
Compiling, Linking, and Running Programs

This chapter contains the following major sections:

- “Compiling and Linking” describes the compilation environment and how to compile and link Fortran programs. This section also contains examples that show how to create separate linkable objects written in Fortran, C, or other languages supported by the compiler system and how to link them into an executable object program.
- “Driver Options” gives an overview of debugging, profiling, optimizing, and other options provided with the Fortran *f77* driver.
- “Object File Tools” briefly summarizes the capabilities of the *elfdump*, *dis*, *nm*, *file*, *size* and *strip* programs that provide listing and other information on object files.
- “Archiver” summarizes the functions of the *ar* program that maintains archive libraries.
- “Run-Time Considerations” describes how to invoke a Fortran program, how the operating system treats files, and how to handle run-time errors.

Also refer to the *Fortran Release Notes* for a list of compiler enhancements, possible compiler errors, and instructions on how to circumvent them.

Compiling and Linking

Drivers

Programs called *drivers* invoke the major components of the compiler system: the Fortran compiler, the optimizing code generator, and the linker. The *f77* command runs the driver that causes your programs to be compiled, optimized, assembled, and linked.

The format of the *f77* driver command is as follows:

```
f77 [option] ... filename [option]
```

For this format:

- f77* invokes the various processing phases that compile, optimize, assemble, and link the program.
- option* represents the driver options through which you provide instructions to the processing phases. They can be anywhere in the command line. These options are discussed later in this chapter.
- fi lename* is the name of the file that contains the Fortran source statements. The filename must always have the suffix **.f**, **.F**, **.for**, **.FOR**, or **.i**. For example, **myprog.f**.

Compilation

The driver command *f77* can both compile and link a source module. Figure 1-1 shows the primary driver's phases. It also shows their principal inputs and outputs for the source modules **more.f**.

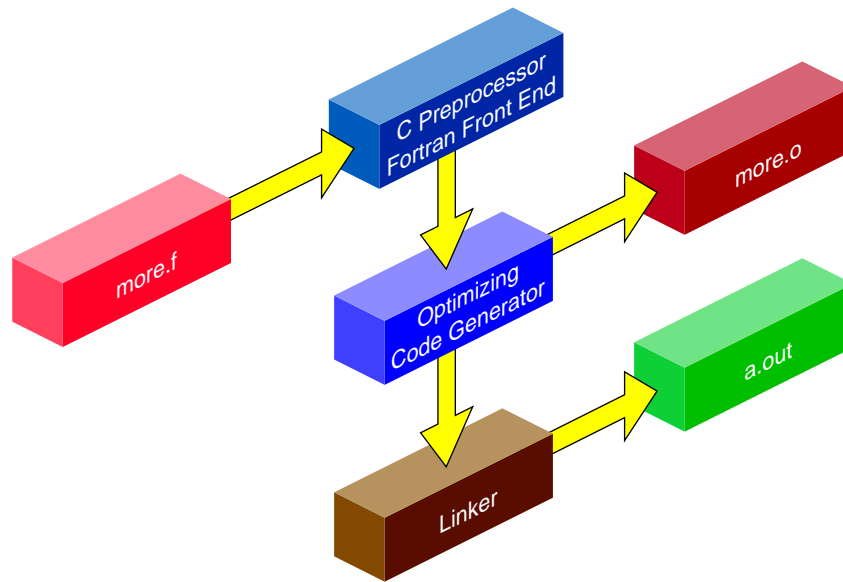


Figure 1-1 Compilation Process

Note the following:

- The source file ends with the required suffixes `.f`, `.F`, `.for`, `.FOR`, or `.i`.
- The Fortran front end has an integrated C preprocessor that provides full *cpp* capabilities.
- The driver produces a linkable object file when you specify the `-c` driver option. This file has the same name as the source file, except with the suffix `.o`. For example, the command line


```
% f77 more.f -c
```

 produces the `more.o` file in the above example.
- The default name of the executable object file is `a.out`. For example, the command line


```
% f77 myprog.f
```

 produces the executable object `a.out`.
- You can specify a name other than `a.out` for the executable object by using the driver option `-o name`, where *name* is the name of the executable object. For example, the command line


```
% f77 myprog.o -o myprog
```

 links the object module `myprog.o` and produces an executable object named `myprog`.
- The command line


```
% f77 myprog.f -o myprog
```

 compiles and links the source module `myprog.f` and produces an executable object named `myprog`.

Compiling Multilanguage Programs

The compiler system provides drivers for other languages, including C and C++. If one of these drivers is installed in your system, you can compile and link your Fortran programs to the language supported by the driver. (See the *MIPSpro Compiling and Performance Tuning Guide* for a list of available drivers and the commands that invoke them. Refer to Chapter 3, “Fortran Program Interfaces,” in this manual for conventions you must follow when writing Fortran program interfaces to C programs.)

When your application has two or more source programs written in different languages, you should compile each program module separately with the appropriate driver and then link them in a separate step. Create objects suitable for linking by specifying the `-c` option, which stops the driver immediately after the assembler phase. For example,

```
% cc -c main.c
% f77 -c rest.f
```

The two command lines shown above produce linkable objects named `main.o` and `rest.o`, as illustrated in Figure 1-2.

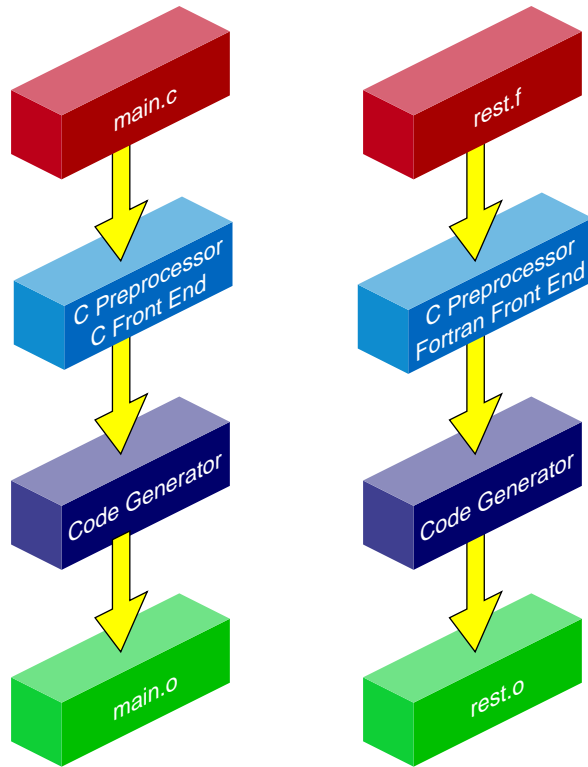


Figure 1-2 Compiling Multilanguage Programs

Linking Objects

You can use the *f77* driver command to link separate objects into one executable program when any one of the objects is compiled from a Fortran source. The driver recognizes the `.o` suffix as the name of a file containing object code suitable for linking and immediately invokes the linker. The following command links the object created in the last example:

```
% f77 -o myprog main.o rest.o
```

You can also use the *cc* driver command, as shown below:

```
% cc -o myprog main.o rest.o -lftn -lm
```

Figure 1-3 shows the flow of control for this link.

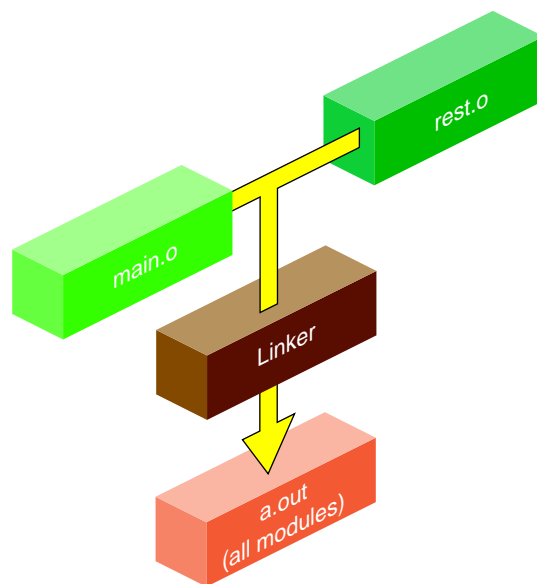


Figure 1-3 Linking

Both *f77* and *cc* use the C link library by default. However, the *cc* driver command does not know the names of the link libraries required by the Fortran objects; therefore, you must specify them explicitly to the linker using the **-l** option as shown in the example. The characters following **-l** are shorthand for link library files, as shown in Table 1-1.

Table 1-1 Link Libraries

-l	Link Library	Contents
ftn	/usr/lib*/nonshared/libftn.a	Intrinsic function, I/O, multiprocessing, IRIX interface, and indexed sequential access method library for nonshared linking and compiling
ftn	/usr/lib*/libftn.so	Same as above, except for shared linking and compiling (this is the default library)
m	/usr/lib*/libm.so	Mathematics library

See the section called “FILES” in the *f77(1)* reference page for a complete list of the files used by the Fortran driver. Also refer to the *ld(1)* reference page for information on specifying the **-l** option.

Specifying Link Libraries

You may need to specify libraries when you use IRIX system packages that are not part of a particular language. Most of the reference pages for these packages list the required libraries. For example, the *getwd(3B)* subroutine requires the BSD compatibility library *libbsd.a*. Specify this library as follows:

```
% f77 main.o more.o rest.o -lbsd
```

To specify a library created with the archiver, type in the pathname of the library as shown below.

```
% f77 main.o more.o rest.o libfft.a
```

Note: The linker searches libraries in the order you specify. Therefore, if you have a library (for example, *libfft.a*) that uses data or procedures from **-lm**, you *must* specify *libfft.a* first.

Driver Options

This section contains an overview of the Fortran-specific driver options. The *f77(1)* reference page has a complete description of the compiler options. This discussion only covers the relationships between some of the options, so as to help you make sense of the many options in the reference page. For more information you can review:

- The *MIPSpro Compiling and Performance Tuning Guide* for a discussion of the compiler options that are common to all MIPSpro compilers.
- The *pfa(1)* reference page for options related to the parallel optimizer.
- The *ld(1)* reference page for a description of the linker options.

Tip: The command *f77 -help* lists all compiler options for quick reference. Use the *-show* option to have the compiler document each phase of execution, showing the exact default and nondefault options passed to each.

Compiling Simple Programs

You need only a very few compiler options when you are compiling a simple program. Examples of simple programs include the following:

- Test cases used to explore algorithms or Fortran language features
- Programs that are principally interactive
- Programs whose performance is limited by disk I/O
- Programs you will execute under a debugger

In these cases you need only specify *-g* for debugging, the target machine architecture, and the word-length. For example, to compile a single source file to execute under *dbx* on a POWER CHALLENGE™ XL, you could use the following commands.

```
f77 -g -mips4 -n32 -o testcase testcase.f
dbx testcase
```

However, a program compiled in this way will take little advantage of the performance features of the machine. In particular, its speed when doing heavy floating-point calculations will be far slower than the machine is capable of. For simple programs, that is not important.

Specifying Source File Format

The options shown in Table 1-2 tell the compiler how to treat the program source file.

Table 1-2 Compile Options for Source File Format

Options	Purpose
<i>-ansi</i>	Report any nonstandard usages.
<i>-backslash</i>	Treat \ in character literals as a character, not as the start of an escape sequence.
<i>-col72, -col120, -extend_source, -noextend_source</i>	Specify margin columns of source lines.
<i>-d_lines</i>	Compile lines with D in column 1.
<i>-Dname, -Dname=def, -Uname</i>	Define, undefine names to the integrated C preprocessor.

Specifying Compiler Input and Output Files

The options summarized in Table 1-3 tell the compiler what output files to generate.

Table 1-3 Compile Options that Select Files

Options	Purpose
<i>-c</i>	Generate a single object file for each input file; do not link.
<i>-E</i>	Run only the macro preprocessor and write its output to standard output.
<i>-I, -Idir, -nostdinc</i>	Specify location of include files.
<i>-listing</i>	Request a listing file.
<i>-MDupdate</i>	Request Makefile dependency output data.
<i>-o</i>	Specify name of output file.
<i>-S</i>	Specify only assembly-language source output.

Using a Defaults Specification File

You can set the Application Binary Interface (ABI), instruction set architecture (ISA), and processor type without explicitly specifying them. Just set the environment variable `COMPILER_DEFAULTS_PATH` to a colon separated list of paths designating where the compiler is to look for a `compiler.defaults` file. If `nocompiler.defaults` file is found or if the environment variable is not set, the compiler looks for `/etc/compiler.defaults`. If this file is not found, the compiler resorts to the built-in defaults.

The `compiler.defaults` file contains a **DEFAULT: option** group specifier that specifies the default ABI, ISA, and processor. The compiler issues a warning if you specify anything other than **DEFAULT: option** in the `compiler.defaults` file.

The format of the **DEFAULT: option** group specifier is as follows:

```
-DEFAULT:[abi={32|n32|64}]:[isa=mipsn]:[proc={r4k|r5k|r8k|r10k}]
```

See the `f77(1)` reference page for an explanation of the `option` group.

Specifying Target Machine Features

The options summarized in Table 1-4 are used to specify the characteristics of the machine where the compiled program will be used.

Table 1-4 Compile Options for Target Machine Features

Options	Purpose
<code>-64, -n32, -32</code>	Whether target machine runs 64-bit mode, “new” 32-bit mode, or 32-bit mode. The <code>-64</code> option is allowed only with the <code>-mips3</code> and <code>-mips4</code> architecture options.
<code>-mips3, -mips4</code>	The instruction architecture available in the target machine: use <code>-mips3</code> for MIPS R4000 [®] and R4400 [®] machines; use <code>-mips4</code> for MIPS R8000 [®] and R10000 [™] machines.
<code>-TARGET:option,...</code>	Specify certain details of the target CPU. Most of these options have correct default values based on the preceding options.
<code>-TENV:option,...</code>	Specify certain details of the software environment in which the source module will execute. Most of these options have correct default values based on other, more general values.

Specifying Memory Allocation and Alignment

The options summarized in Table 1-5 tell the compiler how to allocate memory and how to align variables in it. These options can have a strong effect on both program size and program speed.

Table 1-5 Compile Options for Memory Allocation and Alignment

Options	Purpose
<i>-align8, -align16, -align32, -align64</i>	Align all variables size <i>n</i> on <i>n</i> -byte address boundaries.
<i>-d8, -d16</i>	Specify the size of DOUBLE and DOUBLE COMPLEX variables.
<i>-i2, -i4, -i8</i>	Specify the size of INTEGER and LOGICAL variables.
<i>-r4, -r8</i>	Specify the size of REAL and COMPLEX variables.
<i>-static</i>	Allocate all local variables statically, not dynamically on the stack.
<i>-Gsize, -xgot</i>	Specify use of the global option table.

Specifying Debugging and Profiling

The options summarized in Table 1-6 direct the compiler to include more or less extra information in the object file for debugging or profiling.

Table 1-6 Compile Options for Debugging and Profiling

Options	Purpose
<i>-g0, -g2, -g3, -g</i>	Leave more or less symbol-table information in the object file for use with <i>dbx</i> or Workshop Pro <i>cvd</i> .
<i>-p</i>	Cause profiling to be enabled when the program is loaded.

For more information on debugging and profiling, see the manuals listed in the preface.

Specifying Optimization Levels

The MIPSpro Fortran 77 compiler contains three optimizer phases. One is part of the compiler “back end”; that is, it operates on the generated code, after all syntax analysis and source transformations are complete. The use of this standard optimizer, which is common to all MIPSpro compilers, is discussed in the *MIPSpro Compiling and Performance Tuning Guide*.

In addition, MIPSpro Fortran 77 contains two phases of accelerators, one for scalar optimization and one for parallel array optimization. These operate during the initial phases of the compilation, transforming the source statements before they are compiled to machine language. The options of the scalar optimizer are detailed in the `fopt(1)` reference page. The options of the parallel optimizer are detailed in the `pfa(1)` reference page.

Note: The reason these optimizer phases are documented in separate reference pages is that, when compiling for 32-bit machines, these phases use a separate product, the Power Fortran Accelerator, which has been integrated into the MIPSpro Fortran 77 compiler.

The options summarized in Table 1-7 are used to communicate to the different optimization phases.

Table 1-7 Compile Options for Optimization Control

Options	Purpose
<code>-O, -O0, -O1, -O2, -O3</code>	Select basic level of optimization, setting defaults for all optimization phases.
<code>-GCM:option,...</code>	Specify details of global code motion performed by the back-end optimizer.
<code>-OPT:option,...</code>	Specify miscellaneous details of optimization.
<code>-SWP:option,...</code>	Specify details of pipelining done by back-end optimizer.
<code>-pfa</code>	Request execution of the parallel source-to-source optimizer.
<code>-WK:option,...</code>	Pass options to parallel source-to-source optimizer of MIPSpro Power Fortran 77.

When you use `-O` to specify the optimization level, the compiler assumes default options for the accelerator phases. These defaults are listed in Table 1-8. To see all options that are passed to a compiler phase, use the `-show` option.

Table 1-8 Power Fortran Defaults for Optimization Levels

Optimization Level	Power Fortran Defaults Passed
<code>-O0</code>	<code>-WK,-roundoff=0,-scaleropt=0,-optimize=0</code>
<code>-O1</code>	<code>-WK,-roundoff=0,-scaleropt=0,-optimize=0</code>
<code>-O2</code>	<code>-WK,-roundoff=0,-scaleropt=0,-optimize=0</code>
<code>-O3</code>	<code>-WK,-roundoff=2,-scaleropt=3,-optimize=5</code>

In addition to optimizing options, the compiler system provides other options that can improve the performance of your programs:

- Two linker options, `-G` and `-bestG`, control the size of the global data area, which can produce significant performance improvements. See Chapter 2 of the *MIPSpro Compiling and Performance Tuning Guide* and the `ld(1)` reference page for more information.
- The `-jmpopt` option permits the linker to fill certain instruction delay slots not filled by the compiler front end. This option can improve the performance of smaller programs not requiring extremely large blocks of virtual memory. See the `ld(1)` reference page for more information.

Controlling Compiler Execution

The options summarized in Table 1-9 control the execution of the compiler phases.

Table 1-9 Compile Options for Compiler Phase Control

Options	Purpose
<code>-E, -P</code>	Execute only the integrated C preprocessor.
<code>-fe</code>	Stop compilation immediately after the front-end (syntax analysis) runs.
<code>-M</code>	Run only the macro preprocessor.

Table 1-9 (continued) Compile Options for Compiler Phase Control

Options	Purpose
<i>-Yc,path</i>	Load the compiler phase specified by <i>c</i> from the specified <i>path</i> .
<i>-Wc,option,...</i>	Pass the specified list of options to the compiler phase specified by <i>c</i> .

Object File Tools

The following tools provide information on object files as indicated:

<i>elfdump</i>	Lists headers, tables, and other selected parts of an ELF-format object or archive file.
<i>dis</i>	Disassembles object files into machine instructions.
<i>nm</i>	Prints symbol table information for object and archive files.
<i>file</i>	Lists the properties of program source, text, object, and other files. This tool often erroneously recognizes command files as C programs. It does not recognize Pascal or LISP programs.
<i>size</i>	Prints information about the text, rdata, data, sdata, bss, and sbss sections of the specified object or archive file. See the a.out(4) reference page for a description of the contents and format of section data.
<i>strip</i>	Removes symbol table and relocation bits.

For more information on these tools, see the *MIPSpro Compiling and Performance Tuning Guide* and the [dis\(1\)](#), [elfdump\(1\)](#), [file\(1\)](#), [nm\(1\)](#), [size\(1\)](#), and [strip\(1\)](#) reference pages.

Archiver

An archive library is a file that contains one or more routines in object (**.o**) file format. The term *object* as used in this chapter refers to an **.o** file that is part of an archive library file. When a program calls an object not explicitly included in the program, the link editor *ld* looks for that object in an archive library. The link editor then loads only that object (not the whole library) and links it with the calling program.

The archiver (*ar*) creates and maintains archive libraries and has the following main functions:

- copying new objects into the library
- replacing existing objects in the library
- moving objects about the library
- copying individual objects from the library into individual object files

See the *MIPSpro Compiling and Performance Tuning Guide* and the *ar(1)* reference page for additional information on the archiver.

Run-Time Considerations

Invoking a Program

To run a Fortran program, invoke the executable object module produced by the *f77* command by entering the name of the module as a command. By default, the name of the executable module is **a.out**. If you included the `-o filename` option on the *ld* (or *f77*) command line, the executable object module has the name that you specified.

Maximum Memory Allocations

The total memory allocation for a program, and individual arrays, can exceed 2 gigabytes (2 GB, or 2,048 MB).

Previous implementations of Fortran 77 limited the total program size, as well as the size of any single array, to 2 GB. The current release allows the total memory in use by the program to far exceed this. (For details on the memory use of individual scalar values, see “Alignment, Size, and Value Ranges” on page 21.)

Arrays Larger Than 2 Gigabytes

The MIPSPro 7.1 compilers support arrays that are larger than 2 gigabytes for programs compiled under the *-64* ABI. The arrays can be local, global, and dynamically created as the following example demonstrates. (Note: Initializers are not provided for these arrays.) Large array support is limited to Fortran77, C, and C++.

For example:

```
$cat a2.c

#include <stdlib.h>

int i[0x100000008];

void foo()
{
int k[0x100000008];
k[0x100000007] = 9;
printf("%d \n", k[0x100000007]);
}

main()
{
char *j;
j = malloc(0x100000008);
i[0x100000007] = 7;
j[0x100000007] = 8;
printf("%d \n", i[0x100000007]);
printf("%d \n", j[0x100000007]);
foo();
}
```

You must run this program on a 64-bit operating system with IRIX version 6.2 (or higher). You can verify the system you have by typing **uname -a**. You must have enough swap space to support the working set size and you must have your shell limit `datasize`, `stacksize`, and `vmemoryuse` variables set to values large enough to support the sizes of the arrays (see `sh(1)` reference page).

The following example compiles and runs the above code after setting the `stacksize` to a correct value:

```
$uname -a
IRIX64 cydrome 6.2 03131016 IP19
$cc -64 -mips3 a2.c
$limit
cputime          unlimited
filesize         unlimited
datasize         unlimited
stacksize        65536 kbytesn
coredumpsize     unlimited
```

```
memoryuse
descriptors      200
vmemoryuse      unlimited
$limit stacksize unlimited
$limit
cputime          unlimited
filesize         unlimited
datasize        unlimited
stacksize       unlimited
coredumpsize    unlimited
memoryuse       754544 kbytes
descriptors     200
vmemoryuse      unlimited
$a.out
7
8
9
```

Local Variable (Stack Frame) Sizes

Arrays that are allocated on the process stack must not exceed 2 GB, but the total of all stack variables can exceed that limit. For example,

```
parameter (ndim = 16380)
integer*8 xmat(ndim,ndim), ymat(ndim,ndim), &
          zmat(ndim,ndim)
integer k(1073741824)
integer l(33554432, 256)
```

However, when an array is passed as an argument, it is not limited in size.

```
subroutine abc(k)
integer k(8589934592_8)
```

Static and Common Sizes

When compiling with the `-static` flag, global data is allocated as part of the compiled object (`.o`) file. The total size of any `.o` file may not exceed 2 GB. However, the total size of a program linked from multiple `.o` files may exceed 2 GB.

An individual common block may not exceed 2 GB. However, you can declare multiple common blocks each having that size.

Pointer-based Memory

There is no limit on the size of a pointer-based array. For example,

```
integer *8 ndim
parameter (ndim = 20001)
pointer (xptr, xmat), (yptr, ymat), (zptr, zmat), &
      (aptr, amat)
xptra = malloc(ndim*ndim*8)
yptra = malloc(ndim*ndim*8)
zptr = malloc(ndim*ndim*8)
aptra = malloc(ndim*ndim*8)
```

It is important to make sure that malloc is called with an INTEGER*8 value. A count greater than 2 GB would be truncated if assigned to an INTEGER*4.

File Formats

Fortran supports five kinds of external files:

- sequential formatted
- sequential unformatted
- direct formatted
- direct unformatted
- key indexed file

The operating system implements other files as ordinary files and makes no assumptions about their internal structure.

Fortran I/O is based on records. When a program opens a direct file or key indexed file, the length of the records must be given. The Fortran I/O system uses the length to make the file appear to be made up of records of the given length. When the record length of a direct file is 1 byte, the system treats the file as ordinary system files (as byte strings, in which each byte is addressable). A **READ** or **WRITE** request on such files consumes bytes until satisfied, rather than restricting itself to a single record.

Because of special requirements, sequential unformatted files will probably be read or written only by Fortran I/O statements. Each record is preceded and followed by an integer containing the length of the record in bytes.

During a **READ**, Fortran I/O breaks sequential formatted files into records by using each new line indicator as a record separator. The Fortran 77 standard does not define the required result after reading past the end of a record; the I/O system treats the record as being extended by blanks. On output, the I/O system writes a new line indicator at the end of each record. If a user program also writes a new line indicator, the I/O system treats it as a separate record.

Preconnected Files

Table 1-10 shows the standard preconnected files at program start.

Table 1-10 Preconnected Files

Unit #	Unit	Alternate Unit #
5	Standard input	* (in READ)
6	Standard output	* (in WRITE)
0	Standard error	** (in WRITE)

All other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.n**. These files need not exist, nor will they be created unless their units are used without first executing an open. The default connection is for sequentially formatted I/O.

File Positions

The Fortran 77 standard does not specify where **OPEN** should initially position a file explicitly opened for sequential I/O. The I/O system positions the file to start of file for both input and output. The execution of an **OPEN** statement followed by a **WRITE** on an existing file causes the file to be overwritten, erasing any data in the file. In a program called from a parent process, units 0, 5, and 6 remain where they were positioned by the parent process.

Unknown File Status

When the parameter `STATUS="UNKNOWN"` is specified in an `OPEN` statement, the following occurs:

- If the file does not exist, it is created and positioned at start of file.
- If the file exists, it is opened and positioned at the beginning of the file.

Quad-Precision Operations

When running programs that contain quad-precision operations, you must run the compiler in round-to-nearest mode. Because this mode is the default, you usually do not need to be concerned with setting it. You usually need to set this mode when writing programs that call your own assembly routines. Refer to the `swapRM` reference page for details.

Run-Time Error Handling

When the Fortran run-time system detects an error, the following action takes place

- A message describing the error is written to the standard error unit (unit 0). See Appendix A, "Run-Time Error Messages," for a list of the error messages.
- A core file is produced if the `f77_dump_flag` environment variable is set, as described in Appendix A, "Run-Time Error Messages." You can use `dbx` to inspect this file and determine the state of the program at termination. For more information, see the *dbx User's Guide*.

To invoke `dbx` using the core file, enter the following:

```
% dbx binary-file core
```

where *binary-file* is the name of the object file output (the default is `isa.out`). For more information on `dbx`, see the *dbx User's Guide*

Floating Point Exceptions

The library *libfpe* provides two methods for handling floating point exceptions.

The library provides the subroutine **handle_sigfpe** and the environment variable **TRAP_FPE**. Both methods provide mechanisms for handling and classifying floating point exceptions, and for substituting new values. They also provide mechanisms to count, trace, exit, or abort on enabled exceptions. The **-TENV:check_div** compile flag inserts checks for divide by zero or overflow. See the `handle_sigfpe(3F)` reference page for more information.