.
Process 2 now tries to open the file:

- If process 2 opens the file with Implicit Shared (No Locking) mode, it gets access to the file.
- If process 2 opens the file with Explicit Shared or Exclusive mode, it receives an error.

The Fortran runtime does not coordinate file entry updates during cooperative access. The user needs to coordinate access times among cooperating processes to handle the possibility of simultaneous WRITE and REWRITE statements on the same record positions.

Specifying the Initial Record Position

When you open a disk file, you can use the OPEN statement POSITION specifier to request one of the following initial record positions within the file:

- The initial position before the first record (POSITION='REWIND'). A sequential access READ or WRITE statement will read or write the first record in the file.
- A point beyond the last record in the file (POSITION='APPEND'), just before the end-of-file record, if one exists. For a new file, this is the initial position before the first record (same as 'REWIND'). You might specify 'APPEND' before you write records to an existing sequential file using sequential access.
- The current position (POSITION='ASIS'). This is usually used only to maintain the current record position when reconnecting a file. The second OPEN specifies the same unit number and specifies the same file name (or omits it), which leaves the file open, retaining the current record position. However, if the second OPEN specifies a different file name for the same unit number, the current file will be closed and the different file will be opened.

The following I/O statements allow you to change the current record position:

- REWIND sets the record position to the initial position before the first record. A sequential access READ or WRITE statement would read or write the first record in the file.
- BACKSPACE sets the record position to the previous record in a file. Using sequential access, if you wrote record 5, issued a BACKSPACE to that unit, and then read from that unit, you would read record 5.
- ENDFILE writes an end-of-file marker. This is typically done after writing records using sequential access just before you close the file.

Unless you use nonadvancing I/O, reading and writing records usually advances the current record position by one record. More than one record might be transferred using a single record I/O statement.

Advancing and Nonadvancing Record I/O

After you open a file, if you omit the ADVANCE specifier (or specify ADVANCE= 'YES') in READ and WRITE statements, advancing I/O (normal Fortran I/O) will be used for record access. When using advancing I/O:

- Record I/O statements transfer one entire record (or multiple records).
- Record I/O statements advance the current record position to a position before the next record.

You can request nonadvancing I/O for the file by specifying the ADVANCE= 'NO' specifier in a READ and WRITE statement. You can use nonadvancing I/O only for sequential access to external files using formatted I/O (not list-directed or namelist).

When you use nonadvancing I/O, the current record position does not change, and part of the record might be transferred, unlike advancing I/O where one or more entire records are always transferred.

You can alternate between advancing and nonadvancing I/O by specifying different values for the ADVANCE specifier ('YES' and 'NO') in the READ and WRITE record I/O statements.

When reading records with either advancing or nonadvancing I/O, you can use the END specifier to branch to a specified label when the end of the file is read.

Because nonadvancing I/O might not read an entire record, it also supports an EOR specifier to branch to a specified label when the end of the record is read. If you omit the EOR and the IOSTAT specifiers when using nonadvancing I/O, an error results when the end-of-record is read.

When using nonadvancing input, you can use the SIZE specifier to return the number of characters read. For example, in the following READ statement, SIZE=X (where variable X is an integer) returns the number of characters read in X and an end-of-record condition causes a branch to label 700:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

User-Supplied OPEN Procedures: USEROPEN Specifier

You can use the USEROPEN specifier in an 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 Linux* OS-style 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 UOPEN is a Fortran function, its name is decorated appropriately for Fortran.

Likewise, if UOPEN is a C function, its name is decorated appropriately for C, as long as the following line is included in the above code:

!DEC$ATTRIBUTES C::UOPEN

Compiling and Linking the C and Intel Fortran Programs

Use the icc or icl command to compile the called `uopen` C function `uopen.c` and the ifort command to compile the Intel Fortran calling program `ex1.f`. The same ifort command also links both object files by using the appropriate libraries:

```
icc -c uopen.c (Linux* OS)
icl -c uopen.c (Windows* OS)
ifort ex1.f uopen.o
```

Example Source Code

```
      program UserOpenSample
!DEC$ FREEFORM
      IMPLICIT NONE
      EXTERNAL         UOPEN
      INTEGER(4)       UOPEN
      CHARACTER*10  :: FileName="UOPEN.DAT"
      INTEGER*4     :: IOS
```

```
      Character*255 :: InqFullName
      Character*100 :: InqFileName
      Integer       :: InqLun
      Character*30  :: WriteOutBuffer="Write_One_Record_to_the_File. "
      Character*30  :: ReadInBuffer  ="??????????????????????????????"
110   FORMAT( X,A, ": Created (iostat=",I0,")")
115   FORMAT( X,A, ": Creation Failed (iostat=",I0,")")
120   FORMAT( X,A, ": ERROR: INQUIRE Returned Wrong FileName")
130   FORMAT( X,A, ": ERROR: ReadIn and WriteOut Buffers Do Not Match")
190   FORMAT( X,A, ": Completed.")
      WRITE(*,'(X,"Test the USEROPEN Facility of Open")')
      OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='REPLACE',USEROPEN=UOPEN, &
           IOSTAT=ios, ACTION='READWRITE')

!     When the OPEN statement is executed,
!     the UOPEN function receives control.
!     The function opens the file by calling CreateFile( ),
!     performs whatever operations were specified, and subsequently
!     returns control (with the handle returned by CreateFile( ))
!     to the calling Fortran program.
      IF (IOS .EQ. 0)  THEN
         WRITE(*,110) TRIM(FileName), IOS
         INQUIRE(10, NAME=InqFullName)
         CALL ParseForFileName(InqFullName,InqFileName)
         IF (InqFileName .NE. FileName) THEN
            WRITE(*,120) TRIM(FileName)
         END IF
      ELSE
         WRITE(*,115) TRIM(FileName), IOS
         GOTO 9999
      END IF
      WRITE(10,*) WriteOutBuffer
      REWIND(10)
      READ(10,*) ReadInBuffer
      IF (ReadinBuffer .NE. WriteOutbuffer) THEN
         WRITE(*,130) TRIM(FileName)
      END IF
      CLOSE(10, DISPOSE='DELETE')
      WRITE(*,190) TRIM(FileName)
      WRITE(*,'(X,"Test of USEROPEN Completed")')
9999  CONTINUE
      END
!DEC$ IF DEFINED(_WIN32)
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
! Here is the UOPEN function for WIN32:
!
! The UOPEN function is declared to use the cdecl calling convention,
! so it matches the Fortran rtl declaration of a useropen routine.
!
! The following function definition and arguments are passed from the Intel
! Fortran Run-time Library to the function named in USEROPEN:
!
! The first 7 arguments correspond to the CreateFile( ) api arguments.
! The value of these arguments is set according the caller's OPEN( )
! arguments:
!
! FILENAME
!      Is the address of a null terminated character string that
!      is the name of the file.
! DESIRED_ACCESS
!      Is the desired access (read-write) mode passed by reference.
! SHARE_MODE
!      Is the file sharing mode passed by reference.
! A_NULL
!      Is always null. The Fortran runtime library always passes a NULL
!      for the pointer to a SECURITY_ATTRIBUTES structure in its
!      CreateFile( ) call.
! CREATE_DISP
!      Is the creation disposition specifying what action to take on files
!      that exist, and what action to take on files
!      that do not exist. It is passed by reference.
! FLAGS_ATTR
!      Specifies the file attributes and flags for the file. It is passed
!      by reference.
! B_NULL
!      Is always null. The Fortran runtime library always passes a NULL
!      for the handle to a template file in it's CreateFile( ) call.
! The last 2 arguments are the Fortran unit number and length of the
! file name:
! UNIT
!      Is the Fortran unit number on which this OPEN is being done. It is
!      passed by reference.
! FLEN
!      Is the length of the file name, not counting the terminating null,
!      and passed by value.
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      INTEGER(4)     FUNCTION UOPEN( FILENAME,       &
                                DESIRED_ACCESS, &
```

```
                                SHARE_MODE,     &
                                A_NULL,         &
                                CREATE_DISP,    &
                                FLAGS_ATTR,     &
                                B_NULL,         &
                                UNIT,           &
                                FLEN )
      !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'UOPEN' :: UOPEN
      !DEC$ATTRIBUTES REFERENCE :: FILENAME
      !DEC$ATTRIBUTES REFERENCE :: DESIRED_ACCESS
      !DEC$ATTRIBUTES REFERENCE :: SHARE_MODE
      !DEC$ATTRIBUTES REFERENCE :: CREATE_DISP
      !DEC$ATTRIBUTES REFERENCE :: FLAGS_ATTR
      !DEC$ATTRIBUTES REFERENCE :: UNIT
      USE KERNEL32
      IMPLICIT NONE
      INTEGER*4  DESIRED_ACCESS
      INTEGER*4  SHARE_MODE
      INTEGER*4  A_NULL
      INTEGER*4  CREATE_DISP
      INTEGER*4  FLAGS_ATTR
      INTEGER*4  B_NULL
      INTEGER*4  UNIT
      INTEGER*4  FLEN
      CHARACTER*(FLEN) FILENAME
      INTEGER(4) ISTAT
      TYPE(T_SECURITY_ATTRIBUTES), POINTER :: NULL_SEC_ATTR
140   FORMAT( X, "ERROR: USEROPEN Passed Wrong Unit Number",I)
!     Sanity check
      IF (UNIT .NE. 10) THEN
         WRITE(*,140) UNIT
      END IF
      !! WRITE(*,*) "FILENAME=",FILENAME !! prints the full path of the filename
! Set the FILE_FLAG_WRITE_THROUGH bit in the flag attributes to CreateFile( )
! (for whatever reason)
!     FLAGS_ATTR = FLAGS_ATTR + FILE_FLAG_WRITE_THROUGH
! Do the CreateFile( ) call and return the status to the Fortran rtl
      ISTAT = CreateFile( FILENAME,             &
                          DESIRED_ACCESS,       &
                          SHARE_MODE,           &
                          NULL_SEC_ATTR,        &
                          CREATE_DISP,          &
                          FLAGS_ATTR,           &
                          0 )
```

```
      if (ISTAT == INVALID_HANDLE_VALUE) then
          write(*,*) "Could not open file (error ", GetLastError(),")"
      endif
      UOPEN = ISTAT
      RETURN
      END
!DEC$ ELSE ! LINUX OS or MAC OS X
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! Here is the UOPEN function for Linux OS/Mac OS X:
!
! The UOPEN function is declared to use the cdecl calling convention,
! so it matches the Fortran rtl declaration of a useropen routine.
!
! The following function definition and arguments are passed from the
! Intel Fortran Run-time Library to the function named in USEROPEN:
!
! FILENAME
!      Is the address of a null terminated character string that
!      is the name of the file.
! OPEN_FLAGS
!      read-write flags (see file.h or open(2)).
! CREATE_MODE
!      set if new file (to be created).
! UNIT
!      Is the Fortran unit number on which this OPEN is being done. It is
!      passed by reference.
! FLEN
!      Is the length of the file name, not counting the terminating null,
!      and passed by value.
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      INTEGER FUNCTION UOPEN( FILENAME,       &
                              OPEN_FLAGS,     &
                              CREATE_MODE,    &
                              UNIT,           &
                              FLEN )
      !DEC$ATTRIBUTES C, DECORATE, ALIAS:'uopen' :: UOPEN
      !DEC$ATTRIBUTES REFERENCE :: FILENAME
      !DEC$ATTRIBUTES REFERENCE :: OPEN_FLAGS
      !DEC$ATTRIBUTES REFERENCE :: CREATE_MODE
      !DEC$ATTRIBUTES REFERENCE :: UNIT
      IMPLICIT NONE
      INTEGER*4  OPEN_FLAGS
      INTEGER*4  CREATE_MODE
      INTEGER*4  UNIT
```

```
      INTEGER*4  FLEN
      CHARACTER*(FLEN) FILENAME
      INTEGER*4  ISTAT
      !DEC$ATTRIBUTES C, DECORATE, ALIAS:'open'  :: OPEN
      external   OPEN
      integer*4  OPEN
140   FORMAT( X, "ERROR: USEROPEN Passed Wrong Unit Number",I)
!     Sanity check
      IF (UNIT .NE. 10) THEN
         WRITE(*,140) UNIT
      END IF
!     Call the system OPEN routine
      ISTAT = OPEN ( %ref(FILENAME),     &
                     OPEN_FLAGS,         &
                     CREATE_MODE )
      UOPEN = ISTAT
      RETURN
      END
!DEC$ ENDIF ! End of UOPEN Function
!---------------------------------------------------------------
! SUBROUTINE: ParseForFileName
!             Takes a full pathname and retuns the filename
!             with its extension.
!---------------------------------------------------------------
      SUBROUTINE ParseForFileName(FullName,FileName)
      Character*255 :: FullName
      Character*100 :: FileName
      Integer       :: P
!DEC$ IF DEFINED(_WIN32)
      P = INDEX(FullName,'\',.TRUE.)
      FileName = FullName(P+1:)
!DEC$ ELSE ! LINUX OS/MAC OS X
      P = INDEX(FullName,'/',.TRUE.)
      FileName = FullName(P+1:)
!DEC$ ENDIF
      END
```

Microsoft Fortran PowerStation Compatible Files

When using the `-fpscomp` (Linux OS and Mac OS X) or `/fpscomp` (Windows OS) options for Microsoft* Fortran PowerStation compatibility, the following types of files are possible:

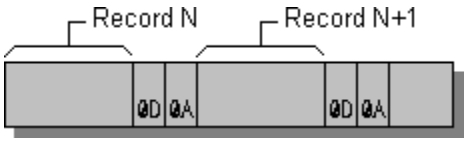
- Formatted Sequential

- Formatted Direct
- Unformatted Sequential
- Unformatted Direct
- Binary Sequential
- Binary Direct

Formatted Sequential Files

A formatted sequential file is a series of formatted records written sequentially and read in the order in which they appear in the file. Records can vary in length and can be empty. They are separated by carriage return (0D) and line feed (0A) characters as shown in the following figure.

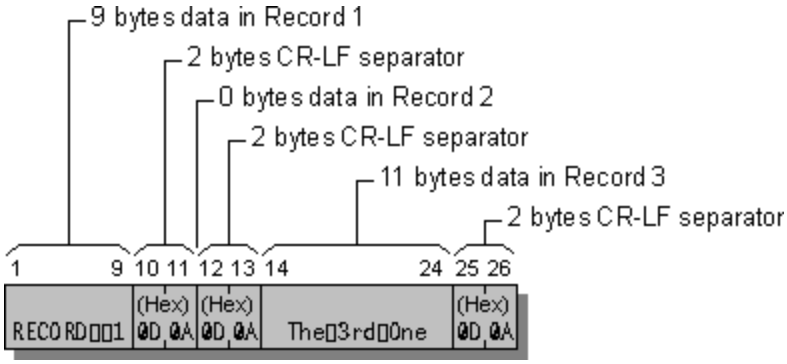
Formatted Records in a Formatted Sequential File



An example of a program writing three records to a formatted sequential file is given below. The resulting file is shown in the following figure.

```
   OPEN (3, FILE='FSEQ')
! FSEQ is a formatted sequential file by default.
   WRITE (3, '(A, I3)') 'RECORD', 1
   WRITE (3, '()')
   WRITE (3, '(A11)') 'The 3rd One'
   CLOSE (3)
   END
```

Formatted Sequential File



Formatted Direct Files

In a formatted direct file, all of the records are the same length and can be written or read in any order. The record size is specified with the RECL option in an OPEN statement and should be equal to or greater than the number of bytes in the longest record.

The carriage return (CR) and line feed (LF) characters are record separators and are not included in the RECL value. Once a direct-access record has been written, you cannot delete it, but you can rewrite it.

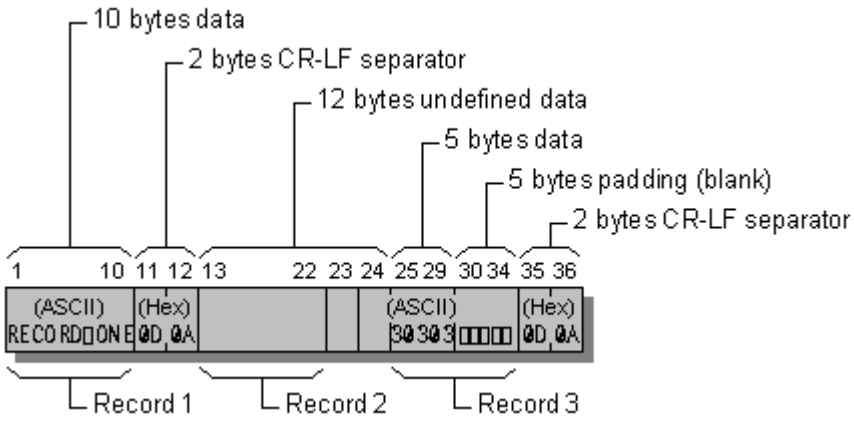
During output to a formatted direct file, if data does not completely fill a record, the compiler pads the remaining portion of the record with blank spaces. The blanks ensure that the file contains only completely filled records, all of the same length. During input, the compiler by default also adds filler bytes (blanks) to the input record if the input list and format require more data than the record contains.

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

An example of a program writing two records, record one and record three, to a formatted direct file is given below. The result is shown in the following figure.

```
OPEN (3,FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT',RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

Formatted Direct File



Unformatted Sequential Files

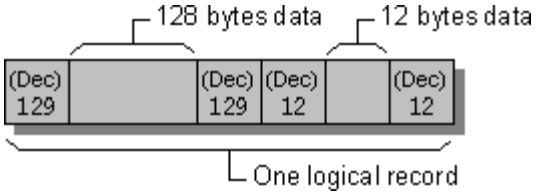
Unformatted sequential files are organized slightly differently on different platforms. This section describes unformatted sequential files created by Intel Fortran when the `-fpscomp` (Linux OS and Mac OS X) or `/fpscomp` (Windows OS) option is specified.

The records in an unformatted sequential file can vary in length. Unformatted sequential files are organized in chunks of 130 bytes or less called physical blocks. Each physical block consists of the data you send to the file (up to 128 bytes) plus two 1-byte "length bytes" inserted by the compiler. The length bytes indicate where each record begins and ends.

A logical record refers to an unformatted record that contains one or more physical blocks. (See the following figure.) Logical records can be as big as you want; the compiler will use as many physical blocks as necessary.

When you create a logical record consisting of more than one physical block, the compiler sets the length byte to 129 to indicate that the data in the current physical block continues on into the next physical block. For example, if you write 140 bytes of data, the logical record has the structure shown in the following figure.

Logical Record in Unformatted Sequential File

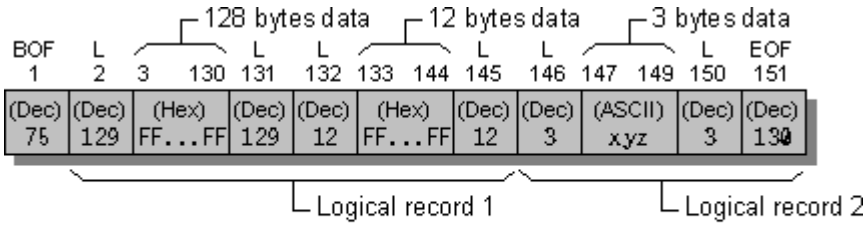


The first and last bytes in an unformatted sequential file are reserved; the first contains a value of 75, and the last holds a value of 130. Fortran uses these bytes for error checking and end-of-file references.

The following program creates the unformatted sequential file shown in the following figure:

```
!   Note: The file is sequential by default
!         -1 is FF FF FF FF hexadecimal.
!
   CHARACTER xyz(3)
   INTEGER(4) idata(35)
   DATA      idata /35 * -1/, xyz /'x', 'y', 'z'/
!
! Open the file and write out a 140-byte record:
! 128 bytes (block) + 12 bytes = 140 for IDATA, then 3 bytes for XYZ.
   OPEN (3, FILE='UFSEQ',FORM='UNFORMATTED')
   WRITE (3) idata
   WRITE (3) xyz
   CLOSE (3)
   END
```

Unformatted Sequential File



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

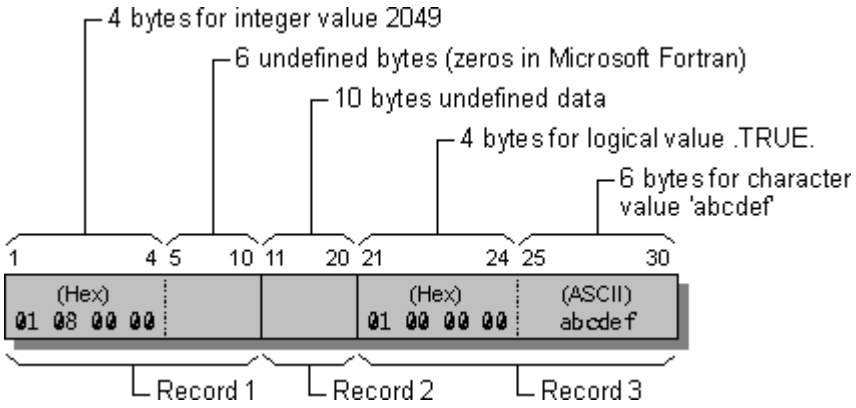
An unformatted direct file is a series of unformatted records. You can write or read the records in any order you choose. All records have the same length, given by the RECL specifier in an OPEN statement. No delimiting bytes separate records or otherwise indicate record structure.

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



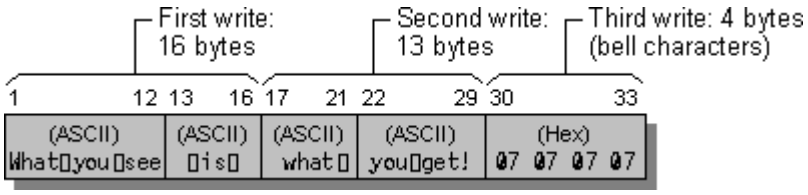
Binary Sequential Files

A binary sequential file is a series of values written and read in the same order and stored as binary numbers. No record boundaries exist, and no special bytes indicate file structure. Data is read and written without changes in form or length. For any I/O data item, the sequence of bytes in memory is the sequence of bytes in the file.

The next program creates the binary sequential file shown in the following figure:

```
!  NOTE: 07 is the bell character
!     Sequential is assumed by default.
!
   INTEGER(1) bells(4)
   CHARACTER(4) wys(3)
   CHARACTER(4) cvar
   DATA bells /4*7/
   DATA cvar /' is '/,wys /'What',' you',' see'/
   OPEN (3, FILE='BSEQ',FORM='BINARY')
   WRITE (3) wys, cvar
   WRITE (3) 'what ', 'you get!'
   WRITE (3) bells
   CLOSE (3)
   END
```

Binary Sequential File



Binary Direct Files

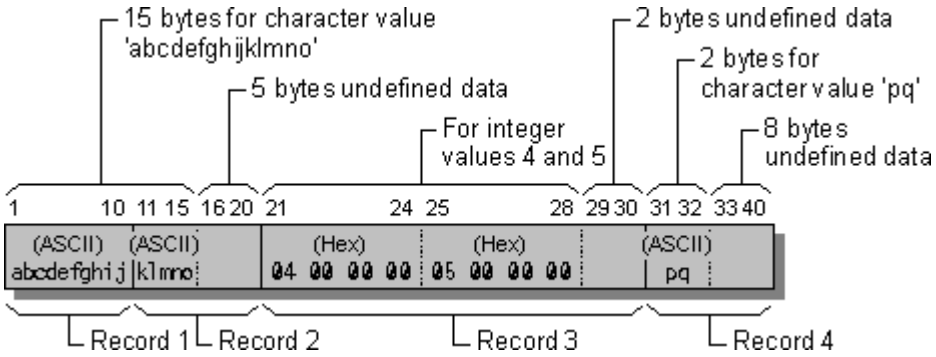
A binary direct file stores records as a series of binary numbers, accessible in any order. Each record in the file has the same length, as specified by the RECL argument to the OPEN statement. You can write partial records to binary direct files; any unused portion of the record will contain undefined data.

A single read or write operation can transfer more data than a record contains by continuing the operation into the next records of the file. Performing such an operation on an unformatted direct file would cause an error. Valid I/O operations for unformatted direct files produce identical results when they are performed on binary direct files, provided the operations do not depend on zero padding in partial records.

The following program creates the binary direct file shown in the following figure:

```
OPEN (3, FILE='BDIR',RECL=10,FORM='BINARY',ACCESS='DIRECT')
WRITE (3, REC=1) 'abcdefghijklmno'
WRITE (3) 4,5
WRITE (3, REC=4) 'pq'
CLOSE (3)
END
```

Binary Direct File



Using Asynchronous I/O

For external files, you can specify that I/O should be asynchronous. By doing this, you allow other statements to execute while an I/O statement is executing.

Note

In order to execute a program that uses asynchronous I/O on Linux* OS or Mac OS* X systems, you must explicitly include one of the following compiler command line options when you compile and link your program:

- `-threads`
- `-reentrancy threaded`
- `-openmp`

On Windows* OS systems, no extra options are needed to execute a program that uses asynchronous I/O.

Using the ASYNCHRONOUS Specifier

Asynchronous I/O is supported for all READ and WRITE operations to external files. However, if you specify asynchronous I/O, you cannot use variable format expressions in formatted I/O operations.

To allow asynchronous I/O for a file, first specify ASYNCHRONOUS='YES' in its OPEN statement, then do the same for each READ or WRITE statement that you want to execute in this manner.

Execution of an asynchronous I/O statement initiates a "pending" I/O operation, which can be terminated in the following ways:

- by an explicit WAIT (initno) statement, which performs a wait operation for the specified pending asynchronous data transfer operation
- by a CLOSE statement for the file
- by a file-positioning statement such as REWIND or BACKSPACE
- by an INQUIRE statement for the file

Use the WAIT statement to ensure that the objects used in the asynchronous data transfer statements are not prematurely deallocated. (This is especially important for local stack objects and allocatable objects which may be deallocated before completion of the pending operation.) If you do not specify the wait operation, the program may terminate with an Access violation error. The following example shows use of the WAIT statement:

```
module mod
   real, allocatable :: X(:)
end module mod
subroutine sbr()
use mod
integer :: Y(500)
   !X and Y initialization
   allocate (X(500))
   call foo1(X, Y)
   !asynchronous writing
   open(1, asynchronous='yes')
   write(1, asynchronous='yes') X, Y
   !some computation
   call foo2()
   !wait operation
   wait(1)
   !X deallocation
   deallocate(X)
   !stack allocated object Y will be deallocated when the routine returns
end subroutine sbr
```

You can use the INQUIRE statement with the keyword of ASYNCHRONOUS (ASYNCHRONOUS=specifier) to determine whether asynchronous I/O is allowed. If it is allowed, a value of YES is returned.

Additionally, you can use the INQUIRE statement with the keyword of PENDING (PENDING= specifier) to determine whether previously pending asynchronous data transfers are complete.

If an ID= specifier appears and the specified data transfer operation is complete, the variable specified by PENDING is assigned the value False and the INQUIRE statement performs a wait operation for the specified data transfer.

If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit are complete, the variable specified by PENDING is assigned the value False and the INQUIRE statement performs wait operations for all previously pending data transfers for the specified unit.

Otherwise, the variable specified by PENDING is assigned the value True and no wait operations are performed. Previously pending data transfers remain pending.

Using the ASYNCHRONOUS Attribute

A data attribute called ASYNCHRONOUS specifies that a variable may be subject to asynchronous input/output. Assigning this attribute to a variable allows certain optimizations to occur.

For more information, see the following topics in the Language Reference:

Asynchronous Specifier

Open: ASYNCHRONOUS Specifier

INQUIRE: ASYNCHRONOUS Specifier

ASYNCHRONOUS Statement and Attributes

Structuring Your Program

Structuring Your Program Overview

There are several ways to organize your projects and the applications that you build with Intel® Fortran. This section introduces several of these options and offers suggestions for when you might want to use them.

See Also

- Creating Fortran Executables
- Using Module (.mod) Files
- Using Include Files
- Advantages of Internal Procedures
- Storing Object Code in Static Libraries
- Storing Routines in Shareable Libraries

Creating Fortran Executables

The simplest way to build an application is to compile all of your Intel® Fortran source files and then link the resulting object files into a single executable file. You can build single-file executables using the ifort command from the command line. For Windows* OS, you can also use the visual development environment.

The executable file you build with this method contains all of the code needed to execute the program, including the run-time library. Because the program resides in a single file, it is easy to copy or install. However, the project contains all of the source and object files for the routines that you used to build the application. If you need to use some of these routines in other projects, you must link all of them again.

Exceptions to this are as follows:

- If you are using shared libraries, all code will not be contained in the executable file.
- On Mac OS* X, the object files contain debug information and would need to be copied along with the executable.

Using Module (.mod) Files

One way to reduce potential confusion when you use the same source code in several projects is to organize the routines into modules. 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 a program's USE statement. Typical applications of modules are the specification of global data or the specification of a derived type and its associated operations.

Modules are excellent ways to organize programs. You can set up separate modules for:

- Commonly used routines
- Data definitions specific to certain operating systems
- System-dependent language extensions

Some programs require modules located in multiple directories. You can use the `-I` (Linux* OS and Mac OS* X) or `/I` compiler (Windows* OS) option when you compile the program to specify the location of the `.mod` files that should be included in the program.

You can use the `-module` *`path`* (Linux OS and Mac OS X) or `/module:path` (Windows OS) option to specify the directory in which to create the module files. If you don't use this option, module files are created in the current directory.

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` (Linux OS and Mac OS X) or `/module:path` (Windows OS) option
3. Current working directory
4. Directories specified by the `-Idir` (Linux OS and Mac OS X) or `/include` (Windows OS) option
5. Directories specified with the `FPATH` (Linux OS and Mac OS X) or `INCLUDE` (Windows OS) environment variable
6. Standard system directories

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` (Linux OS and Mac OS X) or `a.obj` (Windows OS)

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

The following example uses the program program `mod_def.f90` which contains a module defined as follows:

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

Compile the program as follows:

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

This produces the object files `mod_def.o` (Linux OS and Mac OS X) or `mod_def.obj` (Windows OS) and also the .mod file `definedmod.mod`, all in the current directory.

If you need to use the `.mod` file in another directory, do the following:

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

To compile the above program, use the `-I` (Linux OS) or `/I` (Windows OS) option to specify the path to search and locate the `definedmod.mod` file.

Using Include Files

Include files are brought into a program with the `#include` preprocessor directive or a Fortran INCLUDE statement.

Directories are searched for include files in this order:

1. Directory of the source file that contains the include

2. Current working directory
3. Directories specified by the `-I` (Linux OS and Mac OS X) or `/I` (Windows OS) option
4. Directory specified by the `-isystem` option (Linux OS and Mac OS X systems only)
5. Directories specified with the `FPATH` (Linux OS and Mac OS X) or `INCLUDE` (Windows OS) 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.

Specifying and Removing an Include File Path

You can use the `-I` (Linux OS and Mac OS X) or `/I` (Windows OS) option to indicate the location of include files (and also module files).

To prevent the compiler from searching the default path specified by the `FPATH` or the INCLUDE environment variable, use the `-X` or `/noinclude` 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 a specified path instead of the default path, use the following command line:

```
ifort -X -I/alt/include newmain.f (Linux OS and Mac OS X)
ifort /noinclude /IC:/Project2/include newmain.f (Windows OS)
```

Advantages of Internal Procedures

Functions or subroutines that are used in only one program can be organized as internal procedures, following the CONTAINS statement of a program or module.

Internal procedures have the advantage of host association, that is, variables declared and used in the main program are also available to any internal procedure it may contain. For more information on procedures and host association, see Program Units and Procedures.

Internal procedures, like modules, provide a means of encapsulation. Where modules can be used to store routines commonly used by many programs, internal procedures separate functions and subroutines whose use is limited or temporary.

Storing Object Code in Static Libraries

Another way to organize source code used by several projects is to build a static library (for Windows* OS, `.lib` and for Linux* OS and Mac OS* X, `.a`) containing the object files for the reused procedures. You can create a static library by doing the following:

- From the Microsoft Visual Studio* integrated development environment (IDE), create and build a Fortran Static Library project type.
- From the command line, use the `ar` command (on Linux OS and Mac OS X) or the `LIB` command (on Windows OS).

After you have created a static library, you can use it as input to other types of Intel Fortran projects.

Storing Routines in Shareable Libraries

You can organize the code in your application by storing the executable code for certain routines in a shareable library (`.dll` for Windows* OS, `.so` for Linux* OS, `.dylib` for Mac OS*

X). You can then build your applications so that they call these routines from the shareable library.

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

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. It allows you to:

- Call existing code that is written in another language
- Use procedures that may be difficult to implement in a particular language
- 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.

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. If the main program is C/C++, you need to use the `-nofor_main` compiler option to indicate this.

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++.

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.

Mixed-language programming is possible with Intel® Fortran, Visual C/C++*, and Intel® C++, because each language implements functions, subroutines, and procedures in approximately the same way.

Intel Fortran includes several Fortran 2003 features that provide interoperability with C. An entity is considered interoperable if equivalent declarations can be made for it in both languages. Interoperability is provided for variables, derived types, and procedures. The following features are supported:

- BIND attribute and statement, which specifies that an object is interoperable with C and has external linkage.
- language binding in FUNCTION and SUBROUTINE statements
- language binding in derived-type statements

For more information, see Interoperability with C.

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. On Linux OS and Mac OS X systems, if Fortran is not the main routine, the `-nofor-main` compiler option must be specified on the command line.
- To use the same Microsoft* visual development environment for multiple languages, you must have the same version of the visual development environment for your languages.
- 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, CHARACTER, and derived types.
- On Linux* systems, the `-fexceptions` option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.

For more information on mixed language programming using Intel Fortran and C, see the following:

- Compiling and Linking Intel Fortran/C Programs
- Calling C Procedures from an Intel Fortran Program

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.

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:

- Stack considerations
  - Does a routine receive a varying or fixed number of arguments?
  - Which routine clears the stack after a call?
- 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 a number of 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 following is applies to Windows OS only:** The caller routine and the called routine use a calling convention to determine which of them is responsible for adjusting the stack in order to remove arguments when the execution of the called routine is complete. You can specify these conventions with ATTRIBUTES (*c*DEC$ ATTRIBUTES compiler directive) options such as C or STDCALL.
3. The caller routine and the called routine use a calling convention to select the option of passing a variable number of arguments.
4. 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.
5. 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.

ATTRIBUTES Properties and Calling Conventions

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

- The C, STDCALL, REFERENCE, and VARYING properties for an entire routine.
- The VALUE and REFERENCE properties for individual arguments.

By default, Fortran passes all data by reference (except the hidden length argument of strings, which is passed by value). If the C (or, for Windows OS, STDCALL) option is used, the default changes to passing almost all data except arrays by value. However, in addition to the calling-convention options C and STDCALL, you can specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference, regardless of the calling convention option. Arrays can only be passed by reference.

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

It is advisable to use the DECORATE option in combination with the ALIAS option to ensure appropriate name decoration regardless of operating system or architecture. The DECORATE option indicates that the external name specified in ALIAS should have the correct prefix and postfix decorations for the calling mechanism in effect.

Naming conventions are as follows:

- leading (prefix) underscore for Windows operating systems based on IA-32 architecture; no underscores for Windows operating systems based on Intel® 64 architecture and Windows systems based on IA-64 architecture.
- trailing (postfix) underscore for all Linux operating systems
- leading and trailing underscores for all Mac OS* X operating systems

For example:

```
INTERFACE
   SUBROUTINE MY_SUB (I)
       !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'My_Sub' :: MY_SUB
       INTEGER I
   END SUBROUTINE MY_SUB
END INTERFACE
```

This code declares a subroutine named MY_SUB with the C property. The external name will be appropriately decorated for the operating system and platform.

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

Calling Conventions for ATTRIBUTES Options

Argument	Default	C	C, REFERENCE	STDCALL (Windows OS IA-32 architecture)	STDCALL, REFERENCE (Windows OS IA-32 architecture)
Scalar	Reference	Value	Reference	Value	Reference
Scalar [value]	Value	Value	Value	Value	Value
Scalar [reference]	Reference	Reference	Reference	Reference	Reference
String	Reference, either Len:End or Len:Mixed	String(1:1)	Reference, either Len:End or Len:Mixed	String(1:1)	String(1:1)
String [value]	Error	String(1:1)	String(1:1)	String(1:1)	String(1:1)
String [reference]	Reference, either No Len or Len:Mixed	Reference, No Len	Reference, No Len	Reference, No Len	Reference, No Len
Array	Reference	Reference	Reference	Reference	Reference
Array [value]	Error	Error	Error	Error	Error
Array [reference]	Reference	Reference	Reference	Reference	Reference
Derived Type	Reference	Value, size dependent	Reference	Value, size dependent	Reference
Derived Type	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent

[value]					
Derived Type [reference]	Reference	Reference	Reference	Reference	Reference
F90 Pointer	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
F90 Pointer [value]	Error	Error	Error	Error	Error
F90 Pointer [reference]	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
Naming Conventions					
Prefix	_ (Windows operating systems using IA-32 architecture, Mac OS X operating systems) *none* for all others	_ (Windows operating systems using IA-32 architecture, Mac OS X operating systems) *none* for all others	_ (Windows operating systems using IA-32 architecture, Mac OS X operating systems) *none* for all others	_	_
Suffix	*none* (Windows OS) _ (Linux OS, Mac OS X)	*none*	*none*	*@n*	*@n*
Case	Upper Case	Lower Case	Lower Case	Lower Case	Lower Case
Stack Cleanup	Caller	Caller	Caller	Callee	Callee

The terms in the above table mean the following:

[value] Argument assigned the VALUE attribute.

[reference] Argument assigned the REFERENCE attribute.

Value The argument value is pushed on the stack. All values are padded to the next 4-byte boundary.

Reference On systems using IA-32 architecture, the 4-byte argument address is pushed on the stack.
On systems using Intel® 64 and IA-64 architectures, the 8-byte argument address is pushed on the stack.

Len:End or Len:Mixed	For certain string arguments: • Len:End applies when -nomixed-str-len-arg (Linux OS and Mac OS X) or /iface:nomixed_str_len_arg (Windows) is set. The length of the string is pushed (by value) on the stack after all of the other arguments. This is the default. • Len:Mixed applies when -mixed-str-len-arg (Linux OS and Mac OS X) or /iface:mixed_str_len_arg (Windows) is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
No Len or Len:Mixed	For certain string arguments: • No Len applies when nomixed-str-len-arg (Linux OS and Mac OS X) or /iface:nomixed_str_len_arg (Windows) is set. The length of the string is not available to the called procedure. This is the default. • Len:Mixed applies when mixed-str-len-arg (Linux OS and Mac OS X) or /iface:mixed_str_len_arg (Windows) is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
No Len	For string arguments, the length of the string is not available to the called procedure.
String(1:1)	For string arguments, the first character is converted to INTEGER(4) as in ICHAR(string(1:1)) and pushed on the stack by value.
Error	Produces a compiler error.
Descriptor	On systems using IA-32 architecture, the 4-byte address of the array descriptor. On systems using Intel® 64 architecture and systems using IA-64 architecture, the 8-byte address of the array descriptor.
@n	On systems using IA-32 architecture, the at sign (@) followed by the number of bytes (in decimal) required for the argument list.
Size dependent	On systems using IA-32 architecture, derived-type arguments specified by value are passed as follows: • Arguments from 1 to 4 bytes are passed by value. • Arguments from 5 to 8 bytes are passed by value in two registers (two arguments). • Arguments more than 8 bytes provide value semantics by passing a temporary storage address by reference.
Upper Case	Procedure name in all uppercase.
Lower Case	Procedure name in all lowercase.
Callee	The procedure being called is responsible for removing arguments from the

stack before returning to the caller.

Caller — The procedure doing the call is responsible for removing arguments from the stack after the call is over.

The following table shows which Fortran ATTRIBUTES options match other language calling conventions.

Matching Calling Conventions

Other Language Calling Convention	Matching ATTRIBUTES Option
C/C++ cdecl (default)	C
C/C++ __stdcall (Windows OS only)	STDCALL
MASM C (in PROTO and PROC declarations) (Windows OS only)	C
MASM STDCALL (in PROTO and PROC declarations) (Windows OS only)	STDCALL
Assembly (Linux OS only)	C

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

For Windows operating systems, the compiler option `/iface` also establishes some default argument passing conventions. The `/iface` option has the following choices:

Option	How are arguments passed?	Append @n to names on systems using IA-32 architecture?	Who cleans up stack?	Varargs support?
`/iface:cref`	By reference	No	Caller	Yes
`/iface:stdref`	By reference	Yes	Callee	No
`/iface:default`	By reference	No	Caller	Yes
`/iface:c`	By value	No	Caller	Yes
`/iface:stdcall`	By value	Yes	Callee	No
`/iface:cvf`	By reference	Yes	Callee	No

Adjusting Naming Conventions in Mixed-Language Programming Overview

The ATTRIBUTES options C and, for Windows* OS, STDCALL, determine 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 an .OBJ 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, and MASM), 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.

C/C++ Naming Conventions

C and C++ preserve case sensitivity in their symbol tables while Fortran by default does not; this is 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 case conversion of names 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 ); }
```

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 for Fortran, C, C++, and MASM

The following table summarizes how Fortran, C/C++ and, for Windows, MASM handle procedure names. Note that for MASM, the table does not apply if the CASEMAP: ALL option is used.

Naming Conventions in Fortran, C, Visual C++, and MASM

Language	Attributes	Name Translated As	Case of Name in .OBJ File
Fortran	*c*DEC$ ATTRIBUTES C	*name* (Linux* OS) *_name* (Windows* OS) *_name* (Mac OS* X)	All lowercase
Fortran (Windows OS)	*c*DEC$ ATTRIBUTES STDCALL	*_name@n*	All lowercase
Fortran	default	*name_* (Linux OS) *_name* (Windows OS) *_name_* (Mac OS X)	All uppercase
C	cdecl (default)	*name* (Linux OS) *_name* (Windows OS) _name (Mac OS X)	Mixed case preserved
C (Windows OS only)	__stdcall	*_name@n*	Mixed case preserved
C++	Default	*name@@decoration* (Linux OS) *_name@@decoration* (Windows OS) __decoration*name*docration (Mac OS X)	Mixed case preserved
Linux OS, Mac OS X Assembly	Default	*name* (Linux OS) *_name* (Mac OS X)	Mixed case preserved
MASM (Windows OS)	C (in PROTO and PROC declarations)	*_name*	Mixed case preserved
MASM (Windows OS)	STDCALL (in PROTO and PROC declarations)	*_name@n*	Mixed case preserved

In the preceding table:

- The leading underscore (such as *_name*) is used on Windows operating systems based on IA-32 architecture only.
- *@n* represents the stack space, in decimal notation, occupied by parameters on Windows operating systems based on IA-32 architecture only.

For example, assume a function is declared in C as:

```
extern int __stdcall Sum_Up( int a, int b, int c );
```

Each integer occupies 4 bytes, so the symbol name placed in the .OBJ file on systems based on IA-32 architecture is:

```
_Sum_Up@12
```

On systems based on Intel® 64 architecture and those based on IA-64 architecture, the symbol name placed in the .OBJ file is:

```
Sum_Up
```

Reconciling the Case of Names

The following summarizes how to reconcile names between languages:

- All-Uppercase Names (default on Windows OS)

  If you call a Fortran routine that uses Fortran defaults and cannot recompile the Fortran code, then in C you must use an all-uppercase name to make the call. In MASM you must either use an all-uppercase name or set the OPTION CASEMAP directive to ALL, which translates all identifiers to uppercase. Use of the __stdcall convention in C code or STDCALL in MASM PROTO and PROC declarations is not enough, because __stdcall and STDCALL always preserve case in these languages. Fortran generates all-uppercase names by default and the C or MASM code must match it.

  For example, these prototypes establish the Fortran function FFARCTAN(angle) where the argument angle has the ATTRIBUTES VALUE property:

  - In C:
    ```
    extern float FFARCTAN( float angle );
    ```
  - In MASM:
    ```
    .MODEL FLAT
    FFARCTAN PROTO, angle: REAL4
    ...
    FFARCTAN PROC, angle: REAL4
    ```
- All-Lowercase Names (default on Linux OS and Mac OS X)

  If the name of the routine appears as all lowercase in C or assembly, then naming conventions are automatically correct. Any case may be used in the Fortran source code, including mixed case since the name is changed to all lowercase.
  In Linux OS/Mac OS X Assembly, the following establishes the Fortran function `ffarctan`:
  ```
  #--Begin ffarctan_
  .globl ffarctan_
  ```

- Mixed-Case Names

  If the name of a routine appears as mixed-case in C or MASM 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 object file.

The following is an example for referring to the C function My_Proc:

```
!DEC$ ATTRIBUTES DECORATE,ALIAS:'My_Proc' :: My_Proc
```

This example uses DECORATE to automatically reconcile the external name for the target platform.

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_(Linux OS and Mac OS X)
_MODULENAME_mp_ENTITY [ @stacksize ] (Windows OS)
```

`modulename` is the name of the module. For Windows* operating systems, the name is uppercase by default.

`entity` is the name of the module procedure or module data contained within *MODULENAME*. For Windows* operating systems, *ENTITY* is uppercase by default.

`_mp_` is the separator between the module and entity names and is always lowercase.

For example:

```
MODULE mymod
   INTEGER a
CONTAINS
   SUBROUTINE b (j)
      INTEGER j
   END SUBROUTINE
END MODULE
```

This results in the following symbols being defined in the compiled object file on Linux operating systems (On Mac OS X operating systems, the symbols would begin with an underscore)::

```
mymod_mp_a_
mymod_mp_b_
```

The following symbols are defined in the compiled object file on Windows operating systems based on IA-32 architecture:

```
_MYMOD_mp_A
_MYMOD_mp_B
```

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

Note

Except for ALIAS, ATTRIBUTES options do not affect the module name.

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

Effect of ATTRIBUTES options on Fortran Module Names

ATTRIBUTES Option Given to Routine 'b'	Procedure Name in .OBJ file on Systems Using IA-32 Architecture	Procedure Name in .OBJ file on Systems Using Intel® 64 Architecture and IA-64 Architecture
None	mymod_mp_b_ (Linux OS) _mymod_mp_b_ (Mac OS X) _MYMOD_mp_B (Windows OS)	mymod_mp_b_ (Linux OS) _mymod_mp_b_ (Mac OS X) MYMOD_mp_B (Windows OS)
C	mymod_mp_b_(Linux OS) _mymod_mp_b_(Mac OS X) MYMOD_mp_b (Windows OS)	mymod_mp_b (Linux OS) _mymod_mp_b (Mac OS X) MYMOD_mp_b (Windows OS)
STDCALL (Windows OS only)	_MYMOD_mp_b@4	MYMOD_mp_b
ALIAS	Overrides all others, name as given in the alias	Overrides all others, name as given in the alias
VARYING	No effect on name	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 or STDCALL convention in Fortran, and if defining a module in another language, using the ALIAS option to match the name within Fortran. For examples, see Using Modules in 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 for a more detailed 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, and, for Windows, STDCALL) 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, DECORATE, 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

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:

- Passing Arguments in Mixed-Language Programming
- Using Modules in Mixed-Language Programming
- Using Common External Data in Mixed-Language Programming

Passing Arguments in Mixed-Language Programming

You can pass data between Fortran and C, C++, and MASM 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 systems based on IA-32 architecture, Fortran, C, and C++ use 4-byte addresses. On systems based on Intel® 64 architecture and those based on IA-64 architecture, these languages use 8-byte addresses.

You need to make sure that for every call, the calling program and the called routine agree on how each argument is passed. Otherwise, the called routine receives bad data.

The Fortran technique for passing arguments changes depending on the calling convention specified. By default, Fortran passes all data by reference (except the hidden length argument of strings, which is passed by value).

If the ATTRIBUTES C option or, for Windows OS, the STDCALL 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 or STDCALL option, all arguments by default are passed by reference.

You can specify also 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.

Note

On Windows operating systems, the compiler option `/iface` also establishes some default argument passing conventions (such as for hidden length of strings). See ATTRIBUTES Properties and Calling Conventions.

Examples of passing by reference and value for C and MASM 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:

For Linux OS:

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

For Windows OS:

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

In MASM (Windows OS only), arguments are passed by value by default. Arguments to be passed by reference are designated with PTR in the PROTO and PROC directives. For example:

```
TESTPROC PROTO, valparm: SDWORD, refparm: PTR SDWORD
```

To use an argument passed by value, use the value of the variable. For example:

```
mov eax, valparm ; Load value of argument
```

This statement places the value of valparm into the EAX register.

To use an argument passed by reference, use the address of the variable. For example:

```
mov ecx, refparm ; Load address of argument
mov eax, [ecx]   ; Load value of argument
```

These statements place the value of refparm into the EAX register.

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 (or, for Windows OS, STDCALL) option	Scalars and derived types	REFERENCE option	Default
	Default	Arrays	Default	Cannot pass by value
	C (or, for Windows OS, STDCALL) option	Arrays	Default	Cannot pass by value
C/C++		Non-arrays	Pointer argument_name	Default
		Arrays	Default	Struct {type} array_name
Assembler	MASM (Windows OS only)	All types	PTR	Default

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` (Linux OS and Mac OS X) or `/iface:mixed_str_len_arg` (Windows OS) is set immediately after the address of the beginning of the string. It also depends on whether `-nomixed-str-len-arg` (Linux OS and Mac OS X) or `/iface:nomixed_str_len_arg` (Windows OS) is set after all arguments.

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 Array Pointers and Allocatable Arrays and Handling Character Strings.

See also:

Handling Character Strings
Handling Numeric, Complex, and Logical Data Types

Using Modules in 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.

The 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
```

The 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:

The C code:

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

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 following Fortran code defines 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 following Fortran code calls this routine using the module CPROC:

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

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.

This section applies only to Fortran/C and, on Windows, Fortran/MASM mixed-language programs.

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 or MASM by declaring it as global (or COMMON) in one language and accessing it as an external variable in the other language.

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)
```

MASM declares a parameter global (PUBLIC) with the syntax:

PUBLIC [*langtype*] *name*

where *name* is the name of the global variable to be referenced, and the optional *langtype* is STDCALL or C. The option *langtype*, if present, overrides the calling convention specified in the .MODEL directive.

Conversely, 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.)

MASM and Assembly can also access Fortran global (COMMON) parameters with the ATTRIBUTES EXTERN directive. The syntax is:

EXTERN [*langtype*] *name*

where *name* is the name of the global variable to be referenced, and the optional *langtype* is STDCALL or C.

Using Fortran Common Blocks and C Structures

To reference C structures from Fortran common blocks and vice versa, you must take into account how common blocks and structures differ in their methods of storing member variables in memory. Fortran places common block variables into memory in order as close together as possible, with the following rules:

- A single BYTE, INTEGER(1), LOGICAL(1), or CHARACTER variable in common block list begins immediately following the previous variable or array in memory.
- All other types of single variables begin at the next even address immediately following the previous variable or array in memory.
- All arrays of variables begin on the next even address immediately following the previous variable or array in memory, except for CHARACTER arrays which always follow immediately after the previous variable or array.
- All common blocks begin on a four-byte aligned address.

Because of these padding rules, you must consider the alignment of C structure elements with Fortran common block elements and assure matching either by making all variables exactly equivalent types and kinds in both languages (using only 4-byte and 8-byte data types in both languages simplifies this) or by using the C pack pragmas in the C code around the C structure to make C data packing like Fortran data packing. For example:

```
#pragma pack(2)
struct {
        int N;
        char INFO[30];
} examp;
#pragma pack()
```

To restore the original packing, you must add #pragma pack( ) at the end of the structure. (Remember: Fortran module data can be shared directly with C structures with appropriate naming.)

Once you have dealt with alignment and padding, you can give C access to an entire common block or set of common blocks. Alternatively, you can pass individual members of a Fortran common block in an argument list, just as you can any other data item. 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 structures:

```
#pragma pack(2)
extern struct {
  float x, y, z[6];
  double ydbl;
} Really;
#pragma pack()
```

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++ module 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:

```
//
#pragma pack(2)
struct block_type
{
  int n;
  double x;
  double y;
};
#pragma pack()
//
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 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 among Fortran, C, and MASM:

Equivalent Data Types

Fortran Data Type	C Data Type	MASM Data Type
REAL(4)	float	REAL4
REAL(8)	double	REAL8
REAL(16)	---	---
CHARACTER(1)	unsigned char	BYTE
CHARACTER*(*)	See Handling Character Strings	

COMPLEX(4)	struct complex4 {
	float real, imag;
	};
		COMPLEX4 STRUCT 4
		real REAL4 0
		imag REAL4 0
		COMPLEX4 ENDS

COMPLEX(8)	struct complex8 { double real, imag;	COMPLEX8 STRUCT 8 real REAL8 0

	};	imag REAL8 0 COMPLEX8 ENDS
COMPLEX(16)	---	---
All LOGICAL types	Use integer types for C, MASM	
INTEGER(1)	char	.sbyte
INTEGER(2)	short	.sword
INTEGER(4)	int	.sdword
INTEGER(8)	_int64	.qword

The following sections describe how to reconcile data types between the different languages:

- Handling Numeric, Complex, and Logical Data Types
- Handling Fortran 90 Array Pointers and Allocatable Arrays
- Handling Integer 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, MASM, and C/C++.

C, C++, and MASM 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

On systems based on IA-32 architecture, 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;
};
```

The MASM structure definitions for the Fortran COMPLEX types follow:

```
 COMPLEX4 STRUCT 4
   real REAL4 0
   imag REAL4 0
 COMPLEX4 ENDS
 COMPLEX8 STRUCT 8
   real REAL8 0
   imag REAL8 0
 COMPLEX8 ENDS
```

A Fortran LOGICAL(2) is stored as a 2-byte indicator value (0=false, and the `-fpscomp [no]logicals` (Linux* OS and Mac OS* X) or `/fpscomp:[no]logicals` (Windows* OS) 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 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` or `DOUBLE COMPLEX` 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

Linux example:

```
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;
}
```

Windows OS example:

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

void WBAT (_single_complex location, float *x, float *y)
{
float realpart;
float imaginarypart;
... program text, producing realpart and
imaginarypart...
*location.real = realpart;
*location.imag = imaginarypart;
}
```

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 .
- 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

Intel® Fortran integer pointers (also known as Cray*-style pointers) are not the same as Fortran 90 pointers, but are instead like C pointers. On systems based on IA-32 architecture, integer pointers are 4-byte INTEGER quantities. On systems based on Intel® 64 architecture and those based on IA-64 architecture, integer pointers are 8-byte INTEGER quantities.

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.

Fortran main program:

```
! Fortran main program.
INTERFACE
SUBROUTINE Ptr_Sub (p)
!DEC$ ATTRIBUTES C, DECORATE, 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
!
```

On systems using Intel® 64 architecture and IA-64 architecture, the declaration for `p` in the INTERFACE block is equivalent to INTEGER(8) p and on systems using IA-32 architecture, it is equivalent to INTEGER (4) p.

C subprogram:

```
   //C subprogram
void Ptr_Sub (float *p)
{
p[3] = 23.5;
}
```

When the main Fortran program and C function are built and executed, the following output appears:

```
A(4) = 23.50000
```

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 variable of the appropriate data type (matching the data type of the passing routine). When inside the Fortran routine, use the pointee variable to set and access what the pointer points to.

Fortran subroutine:

```
! Fortran subroutine.
     SUBROUTINE Iptr_Sub (p)
     !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'Iptr_Sub' :: Iptr_Sub


      INTEGER (KIND=INT_PTR_KIND()) p
        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 )
```

C Main Program:

```
//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. 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.

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)

Handling Arrays in Visual Basic and MASM

The following information on Visual Basic and MASM applies to Windows operating systems only.

To pass an array from Visual Basic to Fortran, pass the first element of the array. By default, Visual Basic passes variables by reference, so passing the first element of the array will give Fortran the starting location of the array, just as Fortran expects. Visual Basic indexes the first array element as 0 by default, while Fortran by default indexes it as 1. Visual Basic indexing can be set to start with 1 using the statement:

```
Option Base 1
```

Alternatively, in the array declaration in either language you can set the array lower bound to any integer in the range -32,768 to 32,767. For example:

```
' In Basic
Declare Sub FORTARRAY Lib "fortarr.dll" (Barray as Single)
DIM barray (1 to 3, 1 to 7) As Single
Call FORTARRAY(barray (1,1))
! In Fortran
Subroutine FORTARRAY(arr)
   REAL arr(1:3,1:7)
```

In MASM, arrays are one-dimensional and array elements must be referenced byte-by-byte. The assembler stores elements of the array consecutively in memory, with the first address referenced by the array name. You then access each element relative to the first, skipping the total number of bytes of the previous elements. For example:

```
xarray    REAL4     1.1, 2.2, 3.3, 4.4 ; initializes
                      ; a four element array with
                      ; each element 4 bytes
```

Referencing xarray in MASM refers to the first element, the element containing 1.1. To refer to the second element, you must refer to the element 4 bytes beyond the first with xarray[4] or xarray+4. Similarly:

```
yarray     BYTE      256 DUP     ; establishes a
                ; 256 byte buffer, no initialization
zarray SWORD 100 DUP(0) ; establishes 100
                ; two-byte elements, initialized to 0
```

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 array

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), using an *implicit* interface allows the array argument to be passed *without* an Intel Fortran descriptor. However, for cases where the called routine needs the information in the Intel Fortran descriptor, declare the routine with an *explicit* interface and specify the dummy array as either an assumed-shape array or with the pointer attribute.

You can associate a Fortran 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.
- If the descriptor was built by the compiler, it cannot be modified by the user. Changing fields of existing descriptors is illegal.

The components of the current Intel Fortran array descriptor on systems using IA-32 architecture are as follows:

- 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 A0 offset. The A0 offset is added to the base address to calculate the address for the element with all indices zero, even if that is outside the bounds of the actual array. This is helpful in computing array element addresses.
- The fourth longword (bytes 12 to 15) contains a set of flags used to store information about the array. This includes:
    - bit 1 (0x01): array is defined -- set if the array has been defined (storage allocated)
    - bit 2 (0x02): no deallocation allowed -- set if the array pointed to cannot be deallocated (that is, it is an explicit array target)
    - bit 3 (0x04): array is contiguous -- set if the array pointed to is a contiguous whole array or slice.
- 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 and should not be explicitly set.
- The remaining longwords (bytes 24 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) contains 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) contains the A0 offset of -112.
- The fourth longword (bytes 12 to 15) contains 3 (array is defined and deallocation is not allowed).

- The fifth longword (bytes 16 to 19) contains 2 (rank).
- The sixth longword (bytes 20-23) 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 systems using Intel® 64 architecture and those using IA-64 architecture is identical to that on systems using IA-32 architecture, except that all fields are 8-bytes long, instead of 4-bytes.

Handling Large Arrays

When compiling a program with an array greater than 2GB on systems using Intel® 64 architecture and running Linux OS, you may need to specify certain compiler options. Specifically, you may need to use either the `-mcmodel=medium` or `-mcmodel=large` compiler option and also the `-shared-intel` option. For more information, see Specifying Memory Models to use with Systems Based on Intel® 64 Architecture.

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 (for systems based on IA-32 architecture) or unsigned 8-byte integer (for systems based on Intel® 64 architecture and those based on IA-64 architecture), 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 Options on Character Strings Passed as Arguments

Argument	Default	C	C, REFERENCE	STDCALL (Windows* OS)	STDCALL, REFERENCE (Windows* OS)
String	Passed by reference, along with length	First character converted to INTEGER(4) and passed by value	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	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	Passed by reference, no length	Passed by reference, no length

The important things to note about the above table are:

- Character strings without the VALUE or REFERENCE attribute that are passed to C or STDCALL routines 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 C or STDCALL routines are not passed by reference. Instead, only the value of the first character is passed.
- For string arguments with default ATTRIBUTES, ATTRIBUTES C, REFERENCE, or ATTRIBUTES STDCALL, REFERENCE:
    - When `-nomixed-str-len-arg` (Linux OS and Mac OS X) or `/iface:nomixed_str_len_arg` (Windows OS) is set, the length of the string is pushed (by value) on the stack after all of the other arguments. This is the default.
    - When `-mixed-str-len-arg` (Linux OS and Mac OS X) or `/iface:mixed_str_len_arg` (Windows OS) 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` (Linux OS and Mac OS X) or `/iface:nomixed_str_len_arg` is set, the length of the string is not available to the called procedure. This is the default.
    - When `-mixed-str-len-arg` (Linux OS and Mac OS X) or `/iface: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, DECORATE,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:

```
   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. An added benefit for using C strings on Windows* operating systems is that Windows API routines and most C library functions expect null-terminated strings.

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.

Handling Character Strings in Visual Basic and MASM

The following information on Visual Basic and MASM applies to Windows operating systems only.

Visual Basic strings must be passed by value to Fortran. Visual Basic strings are actually stored as structures containing length and location information. Passing by value dereferences the structure and passes just the string location, as Fortran expects. For example:

```
! In Basic
 Declare Sub forstr  Lib "forstr.dll" (ByVal Bstring as String)
 DIM bstring As String * 40 Fixed-length string
```

```
 CALL forstr(bstring)
! End Basic code
! In Fortran
 SUBROUTINE forstr(s)
 !DEC$ ATTRIBUTES STDCALL :: forstr
 !DEC$ ATTRIBUTES REFERENCE :: s
 CHARACTER(40) s
 s = 'Hello, Visual Basic!'
 END
```

The Fortran directive !DEC$ ATTRIBUTES STDCALL and the ATTRIBUTES REFERENCE property on variable arguments together inform Fortran not to expect the hidden length arguments to be passed from the Visual Basic calling program. The name in the Visual Basic program is specified as lowercase since STDCALL makes the Fortran name lowercase.

MASM does not add either a string length or a null character to strings by default. To append the string length, use the syntax:

```
lenstring BYTE "String with length", LENGTHOF lenstring
```

To add a null character, append it by hand to the string:

```
nullstring BYTE "Null-terminated string", 0
```

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 blank spaces if necessary.

The called routine must copy its result through the address specified in the first argument. The following example shows the Fortran code for a return character function called `MAKECHARS` and a 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`.
- On Windows, you must use uppercase names for C routines or Microsoft attributes and `INTERFACE` blocks to make the calls using lower case.

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;
```

Intel(R) Fortran/C Mixed-Language Programs

Interoperability with C

It is often desirable to have a program which contains both Fortran and C code, and in which routines written in one language are able to call routines written in the other. The Intel® Fortran compiler supports the Fortran 2003 standardized mechanism for allowing Fortran code to reliably communicate (or *interoperate*) with C code. The following describes interoperability requirements for types, variables, and procedures.

Interoperability of Intrinsic Types

The intrinsic module ISO_C_BINDING contains named constants that hold kind type parameter values for intrinsic types.

The more commonly used types are included in the following table. The following applies:

- Integer types in Fortran are always signed. In C, integer types may be specified as signed or unsigned, but are unsigned by default.
- The values of C_LONG, C_SIZE_T, C_LONG_DOUBLE, AND C_LONG_DOUBLE_COMPLEX are different on different platforms.

Named constant from ISO_C_BINDING (kind type parameter if value is positive)	**C type**	**Equivalent Fortran type**
C_SHORT	short int	INTEGER(KIND=2)
C_INT	int	INTEGER(KIND=4)
C_LONG	long int	INTEGER (KIND=4 or 8)
C_LONG_LONG	long long int	INTEGER(KIND=8)
C_SIGNED_CHAR	signed char unsigned char	INTEGER(KIND=1)
C_SIZE_T	size_t	INTEGER(KIND=4 or 8)
C_INT8_T	int8_t	INTEGER(KIND=1)
C_INT16_T	int16_t	INTEGER(KIND=2)
C_INT32_T	int32_t	INTEGER(KIND=4)
C_INT64_T	int64_t	INTEGER(KIND=8)
C_FLOAT	float	REAL(KIND=4)
C_DOUBLE	double	REAL(KIND=8)
C_LONG_DOUBLE	long double	REAL(KIND=8 or 16)
C_FLOAT_COMPLEX	float _Complex	COMPLEX(KIND=4)
C_DOUBLE_COMPLEX	double _Complex	COMPLEX(KIND=8)
C_LONG_DOUBLE_COMPLEX	long double _Complex	COMPLEX(KIND=8 or 16)
C_BOOL	_Bool	LOGICAL(KIND=1)
C_CHAR	char	CHARACTER(LEN=1)

While there are named constants for all possible C types, every type is not necessarily supported on every processor. Lack of support is indicated by a negative value for the constant in the module.

For a character type to be interoperable, you must either omit the length type parameter or specify it using an initialization expression whose value is one.

Interoperability with C Pointers

For interoperating with C pointers, the module ISO_C_BINDING contains the derived types C_PTR and C_FUNPTR, which are interoperable with C object and function type pointers, respectively.

These types, as well as certain procedures in the module, provide the mechanism for passing dynamic arrays between the two languages. Because its elements need not be contiguous in memory, a Fortran pointer target or assumed-shape array cannot be passed to C. However, you can pass an allocated allocatable array to C, and you can associate an array allocated in C with a Fortran pointer.

Interoperability of Derived Types

For a derived type to be interoperable with C, you must specify the BIND(C) attribute:

```
TYPE, BIND(C) :: MYTYPE
```

Additionally, as shown in the examples that follow, each component must have an interoperable type and interoperable type parameters, must not be a pointer, and must not be allocatable. This allows Fortran and C types to correspond.

```
typedef struct {
int m, n;
float r;
} myctype
```

The above is interoperable with the following:

```
USE, INTRINSIC :: ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
INTEGER(C_INT) :: I, J
REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

Interoperability of Variables

A scalar Fortran variable is interoperable if its type and type parameters are interoperable and it is not a pointer.

An array Fortran variable is interoperable if its type and type parameters are interoperable and it has an explicit shape or assumed size. It interoperates with a C array of the same type, type parameters, and shape, but with subscripts reversed.

For example, a Fortran array declared as `INTEGER :: A(18, 3:7, *)` is interoperable with a C array declared as `int b[][5][18]`.

Interoperability of Procedures

For a procedure to be interoperable, it must have an explicit interface and be declared with the BIND attribute, as shown in the following:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C)
```

In the case of a function, the result must be scalar and interoperable.

A procedure has an associated binding label, which is global in scope. This label is the name by which the C processor knows it and is, by default, the lower-case version of the Fortran name. For example, the above function has the binding label func. You can specify an alternative binding label as follows:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C, NAME='myC_Func')
```

All dummy arguments must be interoperable. Furthermore, you must ensure that either the Fortran routine uses the VALUE attribute for scalar dummy arguments, or that the C routine receives these scalar arguments as pointers to the scalar values. Consider the following call to this C function:

```
intc_func(int x, int *y);
```

As shown here, the interface for the Fortran call to c_func must have x passed with the VALUE attribute, but y should not have the VALUE attribute, since it is received as a pointer:

```
INTERFACE
INTEGER (C_INT) FUNCTION C_FUNC(X, Y) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER (C_INT), VALUE :: X
INTEGER (C_INT) :: Y
END FUNCTION C_FUNC
END INTERFACE
```

Alternatively, the declaration for y can be specified as a C_PTR passed by value:

```
TYPE (C_PTR), VALUE :: Y
```

To pass a scalar Fortran variable of type character, the character length must be one.

Interoperability of Global Data

A module variable or a common block can interoperate with a C global variable if the Fortran entity uses the BIND attribute and the members of that entity are also interoperable. For example, consider the entities C_EXTERN, C2, COM and SINGLE in the following module:

```
MODULE LINK_TO_C_VARS
USE, INTRINSIC :: ISO_C_BINDING
INTEGER(C_INT), BIND(C) :: C_EXTERN
INTEGER(C_LONG) :: C2
BIND(C, NAME='myVariable') :: C2
COMMON /COM/ R,S
REAL(C_FLOAT) :: R,S,T
BIND(C) :: /COM/, /SINGLE/
COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS
```

These can interoperate with the following C external variables:

```
int c_extern;
long myVariable;
struct {float r, s;} com;
float single;
```

Example of Fortran Calling C

The following example calls a C function.

C Function Prototype:

```
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts);
```

Fortran Modules:

```
MODULE FTN_C_1
USE, INTRINSIC :: ISO_C_BINDING
END MODULE FTN_C_1
```

```
MODULE FTN_C_2
INTERFACE
INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
(SENDBUF, SENDCOUNT, RECVCOUNTS) &
BIND(C, NAME='C_Library_Function')
USE FTN_C_1
IMPLICIT NONE
TYPE (C_PTR), VALUE :: SENDBUF
INTEGER (C_INT), VALUE :: SENDCOUNT
TYPE (C_PTR), VALUE :: RECVCOUNTS
END FUNCTION C_LIBRARY_FUNCTION
END INTERFACE
END MODULE FTN_C_2
```

Fortran Calling Sequence:

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
USE FTN_C_2
...
REAL (C_FLOAT), TARGET :: SEND(100)
INTEGER (C_INT) :: SENDCOUNT
INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(100)
...
ALLOCATE( RECVCOUNTS(100) )
...
CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
C_LOC(RECVCOUNTS))
...
```

Example of C Calling Fortran

The following example calls a Fortran subroutine called SIMULATION. This subroutine corresponds to the C void function simulation.

Fortran Code:

```
SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER (C_LONG), VALUE :: ALPHA
REAL (C_DOUBLE), INTENT(INOUT) :: BETA
INTEGER (C_LONG), INTENT(OUT) :: GAMMA
REAL (C_DOUBLE),DIMENSION(*),INTENT(IN) :: DELTA
TYPE, BIND(C) :: PASS
INTEGER (C_INT) :: LENC, LENF
TYPE (C_PTR) :: C, F
END TYPE PASS
TYPE (PASS), INTENT(INOUT) :: ARRAYS
```

```
REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
REAL (C_FLOAT), POINTER :: C_ARRAY(:)
...
! Associate C_ARRAY with an array allocated in C
CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/) )
...
! Allocate an array and make it available in C
ARRAYS%LENF = 100
ALLOCATE (ETA(ARRAYS%LENF))
ARRAYS%F = C_LOC(ETA)
...
END SUBROUTINE SIMULATION
```

C Struct Declaration

```
struct pass {int lenc, lenf; float *c, *f;};
```

C Function Prototype:

```
void simulation(long alpha, double *beta, long *gamma, double delta[], struct
pass *arrays);
```

C Calling Sequence:

```
simulation(alpha, &beta, &gamma, delta, &arrays);
```

Compiling and Linking Intel® Fortran/C Programs

Your application can contain both C and Fortran source files. If your main program is a Fortran source file (myprog.for) that calls a routine written in C (cfunc.c), you can use the following sequence of commands to build your application.

Linux* OS and Mac OS* X:

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

Windows* OS:

```
icl /c cfunc.c
ifort myprog.for cfunc.obj /link /out:myprog.exe
```

The `icc` or `icl` command for Intel® C++ or the `cl` command (for Microsoft Visual C++*) compiles `cfunc.c`. The `-c` or `/c` option specifies that the linker is not called. This command creates `cfunc.o` (Linux OS and Mac OS X) or `cfunc.obj` (Windows OS).

The `ifort` command compiles `myprog.for` and links `cfunc.o` (Linux OS and Mac OS X) or `cfunc.obj` (Windows OS) with the object file created from `myprog.for` to create the executable.

Additionally, on Linux OS and Mac OS X, you may need to specify one or more of the following options:

- Use the -cxxlib compiler option to tell the compiler to link using the C++ run-time libraries provided by gcc. By default, C++ libraries are not linked with Fortran applications.
- Use the `-fexceptions` compiler option to enable C++ exception handling table generation so C++ programs can handle C++ exceptions when there are calls to Fortran routines on the call stack. This option causes additional information to be

added to the object file that is required during C++ exception handling. By default, mixed Fortran/C++ applications abort in the Fortran code if a C++ exception is thrown.

- Use the `-nofor_main` compiler option if your C/C++ program calls an Intel Fortran subprogram, as shown:

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

Calling C Procedures from an Intel® Fortran Program

Naming Conventions

By default, the Fortran compiler converts function and subprogram names to lower case for Linux OS and Mac OS X and upper case for Windows OS. 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.

For Windows systems using IA-32 architecture only, you can alter the default calling convention. You can use either the `/iface:stdcall` option (`stdcall`) or the `/iface:cvf` option (Compaq* and Powerstation compatibility) to change the default calling convention, or the `VALUE` or C attributes in an explicit interface using the ATTRIBUTES directive. For more information on the ATTRIBUTES directive, see the *Intel® Fortran Language Reference*.

Both options cause the routine compiled and routines that it calls to have a @<*n*> appended to the external symbol name, where *n* is the number of bytes of all parameters. Both options assume that any routine called from a Fortran routine compiled this way will do its own stack cleanup, "callee pops." `/iface:cvf` also changes the way that `CHARACTER` variables are passed. With `/iface:cvf`, `CHARACTER` variables are passed as address/length pairs (that is, `/iface:mixed_str_len_arg`).

Using Libraries

Supplied 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.

Intel® Fortran provides different types of libraries, such as static or DLL, single-threaded or multi-threaded.

On Linux* OS and Mac OS* X systems, you can use the `-shared-intel` compiler option on the command line to specify that you want to use the dynamic versions of all Intel libraries.

The tables below show the libraries provided for the compiler. Except where noted, listed libraries apply to systems based on IA-32 architecture, systems based on Intel® 64 architecture and systems based on IA-64 architecture.

The run-time libraries have associated message catalog files, described in Run-Time Library Message Catalog Location.

The file fredist.txt in the <*install-dir*>`/license` folder lists the Intel compiler libraries that are redistributable.

Libraries provided on Windows* OS systems:

File	Description
`ifauto.lib`	Fortran interfaces to Automation objects
`ifcom.lib`	Fortran interfaces to COM support
`ifconsol.lib`	QuickWin stub support
`ifdlg100.dll`	Provides ActiveX* control support to the dialog procedures
`iflogm.lib`	Dialog support
`ifqw_mdi.lib`	QuickWin multi-document support library
`ifqw_sdi.lib`	QuickWin single document support library
`ifqwin.lib`	QuickWin support library
`ifwin.lib`	Miscellaneous Windows support
`libguide.lib`	OpenMP* static library for the parallelizer tool
`libguide40.lib` `libguide40.dll`	These two libraries make up a dynamic library for the parallelizer tool
`libguide_stats.lib`	OpenMP static library for the parallelizer tool with performance statistics and profile information
`libguide40_stats.lib` `libguide40_stats.dll`	These two libraries make up a dynamic library for the parallelizer tool with performance statistics and profile information
`libifcore.lib`	Intel-specific Fortran I/O intrinsic support library
`libifcoremd.lib` `libifcoremd.dll`	...when compiled with `/MD`
`libifcoremdd.lib` `libifcoremdd.dll`	...when compiled with `/MDd`

libifcoremt.lib	...when compiled with /MT
libifcorert.lib libifcorert.dll	...when compiled with /MDs
libifcorertd.lib libifcorertd.dll	...when compiled with /MDsd
libifport.lib	Portability, POSIX*, and NLS* support library
libifportmd.dll libifportmd.lib	...when compiled with /MD
libirc.lib	Intel-specific library (optimizations)
libircmt.lib	Multithreaded Intel-specific library (optimizations)
libm.lib	Math library
libmmd.lib libmmd.dll	These two libraries make up a dynamic library for the multithreaded math library used when compiling with /MD
libmmdd.lib libmmdd.dll	These two libraries make up a debug dynamic library for the multithreaded math library used when compiling with /MD
libmmds.lib	Static math library built multithread
libmmt.lib	Multithreaded math library used when compiling with /MT
libompstub.lib	Library that resolves references to OpenMP* subroutines when OpenMP is not in use
svml_disp.lib	Short-vector math library (used by vectorizer). Not provided on systems based on IA-64 architecture.
svml_dispmt.lib	Multithread short-vector math library (used by vectorizer). Not provided on systems based on IA-64 architecture.

Libraries provided on Linux* OS and Mac OS* X systems:

File	Description
crtxi.o crtxn.o	C initialization support; Linux OS only.
for_main.o	main routine for Fortran programs
icrt.internal.map icrt.link	C link support; Linux OS only
libcxaguard.a libcxaguard.so (.dylib for Mac OS X) libcxaguard.so.5 (Linux IA-32 and Intel® 64	Used for interoperability with the -cxxlib option.

architectures) `libcxaguard.so.6` (Linux IA-64 architecture)	
`libguide.a` `libguide.so` (`.dylib` for Mac OS X)	OpenMP* static library for the parallelizer tool
`libguide_stats.a` `libguide_stats.so` (`.dylib` for Mac OS X)	Support for parallelizer tool with performance and profile information
`libifcore.a` `libifcore.so` (`.dylib` for Mac OS X) `libifcore.so.5` (Linux OS IA-32 and Intel® 64 architectures) `libifcore.so.6` (IA-64 architecture)	Intel-specific Fortran run-time library
`libifcore_pic.a` `libifcoremt_pic.a`	Intel-specific Fortran static libraries; Linux OS 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 X) `libifcoremt.so.5` (Linux OS IA-32 and Intel® 64 architectures) `libifcoremt.so.6` (IA-64 architecture)	Multithreaded Intel-specific Fortran run-time library
`libifport.a` `libifport.so` (`.dylib` for Mac OS X) `libifport.so.5` (Linux OS IA-32 and Intel® 64 architectures) `libifport.so.6` (IA-64 architecture) `libifportmt.dylib` (Mac OS X only)	Portability and POSIX support
`libimf.a` `libimf.so` (`.dylib` for Mac OS X)	Math library

libirc.a libirc_s.a libirc.dylib (Mac OS X)	Intel-specific library (optimizations)
libintlc.so (.dylib for Mac OS X)	Dynamic versions of libirc
libompstub.a	Library that resolves references to OMP subroutines when OMP is not in use
libsvml.a libsvml.dylib (Mac OS X)	Short vector math library

Creating 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.

When compiling a static library from the ifort command line, include the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) compiler option to suppress linking. Without this option, the compiler generates an error because the library does not contain a main program.

To build a static library (Linux OS):

1. Use the -c option to generate object files from the source files:
   ifort -c my_source1.f90 my_source2.f90 my_source3.f90
2. Use the 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

To build a static library (Mac OS X):

1. Use the following command line to generate object files and create the library file:
   ifort -o my_lib.a -staticlib mysource1.f90 mysource2.f90 mysource3.f90
2. Compile and link your project with your new library:
   ifort main.f90 my_lib.a

If your library file and source files are in different directories, use the `-Ldir` option to indicate where your library is located:

ifort -L/for/libs main.f90 my_lib.a

To build a static library (Windows OS):

To build a static library from the integrated development environment (IDE), select the Fortran Static Library project type.

To build a static library using the command line:

1. Use the /c option to generate object files from the source files:
   ifort /c my_source1.f90 my_source2.f90
2. Use the Microsoft LIB tool to create the library file from the object files:
   lib /out:my_lib.lib my_source1.obj my_source2.obj

3. Compile and link your project with your new library:
   ifort main.f90 my_lib.lib

Creating Shared Libraries
Shared libraries, also referred to as dynamic libraries, are linked differently than static libraries. At compile time, the linker insures that all the necessary symbols are either linked into the executable, or can be linked at runtime from the shared library. Executables compiled from shared libraries are smaller, but the shared libraries must be included with the executable to function correctly. When multiple programs use the same shared library, only one copy of the library is required in memory.

To create a shared library from a Fortran source file, process the files using the `ifort` command:

- You must specify the `-shared` option (Linux* OS) or the `-dynamiclib` option (Mac OS* X) to create the `.so` or `.dylib` file. On Linux OS and Mac OS X operating systems using either IA-32 architecture or Intel® 64 architecture, 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 OS)
ifort -dynamiclib octagon.f90 (Mac OS* X)
```

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 OS) or `octagon.dylib` (Mac OS X).

You can use the `-static-intel` option to force the linker to use the static versions of the Intel-supplied libraries.

You can also create a shared library file with a combination of ifort and ld (Linux OS) or libtool (Mac OS X) 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 OS) or `libtool` (Mac OS X) command to create the shared library. The following example shows the command to create a shared library named octagon.so on a Linux operating system:

```
ld -shared octagon.o \
     -lifport -lifcoremt -limf -lm -lcxa \
```

```
    -lpthread -lirc -lunwind -lc -lirc_s
```

Note the following:

- When you use `ld`, you need to list all Fortran libraries. It is easier and safer to use the ifort command. On Mac OS X, you would use libtool.
- The `-shared` option is required to create a shared library. On Mac OS X, 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 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 X, 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`.

Calling Library Routines

The following table shows the groups of Intel Fortran library routines and the USE statement required to include the interface definitions for the routines in that group:

Routines	USE statement
Portability	USE IFPORT
POSIX*	USE IFPOSIX
Miscellaneous Run-Time	USE IFCORE
The following are Windows only:	
Automation (AUTO) (systems using IA-32 architecture only)	USE IFAUTO
Component Object Model (COM) (systems using IA-32 architecture only)	USE IFCOM
Dialog (systems using IA-32 architecture only)	USE IFLOGM
Graphics	USE IFQWIN
National Language Support	USE IFNLS
QuickWin	USE IFQWIN
Serial port I/O (SPORT) (systems using IA-32 architecture only)	USE IFPORT

Module Routines lists topics that provide an overview of the different groups of library routines as well as calling syntax for the routines. For example, add the following USE statement (before any data declaration statements, such as IMPLICIT NONE or INTEGER):

```
USE IFPORT
```

If you want to minimize compile time for source files that use the Intel Fortran library routines, add the ONLY keyword to the USE statement. For example:

```
USE IFPORT, only: getenv
```

Using the ONLY keyword limits the number of interfaces for that group of library routines.

To view the actual interface definitions, view the `.f90` file that corresponds to the `.mod` file. For example, if a routine requires a USE IFCORE, locate and use a text editor to view the file `ifcore.f90` in the standard INCLUDE directory.

You should avoid copying the actual interface definitions contained in the `ifport.f90` (or `ifcore.f90`, ...) into your program because future versions of Intel Fortran might change these interface definitions.

Similarly, some of the library interface `.f90` files contain USE statements for a subgrouping of routines. However, if you specify a USE statement for such a subgroup, this module name may change in future version of Intel Fortran. Although this will make compilation times faster, it might not be compatible with future versions of Intel Fortran.

Using the Portability Library, IFPORT.LIB

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.LIB` (Windows* OS) or `libifport.a` (Linux* OS and Mac OS* X). Frequently used functions are included in a portability module called IFPORT.

The portability library also contains IEEE* POSIX library functions. These functions are included in a module called IFPOSIX.

You can use the portability library in one of two ways:

- Add the statement USE IFPORT to your program. This statement includes the IFPORT module.
- Call portability routines using the correct parameters and return value.

The portability library is passed to the linker by default during linking. To prevent this, specify the `-fpscomp nolibs` (Linux OS and Mac OS X) or `/fpscomp:nolibs` (Windows OS) option.

Using the IFPORT mod file provides interface blocks and parameter definitions for the routines, as well as compiler verification of calls.

See these topics:

- Using the IFPORT Portability Module
- Portability Routines

Using the IFPORT Portability Module

Using the IFPORT module provides interface blocks and parameter definitions for the portability 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 module contains generic interface blocks that give procedure definitions for these routines.

Fortran 95/90 contains intrinsic procedures for many of the portability functions. 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.

Refer to the Portability Routines table as you read through this topic.

Information Retrieval Routines

Information retrieval procedures 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* OS and Mac OS* X systems, FORK creates a duplicate image of the parent process. Child and parent processes each have their own copies of resources, and become independent from one another. On Windows* OS systems, you can create a child process (called a thread), but both parent and child processes share the same address space and share system resources.

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, in which case the standard Fortran routines should be used.

Data object conversion can be accomplished by using the INT intrinsic function instead of LONG or SHORT. The intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED perform the

same functions as the random number functions listed in the table showing numeric values and conversion routines.

Other bit manipulation functions such as AND, XOR, OR, LSHIFT, and RSHIFT are intrinsic functions. You do not need the IFPORT module to access them. Standard Fortran 95/90 includes many bit operation routines, which are listed in the Bit Operation and Representation Routines table.

Input and Output Routines

The portability library contains routines that change file properties, read and write characters and buffers, and change the offset position in a file. These input and output routines can be used with standard Fortran input or output statements such as READ or WRITE on the same files, provided that you take into account the following:

- When used with direct files, after an FSEEK, GETC, or PUTC operation, the record number is the number of the next whole record. Any subsequent normal Fortran I/O to that unit occurs at the next whole record. For example, if you seek to absolute location 1 of a file whose record length is 10, the NEXTREC returned by an INQUIRE would be 2. If you seek to absolute location 10, NEXTREC would still return 2.
- On units with CARRIAGECONTROL='FORTRAN' (the default), PUTC and FPUTC characters are treated as carriage control characters if they appear in column 1.
- On sequentially formatted units, the C string "\n"c, which represents the carriage return/line feed escape sequence, is written as CHAR(13) (carriage return) and CHAR(10) (line feed), instead of just line feed, or CHAR(10). On input, the sequence 13 followed by 10 is returned as just 10. (The length of character string "\n"c is 1 character, whose ASCII value, indicated by ICHAR('\n'c), is 10.)
- Reading and writing is in a raw form for direct files. Separators between records can be read and overwritten. Therefore, be careful if you continue using the file as a direct file.

I/O errors arising from the use of these routines result in an Intel Fortran run-time error.

Some portability file I/O routines have equivalents in standard Fortran 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 specifier to check for similar information. (The ACCESS function always returns 0 for read permission on FAT files, meaning that all files have read permission.)

Date and Time Routines

Various date and time routines are available to determine system time, or convert it to local time, Greenwich Mean Time, arrays of date and time elements, or an ASCII character string.

DATE and TIME are available as either a function or subroutine. Because of the name duplication, if your programs do not include the USE IFPORT statement, each separately compiled program unit can use only one of these versions. For example, if a program calls the subroutine TIME once, it cannot also use TIME as a function.

Standard Fortran 95/90 includes date and time intrinsic subroutines. For more information, see DATE_AND_TIME.

Error Handling Routines

Error handling routines detect and report errors.

IERRNO error codes are analogous to `errno` on Linux* OS and Mac OS* X 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 the functions listed below. File names can be long file names or UNC file names. A forward slash in a path name is treated as a backslash. All path names can contain drive specifications.

Standard Fortran 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.

Serial Port Routines (Windows only)

The serial port I/O (SPORT_*xxx*) routines help you perform basic input and output to serial ports. These routines are available only on systems using IA-32 architecture. For more information, see Using the Serial Port I/O Routines.

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 Portability Routines table.

Math Libraries
On Linux* OS and Mac OS* X, `libimf.a` is the math library provided by Intel. This is in addition to `libm.a`, which is the math library provided with gcc*.

Both of these libraries are linked in by default because certain math functions supported by the GNU* math library are not available in the Intel math library. This linking arrangement allows the GNU users to have all functions available when using `ifort`, with Intel optimized versions available when supported.

`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.

On Windows* OS, Intel provides `libm.lib` (static library) and `libmmd.dll` (the DLL version).

Error Handling

Handling Compile Time Errors

Understanding Errors During the Build Process

The Intel® Fortran Compiler identifies syntax errors and violations of language rules in the source program.

Compiler Diagnostic Messages

These messages describe diagnostics that are reported during the processing of the source file. Compiler diagnostic messages usually provide enough information for you to determine the cause of an error and correct it. These messages generally have the following format:

filename(*linenum*:) *severity*: *message*

filename Indicates the name of the source file currently being processed.

linenum Indicates the source line where the compiler detects the condition.

severity Indicates the severity of the diagnostic message: Warning, Error, or Fatal error.

message Describes the problem.

The following is an example of an error message showing the format and message text:

```
echar.for(7): Severe: Unclosed DO loop or IF block
        DO I=1,5
--------^
```

The pointer (---^) indicates the exact place on the source program line where the error was found, in this case where an END DO statement was omitted.

To view the passes as they execute on the command line, specify `-watch` (Linux* OS and Mac OS* X) or `/watch` (Windows* OS).

Note

You can perform compile-time procedure interface checking between routines with no explicit interfaces present. To do this, generate a module containing the interface for each compiled routine using the `-gen-interfaces` (Linux OS and Mac OS X) or `/gen-interfaces` (Windows OS) option and check implicit interfaces using the `-warn interfaces` (Linux OS and Mac OS X) or `/warn:interfaces` (Windows OS) option.

Controlling Compiler Diagnostic Warning and Error Messages

You can use a number of compiler options to control the diagnostic messages issued by the compiler. For example, the `-WB` (Linux OS and Mac OS X) or `/WB` (Windows OS) compiler option turns compile time bounds errors into warnings. To control compiler diagnostic messages (such as warning messages), use `-warn` (Linux OS and Mac OS X) or `/warn` (Windows OS). The `-warn [`*keyword*`]` (Linux OS and Mac OS X) or `/warn:`*keyword* (Windows

OS) option controls warnings issued by the compiler and 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 also control the display of diagnostic information with variations of the `-diag` (Linux OS and Mac OS X) or `/Qdiag` (Windows OS) compiler option. This compiler option accepts numerous arguments and values, allowing you wide control over displayed diagnostic messages and reports.

Some of the most common variations include the following:

Linux OS and Mac OS X	Windows OS	Description
`-diag-enable` *`list`*	`/Qdiag-enable:`*`list`*	Enables a diagnostic message or a group of messages
`-diag-disable` *`list`*	`/Qdiag-disable:`*`list`*	Disables a diagnostic message or a group of messages
`-diag-warning` *`list`*	`/Qdiag-warning:`*`list`*	Tells the compiler to change diagnostics to warnings
`-diag-error` *`list`*	`/Qdiag-error:`*`list`*	Tells the compiler to change diagnostics to errors
`-diag-remark` *`list`*	`/Qdiag-remark:`*`list`*	Tells the compiler to change diagnostics to remarks (comments)

The *`list`* items can be specific diagnostic IDs, one of the keywords `warn`, `remark`, or `error`, or a keyword specifying a certain group (`par, vec, driver, cpu-dispatch, sv`). For more information, see -diag,/Qdiag.

Additionally, you can use the following related options:

Linux OS and Mac OS X	Windows OS	Description
`-diag-dump`	`/Qdiag-dump`	Tells the compiler to print

		all enabled diagnostic messages and stop compilation
`-diag-file[=file]`	`/Qdiag-file[:file]`	Causes the results of diagnostic analysis to be output to a file
`-diag-file-append[=file]`	`/Qdiag-file-append[:file]`	Causes the results of diagnostic analysis to be appended to a file
`-diag-error-limit n`	`/Qdiag-error-limit:n`	Specifies the maximum number of errors allowed before compilation stops.

Linker Diagnostic Errors

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors occur, the linker does not produce an executable file. Linker messages are descriptive, and you do not normally need additional information to determine the specific error.

To view the libraries being passed to the linker on the command line, specify `-watch` or `/watch`.

Error Severity Levels

Comment Messages

These messages indicate valid but inadvisable use of the language being compiled. The compiler displays comments by default. You can suppress comment messages with the `-warn nousage` (Linux OS and Mac OS X) or `/warn:nousage` (Windows OS) option.

Comment messages do not terminate translation or linking, they do not interfere with any output files either. Some examples of the comment messages are:

```
Null CASE construct
The use of a non-integer DO loop variable or expression
Terminating a DO loop with a statement other than CONTINUE or ENDDO
```

Warning Messages

These messages report valid but questionable use of the language being compiled. The compiler displays warnings by default. You can suppress warning messages by using the `-warn` or `/warn` option. Warnings do not stop translation or linking. Warnings do not interfere with any output files. Some representative warning messages are:

```
constant truncated - precision too great
non-blank characters beyond column 72 ignored
```

```
Hollerith size exceeds that required by the context
```

Error Messages

These messages report syntactic or semantic misuse of Fortran.

Errors suppress object code for the error containing the error and prevent linking, but they do not stop from parsing to continue to scan for any other errors. Some typical examples of error messages are:

```
line exceeds 132 characters
unbalanced parenthesis
incomplete string
```

Fatal Errors

Fatal messages indicate environmental problems. Fatal error conditions stop translation, assembly, and linking. If a fatal error ends compilation, the compiler displays a termination message on standard error output. Some representative fatal error messages are:

```
Disk is full, no space to write object file
Incorrect number of intrinsic arguments
Too many segments, object format cannot support this many segments
```

Using the Command Line

If you are using the command line, messages are written to the standard error output file.

When using the command line:

- Make sure that the appropriate environment variables have been set by executing the `ifortvars.sh` (Linux OS and Mac OS X) or `IFORTVARS.BAT` (Windows OS) file. For example, this BAT file sets the environment variables for the include and library directory paths. For Windows OS, these environment variables are preset if you use the Fortran Command Prompt window in the Intel® Visual Fortran program folder. For a list of environment variables used by the ifort command during compilation, see Setting Compile-Time Environment Variables.
- Specify the libraries to be linked against using compiler options.
- You can specify libraries (include the path, if needed) as file names on the command line.

Using the Visual Studio IDE (Windows OS)

If you are using the Microsoft Visual Studio* integrated development environment (IDE), compiler and linker errors are displayed in the Build pane of the Output window. To display the Output window, choose **View>Other Windows>Output**. You can also use the Task List window (**View>Other Windows>Task List**) to view display links to problems encountered during the build process. Click these tasks to jump to code that caused build problems. You can also check the Build log for more information.

To quickly locate the source line causing the error, follow these steps:

1. Double-click the error message text in the Build pane of the Output window.

- or -

2. Press F8. The editor window appears with a marker in the left margin that identifies the line causing the error. You can continue to press F8 to scroll through additional errors.

After you have corrected any compiler errors reported during the previous build, choose **Build *project name*** from the **Build** menu. The build engine recompiles only those files that have changed, or which refer to changed include or module files. If all files in your project compile without errors, the build engine links the object files and libraries to create your program or library.

You can force the build engine to recompile all source files in the project by selecting **Rebuild *project name*** from the **Build** menu. This is useful to verify that all of your source code is clean, especially if you are using a makefile, or if you use a new set of compiler options for all of the files in your project.

When using the IDE:

- Make sure that you have specified the correct path, library, and include directories. For more information, see Specifying Path, Library and Include Directories.
- If a compiler option is not available through **Project>Properties** in the Intel Fortran Property pages, you can type the option in the Command Line category. Use the lower part of the window under **Additional Options:** just as you would using the command line. For example, you can enter the linker option `/link /DEFAULTLIB` to specify an additional library.

Using Static Verification Diagnostic Options

Static Verification Overview

Static verification is an additional diagnostic capability to help you debug your programs. You can use the static verification options to detect potential errors in your compiled code including:

- incorrect usage of OpenMP* directives
- inconsistent object declarations in different program units
- boundary violations
- uninitialized memory
- memory corruptions
- memory leaks
- incorrect usage of pointers and allocatable arrays
- dead code and redundant executions
- typographical errors or uninitialized variables

Static verification options can be used to analyze and find issues with source files; these source files need not form a whole program. (For instance, you can pass sources for libraries on the static verification command line.) In such cases, because only partial information is

available about usage and modification of global objects, calls to routines, and so forth, analysis will be less exact.

Your code must successfully compile, with no errors, for static verification options to take effect.

The current usage model is that static verification is added as an alternate build option to produce a diagnostic report. When you enable static verification, the compiler will not create an executable file. Object files that are produced when using static verification are not valid and should not be used for generating real executable or static/dynamic link libraries.

Static verification cannot be used in conjunction with cross-file interprocedural optimization (`/Qipo` or `-ipo`).

Due to generalization of input data (static verification is performed for all possible input data), static verification cannot detect all possible errors that may appear during program execution.

Static verification uses a set of heuristics associated with the modification and usage of program objects. For example, when an undefined element of an array is set to some value, it is assumed that any arbitrary element of the array has been set to the same value. Similarly, after an undefined procedure call, it is assumed that all arguments substituted by reference are used and possibly modified.

Run-time checking executes a program with a fixed set of values for its input variables so it is difficult to check all edge effects. However, static verification performs a general overview check of a program for all possible values simultaneously.

Using the Static Verification Options

Use the `-diag-enable sv{[1|2|3]}` (Linux* and Mac OS* X) or `/Qdiag-enable:sv{[1|2|3]}` (Windows*) compiler option to enable static verification. The number specifies the severity level of the diagnostics (1=all critical errors, 2=all errors, and 3=all errors and warnings).

The command line can contain multiple source files; object files are created for each source file. All object files are processed simultaneously.

Other related options include the following:

`-diag-disable sv` `/Qdiag-disable:sv`	Disables static verification
`-diag-disable warn` `/Qdiag-disable:warn`	Suppresses all warnings, cautions and comments (issues errors only), including those specific to static verification
`-diag-disable` *num-list* `/Qdiag-disable:`*num-list*	Suppresses messages by number list, where *num-list* is either a single message or a list of message numbers separated by commas and enclosed in parentheses.
`-diag-file [`*file*`]` `/Qdiag-file[:`*file*`]`	Directs diagnostic results to *file* with .diag as the default extension. You need to enable static verification diagnostics before you can send them to a file. If a file name is not specified, the diagnostics are sent to *name-of-the-first-source-file*.diag.

`-diag-file-append[=file]` `/Qdiag-file-append[:file]`	Appends diagnostic results to *file* with .diag as the default extension. You need to enable static verification diagnostics before you can send the results to a file. If you do not specify a path, the current working directory will be searched. If the named file is not found, a new file with that name will be created. If a file name is not specified, the diagnostics are sent to *name-of-the-first-source-file*.diag.
`-diag-enable sv-include` `/Qdiag-enable:sv-include`	Diagnoses include files as well as source(s)

Using -diag-enable sv with other Compiler Options

If the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) compiler option is used on the command line along with a command line option to enable static verification, an object file is created; static verification diagnostics are not produced. This object file may be used in a further invocation of static verification. This feature is useful when a program consists of files written in different languages (C/C++ and Fortran).

To analyze OpenMP directives, add the `-openmp` (Linux OS and Mac OS X) or `/Qopenmp` (Windows OS) option to the command line.

Using Static Verification within the IDE

When static verification support is enabled within the IDE, the customary final build target (e.g. an executable image) is not created. Therefore, create a separate "Static Verification" configuration.

In the Microsoft Visual Studio Environment, modify the existing Debug (development) configuration, as follows:

1. Select the **Project Configuration Manager** either from the **Build** menu or from the **Project Properties** window.
2. In the **Active Solution Configuration** dropdown, select **New...**
3. Provide a name for this configuration.
4. Specify to "Copy Settings from" the Debug configuration.
5. Exit the Configuration Manager.

With the new configuration active, navigate to the **Intel(R) Fortran > Diagnostics** property page. Use the **Level of Static Analysis** and **Analyze Include Files** properties to control static verification.

In the Xcode* IDE, modify the existing debug (development) configuration, as follows:

1. Double click the Target to **Get Info**.
2. Navigate to the **Build** properties page.
3. Expand the Configuration drop down and select **Edit Configurations...**
4. Select the **Debug** configuration.
5. Click the **Duplicate** button at the bottom of the page.

6. Optionally change the configuration name; the default name is `Debug copy`.
7. Exit the page.

With the Configuration set to the new configuration, navigate to the Intel compiler's DIAGNOSTIC collection of properties. Use the **Level of Static Analysis** and **Analyze Include Files** properties to control static verification.

Set the **Active Target** and **Active Build Configuration** appropriately to BUILD the static verification enabled configuration.

Static Verification Capabilities

Static verification capabilities include the following:

- Interprocedural Analysis for detecting inconsistent objects declarations in different program units
- Local Program Analysis for analyzing each program unit separately and checking for various kinds of problems that will errors or warnings.
- C/C++ specific Analysis for analyzing C/C++ source code and checking for C/C++ specific error and warning conditions. Static verification also detects improper code style and flaws in object-oriented design solutions.
- Fortran-specific Analysis for analyzing Fortran source code and checking for Fortran-specific error and warning conditions.
- OpenMP* Analysis for checking for OpenMP API restrictions.

Interprocedural Analysis

Static verification detects inconsistent object declarations in different program units, for example:

- Different external objects with the same name.
- Inconsistent declarations of a COMMON block.
- Mismatched number of arguments.
- Mismatched type, rank, shape, or/and size of an argument.
- Inconsistent declaration of a procedure and predeclarations (or interfaces) of this procedure.

The following example illustrates interprocedural analysis.

Example: Wrong number of arguments

File controlf.c contains function declaration in the following line:

```
controlf()
```

File uloop2.c contains the following line:

```
65      fds = controlf( 1 ) ;
```

Static verification issues the following message:

```
uloop2.c(65): error #12020: [SV] number of actual arguments (1) in call of
"controlf" doesn't match the number of formal arguments (0); "controlf" is
defined at (file:controlf.c line:4)
```

Local Program Analysis

With static verification, the compiler analyzes each program unit separately and checks for various kinds of errors, warnings, and/or debatable points in user program. Examples of these errors are:

- Incorrect use or modification of an object
- Problems with memory (for example, leaks, corruptions, uninitialized memory)
- Incorrect use with pointers
- Boundaries violations
- Wrong value of an argument in an intrinsic call
- Dead code and redundant executions

The following examples illustrate local program analysis.

Example 1: Object is smaller than required size

File “chess.h” contains the following:

```
278    typedef struct {
279        int path[MAXPLY];
280        unsigned char path_hashed;
281        unsigned char path_length;
282        unsigned char path_iteration_depth;
283    } CHESS_PATH;
```

File initdata.h contains the following:

```
237      CHESS_PATH     pv[MAXPLY];
```

File quiesce.c contains the following:

```
153      memcpy(&pv[ply-1].path_hashed,&pv[ply].path_hashed,3);
```

Static verification issues the following message:

```
quiesce.c(153): error #12224: [SV] Buffer overflow: size of object "path_hashed"
(1 bytes) is less than required size (3 bytes)
```

Example 2: Wrong type of intrinsic argument

File makefile contains the following:

```
31    CFLAGS2 = $(CFLAGS) -DVERSION=9 -DCOMPDATE=1994
```

File version.c contains the following:

```
20 fprintf (stderr, "%s: version: %d, compiled: %s, cflags: %s\n",
21 ego, VERSION, COMPDATE, "CFLAGS");
```

Static verification issues the following message:

```
version.c(21): error #12037: [SV] actual argument 5 in call of "fprintf" should
be a pointer to "char"
```

Fortran-specific Analysis

Static verification analyzes Fortran source code and checks for various kinds of errors, warnings, and/or debatable points in user program.

Static verification detects issues with the following:

- Function result
- COMMON blocks

The following example illustrates Fortran-specific analysis.

Example 1: Undefined function result

File dlarnd.f contains the following lines:

```
2           REAL*8 FUNCTION DLARND( IDIST, ISEED )
               . . .
82           T1 = DLARAN( ISEED )
84           IF( IDIST.EQ.1 ) THEN
88              DLARND = T1
89           ELSE IF( IDIST.EQ.2 ) THEN
93              DLARND = TWO*T1 - ONE
94           ELSE IF( IDIST.EQ.3 ) THEN
98              T2 = DLARAN( ISEED )
99              DLARND = SQRT( -TWO*LOG( T1 ) )*COS( TWOPI*T2 )
100          END IF
101          RETURN
105          END
```

Static verification issues the following message:

```
dlarnd.f(105): error #12077: [SV] function result is possibly not set
```

C/C++ specific Analysis

Static verification analyzes C/C++ source code and checks for various kinds of errors, warnings, and/or debatable points in your program. It also points out places of improper code style and flaws in object-oriented design solutions.

Static verification detects issues with the following:

- Memory management (leaks, mixing C and C++ memory management routines, smart pointer usage)
- C++ exception handling (uncaught exception, exception specification, exception from destructor/operator delete)
- operator new/operator delete

The following example illustrates C/C++ specific analysis.

```
1          #include "stdio.h"
2          class A {
3          public:
4                    A() { destroy(); }
5
6                    void destroy() { clear0();}
7
8                    virtual void clear()=0;
9
10                   void clear0() { clear(); };
```

```
11      };
12
13      class B : public A {
14      public:
15                B(){ }
16
17                virtual void clear(){ printf("overloaded
clear"); }
18
19                virtual   ~B() { }
20      };
21
22      int main() {
23      B b;
24      return 0;
25      }
```

Static verification issues the following message:

```
test2.cpp(10): warning #12327: [SV] pure virtual function
"clear" is called from constructor (file:test2.cpp line:4)
```

OpenMP* Analysis

The compiler detects some restrictions noted in the OpenMP* API Versions 2.0 and 2.5. When static verification is enabled, the compiler performs some additional checks against restrictions in the OpenMP API, including checks for the correct use of the following:

- nested parallel regions including dynamic extent of parallel regions
- private variables in parallel regions
- thread-private variables
- expressions which are used in OpenMP clauses

Example: Incorrect usage of OpenMP directives

File gafort.f90 contains the following lines:

```
310     !$OMP PARALLEL DO ORDERED
311     ! create an array of locks
312     !$     DO i = 1,indmax
313     !$        CALL omp_init_lock(lck(i))
314     !$     ENDDO
315     !$OMP END PARALLEL DO
```

The parallel region has the clause ORDERED but has no corresponding ORDERED OpenMP directive. Static verification issues the following message:

```
gafort.f90(310): error #12204: [SV] ORDERED clause is used in the dynamic extent
of non-ORDERED DO construct
```

Handling Run-Time Errors

Understanding Run-Time Errors

During execution, your program may encounter errors or exception conditions. These conditions can result from any of the following:

- Errors that occur during I/O operations
- Invalid input data
- Argument errors in calls to the mathematical library
- Arithmetic errors
- Other system-detected errors

The Intel® Fortran run-time system (Run-Time Library or RTL) generates appropriate messages and takes action to recover from errors whenever possible.

For a description of each Intel Fortran run-time error message, see List of Run-Time Error Messages.

There are a few tools and aids that are helpful when an application fails and you need to diagnose the error. Compiler-generated machine code listings and linker-generated map files can help you understand the effects of compiler optimizations and to see how your application is laid out in memory. They may help you interpret the information provided in a stack trace at the time of the error. See Generating Listing and Map Files.

Forcing a Core Dump for Severe Errors

You can force a core dump for severe errors that do not usually cause a `core` file to be created. Before running the program, set the `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.

See these topics:

- Run-Time Default Error Processing
- Run-Time Message Display and Format
- Values Returned at Program Termination
- Methods of Handling Errors
- Using the END, EOR, and ERR Branch Specifiers

- Using the IOSTAT Specifier and Fortran Exit Codes
- Locating Run-Time Errors
- List of Run-Time Error Messages
- Signal Handling
- Overriding the Default Run-Time Library Exception Handler
- Using Traceback Information and related topics

Run-Time Default Error Processing

The Intel® Fortran run-time system processes a number of errors that can occur during program execution. A default action is defined for each error recognized by the Intel Fortran run-time system. The default actions described throughout this section occur unless overridden by explicit error-processing methods.

The way in which the Intel Fortran run-time system actually processes errors depends upon the following factors:

- The severity of the error. For instance, the program usually continues executing when an error message with a severity level of warning or info (informational) is detected.
- For certain errors associated with I/O statements, whether or not an I/O error-handling specifier was specified.
- For certain errors, whether or not the default action of an associated signal was changed.
- For certain errors related to arithmetic operations (including floating-point exceptions), compilation options can determine whether the error is reported and the severity of the reported error.

How arithmetic exception conditions are reported and handled depends on the cause of the exception and how the program was compiled. Unless the program was compiled to handle exceptions, the exception might not be reported until *after* the instruction that caused the exception condition.

See Also

- About where Intel Fortran run-time messages are displayed and their format, see Run-Time Message Display and Format.
- On the Intel Fortran return values at program termination, see Values Returned at Program Termination.
- On locating errors and the compiler options related to handling errors and exceptions, see Locating Run-Time Errors and Using Traceback Information.
- On Intel Fortran intrinsic data types and their ranges, see Data Representation.

Run-Time Message Display and Format

Fortran run-time messages have the following format:

```
forrtl: severity (number): message-text
```

where:

- `forrtl`
  Identifies the source as the Intel Fortran run-time system (Run-Time Library or RTL).
- *`severity`*
  The severity levels are*: severe, error, warning, or info.*
- *`number`*
  This is the message number; also the IOSTAT value for I/O statements.
- *`message-text`*
  Explains the event that caused the message.

The following table explains the severity levels of run-time messages, in the order of greatest to least severity. The severity of the run-time error message determines whether program execution continues:

Severity	Description
severe	Must be corrected. The program's execution is terminated when the error is encountered unless the program's I/O statements use the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier. (See Using the END, EOR, and ERR Branch Specifiers and Using the IOSTAT Specifier and Fortran Exit Codes and Methods of Handling Errors.) For severe errors, stack trace information is produced by default, unless the environment variable `FOR_DISABLE_STACK_TRACE` is set. If the command line option -traceback (Linux* OS and Mac OS* X) or /traceback (Windows* OS) is specified, 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.
error	Should be corrected. The program might continue execution, but the output from this execution might be incorrect. For errors of severity type error, stack trace information is produced by default, unless the environment variable `FOR_DISABLE_STACK_TRACE` is set. If the command line option -traceback (Linux OS and Mac OS X) or /traceback (Windows) is specified, 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.
warning	Should be investigated. The program continues execution, but output from this execution might be incorrect.
info	For informational purposes only; the program continues.

For a description of each Intel Fortran run-time error message, see Run-Time Default Error Processing and related topics.

The following example applies to Linux OS and Mac OS X:

In some cases, stack trace information is 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.)

The following program generates an error at line 12:

```
program ovf
real*4 x(5),y(5)
integer*4 i

x(1) = -1e32
x(2) = 1e38
x(3) = 1e38
x(4) = 1e38
x(5) = -36.0

do i=1,5
y(i) = 100.0*(x(i))
print *, 'x = ', x(i), ' x*100.0 = ',y(i)
end do
end
```

The following command line produces stack trace information for the program executable.

```
> ifort -O0 -fpe0 -traceback ovf.f90 -o ovf.exe
> ovf.exe

x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34
forrtl: error (72): floating overflow
Image        PC         Routine       Line      Source
ovf.exe      08049E4A   MAIN__             14   ovf.f90
ovf.exe      08049F08   Unknown       Unknown   Unknown
ovf.exe      400B3507   Unknown       Unknown   Unknown
ovf.exe      08049C51   Unknown       Unknown   Unknown
Abort
```

The following suppresses stack trace information because the FOR_DISABLE_STACK_TRACE environment variable is set.

```
> setenv FOR_DISABLE_STACK_TRACE true
> ovf.exe

x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34
forrtl: error (72): floating overflow
Abort
```

Run-Time Library Message Catalog File Location

The libifcore, libirc, and libm run-time libraries ship message catalogs. When a message by one of these libraries is to be displayed, the library searches for its message catalog in a directory specified by either the NLSPATH (Linux OS and Mac OS X), or %PATH% (Windows OS) environment variable. If the message catalog cannot be found, the message is displayed in English.

The names of the three message catalogs are as follows:

libifcore message catalogs and related text message	Linux OS and Mac OS	ifcore_msg.cat

files	X	ifcore_msg.msg
	Windows OS	ifcore_msg.dll ifcore_msg.mc
libirc message catalogs and related text message files	Linux OS and Mac OS X	irc_msg.cat irc_msg.msg
	Windows OS	irc_msg.dll irc_msg.mc
libm message catalogs and related text message files	Linux OS and Mac OS X	libm.cat libm.msg
	Windows OS	libmUI.dll libmUI.mc

Values Returned at Program Termination

An Intel Fortran program can terminate in a number of ways. On Linux OS and Mac OS X, values are returned to the shell.

- The program runs to normal completion. A value of zero is returned.
- The program stops with a STOP statement. If an integer *stop-code* is specified, a status equal to the code is returned; if no *stop-code* is specified, a status of zero is returned.
- The program stops because of a signal that is caught but does not allow the program to continue. A value of 1 is returned.
- The program stops because of a severe run-time error. The error number for that run-time error is returned. See Understanding Run-Time Errors and related topics.
- The program stops with a `CALL EXIT` statement. The value passed to `EXIT` is returned.
- The program stops with a `CALL ABORT` statement. A value of 134 is returned.

Methods of Handling Errors

Whenever possible, the Intel® Fortran RTL does certain error handling, such as generating appropriate messages and taking necessary action to recover from errors. You can explicitly supplement or override default actions by using the following methods:

- To transfer control to error-handling code within the program, use the END, EOR, and ERR branch specifiers in I/O statements. See Using the END, EOR, and ERR Branch Specifiers.
- To identify Fortran-specific I/O errors based on the value of Intel Fortran RTL error codes, use the I/O status specifier (IOSTAT) in I/O statements (or call the ERRSNS subroutine). See Using the IOSTAT Specifier and Fortran Exit Codes.
- Obtain system-level error codes by using the appropriate library routines.
- For certain error conditions, use the signal handling facility to change the default action to be taken.

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 Visual Fortran RTL transfers control to the statement at label 400. Similarly, you can use the END specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

```
    READ (12,70,END=550)
```

When using 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 and Fortran Exit Codes

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, which is returned as an exit code if the program terminates:

- 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 List of Run-Time Error Messages, which lists the 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 the `for_iosdef.for` file 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).

Error Handling OPEN Statement File Name

```
     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. Locate correct file with Explorer 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. The following compiler options are related to handling errors and exceptions:

- The `-check [keyword]` (Linux OS Mac OS X) or `/check[:keyword]` (Windows OS) option generates extra code to catch certain conditions at run time. For example, if you specify the keyword of `bounds`, the debugger will catch and stop at array or character string bounds errors. You can specify the keyword of `bounds` to generate code to perform compile-time and run-time checks on array subscript and character substring expressions. An error is reported if the expression is outside the dimension of the array or the length of the string. The keyword of `uninit` generates code for dynamic checks of uninitialized variables. If a variable is read before written, a run-time error routine will be called. The `noformat` and `nooutput_conversion` keywords reduce the severity level of the associated run-time error to allow program continuation. The `pointers` keyword generates code to test for disassociated pointers and unallocatable arrays.
  The following `-check pointers` (Linux OS and Mac OS X) or `/check:pointers` (Windows OS) examples result in various output messages.

  Example 1: Allocatable variable not allocated

```
        real, allocatable:: a(:)
!       allocate(a(4)) ! if a is unallocated, the next statement gets an
error with "check pointers"
        a=17
        print *,a
        end
 Output 1:
forrtl: severe (408): fort: (8): Attempt to fetch from allocatable
variable A when it is not allocated
```

  Example 2: Pointer not associated

```
        real, pointer:: a(:)
        allocate(a(5))
        a=17
        print *,a
        deallocate(a)   ! once a is deallocated, the next statement gets
an error with "check pointers"
        a=20
        print *,a
        end


Output 2:
   17.00000       17.00000       17.00000       17.00000       17.00000
forrtl: severe (408): fort: (7): Attempt to use pointer A when it is not
associated with a target
```

  Example 3: Cray pointer with zero value

```
        pointer(p,a)
        real, target:: b
!       p=loc(b)        ! if integer pointer p has no address assigned to
it,
```

```
!                       ! the next statement gets an error with "check
pointers"
        b=17.
        print *,a
        end

Output 3:
forrtl: severe (408): fort: (9): Attempt to use pointee A when its
corresponding integer pointer P has the value zero
```

- The `-ftrapuv` (Linux OS and Mac OS X) or `/Qtrapuv` (Windows OS) 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` (Linux OS and Mac OS X) or `/traceback` (Windows OS) option generates extra information in the object file to provide source file traceback information when a severe error occurs at run time. This simplifies the task of locating the cause of severe run-time errors. Without `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. Certain traceback-related information accompanies severe run-time errors, as described in Using Traceback Information.
- The `-fpe` (Linux OS and Mac OS X) or `/fpe` (Windows OS) option controls the handling of floating-point arithmetic exceptions (IEEE arithmetic) at run time. If you specify the `-fpe3` (Linux OS and Mac OS X) or `/fpe:3` (Windows OS) compiler option, all floating-point exceptions are disabled, allowing IEEE exceptional values and program continuation. In contrast, specifying `-fpe0` or `/fpe:0` stops execution when an exceptional value (such as a NaN) is generated or when attempting to use a denormalized number, which usually allows you to localize the cause of the error.
- The `-warn` and `-nowarn` (Linux OS and Mac OS X) or `/warn` and `/nowarn` (Windows OS) options control compile-time warning messages, which, in some circumstances, can help determine the cause of a run-time error.
- On Linux OS and Mac OS X, the `-fexceptions` option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.
- On Windows OS, the Compilation Diagnostics Options in the IDE control compile-time diagnostic messages, which, in some circumstances can help determine the cause of a run-time error.

See also Understanding Run-Time Errors.

List of Run-Time Error Messages

This section 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:

`for_iosdef.for`

As described in the table, the severity of the message determines which of the following occurs:

- with `info` and `warning`, program execution continues
- with `error`, the results may be incorrect
- with `severe`, program execution stops (unless a recovery method is specified)

In the last case, to prevent program termination, you must include either an appropriate I/O error-handling specifier and recompile or, for certain errors, change the default action of a signal before you run the program again.

In the following table, the first column lists error numbers returned to IOSTAT variables when an I/O error is detected.

The first line of the second column provides the message as it is displayed (following *forrtl:*), including the severity level, message number, and the message text. The following lines of the second column contain the status condition symbol (such as FOR$IOS_INCRECTYP) and an explanation of the message.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
1[1]	**severe (1): Not a Fortran-specific error** FOR$IOS_NOTFORSPE. An error in the user program or in the RTL was not an Intel Fortran-specific error and was not reportable through any other Intel Fortran run-time messages.
8	**severe (8): Internal consistency check failure** FOR$IOS_BUG_CHECK. Internal error. Please check that the program is correct. Recompile if an error existed in the program. If this error persists, submit a problem report.
9	**severe (9): Permission to access file denied** FOR$IOS_PERACCFIL. Check the permissions of the specified file and whether the network device is mapped and available. Make sure the correct file and device was being accessed. Change the protection, specific file, or process used before rerunning the program.
10	**severe (10): Cannot overwrite existing file** FOR$IOS_CANOVEEXI. Specified file xxx already exists when OPEN statement specified STATUS='NEW' (create new file) using I/O unit x. Make sure correct file name, directory path, unit, and so forth were specified in the source program. Decide whether to: • Rename or remove the existing file before rerunning the program.

- Modify the source file to specify different file specification, I/O unit, or OPEN statement STATUS.

11[1] **info (11): Unit not connected**

FOR$IOS_UNINOTCON. The specified unit was not open at the time of the attempted I/O operation. Check if correct unit number was specified. If appropriate, use an OPEN statement to explicitly open the file (connect the file to the unit number).

17 **severe (17): Syntax error in NAMELIST input**

FOR$IOS_SYNERRNAM. The syntax of input to a namelist-directed READ statement was incorrect.

18 **severe (18): Too many values for NAMELIST variable**

FOR$IOS_TOOMANVAL. An attempt was made to assign too many values to a variable during a namelist READ statement.

19 **severe (19): Invalid reference to variable in NAMELIST input**

FOR$IOS_INVREFVAR. One of the following conditions occurred:

- The variable was not a member of the namelist group.
- An attempt was made to subscript a scalar variable.
- A subscript of the array variable was out-of-bounds.
- An array variable was specified with too many or too few subscripts for the variable.
- 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:

- An Intel Fortran RTL I/O system end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification.
- An end-of-file record written by the ENDFILE statement was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification.
- An attempt was made to read past the end of an internal file character string or array during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification.

This error is returned by END and ERRSNS.

25 **severe (25): Record number outside range**

FOR$IOS_RECNUMOUT. A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was opened.

26 **severe (26): OPEN or DEFINE FILE required**

FOR$IOS_OPEDEFREQ. A direct access READ, WRITE, or FIND statement was attempted for a file when no prior DEFINE FILE or OPEN statement with ACCESS='DIRECT' was performed for that file.

27 **severe (27): Too many records in I/O statement**

FOR$IOS_TOOMANREC. An attempt was made to do one of the following:

- Read or write more than one record with an ENCODE or DECODE statement.
- Write more records than existed.

28 **severe (28): CLOSE error**

FOR$IOS_CLOERR. An error condition was detected by the Intel Fortran RTL I/O system during execution of a CLOSE statement.

29 **severe (29): File not found**

FOR$IOS_FILNOTFOU. A file with the specified name could not be found during an open operation.

30 **severe (30): Open failure**

FOR$IOS_OPEFAI. An error was detected by the Intel Fortran RTL I/O system while attempting to open a file in an OPEN, INQUIRE, or other I/O statement. This message is issued when the error condition is not one of the more common conditions for which specific error messages are provided. It can occur when an OPEN operation was attempted for one of the following:

- Segmented file that was not on a disk or a raw magnetic tape
- Standard I/O file that had been closed

31 **severe (31): Mixed file access modes**

FOR$IOS_MIXFILACC. An attempt was made to use any of the following combinations:

- Formatted and unformatted operations on the same unit
- An invalid combination of access modes on a unit, such as direct and sequential
- An Intel Fortran RTL I/O statement on a logical unit that was opened by a program coded in another language

32 **severe (32): Invalid logical unit number**

FOR$IOS_INVLOGUNI. A logical unit number greater than 2,147,483,647 or less than zero was used in an I/O statement.

33 **severe (33): ENDFILE error**

FOR$IOS_ENDFILERR. One of the following conditions occurred:

- The file was not a sequential organization file with variable-length records.
- The file was not opened for sequential, append, or direct access.
- An unformatted file did not contain segmented records.
- The Intel Fortran RTL I/O system detected an error during execution of an ENDFILE statement.

34 **severe (34): Unit already open**

FOR$IOS_UNIALROPE. A DEFINE FILE statement specified a logical unit that was already opened.

35 **severe (35): Segmented record format error**

FOR$IOS_SEGRECFOR. An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or was created by a program written in a language other than Fortran or Fortran 90.

36 **severe (36): Attempt to access non-existent record**

FOR$IOS_ATTACCNON. A direct-access READ or **FIND** statement attempted to access beyond the end of a relative file (or a sequential file on disk with fixed-length records) or access a record that was previously deleted from a relative file.

37 **severe (37): Inconsistent record length**

FOR$IOS_INCRECLEN. An attempt was made to open a direct access file without specifying a record length.

38 **severe (38): Error during write**

FOR$IOS_ERRDURWRI. The Intel Fortran RTL I/O system detected an error condition during execution of a WRITE statement.

39 **severe (39): Error during read**

FOR$IOS_ERRDURREA. The Intel Fortran RTL I/O system detected an error condition during execution of a READ statement.

40 **severe (40): Recursive I/O operation**

FOR$IOS_RECIO_OPE. While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted, such as a function subprogram that performs I/O to the same logical unit that was referenced in an expression in an I/O list or variable format expression.

41 **severe (41): Insufficient virtual memory**

FOR$IOS_INSVIRMEM. The Intel Fortran RTL attempted to exceed its available virtual memory while dynamically allocating space. To overcome this problem, investigate increasing the data limit. Before you try to run this program again, wait until the new system resources take effect.

Note: This error can be returned by STAT in an ALLOCATE or a DEALLOCATE statement.

42 **severe (42): No such device**

FOR$IOS_NO_SUCDEV. A pathname included an invalid or unknown device name when an OPEN operation was attempted.

43 **severe (43): File name specification error**

FOR$IOS_FILNAMSPE. A pathname or file name given to an OPEN or INQUIRE statement was not acceptable to the Intel Fortran RTL I/O system.

44 **severe (44): Inconsistent record type**

FOR$IOS_INCRECTYP. The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.

45 **severe (45): Keyword value error in OPEN statement**

FOR$IOS_KEYVALERR. An improper value was specified for an OPEN or CLOSE statement specifier requiring a value.

46 **severe (46): Inconsistent OPEN/CLOSE parameters**

FOR$IOS_INCOPECLO. Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations follow:

- READONLY or ACTION='READ' with STATUS='NEW' or STATUS='SCRATCH'
- READONLY with STATUS='REPLACE', ACTION='WRITE', or ACTION='READWRITE'

- ACCESS='APPEND' with READONLY, ACTION='READ', STATUS='NEW', or STATUS='SCRATCH'
- DISPOSE='SAVE', 'PRINT', or 'SUBMIT' with STATUS='SCRATCH'
- DISPOSE='DELETE' with READONLY
- CLOSE statement STATUS='DELETE' with OPEN statement READONLY
- ACCESS='APPEND' with STATUS='REPLACE'
- ACCESS='DIRECT' with POSITION='APPEND' or 'ASIS'

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 more information, see the FORMAT statement.

59 **severe (59): List-directed I/O syntax error**

FOR$IOS_LISIO_SYN. The data in a list-directed input record had an invalid format, or the type of the constant was incompatible with the corresponding variable. The value of the variable was unchanged.

Note: The ERR transfer is taken after completion of the I/O statement for error number 59. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

60 **severe (60): Infinite format loop**

FOR$IOS_INFFORLOO. The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.

61 **severe or info (61): Format/variable-type mismatch**

FOR$IOS_FORVARMIS. An attempt was made either to read or write a real variable with an integer field descriptor (I, L, O, Z, B), or to read or write an integer or logical variable with a real field descriptor (D, E, or F). To suppress this error message, see the description of /check:noformat.

Note: The severity depends on the -check keywords or /check:keywords option used during the compilation command. The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

62 **severe (62): Syntax error in format**

FOR$IOS_SYNERRFOR. A syntax error was encountered while the RTL was processing a format stored in an array or character variable.

63 **error or info (63): Output conversion error**

FOR$IOS_OUTCONERR. During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. When this situation is encountered, the overflowed field is filled with asterisks to indicate the error in the output record. If no ERR address has been defined for this error, the program continues after the error message is displayed. To suppress this error message, see the description of /check:nooutput_conversion.

Note: The severity depends on the -check keywords or /check:keywords option used during the compilation command. The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

64 **severe (64): Input conversion error**

FOR$IOS_INPCONERR. During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable was set to zero.

Note: The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer

as soon as the error is detected, so file status and record position are undefined.

65 **error (65): Floating invalid**

FOR$IOS_FLTINV. During an arithmetic operation, the floating-point values used in a calculation were invalid for the type of operation requested or invalid exceptional values. For example, the error can occur if you request a log of the floating-point values 0.0 or a negative number. For certain arithmetic expressions, specifying the /check:nopower option can suppress this message.

66 **severe (66): Output statement overflows record**

FOR$IOS_OUTSTAOVE. An output statement attempted to transfer more data than would fit in the maximum record size.

67 **severe (67): Input statement requires too much data**

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. The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of 1 was assumed, except for a P edit descriptor, for which a value of zero was assumed.

Note: The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.

69[1] **error (69): Process interrupted (SIGINT)**

FOR$IOS_SIGINT. The process received the signal SIGINT. Determine source of this interrupt signal (described in `signal(3)`).

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 /integer-size:size option).

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. See Data Representation for ranges of the various data types.

73[1] **error (73): Floating divide by zero**

FOR$IOS_FLTDIV. During a floating-point arithmetic operation, an attempt was made to divide by zero.

74[1] **error (74): Floating underflow**

FOR$IOS_FLTUND. During an arithmetic operation, a floating-point value became less than the smallest finite value for that data type. Depending on the values of the /fpe:*n* option, the underflowed result was either set to zero or allowed to gradually underflow. See the Data Representation for ranges of the various data types.

75[1] **error (75): Floating point exception**

FOR$IOS_SIGFPE. A floating-point exception occurred. Possible causes include:

- Division by zero
- Overflow
- An invalid operation, such as subtraction of infinite values, multiplication of zero by infinity without signs), division of zero by zero or infinity by infinity
- Conversion of floating-point to fixed-point format when an overflow prevents conversion

76[1] **error (76): IOT trap signal**
FOR$IOS_SIGIOT. Core dump file created. Examine core dump for possible cause of this IOT signal.

77[1] **severe (77): Subscript out of range**

FOR$IOS_SUBRNG. An array reference was detected outside the declared array bounds.

78[1] **error (78): Process killed**

FOR$IOS_SIGTERM. The process received a signal requesting termination of this process. Determine the source of this software termination signal.

79[1] **error (79): Process quit**

FOR$IOS_SIGQUIT. The process received a signal requesting termination of itself. Determine the source of this quit signal.

95[1] **info (95): Floating-point conversion failed**

FOR$IOS_FLOCONFAI. The attempted unformatted read or write of nonnative floating-point data failed because the floating-point value:

- Exceeded the allowable maximum value for the equivalent native format and was set equal to infinity (plus or minus)
- Was infinity (plus or minus) and was set to infinity (plus or minus)
- Was invalid and was set to not a number (NaN)

Very small numbers are set to zero (0). This error could be caused by the specified nonnative floating-point format not matching the floating-point format found in the

specified file. Check the following:

- The correct file was specified.
- The record layout matches the format Intel Fortran is expecting.
- The ranges for the data being used (see Data Representation).
- The correct nonnative floating-point data format was specified (see Supported Native and Nonnative Numeric Formats).

96 **info (96): F_UFMTENDIAN environment variable was ignored:erroneous syntax**

FOR$IOS_UFMTENDIAN. Syntax for specifying whether little endian or big endian conversion is performed for a given Fortran unit was incorrect. Even though the program will run, the results might not be correct if you do not change the value of F_UFMTENDIAN. For correct syntax, see Environment Variable F_UFMTENDIAN Method.

108 **Severe (108): Cannot stat file**

FOR$IOS_CANSTAFILE. 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**

FOR$IOS_BRK_RANGE. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.

139[1] **severe: (139): Array index out of bounds for index nn**

FOR$IOS_BRK_RANGE2. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.

140[1] **error (140): Floating inexact**

FOR$IOS_FLTINE. A floating-point arithmetic or conversion operation gave a result that differs from the mathematically exact result. This trap is reported if the rounded result of an IEEE operation is not exact.

144[1] **severe (144): Reserved operand**

FOR$IOS_ROPRAND. The Intel Fortran RTL encountered a reserved operand while executing your program. Please report the problem to Intel.

145[1] **severe (145): Assertion error**

FOR$IOS_ASSERTERR. The Intel Fortran RTL encountered an assertion error. Please report the problem to Intel.

146[1] **severe (146): Null pointer error**

FOR$IOS_NULPTRERR. Attempted to use a pointer that does not contain an address. Modify the source program, recompile, and relink.

147[1] **severe (147): Stack overflow**

FOR$IOS_STKOVF. The Intel Fortran RTL encountered a stack overflow while executing your program.

148[1] **severe (148): String length error**

FOR$IOS_STRLENERR. During a string operation, an integer value appears in a context where the value of the integer is outside the permissible string length range. Recompile with the /check:bounds option.

149[1] **severe (149): Substring error**

FOR$IOS_SUBSTRERR. An array subscript is outside the dimensioned boundaries of an array. Recompile with the /check:bounds option.

150[1] **severe (150): Range error**

FOR$IOS_RANGEERR. An integer value appears in a context where the value of the integer is outside the permissible range.

151[1] **severe (151): Allocatable array is already allocated**

FOR$IOS_INVREALLOC. An allocatable array must not already be allocated when you attempt to allocate it. You must deallocate the array before it can again be allocated.

Note: This error can be returned by STAT in an ALLOCATE statement.

152[1] **severe (152): Unresolved contention for DEC Fortran RTL global resource**

FOR$IOS_RESACQFAI. Failed to acquire an Intel Fortran RTL global resource for a reentrant routine. For a multithreaded program, the requested global resource is held by a different thread in your program. For a program using asynchronous handlers, the requested global resource is held by the calling part of the program (such as main program) and your asynchronous handler attempted to acquire the same global resource.

153[1] **severe (153): Allocatable array or pointer is not allocated**

FOR$IOS_INVDEALLOC. A 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.

Note: This error can be returned by STAT in a DEALLOCATE statement.

154[1] **severe(154): Array index out of bounds**

FOR$IOS_RANGE. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.

155[1] **severe(155): Array index out of bounds for index nn**

FOR$IOS_RANGE2. An array subscript is outside the dimensioned boundaries of

that array. Recompile with the /check:bounds option set.

156[1] **severe(156): GENTRAP code = hex dec**

FOR$IOS_DEF_GENTRAP. The Intel Fortran RTL has detected an unknown GENTRAP code. The cause is most likely a software problem due to memory corruption, or software signalling an exception with an incorrect exception code. Try recompiling with the /check:bounds option set to see if that finds the problem.

157[1] **severe(157): Program Exception - access violation**

FOR$IOS_ACCVIO. The program tried to read from or write to a virtual address for which it does not have the appropriate access. Try recompiling with the /check:bounds option set, to see if the problem is an out-of-bounds memory reference or a argument mismatch that causes data to be treated as an address.

Other causes of this error include:

- Mismatches in C vs. STDCALL calling mechanisms, causing the stack to become corrupted
- References to unallocated pointers
- Attempting to access a protected (for example, read-only) address

158[1] **severe(158): Program Exception - datatype misalignment**

FOR$IOS_DTYPE_MISALIGN. The Intel Fortran RTL has detected data that is not aligned on a natural boundary for the data type specified. For example, a REAL(8) data item aligned on natural boundaries has an address that is a multiple of 8. To ensure naturally aligned data, use the /align option.

This is an operating system error. See your operating system documentation for more information.

159[1] **severe(159): Program Exception - breakpoint**

FOR$IOS_PGM_BPT. The Intel Fortran RTL has encountered a breakpoint in the program.

This is an operating system error. See your operating system documentation for more information.

160[1] **severe(160): Program Exception - single step**

FOR$IOS_PGM_SS. A trace trap or other single-instruction mechanism has signaled that one instruction has been executed.

This is an operating system error. See your operating system documentation for more information.

161[1] **severe(161): Program Exception - array bounds exceeded**

FOR$IOS_PGM_BOUNDS. The program tried to access an array element that is outside the specified boundaries of the array. Recompile with the /check:bounds option set.

162[1] **severe(162): Program Exception - denormal floating-point operand**

FOR$IOS_PGM_DENORM. A floating-point arithmetic or conversion operation has a

denormalized number as an operand. A denormalized number is smaller than the lowest value in the normal range for the data type specified. See Data Representation for ranges for floating-point types.

Either locate and correct the source code causing the denormalized value or, if a denormalized value is acceptable, specify a different value for the /fpe compiler option to allow program continuation.

163[1] **severe(163): Program Exception - floating stack check**

FOR$IOS_PGM_FLTSTK. During a floating-point operation, the floating-point register stack on systems using IA-32 architecture overflowed or underflowed. This is a fatal exception. The most likely cause is calling a REAL function as if it were an INTEGER function or subroutine, or calling an INTEGER function or subroutine as if it were a REAL function.

Carefully check that the calling code and routine being called agree as to how the routine is declared. If you are unable to resolve the issue, please send a problem report with an example to Intel.

164[1] **severe(164): Program Exception - integer divide by zero**

FOR$IOS_PGM_INTDIV. During an integer arithmetic operation, an attempt was made to divide by zero. Locate and correct the source code causing the integer divide by zero.

165[1] **severe(165): Program Exception - integer overflow**

FOR$IOS_PGM_INTOVF. During an arithmetic operation, an integer value exceeded the largest representable value for that data type. See Data Representation for ranges for INTEGER types.

166[1] **severe(166): Program Exception - privileged instruction**

FOR$IOS_PGM_PRIVINST. The program tried to execute an instruction whose operation is not allowed in the current machine mode.

This is an operating system error. See your operating system documentation for more information.

167[1] **severe(167): Program Exception - in page error**

FOR$IOS_PGM_INPGERR. The program tried to access a page that was not present, so the system was unable to load the page. For example, this error might occur if a network connection was lost while trying to run a program over the network.

This is an operating system error. See your operating system documentation for more information.

168[1] **severe(168): Program Exception - illegal instruction**

FOR$IOS_PGM_ILLINST. The program tried to execute an invalid instruction.

This is an operating system error. See your operating system documentation for more information.

169[1] **severe(169): Program Exception - noncontinuable exception**

FOR$IOS_PGM_NOCONTEXCP. The program tried to continue execution after a noncontinuable exception occurred.

This is an operating system error. See your operating system documentation for more information.

170[1] **severe(170): Program Exception - stack overflow**

FOR$IOS_PGM_STKOVF. The Intel Fortran RTL has detected a stack overflow while executing your program. See your Release Notes for information on how to increase stack size.

171[1] **severe(171): Program Exception - invalid disposition**

FOR$IOS_PGM_INVDISP. An exception handler returned an invalid disposition to the exception dispatcher. Programmers using a high-level language should never encounter this exception.

This is an operating system error. See your operating system documentation for more information.

172[1] **severe(172): Program Exception - exception code = hex dec**

FOR$IOS_PGM_EXCP_CODE. The Intel Fortran RTL has detected an unknown exception code.

This is an operating system error. See your operating system documentation for more information.

173[1] **severe(173): A pointer passed to DEALLOCATE points to an array that cannot be deallocated**

FOR$IOS_INVDEALLOC2. A pointer that was passed to DEALLOCATE pointed to an explicit array, an array slice, or some other type of memory that could not be deallocated in a DEALLOCATE statement. Only whole arrays previous allocated with an ALLOCATE statement may be validly passed to DEALLOCATE.

Note: This error can be returned by STAT in a DEALLOCATE statement.

174[1] **severe (174): SIGSEGV, message-text**

FOR$IOS_SIGSEGV. One of two possible messages occurs for this error number:

- `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] **severe(179): Cannot allocate array - overflow on array size calculation**

FOR$IOS_ARRSIZEOVF. An attempt to dynamically allocate storage for an array failed because the required storage size exceeds addressable memory.

Note: This error can be returned by STAT in an ALLOCATE statement.

256 **severe (256): Unformatted I/O to unit open for formatted transfers**

FOR$IOS_UNFIO_FMT. Attempted unformatted I/O to a unit where the OPEN statement (FORM specifier) indicated the file was formatted. Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains unformatted data, specify FORM='UNFORMATTED'in the OPEN statement. Otherwise, if appropriate, use formatted I/O (such as list-directed or namelist I/O).

257 **severe (257): Formatted I/O to unit open for unformatted transfers**

FOR$IOS_FMTIO_UNF. Attempted formatted I/O (such as list-directed or namelist I/O) to a unit where the OPEN statement indicated the file was unformatted (FORM specifier). Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains formatted data, specify FORM='FORMATTED' in the OPEN statement. Otherwise, if appropriate, use unformatted I/O.

259 **severe (259): Sequential-access I/O to unit open for direct access**

FOR$IOS_SEQIO_DIR. The OPEN statement for this unit number specified direct

access and the I/O statement specifies sequential access. Check the OPEN statement and make sure the I/O statement uses the correct unit number and type of access.

264 **severe (264): operation requires file to be on disk or tape**

FOR$IOS_OPEREQDIS. Attempted to use a BACKSPACE statement on such devices as a terminal.

265 **severe (265): operation requires sequential file organization and access**

FOR$IOS_OPEREQSEQ. Attempted to use a BACKSPACE statement on a file whose organization was not sequential or whose access was not sequential. A BACKSPACE statement can only be used for sequential files opened for sequential access.

266[1] **error (266): Fortran abort routine called**

FOR$IOS_PROABOUSE. The program called the abort routine to terminate itself.

268[1] **severe (268): End of record during read**

FOR$IOS_ENDRECDUR. An end-of-record condition was encountered during execution of a nonadvancing I/O **READ** statement that did not specify the EOR branch specifier.

296[1] **info(296): nn floating inexact traps**

FOR$IOS_FLOINEEXC. The total number of floating-point inexact data traps encountered during program execution was *nn*. This summary message appears at program completion.

297[1] **info (297): nn floating invalid traps**

FOR$IOS_FLOINVEXC. The total number of floating-point invalid data traps encountered during program execution was *nn*. This summary message appears at program completion.

298[1] **info (298): nn floating overflow traps**

FOR$IOS_FLOOVFEXC. The total number of floating-point overflow traps encountered during program execution was *nn*. This summary message appears at program completion.

299[1] **info (299): nn floating divide-by-zero traps**

FOR$IOS_FLODIV0EXC. The total number of floating-point divide-by-zero traps encountered during program execution was *nn*. This summary message appears at program completion.

300[1] **info (300): nn floating underflow traps**

FOR$IOS_FLOUNDEXC. The total number of floating-point underflow traps encountered during program execution was *nn*. This summary message appears at program completion.

540 **severe (540): Array or substring subscript expression out of range**

FOR$IOS_F6096. An expression used to index an array was smaller than the lower

dimension bound or larger than the upper dimension bound.

541 **severe (541): CHARACTER substring expression out of range**

FOR$IOS_F6097. An expression used to index a character substring was illegal.

542 **severe (542): Label not found in assigned GOTO list**

FOR$IOS_F6098. The label assigned to the integer-variable name was not specified in the label list of the assigned GOTO statement.

543 **severe (543): INTEGER arithmetic overflow**

FOR$IOS_F6099. This error occurs whenever integer arithmetic results in overflow.

544 **severe (544): INTEGER overflow on input**

FOR$IOS_F6100. An integer item exceeded the legal size limits.

An INTEGER (1) item must be in the range -127 to 128. An INTEGER (2) item must be in the range -32,767 to 32,768. An INTEGER (4) item must be in the range -2,147,483,647 to 2,147,483,648.

545 **severe (545): Invalid INTEGER**

FOR$IOS_F6101. Either an illegal character appeared as part of an integer, or a numeric character larger than the radix was used in an alternate radix specifier.

546 **severe (546): REAL indefinite (uninitialized or previous error)**

FOR$IOS_F6102. An invalid real number was read from a file, an internal variable, or the console. This can happen if an invalid number is generated by passing an illegal argument to an intrinsic function -- for example, SQRT(-1) or ASIN(2). If the invalid result is written and then later read, the error will be generated.

547 **severe (547): Invalid REAL**

FOR$IOS_F103. An illegal character appeared as part of a real number.

548 **severe (548): REAL math overflow**

FOR$IOS_F6104. A real value was too large. Floating-point overflows in either direct or emulated mode generate NaN (Not-A-Number) exceptions, which appear in the output field as asterisks (*) or the letters NAN.

550 **severe (550): INTEGER assignment overflow**

FOR$IOS_F6106. This error occurs when assignment to an integer is out of range.

551 **severe (551): Formatted I/O not consistent with OPEN options**

FOR$IOS_F6200. The program tried to perform formatted I/O on a unit opened with FORM='UNFORMATTED' or FORM='BINARY'.

552 **severe (552): List-directed I/O not consistent with OPEN options**

FOR$IOS_F6201. The program tried to perform list-directed I/O on a file that was not opened with FORM='FORMATTED' and ACCESS='SEQUENTIAL'.

553 **severe (553): Terminal I/O not consistent with OPEN options**

FOR$IOS_F6202. When a special device such as CON, LPT1, or PRN is opened in an OPEN statement, its access must be sequential and its format must be either formatted or binary. By default ACCESS='SEQUENTIAL' and FORM='FORMATTED' in OPEN statements.

To generate this error the device's OPEN statement must contain an option not appropriate for a terminal device, such as ACCESS='DIRECT' or FORM='UNFORMATTED'.

554 **severe (554): Direct I/O not consistent with OPEN options**

FOR$IOS_F6203. A REC= option was included in a statement that transferred data to a file that was opened with the ACCESS='SEQUENTIAL' option.

555 **severe (555): Unformatted I/O not consistent with OPEN options**

FOR$IOS_F6204. If a file is opened with FORM='FORMATTED', unformatted or binary data transfer is prohibited.

556 **severe (556): A edit descriptor expected for CHARACTER**

FOR$IOS_F6205. The A edit descriptor was not specified when a character data item was read or written using formatted I/O.

557 **severe (557): E, F, D, or G edit descriptor expected for REAL**

FOR$IOS_F6206. The E, F, D, or G edit descriptor was not specified when a real data item was read or written using formatted I/O.

558 **severe (558): I edit descriptor expected for INTEGER**

FOR$IOS_F6207. The I edit descriptor was not specified when an integer data item was read or written using formatted I/O.

559 **severe (559): L edit descriptor expected for LOGICAL**

FOR$IOS_F6208. The L edit descriptor was not specified when a logical data item was read or written using formatted I/O.

560 **severe (560): File already open: parameter mismatch**

FOR$IOS_F6209. An **OPEN** statement specified a connection between a unit and a filename that was already in effect. In this case, only the BLANK= option can have a different setting.

561 **severe (561): Namelist I/O not consistent with OPEN options**

FOR$IOS_F6210. The program tried to perform namelist I/O on a file that was not opened with FORM='FORMATTED' and ACCESS='SEQUENTIAL.'

562 **severe (562): IOFOCUS option illegal with non-window unit**

FOR$IOS_F6211. IOFOCUS was specified in an OPEN or INQUIRE statement for a non-window unit. The IOFOCUS option can only be used when the unit opened or inquired about is a QuickWin child window.

563 **severe (563): IOFOCUS option illegal without QuickWin**

FOR$IOS_F6212. IOFOCUS was specified in an OPEN or INQUIRE statement for a

non-QuickWin application. The IOFOCUS option can only be used when the unit opened or inquired about is a QuickWin child window.

564 **severe (564): TITLE illegal with non-window unit**

FOR$IOS_F6213. TITLE was specified in an OPEN or INQUIRE statement for a non-window unit. The TITLE option can only be used when the unit opened or inquired about is a QuickWin child window.

565 **severe (565): TITLE illegal without QuickWin**

FOR$IOS_F6214. TITLE was specified in an OPEN or INQUIRE statement for a non-QuickWin application. The TITLE option can only be used when the unit opened or inquired about is a QuickWin child window.

566 **severe (566): KEEP illegal for scratch file**

FOR$IOS_F6300. STATUS='KEEP' was specified for a scratch file; this is illegal because scratch files are automatically deleted at program termination.

567 **severe (567): SCRATCH illegal for named file**

FOR$IOS_F6301. STATUS='SCRATCH' should not be used in a statement that includes a filename.

568 **severe (568): Multiple radix specifiers**

FOR$IOS_F6302. More than one alternate radix for numeric I/O was specified. F6302 can indicate an error in spacing or a mismatched format for data of different radices.

569 **severe (569): Illegal radix specifier**

FOR$IOS_F6303. A radix specifier was not between 2 and 36, inclusive. Alternate radix constants must be of the form n#ddd... where n is a radix from 2 to 36 inclusive and ddd... are digits with values less than the radix. For example, 3#12 and 34#7AX are valid constants with valid radix specifiers. 245#7A and 39#12 do not have valid radix specifiers and generate error 569 if input.

570 **severe (570): Illegal STATUS value**

FOR$IOS_F6304. An illegal value was used with the STATUS option.

STATUS accepts the following values:

- 'KEEP' or 'DELETE' when used with CLOSE statements
- 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN' when used with OPEN statements

571 **severe (571): Illegal MODE value**

FOR$IOS_F6305. An illegal value was used with the MODE option.

MODE accepts the values 'READ', 'WRITE', or 'READWRITE'.

572 **severe (572): Illegal ACCESS value**

FOR$IOS_F6306. An illegal value was used with the ACCESS option.

ACCESS accepts the values 'SEQUENTIAL' and 'DIRECT'.

573 **severe (573): Illegal BLANK value**

FOR$IOS_F6307. An illegal value was used with the BLANK option.

BLANK accepts the values 'NULL' and 'ZERO'.

574 **severe (574): Illegal FORM value**

FOR$IOS_F6308. An illegal value was used with the FORM option.

FORM accepts the following values: 'FORMATTED', 'UNFORMATTED', and 'BINARY'.

575 **severe (575): Illegal SHARE value**

FOR$IOS_F6309. An illegal value was used with the SHARE option.

SHARE accepts the values 'COMPAT', 'DENYRW', 'DENYWR', 'DENYRD', and 'DENYNONE'.

577 **severe (577): Illegal record number**

FOR$IOS_F6311. An invalid number was specified as the record number for a direct-access file.

The first valid record number for direct-access files is 1.

578 **severe (578): No unit number associated with ***

FOR$IOS_F6312. In an INQUIRE statement, the NUMBER option was specified for the file associated with * (console).

580 **severe (580): Illegal unit number**

FOR$IOS_F6314. An illegal unit number was specified.

Legal unit numbers can range from 0 through 2**31-1, inclusive.

581 **severe (581): Illegal RECL value**

FOR$IOS_F6315. A negative or zero record length was specified for a direct file.

The smallest valid record length for direct files is 1.

582 **severe (582): Array already allocated**

FOR$IOS_F6316. The program attempted to **ALLOCATE** an already allocated array.

583 **severe (583): Array size zero or negative**

FOR$IOS_F6317. The size specified for an array in an ALLOCATE statement must be greater than zero.

584 **severe (584): Non-HUGE array exceeds 64K**

FOR$IOS_F6318.

585 **severe (585): Array not allocated**

FOR$IOS_F6319. The program attempted to **DEALLOCATE** an array that was never allocated.

586 **severe (586): BACKSPACE illegal on terminal device**

FOR$IOS_F6400. A BACKSPACE statement specified a unit connected to a terminal device such as a terminal or printer.

587 **severe (587): EOF illegal on terminal device**

FOR$IOS_F6401. An EOF intrinsic function specified a unit connected to a terminal device such as a terminal or printer.

588 **severe (588): ENDFILE illegal on terminal device**

FOR$IOS_F6402. An ENDFILE statement specified a unit connected to a terminal device such as a terminal or printer.

589 **severe (589): REWIND illegal on terminal device**

FOR$IOS_F6403. A REWIND statement specified a unit connected to a terminal device such as a terminal or printer.

590 **severe (590): DELETE illegal for read-only file**

FOR$IOS_F6404. A CLOSE statement specified STATUS='DELETE' for a read-only file.

591 **severe (591): External I/O illegal beyond end of file**

FOR$IOS_F6405. The program tried to access a file after executing an ENDFILE statement or after it encountered the end-of-file record during a read operation.

A BACKSPACE, REWIND, or OPEN statement must be used to reposition the file before execution of any I/O statement that transfers data.

592 **severe (592): Truncation error: file closed**

FOR$IOS_F6406.

593 **severe (593): Terminal buffer overflow**

FOR$IOS_F6407. More than 131 characters were input to a record of a unit connected to the terminal (keyboard). Note that the operating system may impose additional limits on the number of characters that can be input to the terminal in a single record.

594 **severe (594): Comma delimiter disabled after left repositioning**

FOR$IOS_F6408. If you have record lengths that exceed the buffer size associated with the record, (for instance, the record is a file with the buffer set by BLOCKSIZE in the OPEN statement), either you should not do left tabbing within the record, or you should not use commas as field delimiters. This is because commas are disabled as input field delimiters if left tabbing leaves the record positioned in a previous buffer.

For example, consider you have a file LONG.DAT that is one continuous record with data fields separated by commas. You then set the buffer associated with the file to 512 bytes, read more than one buffer size of data, tab left to data in the previous buffer, and attempt to read further data, as follows:

```
INTEGER value(300)
OPEN (1, FILE = 'LONG.DAT', BLOCKSIZE = 512)
```

```
    READ (1, 100) (value(i), i = 1, 300)
100 FORMAT (290I2,TL50,10I2)
```

In this case, error 594 occurs.

599 **severe (599): File already connected to a different unit**

FOR$IOS_F6413. The program tried to connect an already connected file to a new unit.

A file can be connected to only one unit at a time.

600 **severe (600): Access not allowed**

FOR$IOS_F6414.

This error can be caused by one of the following:

- The filename specified in an OPEN statement was a directory.
- An OPEN statement tried to open a read-only file for writing.
- The file was opened with SHARE='DENYRW' by another process.

601 **severe (601): File already exists**

FOR$IOS_F6415. An OPEN statement specified STATUS='NEW' for a file that already exists.

602 **severe (602): File not found**

FOR$IOS_F6416. An OPEN statement specified STATUS='OLD' for a specified file or a directory path that does not exist.

603 **severe (603): Too many open files**

FOR$IOS_F6417. The program exceeded the number of open files the operating system allows.

604 **severe (604): Too many units connected**

FOR$IOS_F6418. The program exceeded the number of units that can be connected at one time. Units are connected with the OPEN statement.

605 **severe (605): Illegal structure for unformatted file**

FOR$IOS_F6419. The file was opened with FORM='UNFORMATTED' and ACCESS='SEQUENTIAL', but its internal physical structure was incorrect or inconsistent. Possible causes: the file was created in another mode or by a non-Fortran program.

606 **severe (606): Unknown unit number**

FOR$IOS_F6420. A statement such as BACKSPACE or ENDFILE specified a file that had not yet been opened. (The READ and WRITE statements do not cause this problem because they prompt you for a file if the file has not been opened yet.)

607 **severe (607): File read-only or locked against writing**

FOR$IOS_F6421. The program tried to transfer data to a file that was opened in

read-only mode or locked against writing.

The error message may indicate a CLOSE error when the fault is actually coming from WRITE. This is because the error is not discovered until the program tries to write buffered data when it closes the file.

608 **severe (608): No space left on device**

FOR$IOS_F6422. The program tried to transfer data to a file residing on a device (such as a hard disk) that was out of storage space.

609 **severe (609): Too many threads**

FOR$IOS_F6423. Too many threads were active simultaneously. At most, 32 threads can be active at one time. Close any unnecessary processes or child windows within your application.

610 **severe (610): Invalid argument**

FOR$IOS_F6424.

611 **severe (611): BACKSPACE illegal for SEQUENTIAL write-only files**

FOR$IOS_F6425. The BACKSPACE statement is not allowed in files opened with MODE='WRITE' (write-only status) because BACKSPACE requires reading the previous record in the file to provide positioning.

Resolve the problem by giving the file read access or by avoiding the BACKSPACE statement. Note that the REWIND statement is valid for files opened as write-only.

612 **severe (612): File not open for reading or file locked**

FOR$IOS_F6500. The program tried to read from a file that was not opened for reading or was locked.

613 **severe (613): End of file encountered**

FOR$IOS_F6501. The program tried to read more data than the file contains.

614 **severe (614): Positive integer expected in repeat field**

FOR$IOS_F6502. When the *i*c* form is used in list-directed input, the *i* must be a positive integer. For example, consider the following statement:

```
READ(*,*) a, b
```

Input 2*56.7 is accepted, but input 2.1*56.7 returns error 614.

615 **severe (615): Multiple repeat field**

FOR$IOS_F6503. In list-directed input of the form *i*c*, an extra repeat field was used. For example, consider the following:

```
READ(*,*) I, J, K
```

Input of 2*1*3 returns this error. The 2*1 means send two values, each 1; the *3 is an error.

616 **severe (616): Invalid number in input**

FOR$IOS_F6504. Some of the values in a list-directed input record were not numeric. For example, consider the following:

```
READ(*,*) I, J
```

The preceding statement would cause this error if the input were: 123 'abc'.

617 **severe (617): Invalid string in input**

FOR$IOS_F6505. A string item was not enclosed in single quotation marks.

618 **severe (618): Comma missing in COMPLEX input**

FOR$IOS_F6506. When using list-directed input, the real and imaginary components of a complex number were not separated by a comma.

619 **severe (619): T or F expected in LOGICAL read**

FOR$IOS_F6507. The wrong format was used for the input field for logical data.

The input field for logical data consists of optional blanks, followed by an optional decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field, so that .TRUE. and .FALSE. are acceptable input forms.

620 **severe (620): Too many bytes read from unformatted record**

FOR$IOS_F6508. The program tried to read more data from an unformatted file than the current record contained. If the program was reading from an unformatted direct file, it tried to read more than the fixed record length as specified by the RECL option. If the program was reading from an unformatted sequential file, it tried to read more data than was written to the record.

621 **severe (621): H or apostrophe edit descriptor illegal on input**

FOR$IOS_F6509. Hollerith (H) or apostrophe edit descriptors were encountered in a format used by a READ statement.

622 **severe (622): Illegal character in hexadecimal input**

FOR$IOS_F6510. The input field contained a character that was not hexadecimal.

Legal hexadecimal characters are 0 - 9 and A - F.

623 **severe (623): Variable name not found**

FOR$IOS_F6511. A name encountered on input from a namelist record is not declared in the corresponding NAMELIST statement.

624 **severe (624): Invalid NAMELIST input format**

FOR$IOS_F6512. The input record is not in the correct form for namelist input.

625 **severe (625): Wrong number of array dimensions**

FOR$IOS_F6513. In namelist input, an array name was qualified with a different number of subscripts than its declaration, or a non-array name was qualified.

626 **severe (626): Array subscript exceeds allocated area**

FOR$IOS_F6514. A subscript was specified in namelist input which exceeded the declared dimensions of the array.

627 **severe (627): Invalid subrange in NAMELIST input**

FOR$IOS_F6515. A character item in namelist input was qualified with a subrange that did not meet the requirement that 1 <= e1 <= e2 <= len (where "len" is the length of the character item, "e1" is the leftmost position of the substring, and "e2" is the rightmost position of the substring).

628 **severe (628): Substring range specified on non-CHARACTER item**

FOR$IOS_F6516. A non-CHARACTER item in namelist input was qualified with a substring range.

629 **severe (629): Internal file overflow**

FOR$IOS_F6600. The program either overflowed an internal-file record or tried to write to a record beyond the end of an internal file.

630 **severe (630): Direct record overflow**

FOR$IOS_F6601. The program tried to write more than the number of bytes specified in the RECL option to an individual record of a direct-access file.

631 **severe (631):Numeric field bigger than record size**

FOR$IOS_F6602. The program tried to write a noncharacter item across a record boundary in list-directed or namelist output. Only character constants can cross record boundaries.

632 **severe (632): Heap space limit exceeded**

FOR$IOS_F6700. The program ran out of heap space. The ALLOCATE statement and various internal functions allocate memory from the heap. This error will be generated when the last of the heap space is used up.

633 **severe (633): Scratch file name limit exceeded**

FOR$IOS_F6701. The program exhausted the template used to generate unique scratch-file names. The maximum number of scratch files that can be open at one time is 26.

634 **severe (634): D field exceeds W field in ES edit descriptor**

FOR$IOS_F6970. The specified decimal length D exceeds the specified total field width W in an ES edit descriptor.

635 **severe (635): D field exceeds W field in EN edit descriptor**

FOR$IOS_F6971. The specified decimal length D exceeds the specified total field width W in an EN edit descriptor.

636 **severe (636): Exponent of 0 not allowed in format**

FOR$IOS_F6972.

637 **severe (637): Integer expected in format**

FOR$IOS_F6980. An edit descriptor lacked a required integer value. For example, consider the following:

```
     WRITE(*, 100) I, J
 100 FORMAT (I2, TL, I2)
```

The preceding code will cause this error because an integer is expected after TL.

638 **severe (638): Initial left parenthesis expected in format**

FOR$IOS_F6981. A format did not begin with a left parenthesis (( ).

639 **severe (639): Positive integer expected in format**

FOR$IOS_F6982. A zero or negative integer value was used in a format.

Negative integer values can appear only with the P edit descriptor. Integer values of 0 can appear only in the *d* and *m* fields of numeric edit descriptors.

640 **severe (640): Repeat count on nonrepeatable descriptor**

FOR$IOS_F6983. One or more BN, BZ, S, SS, SP, T, TL, TR, /, $, :, or apostrophe (') edit descriptors had repeat counts associated with them.

641 **severe (641): Integer expected preceding H, X, or P edit descriptor**

FOR$IOS_F6984. An integer did not precede a (nonrepeatable) H, X, or P edit descriptor.

The correct formats for these descriptors are *n*H, *n*X, and *k*P, respectively, where *n* is a positive integer and *k* is an optionally signed integer.

642 **severe (642): N or Z expected after B in format**

FOR$IOS_F6985. To control interpretation of embedded and trailing blanks within numeric input fields, you must specify BN (to ignore them) or BZ (to interpret them as zeros).

643 **severe (643): Format nesting limit exceeded**

FOR$IOS_F6986. More than 16 sets of parentheses were nested inside the main level of parentheses in a format.

644 **severe (644): '.' expected in format**

FOR$IOS_F6987. No period appeared between the *w* and *d* fields of a D, E, F, or G edit descriptor.

645 **severe (645): Unexpected end of format**

FOR$IOS_F6988. An incomplete format was used.

Improperly matched parentheses, an unfinished Hollerith (H) descriptor, or another incomplete descriptor specification can cause this error.

646 **severe (646): Unexpected character in format**

FOR$IOS_F6989. A character that cannot be interpreted as part of a valid edit

descriptor was used in a format. For example, consider the following:

```
      WRITE(*, 100) I, J
  100 FORMAT (I2, TL4.5, I2)
```

The code will generate this error because TL4.5 is not a valid edit descriptor. An integer must follow TL.

647 **severe (647): M field exceeds W field in I edit descriptor**

FOR$IOS_F6990. In syntax I*w*.*m*, the value of *m* cannot exceed the value of *w*.

648 **severe (648): Integer out of range in format**

FOR$IOS_F6991. An integer value specified in an edit descriptor was too large to represent as a 4-byte integer.

649 **severe (649): format not set by ASSIGN**

FOR$IOS_F6992. The format specifier in a READ, WRITE, or PRINT statement was an integer variable, but an ASSIGN statement did not properly assign it the statement label of a FORMAT statement in the same program unit.

650 **severe (650): Separator expected in format**

FOR$IOS_F6993. Within format specifications, edit descriptors must be separated by commas or slashes (/).

651 **severe (651): %c or $: nonstandard edit descriptor in format**

FOR$IOS_F6994.

652 **severe (652): Z: nonstandard edit descriptor in format**

FOR$IOS_F6995. Z is not a standard edit descriptor in format.

If you want to transfer hexadecimal values, you must use the edit descriptor form Z*w*[.*m*], where *w* is the field width and *m* is the minimum number of digits that must be in the field (including leading zeros).

653 **severe (653): DOS graphics not supported under Windows NT**

FOR$IOS_F6996.

654 **severe (654): Graphics error**

FOR$IOS_F6997. An OPEN statement in which IOFOCUS was TRUE, either explicitly or by default, failed because the new window could not receive focus. The window handle may be invalid, or closed, or there may be a memory resource problem.

655 **severe (655): Using QuickWin is illegal in console application**

FOR$IOS_F6998. A call to QuickWin from a console application was encountered during execution.

656 **severe (656): Illegal 'ADVANCE' value**

FOR$IOS_F6999. The ADVANCE option can only take the values 'YES' and 'NO'.

ADVANCE='YES' is the default. ADVANCE is a READ statement option.

657 **severe (657): DIM argument to SIZE out of range**

FOR$IOS_F6702. The argument specified for DIM must be greater than or equal to 1, and less than or equal to the number of dimensions in the specified array. Consider the following:

```
i = SIZE (array, DIM = dim)
```

In this case, 1 <= dim <= *n*, where *n* is the number of dimensions in array.

658 **severe (657): Undefined POINTER used as argument to ASSOCIATED function**

FOR$IOS_F6703. A POINTER used as an argument to the ASSOCIATED function must be defined; that is, assigned to a target, allocated, or nullified.

659 **severe (659): Reference to uninitialized POINTER**

FOR$IOS_F6704. Except in an assignment statement, a pointer must not be referenced until it has been initialized: assigned to a target, allocated or nullified.

660 **severe (660): Reference to POINTER which is not associated**

FOR$IOS_F6705. Except in an assignment statement and certain procedure references, a pointer must not be referenced until it has been associated: either assigned to a target or allocated.

661 **severe (661): Reference to uninitialized POINTER 'pointer'**

FOR$IOS_F6706. Except in an assignment statement, a pointer must not be referenced until it has been initialized: assigned to a target, allocated or nullified.

662 **severe (662): reference to POINTER 'pointer' which is not associated**

FOR$IOS_F6707. Except in an assignment statement and certain procedure references, a pointer must not be referenced until it has been associated: either assigned to a target or allocated.

663 **severe (663): Out of range: substring starting position 'pos' is less than 1**

FOR$IOS_F6708. A substring starting position must be a positive integer variable or expression that indicates a position in the string: at least 1 and no greater than the length of the string.

664 **severe (664): Out of range: substring ending position 'pos' is greater than string length 'len'**

FOR$IOS_F6709. A substring ending position must be a positive integer variable or expression that indicates a position in the string: at least 1 and no greater than the length of the string.

665 **severe (665): Subscript 'n' of 'str' (value 'val') is out of range ('first:last')**

FOR$IOS_F6710. The subscript for a substring within a string is not a valid string position: at least 1 and no greater than the length of the string.

666 **severe (666): Subscript 'n' of 'str' (value 'val') is out of range ('first:*')**

FOR$IOS_F6711. The subscript for a substring within a string is not a valid string position: at least 1 and no greater than the length of the string.

667 **severe (667): VECTOR argument to PACK has incompatible character length**

FOR$IOS_F6712. The character length of elements in the VECTOR argument to PACK is not the same as the character length of elements in the array to be packed.

668 **severe (668): VECTOR argument to PACK is too small**

FOR$IOS_F6713. The VECTOR argument to PACK must have at least as many elements as there are true elements in MASK (the array that controls packing).

669 **severe (669): SOURCE and PAD arguments to RESHAPE have different character lengths**

FOR$IOS_F6714. The character length of elements in the SOURCE and PAD arguments to PACK must be the same.

670 **severe (670): Element 'n' of SHAPE argument to RESHAPE is negative**

FOR$IOS_F6715. The SHAPE vector specifies the shape of the reshaped array. Since an array cannot have a negative dimension, SHAPE cannot have a negative element.

671 **severe (671): SOURCE too small for specified SHAPE in RESHAPE, and no PAD**

FOR$IOS_F6716. If there is no PAD array, the SOURCE argument to RESHAPE must have enough elements to make an array of the shape specified by SHAPE.

672 **severe (672): Out of memory**

FOR$IOS_F6717. The system ran out of memory while trying to make the array specified by RESHAPE. If possible, reset your virtual memory size through the Windows Control Panel, or close unneccessary applications and deallocate all allocated arrays that are no longer needed.

673 **severe (673): SHAPE and ORDER arguments to RESHAPE have different sizes ('size1' and 'size2')**

FOR$IOS_F6718. ORDER specifies the order of the array dimensions given in SHAPE, and they must be vectors of the same size.

674 **severe (674): Element 'n' of ORDER argument to RESHAPE is out of range ('range')**

FOR$IOS_F6719. The ORDER argument specifies the order of the dimensions of the reshaped array, and it must be a permuted list of (1, 2, ..., *n*) where *n* is the highest dimension in the reshaped array.

675 **severe (675): Value 'val' occurs twice in ORDER argument to RESHAPE**

FOR$IOS_F6720. The ORDER vector specifies the order of the dimensions of the

reshaped array, and it must be a permuted list of (1, 2, ..., *n*) where *n* is the highest dimension in the reshaped array. No dimension can occur twice.

676 **severe (676): Impossible nextelt overflow in RESHAPE**

FOR$IOS_F6721.

677 **severe (677): Invalid value 'dim' for argument DIM for SPREAD of rank 'rank' source**

FOR$IOS_F6722. The argument specified for DIM to SPREAD must be greater than or equal to 1, and less than or equal to one larger than the number of dimensions (rank) of SOURCE. Consider the following statement:

```
result = SPREAD (SOURCE= array, DIM = dim, NCOPIES = k)
```

In this case, 1 <= dim <= *n* + 1, where *n* is the number of dimensions in array.

678 **severe (678): Complex zero raised to power zero**

FOR$IOS_F6723. Zero of any type (complex, real, or integer) cannot be raised to zero power.

679 **severe (679): Complex zero raised to negative power**

FOR$IOS_F6724. Zero of any type (complex, real, or integer) cannot be raised to a negative power. Raising to a negative power inverts the operand.

680 **severe (680): Impossible error in NAMELIST input**

FOR$IOS_F6725.

681 **severe (681):DIM argument to CSHIFT ('dim') is out of range**

FOR$IOS_F6726. The optional argument DIM specifies the dimension along which to perform the circular shift, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array to be shifted. That is, 1 <=DIM <= *n*, where *n* is the number of dimensions in the array to be shifted.

682 **severe (682): DIM argument ('dim') to CSHIFT is out of range (1:'n')**

FOR$IOS_F6727. The optional argument DIM specifies the dimension along which to perform the circular shift, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array to be shifted. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in the array to be shifted.

683 **severe (683): Shape mismatch (dimension 'dim') between ARRAY and SHIFT in CSHIFT**

FOR$IOS_F6728. The SHIFT argument to CSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the SHIFT must conform to the shape of the array being shifted in every dimension except the one being shifted along.

684 **severe (684): Internal error - bad arguments to CSHIFT_CA**

FOR$IOS_F6729.

685 **severe (685): Internal error - bad arguments to CSHIFT_CAA**

FOR$IOS_F6730.

686 **severe (686): DATE argument to DATE_AND_TIME is too short (LEN='len')**

FOR$IOS_F6731. The character DATE argument must have a length of at least 8 to contain the complete value.

687 **severe (687): TIME argument to DATE_AND_TIME is too short (LEN='len')**

FOR$IOS_F6732. The character TIME argument must have a length of at least 10 to contain the complete value.

688 **severe (688): ZONE argument to DATE_AND_TIME is too short (LEN='len')**

FOR$IOS_F6733. The character ZONE argument must have a length of at least 5 to contain the complete value.

689 **severe (689): VALUES argument to DATE_AND_TIME is too small ('size' elements)**

FOR$IOS_F6734. The integer VALUES argument must be a one-dimensional array with a size of at least 8 to hold all returned values.

690 **severe (690): Out of range: DIM argument to COUNT has value 'dim'**

FOR$IOS_F6735. The optional argument DIM specifies the dimension along which to count true elements of MASK, and must be greater than or equal to 1 and less than or equal to the number of dimensions in MASK. That is, 1 <= DIM <= n, where n is the number of dimensions in MASK.

691 **severe (691): Out of range: DIM argument to COUNT has value 'dim' with MASK of rank 'rank'**

FOR$IOS_F6736. The optional argument DIM specifies the dimension along which to count true elements of MASK, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in MASK. That is, 1 <= DIM <= n, where n is the number of dimensions in MASK.

692 **severe (692): Out of range: DIM argument to PRODUCT has value 'dim'**

FOR$IOS_F6737. The optional argument DIM specifies the dimension along which to compute the product of elements in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array holding the elements to be multiplied.

693 **severe (693): Out of range: DIM argument to PRODUCT has value 'dim' with ARRAY of rank 'rank'**

FOR$IOS_F6738. The optional argument DIM specifies the dimension along which to compute the product of elements in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) of the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array holding the elements to be multiplied.

694 **severe (694): Out of range: DIM argument to SUM has value 'dim' with ARRAY of rank 'rank'**

FOR$IOS_F6739. The optional argument DIM specifies the dimension along which to sum the elements of an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) of the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array holding the elements to be summed.

695 **severe (695): Real zero raised to zero power**

FOR$IOS_F6740. Zero of any type (real, complex, or integer) cannot be raised to zero power.

696 **severe (696): Real zero raised to negative power**

FOR$IOS_F6741. Zero of any type (real, complex, or integer) cannot be raised to a negative power. Raising to a negative power inverts the operand.

697 **severe (697): Out of range: DIM argument to SUM has value 'dim'**

FOR$IOS_F6742. The optional argument DIM specifies the dimension along which to sum the elements of an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array holding the elements to be summed.

698 **severe (698): DIM argument ('dim') to EOSHIFT is out of range (1:'n')**

FOR$IOS_F6743. The optional argument DIM specifies the dimension along which to perform an end-off shift in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array holding the elements to be shifted.

699 **severe (699): Shape mismatch (dimension 'dim') between ARRAY and BOUNDARY in EOSHIFT**

FOR$IOS_F6744. The BOUNDARY argument to EOSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the BOUNDARY must conform to the shape of the array being shifted in every dimension except the one being shifted along.

700 **severe (700): DIM argument to EOSHIFT is out of range ('dim')**

FOR$IOS_F6745. The optional argument DIM specifies the dimension along which to perform an end-off shift in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array holding the elements to be shifted.

701 **severe (701): Shape mismatch (dimension 'dim') between ARRAY and SHIFT in EOSHIFT**

FOR$IOS_F6746. The SHIFT argument to EOSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the SHIFT must conform to the shape of the array being shifted in every dimension

except the one being shifted along.

702 **severe (702): BOUNDARY argument to EOSHIFT has wrong LEN ('len1 instead of len2')**

FOR$IOS_F6747. The character length of elements in the BOUNDARY argument and in the array being end-off shifted must be the same.

703 **severe (703): BOUNDARY has LEN 'len' instead of 'len' to EOSHIFT**

FOR$IOS_F6748.

704 **severe (704): Internal error - bad arguments to EOSHIFT**

FOR$IOS_F6749.

705 **severe (705): GETARG: value of argument 'num' is out of range**

FOR$IOS_F6750. The value used for the number of the command-line argument to retrieve with GETARG must be 0 or a positive integer. If the number of the argument to be retrieved is greater than the actual number of arguments, blanks are returned, but no error occurs.

706 **severe (706): FLUSH: value of LUNIT 'num' is out of range**

FOR$IOS_F6751. The unit number specifying which I/O unit to flush to its associated file must be an integer between 0 and 2**31-1, inclusive. If the unit number is valid, but the unit is not opened, error F6752 is generated.

707 **severe (707): FLUSH: Unit 'n' is not connected**

FOR$IOS_F6752. The I/O unit specified to be flushed to its associated file is not connected to a file.

708 **severe (708): Invalid string length ('len') to ICHAR**

FOR$IOS_F6753. The character argument to ICHAR must have length 1.

709 **severe (709): Invalid string length ('len') to IACHAR**

FOR$IOS_F6754. The character argument to IACHAR must have length 1.

710 **severe (710): Integer zero raised to negative power**

FOR$IOS_F6755. Zero of any type (integer, real, or complex) cannot be raised to a negative power. Raising to a negative power inverts the operand.

711 **severe (711): INTEGER zero raised to zero power**

FOR$IOS_F6756. Zero of any type (integer, real, or complex) cannot be raised to zero power.

712 **severe (712): SIZE argument ('size') to ISHFTC intrinsic out of range**

FOR$IOS_F6757. The argument SIZE must be positive and must not exceed the bit size of the integer being shifted. The bit size of this integer can be determined with the function BIT_SIZE.

713 **severe (713): SHIFT argument ('shift') to ISHFTC intrinsic out of range**

FOR$IOS_F6758. The argument SHIFT to ISHFTC must be an integer whose absolute value is less than or equal to the number of bits being shifted: either all bits in the number being shifted or a subset specified by the optional argument SIZE.

714 **severe (714): Out of range: DIM argument to LBOUND has value 'dim'**

FOR$IOS_F6759. The optional argument DIM specifies the dimension whose lower bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array.

715 **severe (715): Out of range: DIM argument ('dim') to LBOUND greater than ARRAY rank 'rank'**

FOR$IOS_F6760. The optional argument DIM specifies the dimension whose lower bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array.

716 **severe (716): Out of range: DIM argument to MAXVAL has value 'dim'**

FOR$IOS_F6761. The optional argument DIM specifies the dimension along which maximum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array.

717 **severe (717): Out of range: DIM argument to MAXVAL has value 'dim' with ARRAY of rank 'rank'**

FOR$IOS_F6762. The optional argument DIM specifies the dimension along which maximum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= n, where n is the number of dimensions in array.

718 **severe (718): Cannot allocate temporary array -- out of memory**

FOR$IOS_F6763. There is not enough memory space to hold a temporary array.

Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

719 **severe (719): Attempt to DEALLOCATE part of a larger object**

FOR$IOS_F6764. An attempt was made to DEALLOCATE a pointer to an array subsection or an element within a derived type. The whole data object must be deallocated; parts cannot be deallocated.

720 **severe (720): Pointer in DEALLOCATE is ASSOCIATED with an ALLOCATABLE array**

FOR$IOS_F6765. Deallocating a pointer associated with an allocatable target is illegal. Instead, deallocate the target the pointer points to, which frees memory

and disassociates the pointer.

721 **severe (721): Attempt to DEALLOCATE an object which was not allocated**

FOR$IOS_F6766. You cannot deallocate an array unless it has been previously allocated. You cannot deallocate a pointer whose target was not created by allocation. The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated.

722 **severe (722): Cannot ALLOCATE scalar POINTER -- out of memory**

FOR$IOS_F6767. There is not enough memory space to allocate the pointer.

Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

723 **severe (723): DEALLOCATE: object not allocated/associated**

FOR$IOS_F6768. You cannot deallocate an array unless it has been previously allocated. You cannot deallocate a pointer whose target was not created by allocation, or a pointer that has undefined association status.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated.

724 **severe (724): Cannot ALLOCATE POINTER array -- out of memory**

FOR$IOS_F6769. There is not enough memory space to allocate the POINTER array.

Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

725 **severe (725): DEALLOCATE: Array not allocated**

FOR$IOS_F6770. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.

726 **severe (726): DEALLOCATE: Character array not allocated**

FOR$IOS_F6771. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.

727 **severe (727): Cannot ALLOCATE allocatable array -- out of memory**

FOR$IOS_F6772. There is not enough memory space to hold the array.

Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size

through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

728 **severe (728): Cannot allocate automatic object -- out of memory**

FOR$IOS_F6773. There is not enough memory space to hold the automatic data object.

Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

An automatic data object is an object that is declared in a procedure subprogram or interface, is not a dummy argument, and depends on a nonconstant expression. For example:

```
SUBROUTINE  EXAMPLE (N)
   DIMENSION A (N, 5), B(10*N)
```

The arrays A and B in the example are automatic data objects.

729 **severe (729): DEALLOCATE failure: ALLOCATABLE array is not ALLOCATED**

FOR$IOS_F6774. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.

730 **severe (730): Out of range: DIM argument to MINVAL has value 'dim'**

FOR$IOS_F6775. The optional argument DIM specifies the dimension along which minimum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.

731 **severe (731): Out of range: DIM argument to MINVAL has value 'dim' with ARRAY of rank 'rank** /p>

FOR$IOS_F6776. The optional argument DIM specifies the dimension along which minimum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.

732 **severe (732): P argument to MOD is double precision zero**

FOR$IOS_F6777. MOD(A,P) is computed as A - INT(A/P) * P. So, P cannot be zero.

733 **severe (733): P argument to MOD is integer zero**

FOR$IOS_F6778. MOD(A,P) is computed as A - INT(A/P) * P. So, P cannot be zero.

734 **severe (734): P argument to MOD is real zero**

FOR$IOS_F6779. MOD(A,P) is computed as A - INT(A/P) * P. So, P cannot be zero.

735 **severe (735): P argument to MODULO is real zero**

FOR$IOS_F6780. >MODULO(A,P) for real numbers is computed as A - FLOOR(A/P) * P. So, P cannot be zero.

736 **severe (736): P argument to MODULO is zero**

FOR$IOS_F6781. In the function, MODULO(A,P), P cannot be zero.

737 **severe (737): Argument S to NEAREST is zero**

FOR$IOS_F6782. The sign of the S argument to NEAREST(X,S) determines the direction of the search for the nearest number to X, and cannot be zero.

738 **severe (738): Heap storage exhausted**

FOR$IOS_F6783.

739 **severe (739): PUT argument to RANDOM_SEED is too small**

FOR$IOS_F6784. The integer array PUT must be greater than or equal to the number of integers the processor uses to set the seed value. This number can be determined by calling RANDOM_SEED with the SIZE argument. For example:

```
INTEGER, ALLOCATABLE SEED
CALL RANDOM_SEED( )          ! initialize processor
CALL RANDOM_SEED(SIZE = K)    ! get size of seed
ALLOCATE SEED(K)             ! allocate array
CALL RANDOM_SEED(PUT = SEED)  ! set the seed
```

Note that RANDOM_SEED can be called with at most one argument at a time.

740 **severe (740): GET argument to RANDOM_SEED is too small**

FOR$IOS_F6785. The integer array GET must be greater than or equal to the number of integers the processor uses to set the seed value. This number can be determined by calling RANDOM_SEED with the SIZE argument. For example:

```
INTEGER, ALLOCATABLE SEED
CALL RANDOM_SEED( )          ! initialize processor
CALL RANDOM_SEED(SIZE = K)    ! get size of seed
ALLOCATE SEED(K)             ! allocate array
CALL RANDOM_SEED(GET = SEED)  ! get the seed
```

Note that RANDOM_SEED can be called with at most one argument at a time.

741 **severe (741): Recursive I/O reference**

FOR$IOS_F6786.

742 **severe (742): Argument to SHAPE intrinsic is not PRESENT**

FOR$IOS_F6787.

743 **severe (743): Out of range: DIM argument to UBOUND had value 'dim'**

FOR$IOS_F6788. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or

equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.

744 **severe (744): DIM argument ('dim') to UBOUND greater than ARRAY rank 'rank'**

FOR$IOS_F6789. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.

745 **severe (745): Out of range: UBOUND of assumed-size array with DIM==rank ('rank')**

FOR$IOS_F6790. The optional argument DIM specifies the dimension whose upper bound is to be returned.

An assumed-size array is a dummy argument in a subroutine or function, and the upper bound of its last dimension is determined by the size of actual array passed to it. Assumed-size arrays have no determined shape, and you cannot use UBOUND to determine the extent of the last dimension. You can use UBOUND to determine the upper bound of one of the fixed dimensions, in which case you must pass the dimension number along with the array name.

746 **severe (746): Out of range: DIM argument ('dim') to UBOUND greater than ARRAY rank**

FOR$IOS_F6791. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.

747 **severe (747): Shape mismatch: Dimension 'shape' extents are 'ext1' and 'ext2'**

FOR$IOS_F6792.

748 **severe (748): Illegal POSITION value**

FOR$IOS_F6793. An illegal value was used with the POSITION specifier.

POSITION accepts the following values:

- 'ASIS' (the default)
- 'REWIND' - on Fortran I/O systems, this is the same as 'ASIS'
- 'APPEND'

749 **severe (749): Illegal ACTION value**

FOR$IOS_F6794. An illegal value was used with the ACTION specifier.

ACTION accepts the following values:

- 'READ'
- 'WRITE'
- 'READWRITE' - the default

750 **severe (750): DELIM= specifier not allowed for an UNFORMATTED file**

FOR$IOS_F6795. The DELIM specifier is only allowed for files connected for formatted data transfer. It is used to delimit character constants in list-directed an namelist output.

751 **severe (751): Illegal DELIM value**

FOR$IOS_F6796. An illegal value was used with the DELIM specifier.

DELIM accepts the following values:

- 'APOSTROPHE'
- 'QUOTE'
- 'NONE' - the default

752 **severe (752): PAD= specifier not allowed for an UNFORMATTED file**

FOR$IOS_F6797. The PAD specifier is only allowed for formatted input records. It indicates whether the formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.

753 **severe (753): Illegal PAD= value**

FOR$IOS_F6798. An illegal value was used with the PAD specifier.

PAD accepts the following values:

- 'NO'
- 'YES' - the default

754 **severe (754): Illegal CARRIAGECONTROL= value**

FOR$IOS_F6799. An illegal value was used with the CARRIAGECONTROL specifier.

CARRIAGECONTROL accepts the following values:

- 'FORTRAN' - default if the unit is connected to a terminal or console
- 'LIST' - default for formatted files
- 'NONE' - default for unformatted files

755 **severe (755): SIZE= specifier only allowed with ADVANCE='NO'**

FOR$IOS_F6800. The SIZE specifier can only appear in a formatted, sequential READ statement that has the specifier ADVANCE='NO' (indicating nonadvancing input).

756 **severe (756): Illegal character in binary input**

FOR$IOS_F6801.

757 **severe (757): Illegal character in octal input**

FOR$IOS_F6802.

758 **severe (758): End of record encountered**

FOR$IOS_F6803.

759	**severe (759): Illegal subscript in namelist input record** FOR$IOS_F6804.

Footnotes:

1 Identifies errors not returned by IOSTAT.

Signal Handling (Linux* OS and Mac OS* X only)

A *signal* is an abnormal event generated by one of various sources, such as:

- A user of a terminal
- Program or hardware error
- Request of another program
- When a process is stopped to allow access to the control terminal

You can optionally set certain events to issue signals, for example:

- When a process resumes after being stopped
- When the status of a child process changes
- When input is ready at the terminal

Some signals terminate the receiving process if no action is taken (optionally creating a `core` file), while others are simply ignored unless the process has requested otherwise.

Except for certain signals, calling the `signal` or `sigaction` routine allows specified signals to be ignored or causes an interrupt (transfer of control) to the location of a user-written signal handler.

You can establish one of the following actions for a signal with a call to `signal`:

- Ignore the specified signal (identified by number).
- Use the default action for the specified signal, which can reset a previously established action.
- Transfer control from the specified signal to a procedure to receive the signal, specified by name.

Calling the `signal` routine lets you change the action for a signal, such as intercepting an operating system signal and preventing the process from being stopped.

The table below shows the signals that the Intel Fortran RTL arranges to catch when a program is started:

Signal	Intel Fortran RTL message
SIGFPE	`Floating-point exception` (number 75)
SIGINT	`Process interrupted` (number 69)
SIGIOT	`IOT trap signal` (number 76)
SIGQUIT	`Process quit` (number 79)
SIGSEGV	`Segmentation fault` (number 174)

SIGTERM `Process killed` (number 78)

Calling the `signal` routine (specifying the numbers for these signals) results in overwriting the signal-handling facility set up by the Intel Fortran RTL. The only way to restore the default action is to save the returned value from the first call to `signal`.

When using a debugger, it may be necessary to enter a command to allow the Intel Fortran RTL to receive and handle the appropriate signals.

Overriding the Default Run-Time Library Exception Handler

To override the default run-time library exception handler on Linux OS and Mac OS X, 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
```

Using Traceback Information

Using Traceback Information Overview

When a Fortran program terminates due to a severe error condition, the Fortran run-time system displays additional diagnostic information after the run-time message.

The Fortran run-time system attempts to walk back up the call chain and produce a report of the calling sequence leading to the error as part of the default diagnostic message report. This is known as traceback. The minimum information displayed includes:

- The standard Fortran run-time error message text that explains the error condition.

- A tabular report that contains one line per call stack frame. This information includes at least the image name and a hexadecimal PC in that image.

The information displayed under the Routine, Line, and Source columns depends on whether your program was compiled with the `-traceback` (Linux* OS and Mac OS* X) or `/traceback` (Windows* OS) option.

For example, if `-traceback` or `/traceback` is specified, the displayed information might resemble the following:

```
 forrtl: severe (24): end-of-file during read, unit 10, file
E:\USERS\xxx.dat
 Image              PC        Routine         Line     Source
 libifcorert.dll    1000A3B2  Unknown         Unknown  Unknown
 libifcorert.dll    1000A184  Unknown         Unknown  Unknown
 libifcorert.dll    10009324  Unknown         Unknown  Unknown
 libifcorert.dll    10009596  Unknown         Unknown  Unknown
 libifcorert.dll    10024193  Unknown         Unknown  Unknown
 teof.exe           004011A9  AGAIN                21  teof.for
 teof.exe           004010DD  GO                   15  teof.for
 teof.exe           004010A7  WE                   11  teof.for
 teof.exe           00401071  HERE                  7  teof.for
 teof.exe           00401035  TEOF                  3  teof.for
 teof.exe           004013D9  Unknown         Unknown  Unknown
 teof.exe           004012DF  Unknown         Unknown  Unknown
 KERNEL32.dll       77F1B304  Unknown         Unknown  Unknown
```

If the same program is not compiled with the `-traceback` or `/traceback` option:

- The Routine name, Line number, and Source file columns would be reported as "Unknown."
- A link map file is usually needed to locate the cause of the error.

The `-traceback` or `/traceback` option provides program counter (PC) to source file line correlation information to appear in the displayed error message information, which simplifies the task of locating the cause of severe run-time errors.

For Fortran objects generated with `-traceback` or `/traceback`, the compiler generates additional information used by the Fortran run-time system to automatically correlate PC values to the routine name in which they occur, Fortran source file, and line number in the source file. This information is displayed in the run-time error diagnostic report.

Automatic PC correlation is only supported for Fortran code. For non-Fortran code, only the hexadecimal PC locations are reported.

Tradeoffs and Restrictions in Using Traceback

This topic describes tradeoffs and restrictions that apply to using traceback.

Effect on Image Size

Using the `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS) option to get automatic PC correlation increases the size of an image. For any application, the developer must decide if the increase in image size is worth the benefit of automatic PC correlation or if manually correlating PCs with a map file is acceptable.

The approach of providing automatic correlation information in the image was used so that no run-time penalty is incurred by building the information "on the fly" as your application executes. No run-time diagnostic code is invoked unless your application is terminating due to a severe error.

C Compiler Omit Frame Pointer Option on Systems Using IA-32 Architecture

The following routines are used to walk the stack:

- For Windows OS, the Windows API routine StackWalk( ) in imagehlp.dll
- For Linux OS and Mac OS X, _Unwind_ForcedUnwind( ), _Unwind_GetIP( ), _Unwind_GetRegionStart( ) and _Unwind_GetGr( ) routines in libunwind.so

In an environment using IA-32 architecture, there are no firm software calling standards documented. Compiler developers are under no constraints to use machine registers in any particular way or to hook up procedures in any particular way. The stack walking routines listed above use a set of heuristics to determine how to walk the call stack. That is, they make a "best guess" to determine how a program reached a particular point in the call chain. With C code that has been compiled with Visual C++* using the Omit Frame Pointer option -- either `-fomit-frame-pointer` (Linux OS and Mac OS X) or `/Oy` (Windows OS) -- this "best guess" is not usually the correct one.

If you are mixing Fortran and C code and you are concerned about stack tracing, consider disabling the `-fomit-frame-pointer` or `/Oy` option in your C compilations. Otherwise, traceback will most likely not work for you.

Stack Trace Failure

Programs can fail for a number of reasons, often with unpredictable consequences. Memory corruption by erroneously executing code is one possibility. Stack memory can be corrupted in such a way that the attempt to trace the call stack will result in access violations or other undesirable consequences. The stack-tracing run-time code is guarded with a local exception filter. If the traceback attempt fails due to a hard detectable condition, the run-time will report this in its diagnostic output message as:

```
Stack trace terminated abnormally
```

Be forewarned, however: It is also possible for memory to be corrupted in such a way that a stack trace can seem to complete successfully with no hint of a problem. The bit patterns it finds in corrupted memory where the stack used to be, and then uses to access memory, may constitute perfectly valid memory addresses for the program to be accessing. They just do not happen to have any connection to what the stack used to look like. So, if it appears that the stack walk completed normally, but the reported PCs make no sense to you, then consider ignoring the stack trace output in diagnosing your problem.

Another condition that will disable the stack trace process is your program exiting because it has exhausted virtual memory resources.

The stack trace can fail if the run-time system cannot dynamically load libunwind.so (Linux OS and Mac OS X) or imagehlp.dll (Windows OS) or cannot find the necessary routines from that

library. In this case, you still get the basic run-time diagnostic message; you will not get any call stack information.

Linker /incremental:no Option on Windows Operating Systems

The following applies to Windows operating systems only.

When incremental linking is enabled, automatic PC correlation does not work. Use of incremental linking always disables automatic PC correlation even if you specify `/traceback` during compilation.

When you use incremental linking, the default hexadecimal (hex) PC values will still appear in the output. To correlate from the hexadecimal PC values to routine containing the PC addresses requires use of a linker map file. However, if you request a map file during linking, incremental linking becomes disabled. Thus to allow any PC values generated for a run-time problem to be helpful, incremental linking must be disabled.

In the integrated development environment, you can use the Call stack display, so incremental linking is not a problem.

Sample Programs and Traceback Information

The following sections provide sample programs that show the use of traceback to locate the cause of the error:

- Example: End-of-File Condition, Program teof
- Example: Machine Exception Condition, Program ovf
- Example: Using Traceback in Mixed Fortran/C Applications, Program FPING and CPONG

Note that the hex PC's and contents of registers displayed in these program outputs are meant as representative examples of typical output. The PC's will change over time, as the libraries and other tools used to create an image change.

Example: End-of-File Condition, Program teof

In the following example, a READ statement creates an End-Of-File error, which the application has not handled:

```
program teof
integer*4 i,res
i=here( )
end


integer*4 function here( )
here = we( )
end


integer*4 function we( )
we = go( )
```

```
  end


  integer*4 function go( )
  go = again( )
  end


  integer*4 function again( )
  integer*4 a
  open(10,file='xxx.dat',form='unformatted',status='unknown')
  read(10) a
  again=a
  end
```

The diagnostic output that results when this program is built with traceback enabled and linked against the single-threaded, shared Fortran run-time library on the IA-32 architecture platform is similar to the following:

```
forrtl: severe (24): end-of-file during read, unit 10, file
E:\USERS\xxx.dat
Image              PC        Routine          Line     Source
libifcorert.dll  1000A3B2  Unknown          Unknown  Unknown
libifcorert.dll  1000A184  Unknown          Unknown  Unknown
libifcorert.dll  10009324  Unknown          Unknown  Unknown
libifcorert.dll  10009596  Unknown          Unknown  Unknown
libifcorert.dll  10024193  Unknown          Unknown  Unknown
teof.exe         004011A9  AGAIN                 21  teof.for
teof.exe         004010DD  GO                    15  teof.for
teof.exe         004010A7  WE                    11  teof.for
teof.exe         00401071  HERE                   7  teof.for
teof.exe         00401035  TEOF                   3  teof.for
teof.exe         004013D9  Unknown          Unknown  Unknown
teof.exe         004012DF  Unknown          Unknown  Unknown
KERNEL32.dll     77F1B304  Unknown          Unknown  Unknown
```

The first line of the output is the standard Fortran run-time error message. What follows is the result of walking the call stack in reverse order to determine where the error originated. Each line of output represents a call frame on the stack. Since the application was compiled with `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS), the PCs that fall in Fortran code are correlated to their matching routine name, line number and source module. PCs that are not in Fortran code are not correlated and are reported as "Unknown."

The first five frames show the calls to routines in the Fortran run-time library (in reverse order). Since the application was linked against the single threaded, shared version of the library, the image name reported is either libifcore.so (Linux OS and Mac OS X) or libifcorert.dll (Windows OS). These are the run-time routines that were called to do the READ and upon detection of the EOF condition, were invoked to report the error. In the case of an unhandled I/O programming error, there will always be a few frames on the call stack down in run-time code like this.

The stack frame of real interest to the Fortran developer is the first frame in image teof.exe which shows that the error originated in the routine named AGAIN in source module `teof.for` at line 21. Looking in the source code at line 21, you can see the Fortran READ statement that incurred the end-of-file condition.

The next four frames show the trail of calls in the Fortran user code that led to the routine that got the error (TEOF->HERE->WE->GO->AGAIN).

Finally, the bottom three frames are routines which handled the startup and initialization of the program.

If this program had been linked against the single-threaded, static Fortran run-time library, the output would then look like:

```
forrtl: severe (24): end-of-file during read, unit 10, file
E:\USERS\xxx.dat
Image          PC        Routine        Line     Source
teof.exe       004067D2  Unknown        Unknown  Unknown
teof.exe       0040659F  Unknown        Unknown  Unknown
teof.exe       00405754  Unknown        Unknown  Unknown
teof.exe       004059C5  Unknown        Unknown  Unknown
teof.exe       00403543  Unknown        Unknown  Unknown
teof.exe       004011A9  AGAIN               21  teof.for
teof.exe       004010DD  GO                  15  teof.for
teof.exe       004010A7  WE                  11  teof.for
teof.exe       00401071  HERE                 7  teof.for
teof.exe       00401035  TEOF                 3  teof.for
teof.exe       004202F9  Unknown        Unknown  Unknown
teof.exe       00416063  Unknown        Unknown  Unknown
KERNEL32.dll   77F1B304  Unknown        Unknown  Unknown
```

Notice that the initial five stack frames now show routines in image `teof.exe`, not libifcore.so (Linux OS and Mac OS X) or `libifcorert.dll` (Windows OS). The routines are the same five run-time routines as previously reported for the shared library case but since the application was linked against the archive library libifcore.a (Linux OS and Mac OS X) or the static Fortran run-time library `libifcore.lib` (Windows OS), the object modules containing these routines were linked into the application image (`teof.exe`). You can use a map file to determine the locations of uncorrelated PCs.

Now suppose the application was compiled without traceback enabled and, once again, linked against the single-threaded, static Fortran library. The diagnostic output would appear as follows:

```
forrtl: severe (24): end-of-file during read, unit 10, file
E:\USERS\xxx.dat
Image            PC        Routine       Line     Source
teof.exe         00406792  Unknown       Unknown  Unknown
teof.exe         0040655F  Unknown       Unknown  Unknown
teof.exe         00405714  Unknown       Unknown  Unknown
teof.exe         00405985  Unknown       Unknown  Unknown
teof.exe         00403503  Unknown       Unknown  Unknown
teof.exe         00401169  Unknown       Unknown  Unknown
```

```
teof.exe          004010A8  Unknown       Unknown  Unknown
teof.exe          00401078  Unknown       Unknown  Unknown
teof.exe          00401048  Unknown       Unknown  Unknown
teof.exe          0040102F  Unknown       Unknown  Unknown
teof.exe          004202B9  Unknown       Unknown  Unknown
teof.exe          00416023  Unknown       Unknown  Unknown
KERNEL32.dll      77F1B304  Unknown       Unknown  Unknown
```

Without the correlation information in the image that `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS) previously supplied, the Fortran run-time system cannot correlate PC's to routine name, line number, and source file. You can still use the map file to at least determine the routine names and what modules they are in.

Remember that compiling with `-traceback` or `/traceback` increases the size of your application's image because of the extra PC correlation information included in the image. You can see if the extra traceback information is included in an image (checking for the presence of a `.trace` section) by entering:

```
objdump -h your_app.exe (Linux OS)
otool -l your_app.exe (Mac OS X)
link -dump -summary your_app.exe (Windows OS)
```

Build your application with and without traceback and compare the file size of each image. Check the file size with a simple directory command.

For this simple `teof.exe` example, the traceback correlation information adds about 512 bytes to the image size. In a real application, this would probably be much larger. For any application, the developer must decide if the increase in image size is worth the benefit of automatic PC correlation or if manually correlating PC's with a map file is acceptable.

If an error occurs when traceback was requested during compilation, the run-time library will produce the correlated call stack display.

If an error occurs when traceback was disabled during compilation, the run-time library will produce the uncorrelated call stack display.

If you do not want to see the call stack information displayed, you can set the environment variable `FOR_DISABLE_STACK_TRACE` to true. You will still get the Fortran run-time error message:

```
forrtl: severe (24): end-of-file during read, unit 10, file
E:\USERS\xxx.dat
```

Example: Machine Exception Condition, Program ovf

The following program generates a floating-point overflow exception when compiled with `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows OS):

```
 program ovf
 real*4 a
 a=1e37
 do i=1,10
    a=hey(a)
 end do
```

```
  print *, 'a= ', a
  end


  real*4 function hey(b)
  real*4 b
  hey = watch(b)
  end


  real*4 function watch(b)
  real*4 b
  watch = out(b)
  end


  real*4 function out(b)
  real*4 b
  out = below(b)
  end


  real*4 function below(b)
  real*4 b
  below = b*10.0e0
  end
```

Assume this program is compiled with the following:

- `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows OS)
- `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS)
- `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS)

On a system based on IA-32 architecture, the traceback output is similar to the following:

```
forrtl: error (72): floating overflow
Image              PC        Routine           Line     Source
ovf.exe            00401161  BELOW                29  ovf.f90
ovf.exe            0040113C  OUT                  24  ovf.f90
ovf.exe            0040111B  WATCH                19  ovf.f90
ovf.exe            004010FA  HEY                  14  ovf.f90
ovf.exe            0040105B  OVF                   7  ovf.f90
ovf.exe            00432429  Unknown         Unknown  Unknown
ovf.exe            00426C74  Unknown         Unknown  Unknown
KERNEL32.dll       77F1B9EA  Unknown         Unknown  Unknown
```

Notice that unlike the previous example of an unhandled I/O programming error, the stack walk can begin right at the point of the exception. There are no run-time routines on the call stack to dig through. The overflow occurs in routine BELOW at PC 00401161, which is correlated to line 29 of the source file `ovf.f90`.

When the program is compiled at a higher optimization level of `O2`, along with `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows) and `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS), the traceback output appears as follows:

```
forrtl: error (72): floating overflow
Image              PC        Routine        Line       Source
ovf.exe            00401070  OVF                 29    ovf.f90
ovf.exe            004323E9  Unknown        Unknown    Unknown
ovf.exe            00426C34  Unknown        Unknown    Unknown
KERNEL32.dll       77F1B9EA  Unknown        Unknown    Unknown
```

With `/O2`, the entire program has been inlined.

The main program, `OVF`, no longer calls routine `HEY`. While the output is not quite what one might have expected intuitively, it is still entirely correct. You need to keep in mind the effects of compiler optimization when you interpret the diagnostic information reported for a failure in a release image.

If the same image were executed again, this time with the environment variable called `TBK_ENABLE_VERBOSE_STACK_TRACE` set to True, you would also see a dump of the exception context record at the time of the error. Here is an excerpt of how that might appear on a system using IA-32 architecture:

```
forrtl: error (72): floating overflow
Hex Dump Of Exception Record Context Information:
Exception Context:  Processor Control and Status Registers.
EFlags:  00010212
CS:  0000001B  EIP:  00401161  SS:   00000023  ESP:  0012FE38  EBP:  00
12FE60
Exception Context:  Processor Integer Registers.
EAX: 00444488  EBX:  00000009  ECX:  00444488  EDX:  00000002
ESI: 0012FBBC  EDI:  F9A70030
Exception Context:  Processor Segment Registers.
DS:  00000023  ES:   00000023  FS:   00000038  GS:   00000000
Exception Context:  Floating Point Control and Status Registers.
ControlWord:  FFFF0262  ErrorOffset:    0040115E  DataOffset:    0012FE
5C
StatusWord:   FFFFF8A8  ErrorSelector:  015D001B  DataSelector:  FFFF00
23
TagWord:      FFFF3FFF  Cr0NpxState:    00000000
Exception Context:  Floating Point RegisterArea.
RegisterArea[00]:  4080BC143F4000000000  RegisterArea[10]:  F7A0FFFFFFF
F77F9D860
RegisterArea[20]:  00131EF0000800060012  RegisterArea[30]:  00000012F7C
002080006
RegisterArea[40]:  02080006000000000000  RegisterArea[50]:  00000000000
00012F7D0
RegisterArea[60]:  00000000000000300000  RegisterArea[70]:  FBBC0000003
00137D9EF
...
```

Example: Using Traceback in Mixed Fortran/C Applications, Program FPING and CPONG

Consider the following example that shows how the traceback output might appear in a mixed Fortran/C application. The main program is a Fortran program named `FPING`. Program `FPING` triggers a chain of function calls which are alternately Fortran and C code. Eventually, the C routine named Unlucky is called, which produces a floating divide-by-zero error.

Source module `FPING.FOR` contains the Fortran function definitions, each of which calls a C routine from source module `CPONG.C`. The program `FPING.FOR` is compiled with the following options:

- `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows OS)
- `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS)
- `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS)

On the IA-32 architecture platform. the program traceback output resembles the following:

```
forrtl: error (73): floating divide by zero
Image              PC        Routine       Line     Source
fping.exe          00401161  Unknown       Unknown  Unknown
fping.exe          004010DC  DOWN4              58  fping.for
fping.exe          0040118F  Unknown       Unknown  Unknown
fping.exe          004010B6  DOWN3              44  fping.for
fping.exe          00401181  Unknown       Unknown  Unknown
fping.exe          00401094  DOWN2              31  fping.for
fping.exe          00401173  Unknown       Unknown  Unknown
fping.exe          00401072  DOWN1              18  fping.for
fping.exe          0040104B  FPING               5  fping.for
fping.exe          004013B9  Unknown       Unknown  Unknown
fping.exe          004012AF  Unknown       Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown       Unknown  Unknown
```

Notice that the stack frames contributed by Fortran routines can be correlated to a routine name, line number, and source module but those frames contributed by C routines cannot be correlated. Remember, even though the stack can be walked in reverse, and PCs reported, the information necessary to correlate the PC to a routine name, line number, and so on, is contributed to the image from the objects generated by the Fortran compiler. The C compiler does not have this capability. Also remember that you only get the correlation information if you specify the `-traceback` or `/traceback` option for your Fortran compiles.

The top stack frame cannot be correlated to a routine name because it is in C code. You can examine the map file for the application; if you do so, you will see that the reported PC, 00401161, is greater than the start of routine _Unlucky, but less than the start of routine _down1_C. This means that the error occurred in routine _Unlucky.

In a similar manner, the other PCs reported as "Unknown" can be correlated to a routine name using the map file.

When examining traceback output (or any type of diagnostic output, for that matter), it is important to keep in mind the effects of compiler optimization. The Fortran source module in

the above example was built with optimization turned off. Look at the output when optimizations are enabled with `-O2` (Linux OS and Mac OS X) or `/O2` (Windows OS):

```
forrtl: error (73): floating divide by zero
Image              PC        Routine     Line     Source
fping.exe          00401111  Unknown     Unknown  Unknown
fping.exe          0040109D  DOWN4            58  fping.for
fping.exe          0040113F  Unknown     Unknown  Unknown
fping.exe          00401082  DOWN3            44  fping.for
fping.exe          00401131  Unknown     Unknown  Unknown
fping.exe          0040106B  DOWN2            31  fping.for
fping.exe          00401123  Unknown     Unknown  Unknown
fping.exe          00401032  FPING            18  fping.for
fping.exe          00401369  Unknown     Unknown  Unknown
fping.exe          0040125F  Unknown     Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown     Unknown  Unknown
```

From the traceback output, it would appear that routine DOWN1 was never called. In fact, it has not been called. At the higher optimization level, the compiler has inlined function DOWN1 so that the call to routine down1_C is now made from FPING. The correlated line number still points to the correct line in the source code.

Finally, suppose the example Fortran code is redesigned with each of the Fortran routines split into separate source modules. Here is what the traceback output would look like with the redesigned code:

```
forrtl: error (73): floating divide by zero
Image              PC        Routine      Line     Source
fpingmain.exe      00401171  Unknown      Unknown  Unknown
fpingmain.exe      004010ED  DOWN4             12  fping4.for
fpingmain.exe      0040119F  Unknown      Unknown  Unknown
fpingmain.exe      004010C1  DOWN3             11  fping3.for
fpingmain.exe      00401191  Unknown      Unknown  Unknown
fpingmain.exe      00401099  DOWN2             11  fping2.for
fpingmain.exe      00401183  Unknown      Unknown  Unknown
fpingmain.exe      00401073  DOWN1             11  fping1.for
fpingmain.exe      0040104B  FPING              5  fpingmain.for
fpingmain.exe      004013C9  Unknown      Unknown  Unknown
fpingmain.exe      004012BF  Unknown      Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown      Unknown  Unknown
```

Notice that the line number and source file correlation information has changed to reflect the new design of the code.

Here are the sources used in the above examples:

```
**********************************
FPING.FOR
**********************************
       program fping
       real*4 a,b
```

```
       a=-10.0
       b=down1(a)
       end
       real*4 function down1(b)
       real*4 b
!DEC$ IF DEFINED(_X86_)
       INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'_down1_C'] (n)
!DEC$ ELSE
       INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'down1_C'] (n)
!DEC$ ENDIF
       REAL*4 n [VALUE]
       END
       real*4 down1_C
       down1 = down1_C(b)
       end
       real*4 function down2(b)
       real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
       INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'_down2_C'] (n)
!DEC$ ELSE
       INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'down2_C'] (n)
!DEC$ ENDIF
       REAL*4 n [VALUE]
       END
       real*4 down2_C
       down2 = down2_C(b)
       end
       real*4 function down3(b)
       real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
       INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'_down3_C'] (n)
!DEC$ ELSE
       INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'down3_C'] (n)
!DEC$ ENDIF
       REAL*4 n [VALUE]
       END
       real*4 down3_C
       down3 = down3_C(b)
       end
       real*4 function down4(b)
       real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
       INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'_Unlucky'] (a,c)
!DEC$ ELSE
       INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'Unlucky'] (a,c)
!DEC$ ENDIF
       REAL*4 a [VALUE]
```

```
        REAL*4 c [REFERENCE]
        END
        real*4 a
        call Unlucky(b,a)
        down4 = a
        end

***********************************
CPONG.C
***********************************
#include <math.h>
extern float __stdcall DOWN2 (float n);
extern float __stdcall DOWN3 (float n);
extern float __stdcall DOWN4 (float n);
int Fact( int n )
{
    if (n > 1)
        return( n * Fact( n - 1 ));
    return 1;
}
void Pythagoras( float a, float b, float *c)
{
    *c = sqrt( a * a + b * b );
}
void Unlucky( float a, float *c)
{
float b=0.0;
    *c = a/b;
}
float down1_C( float a )
{
    return( DOWN2( a ));
}
float down2_C( float a )
{
    return( DOWN3( a ));
}
float down3_C( float a )
{
    return( DOWN4( a ));
}

***********************************
FPINGMAIN.FOR
***********************************
        program fping
        real*4 a,b
```

```
        a=-10.0
        b=down1(a)
        end

**********************************
FPING1.FOR
**********************************
        real*4 function down1(b)
        real*4 b
!DEC$ IF DEFINED(_X86_)
        INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'_down1_C'] (n)
!DEC$ ELSE
        INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'down1_C'] (n)
!DEC$ ENDIF
        REAL*4 n [VALUE]
        END
        real*4 down1_C
        down1 = down1_C(b)
        end

**********************************
FPING2.FOR
**********************************
        real*4 function down2(b)
        real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
        INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'_down2_C'] (n)
!DEC$ ELSE
        INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'down2_C'] (n)
!DEC$ ENDIF
        REAL*4 n [VALUE]
        END
        real*4 down2_C
        down2 = down2_C(b)
        end

**********************************
FPING3.FOR
**********************************
        real*4 function down3(b)
        real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
        INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'_down3_C'] (n)
!DEC$ ELSE
        INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'down3_C'] (n)
!DEC$ ENDIF
        REAL*4 n [VALUE]
```

```
        END
        real*4 down3_C
        down3 = down3_C(b)
        end

***********************************
FPING4.FOR
***********************************
        real*4 function down4(b)
        real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
        INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'_Unlucky'] (a,c)
!DEC$ ELSE
        INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'Unlucky'] (a,c)
!DEC$ ENDIF
        REAL*4 a [VALUE]
        REAL*4 c [REFERENCE]
        END
        real*4 a
        call Unlucky(b,a)
        down4 = a
        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.

Portability Considerations

Portability Considerations Overview

This section presents topics to help you understand how language standards, operating system differences, and computing hardware influence your use of Intel® Fortran and the portability of your programs.

Your program is portable if you can implement it on one hardware-software platform and then move it to additional systems with a minimum of changes to the source code. Correct results on the first system should be correct on the additional systems. The number of changes you might have to make when moving your program varies significantly. You might have no changes at all (strictly portable), or so many (non-portable customization) that it is more efficient to design or implement a new program. Most programs in their lifetime will need to be ported from one system to another, and this section can help you write code that makes this easy.

See these topics:

- Understanding Fortran Language Standards and related topics
- Minimizing Operating System-Specific Information
- Storing and Representing Data
- Formatting Data for Transportability
- Portability Library Overview

Portability Considerations Overview

This section presents topics to help you understand how language standards, operating system differences, and computing hardware influence your use of Intel® Fortran and the portability of your programs.

Your program is portable if you can implement it on one hardware-software platform and then move it to additional systems with a minimum of changes to the source code. Correct results on the first system should be correct on the additional systems. The number of changes you might have to make when moving your program varies significantly. You might have no changes at all (strictly portable), or so many (non-portable customization) that it is more efficient to design or implement a new program. Most programs in their lifetime will need to be ported from one system to another, and this section can help you write code that makes this easy.

See these topics:

- Understanding Fortran Language Standards and related topics
- Minimizing Operating System-Specific Information
- Storing and Representing Data
- Formatting Data for Transportability
- Portability Library Overview

Understanding Fortran Language Standards

Understanding Fortran Language Standards Overview

A language standard specifies the form and establishes the interpretation of programs expressed in the language. Its primary purpose is to promote, among vendors and users, portability of programs across a variety of systems.

The vendor-user community has adopted four major Fortran language standards. ANSI (American National Standards Institute) and ISO (International Standards Organization) are the primary organizations that develop and publish the standards.

The major Fortran language standards are:

- FORTRAN IV

  American National Standard Programming Language FORTRAN, ANSI X3.9-1966. This was the first attempt to standardize the languages called FORTRAN by many vendors.

- FORTRAN 77

  American National Standard Programming Language FORTRAN, ANSI X3.9-1978. This standard added new features based on vendor extensions to FORTRAN IV and addressed problems associated with large-scale projects, such as improved control structures.

- Fortran 90

  American National Standard Programming Language Fortran, ANSI X3.198-1992 and International Standards Organization, ISO/IEC 1539: 1991, Information technology -- Programming languages -- Fortran. This standard emphasizes modernization of the language by introducing new developments. For information about differences between Fortran 90 and FORTRAN 77, see the *Fortran Language Reference Manual*.

- Fortran 95

  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1: 1997(E), Information technology -- Programming languages -- Fortran. This standard introduces certain language elements and corrections into Fortran 90. Fortran 95 includes Fortran 90 and most features of FORTRAN 77. For information about differences between Fortran 95 and Fortran 90, see the *Fortran Language Reference Manual*.

- Fortran 2003
  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1:2004, Information technology -- Programming languages -- Fortran. This standard introduces extended support for exception handling, object-oriented programming, and improved interoperability with the C language. For more information on supported Fortran 2003 features, see the *Fortran Language Reference Manual*.

Although a language standard seeks to define the form and the interpretation uniquely, a standard may not cover all areas of interpretation. It may also include some ambiguities. You

need to carefully craft your program in these cases so that you get the answers that you want when producing a portable program.

Using Standard Features and Extensions

Use standard features to achieve the greatest degree of portability for your Intel Fortran programs. You can design a robust implementation to improve the portability of your program, or you can choose to use extensions to the standard to increase the readability, functionality, and efficiency of your programs. You can ensure your program enforces the Fortran standard by using the `-stand` (Linux* OS and Mac OS* X) or `/stand` (Windows* OS) compiler option with the appropriate keyword (`f90`, `f95`, or `f03`) to flag extensions. The `none` keyword turns off enforcement of a particular Fortran standard. You can also use the following compiler options to set the Fortran standard: `-std90` or `/std90`, `-std95` or `/std95`, and `-std03` or `/std03`. The default is std03, which diagnoses exceptions to the Fortran 2003 standard.

Not all Fortran standard extensions cause problems when porting to other platforms. Many extensions are supported on a wide range of platforms, and if a system you are porting a program to supports an extension, there is no reason to avoid using it. There is no guarantee, however, that the same feature on another system will be implemented in the same way as it is in Intel Fortran. Only the Fortran standard is guaranteed to coexist uniformly on all platforms.

Intel® Fortran supports many language extensions on multiple platforms, including Windows, Linux, and Mac OS X operating systems. The *Intel® Fortran Language Reference Manual* identifies whether each language element is supported on other platforms.

It is a good programming practice to declare any external procedures either in an EXTERNAL statement or in a procedure interface block, for the following reasons:

- The Fortran 90 standard added many new intrinsic procedures to the language. Programs that conformed to the FORTRAN 77 standard may include nonintrinsic functions or subroutines having the same name as new Fortran 90 procedures.
- Some processors include nonstandard intrinsic procedures that might conflict with procedure names in your program.

If you do not explicitly declare the external procedures and the name duplicates an intrinsic procedure, the processor calls the intrinsic procedure, not your external routine. For more information on how the Fortran compiler resolves name definitions, see Resolving Procedure References.

Using Compiler Optimizations

Many Fortran compilers perform code-generation optimizations to increase the speed of execution or to decrease the required amount of memory for the generated code. Although the behaviors of both the optimized and nonoptimized programs fall within the language standard specification, different behaviors can occur in areas not covered by the language standard. Compiler optimization especially can influence floating-point numeric results.

The Intel® Fortran compiler can perform optimizations to increase execution speed and to improve floating-point numerical consistency.

Floating-point consistency refers to obtaining results consistent with the IEEE binary floating-point standards. For more information, see the `-fltconsistency` (Linux OS and Mac OS X) or `/fltconsistency` (Windows OS) option.

Unless you properly design your code, you may encounter numerical difficulties when you optimize for fastest execution. The `-nofltconsistency` or `/nofltconsistency` option uses the floating-point registers, which have a higher precision than stored variables, whenever possible. This tends to produce results that are inconsistent with the precision of stored variables. The `-fltconsistency` or `/fltconsistency` option can improve the consistency of generated code by rounding results of statement evaluations to the precision of the standard data types, but it does produce slower execution times.

See also *Optimizing Applications*.

Minimizing Operating System-Specific Information

The operating system influences your program both externally and internally. For increased portability, you need to minimize the amount of operating-system-specific information required by your program. The Fortran language standards do not specify this information.

Operating-system-specific information consists of nonintrinsic extensions to the language, compiler and linker options, and possibly the graphical user interface of Windows. Input and output operations use devices that may be system-specific, and may involve a file system with system-specific record and file structures.

The operating system also governs resource management and error handling. You can depend on default resource management and error handling mechanisms or provide mechanisms of your own. For information on special library routines to help port your program from one system to another, see Portability Library Overview and related topics.

The minimal interaction with the operating system is for input/output (I/O) operations and usually consists of knowing the standard units preconnected for input and output. You can use default file units with the asterisk (*) unit specifier.

To increase the portability of your programs across operating systems, consider the following:

- Do not assume the use of a particular type of file system.
- Do not embed filenames or paths in the body of your program. Define them as constants at the beginning of the program or read them from input data.
- Do not assume a particular type of standard I/O device or the "size" of that device (number of rows and columns).
- Do not assume display attributes for the standard I/O device. Some environments do not support attributes such as color, underlined text, blinking text, highlighted text, inverse text, protected text, or dim text.

Storing and Representing Data

The Fortran language standard specifies little about the storage of data types.

This loose specification of storage for data types results from a great diversity of computing hardware. This diversity poses problems in representing data and especially in transporting stored data among a multitude of systems. The size (as measured by the number of bits) of a storage unit (a word, usually several bytes) varies from machine to machine. In addition, the ordering of bits within bytes and bytes within words varies from one machine to another. Furthermore, binary representations of negative integers and floating-point representations of real and complex numbers take several different forms.

If you are careful, you can avoid most of the problems involving data storage. The simplest and most reliable means of transferring data between dissimilar systems is in character and

not binary form. Simple programming practices ensure that your data as well as your program is portable.

See also Supported Native and Nonnative Numeric Formats.

Formatting Data for Transportability
You can achieve the highest transportability of your data by formatting it as 8-bit character data. Use a standard character set such as the ASCII standard for encoding your character data. Although this practice is less efficient than using binary data, it will save you from shuffling and converting your data.

If you are transporting your data by means of a record-structured medium, it is best to use the Fortran sequential formatted (as character data) form. You can also use the direct formatted form, but you need to know the record length of your data.

Remember also that some systems use a carriage return/linefeed pair as an end-of-record indicator, while other systems use linefeed only. If you use either the direct unformatted or the sequential unformatted form, there might be system-dependent values embedded within your data that complicate its transport.

Implementing a strictly portable solution requires a careful effort. Maximizing portability may also mean making compromises to the efficiency and functionality of your solution. If portability is not your highest priority, you can use some of the techniques that appear in later sections to ease your task of customizing a solution.

See Also

Supported Native and Nonnative Numeric Formats
Porting Nonnative Data
Methods of Specifying the Data Format

Troubleshooting

Troubleshooting Your Application
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.**
  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 supported 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` (Linux* OS and Mac OS* X) or `/Od` (Windows* OS) 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` (Linux OS and Mac OS X) or `/Od` (Windows OS), but fails with the default `-O2` (Linux OS and Mac OS X) or `/O2` (Windows OS), you need to determine which file or files are causing the problem.
- **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.

Reference Information

Key Compiler Files Summary

The following table lists the key files that are installed for use by the compiler.

\bin Files	
File	**Description**
`codecov`	Executable for the Code-coverage tool
`fortcom`	Executable used by the compiler
`fpp`	Fortran preprocessor
`ias` (Linux* OS and Mac OS* X)	Assembler for systems using IA-64 architecture
`idis` (Windows* OS)	Disassembler for systems using IA-64 architecture
`ifortvars`	File to set environment variables
`ifort.cfg`	Configuration file for use from command line
`ifort`	Intel® Fortran Compiler
`ifortbin` (Linux OS and Mac OS X)	Executable used by the compiler
`map_opts`	Utility used for option translation
`profmerge`	Utility used for Profile Guided Optimizations
`proforder`	Utility used for Profile Guided Optimizations
`tselect`	Test prioritization tool
`uninstall.sh` (Linux OS and Mac OS X)	Uninstall utility
`xiar` (Linux OS) `xilibtool` (Mac OS X) `xilib` (Windows OS)	Tool used for Interprocedural Optimizations
`xild` (Linux OS and Mac OS X) `xilink` (Windows OS)	Tool used for Interprocedural Optimizations

For a list of the files installed in the `lib` directory, see Supplied Libraries.

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 elements per dimension	9,223,372,036,854,775,807 = 2**31-1 on systems using IA-32 architecture; 2**63-1 on systems using Intel® 64 and IA-64 architectures; plus limited by current memory configuration
Constants: character and Hollerith	7198
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
Fortran 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 arguments to MIN and MAX	Limited only by memory constraints
Number of digits in a numeric constant	Limited by statement length
Parentheses nesting in expressions	Limited only by memory constraints
Structure nesting	30
Symbolic name length	63 characters
Width field for a numeric edit descriptor	2**31 - 1

See the product Release Notes for more information on memory limits for large data objects.