Chapter 1

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, Pascal, 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 *odump*, *stdump*, *nm*, *file*, and *size* 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 intermediate code optimizer, the code generator, the assembler, and the link editor. The *f77* command runs the driver that causes your programs to be compiled, optimized, assembled, and link edited.

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

`f77 [`*option*`] …` *filename*`.f [`*option*`]`

where

f77 — invokes the various processing phases that compile, optimize, assemble, and link edit 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.

filename.**f** — is the name of the file that contains the Fortran source statements. The filename must always have the suffix **.f**, for example, **myprog.f**.

Compilation

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

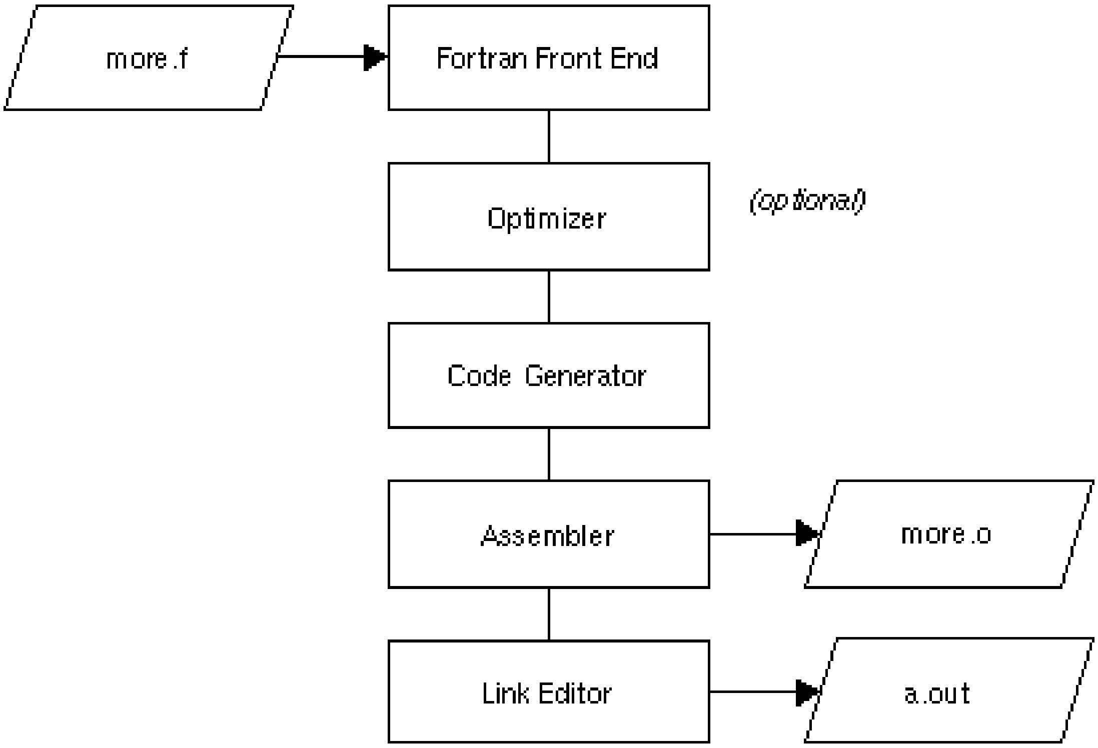

Figure 1–1 Compilation Process

Note the following:

- The source file ends with the required suffixes **.f** or **.F**.
- The source file is passed through the C preprocessor, *cpp*, by default. *cpp* does not accept C–style comments in Hollerith strings. The **-nocpp** option skips the pass through *cpp* and therefore, allows C–style comments in Hollerith strings. (See the**-nocpp** option in "Driver Options" for details.) In the example

  ```
  % f77 myprog.f –nocpp
  ```

  the file **myprog.f** will not be preprocessed by *cpp*.
- 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
  ```

  link edits the object module *myprog.o* and produces an executable object named **myprog**.

- The command line

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

  compiles and link edits 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, Pascal, COBOL, and PL/1. 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 *IRIX Series Compiler Guide* for a list of available drivers and the commands that invoke them; refer to Chapter 3 of this manual for conventions you must follow in writing Fortran program interfaces to C and Pascal 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 link editing 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

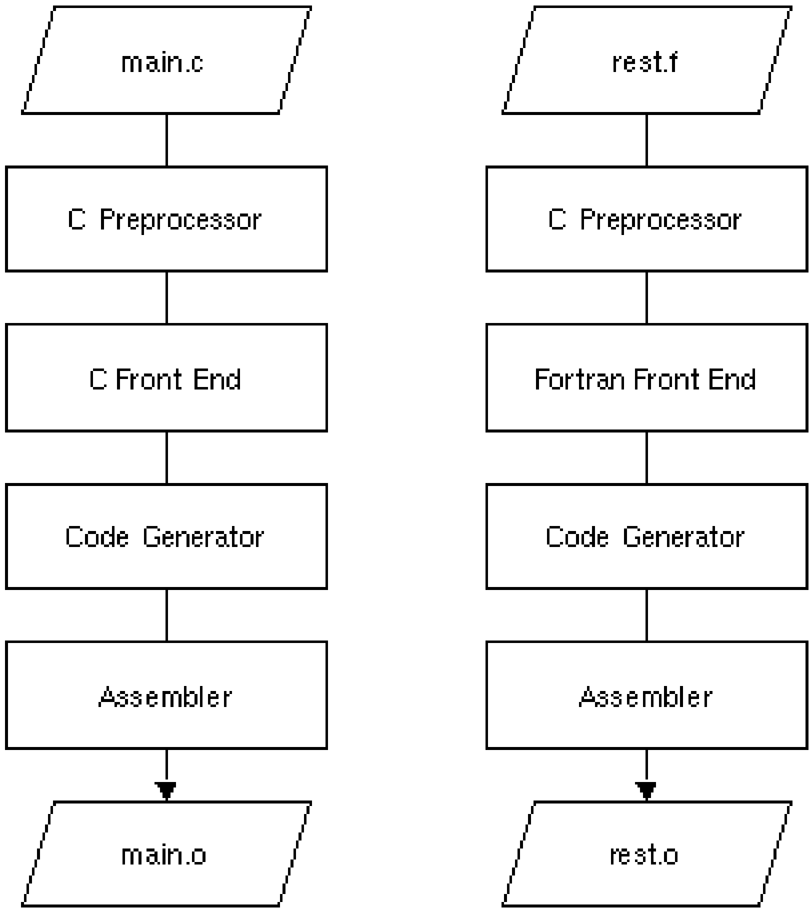

Figure 1–2 Compiling Multilanguage Programs

Linking Objects

You can use the *f77* driver command to link edit 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 link editing and immediately invokes the link editor. The following command link edits 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 -lF77 -lU77 -lI77 -lisam -lm
```

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

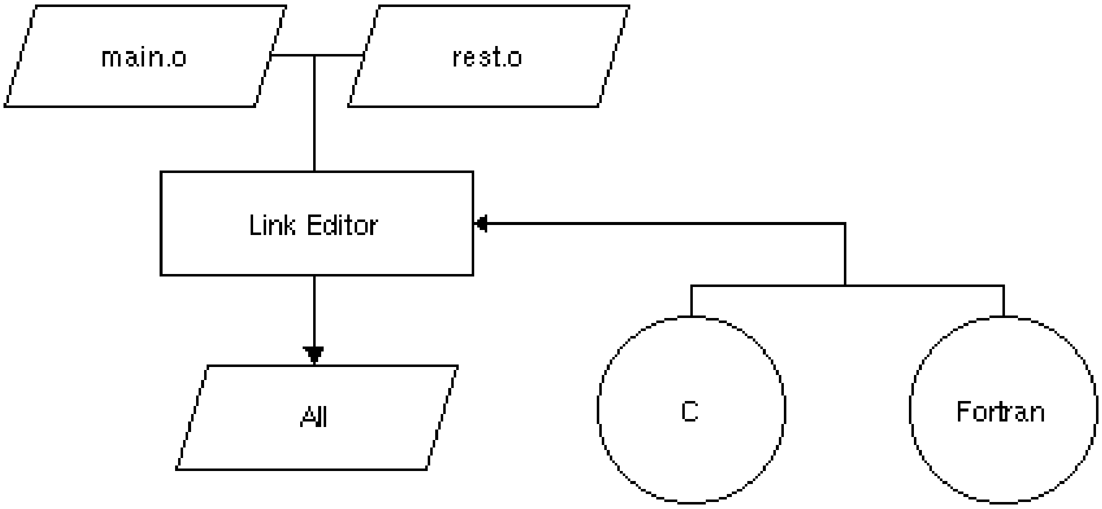

Figure 1–3 Link Editing

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 link editor 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
F77	/usr/lib/libF77.a	Fortran intrinsic function library
I77	/usr/lib/libI77.a	Fortran I/O library
I77_mp	/usr/lib/libI77_mp.a	Fortran multiprocessing I/O library
U77	/usr/lib/libU77.a	Fortran IRIX interface library
isam	/usr/lib/libisam.a	Indexed sequential access method library
fgl	/usr/lib/libfgl/a	Fortran graphics library
m	/usr/lib/libm.a	Mathematics library

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

Specifying Link Libraries

You must explicitly load any required run–time libraries when compiling multilanguage programs. For example, when you link a program written in Fortran and some procedures written in Pascal, you must explicitly load the Pascal library *libp.a* and the math library *libm.a* with the options **-lp** and **-lm** (abbreviations for the libraries *libp.a* and *libm.a*). This procedure is demonstrated in the next example.

```
% f77 main.o more.o rest.o –lp –lm
```

To find the Pascal library, the link editor replaces the **-l** with *lib* and adds an **.a** after **p**. Then, it searches the */lib*, */usr/lib*, and */usr/local/lib* directories for this library. For a list of the libraries that a language uses, see the associated driver manual page, *cc*(1), *f77*(1), or *pc*(1).

You may need to specify libraries when you use IRIX system packages that are not part of a particular language. Most of the manual pages for these packages list the required libraries. For example, the *getwd* (3B) subroutine requires the BSD compatibility library *libbsd.a*. This library is specified 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 link editor searches libraries in the order you specify. Therefore, if you have a library (for example, *libfft.a*) that uses data or procedures from **-lp**, you *must* specify *libfft.a* first.

Driver Options

This section contains a summary of the Fortran-specific driver options. See the *f77*(1) manual page for a complete description of the compiler options; see the *ld*(1) manual page for a description of the link editor options.

-66 Compiles Fortran 66 source programs.

When used at compile time, the following four options generate various degrees of misaligned data in common blocks. They then generate the code to deal with the misalignment.

Note: When specified, these options can degrade program performance;
-align8 causes the greatest degree of degradation, and **-align32** causes the least.

-**align8** Aligns objects larger than 8 bits on 8−bit boundaries. Using this option will have the largest impact on performance.

-**align16** Aligns objects larger than 16 bits on 16−bit boundaries; 16−bit objects must still be aligned on 16−bit boundaries (MC68000−like alignment rules).

-**align32** Aligns objects larger than 32 bits on 32−bit boundaries; 16−bit objects must still be aligned on 16−bit boundaries, and 32−bit objects must still be aligned on 32−bit boundaries.

-**align64** Aligns objects larger than 64 bits on 64 bit boundaries. Objects with size 64 bits or smaller must still be aligned on the corresponding boundaries.

The current default alignment is 32 bits. This number may be changed in the future to take advantage of the new 64−bit architecture.

You must specify the appropriate alignment option in the compilation of all modules that reference or define common blocks with misaligned data. Failure to do so could cause core dumps (if the trap handler is not used) or mismatched common blocks.

To load the system libraries capable of handling misaligned data, use the **-L** */usr/lib/align* switch at load time. The trap handler may be needed to handle misaligned data passed to system libraries that are not included in the */usr/lib/align* directory (see *fixade*(3f) and *unalign*(3x)).

-backslash Allows the backslash character to be used as a normal Fortran character instead of the beginning of an escape sequence.

-C Generates code for run−time subscript range checking. The default suppresses range checking. This option will not cause the program to core dump; it will cause the program to exit with a diagnostic message. For details on how to produce a core dump, refer to the information on the **f77_dump_flag** environment variable in Appendix A, "Run−Time Error Messages."

-check_bounds

Causes an error message to be issued at run time when the value of an array subscript expression exceeds the bounds declared for the array. This is equivalent to the **-C** option.

-chunk=*integer*

Has the same effect as putting a **C$CHUNK=***integer* directive at the beginning of the file. See Chapter 5, "Fortran Enhancements for Multiprocessors," and Chapter 6, "Compiling and Debugging Parallel Fortran," for details.

-col72 Sets the source statement format as described in Table 1−2

Table 1−2 Source Statement Settings for −col72 Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

If the source statement contains fewer than 72 characters, no blank padding occurs; the TAB−format facility is disabled.

This option provides the SVS Fortran 72−column option mode.

-col120 Sets the source statement format as described in Table 1−3

Table 1−3 Source Statement Settings for −col120 Option

Column	Contents
1−5	Statement label
6	Continuation indicator
7−120	Statement body
121−end	Ignored

If the source statement contains fewer than 120 characters, no blank padding occurs; the TAB−format facility is disabled.

This option provides the SVS Fortran default mode.

-cpp Runs the C macro preprocessor *cpp* on all source files, including those created by

RATFOR, before compilation. (This option is enabled by default.)

-d_lines Causes any lines with a D in column 1 to be compiled. By default, the compiler treats all lines with a character in column 1 as comment lines.

-expand_include

Expands all include statements in the Fortran source listing file **.L**. This option is only applicable with the **-listing** option.

-extend_source

Sets the source statement format as described in Table 1−4

Table 1−4 Source Statement Settings for −extend_source Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-132	Statement body
133-end	Warning message issued

If the source statement contains fewer than 132 characters, blanks are assumed at the end; the ability of TAB−formatted lines to extend past column 132 is disabled.

This option provides VMS Fortran 132−column mode, except that a warning, instead of a fatal error message, is generated when text extends beyond column 132.

-E Runs only the C macro preprocessor on the files and sends the results to standard output.

-F Calls the RATFOR preprocessor only and puts the output in a **.f** file. Does not produce **.o** files.

-framepointer

Defines the frame pointer register for each subroutine in the source file.

-i2 All small integer constants become **INTEGER*2**. All variables and functions implicitly or explicitly declared type **INTEGER** or **LOGICAL** (without a size designator, that is, **2*, **4*, and so on) will be **INTEGER *2** or **LOGICAL*2**, respectively.

-listing Produces the source listing file with **.L** suffix containing line numbers, error messages, symbol table information, and cross references.

-m If the generic function results do not determine the precision of an integer−valued intrinsic function, the compiler chooses the precisions that return **INTEGER *2**. The default is **INTEGER *4**. Note that **INTEGER *2** and **LOGICAL *2** quantities do not obey the Fortran standard rules for storage location.

−m4 Applies the M4 macro preprocessor to source files to be transformed with RATFOR. The driver puts the result in a **.p** file. Unless you specify the **-K** option, the compiler removes the **.p** file on completion. See the *m4*(1) manual page for details.

-mp Enable the multiprocessing directives. See Chapter 5, "Fortran Enhancements for Multiprocessors," and Chapter 6, "Compiling and Debugging Parallel Fortran," of this book, and the man page on *f77*(1) for further options affecting multiprocessing compilation.

-mp_schedtype=*type*

Has the same effect as putting a **C$MP_SCHEDTYPE=** *type* directive at the beginning of the file. The supported types are **simple**, **interleave**, **dynamic**, **gss**, and **runtime**. See Chapter 5, "Fortran Enhancements for Multiprocessors," and Chapter 6, "Compiling and Debugging Parallel Fortran," of this manual for more details.

-N[qxscnlC]*nnn*

nnn is a decimal number changing the default size of the static tables in the compiler. See the *f77*(1) manual page for details.

-nocpp Does not run the C preprocessor on the source files. Specifying this option allows you to specify C−style comments inside Hollerith strings. Use this option when you want your program to strictly conform to the Fortran 77 standard.

-noexpopt Excludes floating point constant exponent optimization to achieve the same precision as releases prior to 4D1−4.0.

-noextend_source

Sets the source statement format as described in Table 1−5

Table 1−5 Source Statement Settings for −noextend_source Option

Column	Contents
1-5	Statement label
6	Continuation indicator
7-72	Statement body
73-end	Ignored

If the source statement contains fewer than 72 characters, blanks are assumed at the end; the ability of TAB−formatted lines to extend past Column 72 is disabled.

This option provides VMS Fortran default mode.

-noi4 Same as **-i2** option.

-nof77 Same as **-onetrip** switch except for the following:

- The syntax and behavior of **EXTERNAL** statements are altered.
- The default value for the **BLANK=***clause* in an **OPEN** statement is **ZERO**.
- The default value for the **STATUS=***clause* in an **OPEN** statement is **NEW**.

-noisam Excludes the indexed sequential access library *libisam.a* from being linked to the executable to reduce the size.

-old_rl Interprets the record length specifier for a direct unformatted file as a number of bytes instead of a number of words. This option provides backward compatibility with

4D1−3.1 releases and earlier.

-onetrip Same as **-1** option.

-1 Compiles **DO** loops so that they execute at least once if reached. By default, **DO** loops are not executed if the upper limit is smaller than the lower limit. Similar to the **-nof77** option.

-P Runs only the C macro preprocessor and puts the result of each source file into a corresponding **.i** file. The **.i** file cannot contain # lines.

-pfa Run the *pfa* preprocessor to automatically discover, parallelism in the source code. This also enables the multiprocessing directives. There are two optional arguments:

- **-pfa list** runs *pfa* and produces a listing file with suffix **.l** explaining which loops were parallelized, and if not, why not.
- **-pfa keep** runs *pfa*, produces the listing file, and also keeps the transformed, multiprocessed Fortran intermediate file in a file with suffix **.a**.

-R*flags* *flags* is a valid option for RATFOR; the flags are given in the *ratfor*(1) manual page.

The RATFOR input filename is *filename***.r**. The resulting output is placed in *filename* **.f**. You must specify the **-K** option to retain the output file.

-r8 Uses **REAL*8** and **COMPLEX*16** as the defaults for real and complex variables that are not explicitly declared with a type size.

-static Local variables are saved in a static location, initialized to zeros, and retain values between calls. This option overrides the default **-automatic** option.

-trapeuv Sets unitialized local variables to **0xFFFA5A5A**. This value is treated as a floating point NaN and causes a floating point trap.

-U Causes the compiler to differentiate upper− and lowercase alphabetic characters. For example, the compiler considers a and A as distinct characters. Note that this option causes the compiler to recognize lowercase keywords only. Therefore, lowercase keywords must be used in writing case−sensitive programs (or in writing generic header files).

-u Turns off Fortran default data typing and any data typing explicitly specified in an **IMPLICIT** statement. Forces the explicit declaration of all data types.

-usefpidx Uses the floating point **DO** loop variable as the loop counter instead of a separate integer counter to maintain backward compatibility with releases before 4D1−4.0.

-vms_cc Uses VMS Fortran carriage control interpretation on unit 6.

-vms_endfile

Causes a VMS endfile record to be written when an **ENDFILE** statement is executed, allows records to be written after an endfile record and subsequent reading from an input file after an endfile record is encountered.

-vms_library

Treats subprograms starting with **LIB$**, **OTS$**, and **SMG$** as VMS run−time routines that accept a variable number of arguments.

-vms_stdin Allows rereading from *stdin* after **EOF** has been encountered.

-w Suppresses warning messages.

-w66 Suppresses Fortran 66 compatibility warning messages.

Debugging

The compiler system provides a source−level, interactive debugger called *dbx* that you can use to debug programs as they execute. With *dbx* you can control program execution to set breakpoints, monitor what is happening, modify values, and evaluate results. *dbx* keeps track of variables, subprograms, subroutines, and data types in terms of the symbols used in the source language. You can use this debugger to access the source text of the program, to identify and reference program entities, and to detect errors in the logic of the program.

Reference Information

For a complete list of **-g** driver options, see the *f77*(1) manual page. See the *dbx*(1) manual page for information on the debugger. For a complete description see the *dbx Reference Manual.*

Profiling

The compiler system permits the generation of profiled programs that, when executed, provide operational statistics. This is done through driver option **-p** (which provides pc sampling information) and the *pixie* and *prof* programs.

A variety of options and methods of profiling are available. To learn more about them, read Chapter 2 of the *IRIX Series Compiler Guide*, which describes the advantages and methods of profiling. It also gives examples of the various options and commands to achieve the desired results. See the *prof*(1) manual page for detailed reference information.

Optimizing

The default optimizing option, **-O1**, causes the code generator and assembler phases of compilation to improve the performance of your executable object. You can prevent optimization by specifying **-O0**.

Table 1−6 summarizes the optimizing functions available.

Table 1−6 Optimizer Options

Option	Result
-O3	Performs all optimizations, including global register allocation. With this option, a *ucode* object file is created for each Fortran source file and left in a **.u** file. The newly created *ucode* object files, the *ucode* object files specified on the command lines, the run−time startup routine, and all of the run−time libraries are *ucode* linked. Optimization is done globally on the resulting *ucode* linked file, and then it is linked as normal, producing an *a.out* file. No **.o** file is left from the *ucode* linked result. **-c** cannot be specified with -O3.

-O2	The global optimizer (*uopt*) phase executes. It performs optimization only within the bounds of individual compilation units.
-O1	Default option. The code generator and the assembler perform basic optimizations in a more limited scope.
-O0	No optimization.

The default option, **-O1**, causes the code generator and the assembler to perform basic optimizations such as constant folding, common subexpression elimination within individual statements, and common subexpression elimination between statements.

The global optimizer, invoked with the **-O2** option, is a single program that improves the performance of an object program by transforming existing code into more efficient coding sequences. Although the same optimizer processes all compiler optimizations, it does distinguish between the various languages supported by the compiler system programs to take advantage of the different language semantics involved.

See the *IRIX Series Compiler Guide* for details on the optimization techniques used by the compiler and tips on writing optimal code for optimizer processing.

Performance

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

- The **-feedback** and **-cord** options (see the *f77*(1) manual page) together with the *pixie*(1) and *prof*(1) utilities, can be used to reduce possible machine cache conflicts.
- The link editor **-G num** and **-bestGnum** options control the size of the global data area, which can produce significant performance improvements. See Chapter 2 of the *IRIX Series Compiler Guide* and the *ld*(1) manual page for more information.
- The **-jmpopt** option permits the link editor 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 *ld*(1) for more information.

Object File Tools

The following tools provide information on object files as indicated:

odump	Lists headers, tables, and other selected parts of an object or archive file. Chapters 10 and 11 of the *Assembly Language Programmer's Guide* describe the information provided.
stdump	Lists intermediate–code symbolic information for object files, executables, or symbolic information files.
nm	Prints symbol table information for object and archive files.
file	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.

size — Prints information about the text, rdata, data, sdata, bss, and sbss sections of the specified object or archive files. See Chapter 10 of the *Assembly Language Programmer's Guide* for a description of the contents and format of section data.

For more information on these tools, see the *odump*(1), *stdump*(1), *nm*(1), *file*(1), or *size*(1) manual 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 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 *ar*(1) manual 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.

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, 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 1−7shows the standard preconnected files at program start.

Table 1−7 Preconnected Files

Unit #	Unit
5	Standard input
6	Standard output
0	Standard error

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 are 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 already 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.

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* or *edge* to inspect this file and determine the state of the program at termination. For more information, see the *dbx Reference Manual* and the *edge*(1) manual page.

  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 **a.out**). For more information on *dbx*, see "Debugging".

Trap Handling

The library *libfpe.a* provides two methods for handling floating point exceptions: the subroutine **handle_sigfpes** 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. See the *handle_sigfpes*(3F) manual page for more information.