Chapter 1

Using the Compiler System

This chapter contains these sections:

- "Overview" lists the components of the IRIS-4D compiler system.
- "Object File Format and Dynamic Linking" discusses the major differences between the latest version of IRIX and previous versions.
- "Source File Considerations" explains source file naming conventions and the procedure for including header files.
- "Compiler Drivers" lists and explains the general compiler-driver options.
- "Linking" explains how to manually link-edit programs (using *ld* or a compiler driver) and how to compile multilanguage programs. It also describes Dynamic Shared Objects and how to link them into your programs.
- "Debugging" explains the compiler-driver options for debugging.
- "Getting Information about Object Files" explains how to use the object file tools to analyze object files.
- "Using the Archiver to Create Libraries" explains how to use the archiver, *ar*.

Overview

The IRIS-4D compiler system consists of a set of components that enable you to create executable programs using such languages as C, Fortran 77, and Pascal. Table 1-1 summarizes the IRIS-4D compiler system components and the task each performs.

Table 1-1 Compiler System Functional Components

Tool	Task	Examples
Text editor	Write and edit programs	*vi*, *jot*, *emacs*
Compiler driver	Compile, link, and load programs	*cc*, *f77*, *pc*
Object file analyzer	Analyze object files	*elfdump*, *file*, *nm*, *odump*, *size*
Profiler	Analyze program performance	*prof*, *pixie*
Optimizer	Improve program performance	*uopt*
Archiver	Produce object-file libraries	*ar*
Runtime loader	Link Dynamic Shared Objects at runtime	*rld*
Debugger	Debug programs	*dbx*

A single program called a compiler driver (such as *cc*, *f77*, or *pc*) invokes the following major components of the compiler system (refer to Figure 1-1):

- Macro preprocessor (*cfe*)
- Parallel analyzer (*pca, pfa*)
- Compiler front end (*cfe*, *fcom*, *upas*)
- Ucode tools (*ujoin, uld, umerge*)
- Optimizer (*uopt*)
- Code generator (*ugen*)
- Assembler (*as1*)
- Link editor (*ld*)

Note: C++ has a specialized driver, *CC*, with slightly different options from *cc*, *f77*, and *pc*. Refer to the *C++ Programming Guide* for details. ♦

You can invoke a compiler driver with various options (described later in this chapter) and with one or more source files as arguments. All specified source files are automatically sent to the macro preprocessor.

Note: Preprocessing is now done by *cfe*, but the old preprocessors (*cpp* for “traditional” Kernighan & Ritchie C, or *acpp* for ANSI C) are still available for non-compilation preprocessing in case you want to use them. ♦

Although the macro preprocessor was originally designed for C programs, it is now run by default as part of almost all compilations. To prevent the preprocessor from being run, specify the *–nocpp* option on the driver command line.

If available, the parallel analyzers *pca* and *pfa* produce parallelized source code from standard source code. The result takes advantage of multiple CPUs (when present) to achieve higher computation rates. *pca* and *pfa* are part of the Power C and Power Fortran packages; for more information about these packages and how to obtain them, contact your dealer or sales representative.

The compilers proper, often called “front ends,” translate source code into intermediate code. The available compiler front ends are *cfe* (C), *fcom* (Fortran 77), and *upas* (Pascal). *ujoin*, *uld*, *umerge,* and *uopt* comprise the optimization subsystem of the compiler system. (For more information about optimization and profiling, see Chapter 3, “Improving Program Performance.”) *ugen* and *as1* make up the code-generation subsystem of the compiler system.

The link editor *ld* combines several object files into one, performs relocation, resolves external symbols, and merges symbol table information for symbolic debugging. The driver automatically runs *ld* unless you specify the *–c* option to skip the linking step.

To see the various utilities a program passes through during compilation, invoke the appropriate driver with the *–v* option (or *+v* for the C++ driver *CC*).

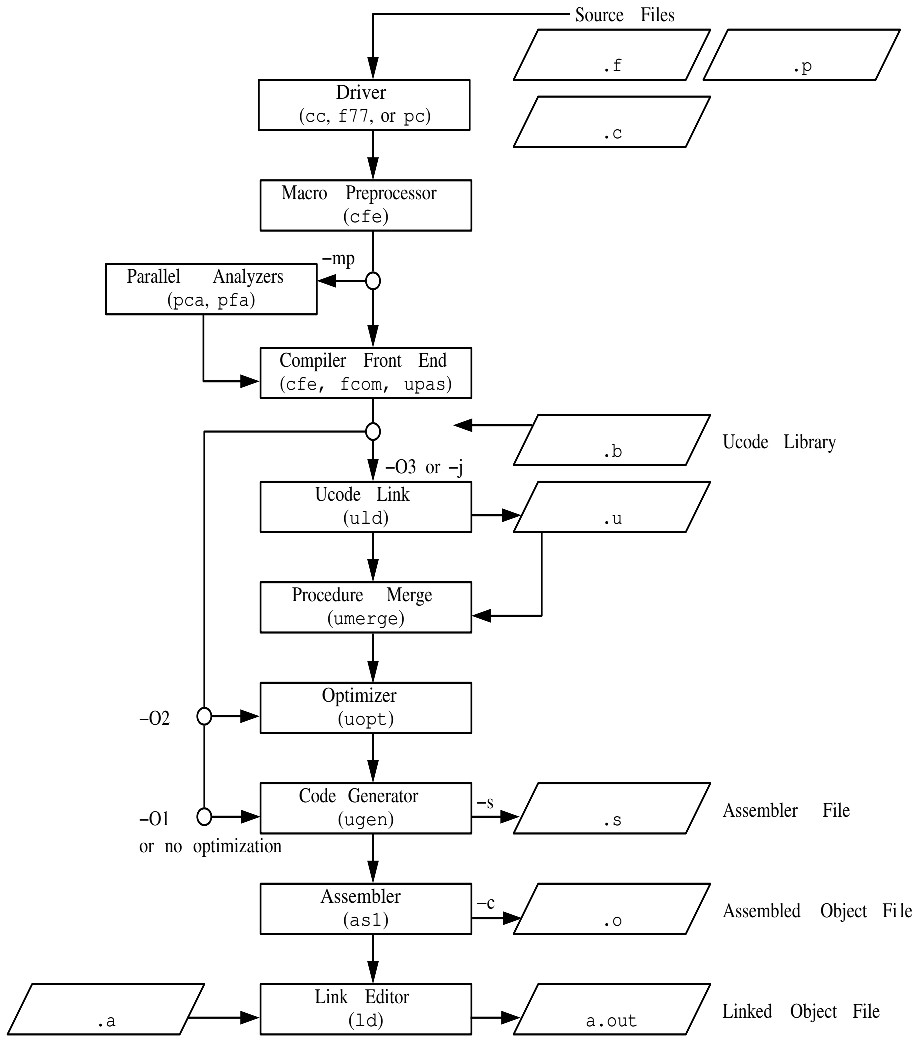

Figure 1-1 Compiler System Flowchart

Object File Format and Dynamic Linking

A new object file format was adopted in IRIX version 5.0. The major differences between the current compiler system and pre-5.0 compiler systems are summarized below:

- The compiler system uses a new format, Executable and Linking Format (ELF), for object files.
- The compiler system uses a new kind of shared library, the Dynamic Shared Object (DSO).
- The compiler system creates Position-Independent Code (PIC) by default to support dynamic linking.

Executable and Linking Format

Previous versions of IRIX used an extended version of the Common Object File Format (COFF) for object files. The current compiler system produces ELF object files instead. ELF is the format specified by the System V Release 4 Applications Binary Interface (the SVR4 ABI). In addition, ELF provides support for Dynamic Shared Objects, described below. There are three kinds of ELF object files:

- Relocatable files contain code and data in a format suitable for linking with other object files to make a shared object or executable.
- Dynamic Shared Objects contain code and data suitable for *dynamic linking*. Relocatable files may be linked with DSOs to create a dynamic executable. At runtime, the runtime linker combines the executable and DSOs to produce a process image.
- Executable files are programs ready for execution. They may or may not be dynamically linked.

COFF executables will continue to run on new releases of IRIX, but the current compiler system has no facility for creating or linking COFF executables. COFF and ELF object files may not be linked together. To take advantage of new IRIX features, you must recompile your code.

IRIX will execute all binaries that are compliant with the SVR4 ABI, as specified in the *System V Applications Binary Interface—Revised Edition* and the *System V ABI MIPS Processor Supplement*; however, binaries compiled under this version of the compiler system are not guaranteed to comply with the SVR4 ABI. The MIPS-specific version of the SVR4 ABI is referred to as the MIPS ABI. Programs that comply with the MIPS ABI can be run on any machine that supports the MIPS ABI.

Dynamic Shared Objects

IRIX 5.0 introduced a new kind of shared object called a *Dynamic Shared Object*, or *DSO*. The object code of a DSO is *Position-Independent Code* (*PIC*), which can be mapped into the virtual address space of several different processes at once. DSOs are loaded at runtime instead of at linking time, by the runtime loader, *rld*. As is true for static shared libraries, the code for DSOs is not included in executable files; thus, executables built with DSOs are smaller than those built with non-shared libraries, and multiple programs may use the same DSO at the same time.

Static shared libraries are only supported under this release for the purposes of running old (COFF) binaries. The current compiler system has no facilities for generating static shared libraries.

Position-Independent Code

Dynamic linking requires that all object code used in the executable be Position-Independent Code. For source files in high-level languages, you just need to recompile to produce PIC. Assembly language files must be modified to produce PIC; see Appendix A, “Position-Independent Coding in Assembly Language,” for details.

Position-Independent Code satisfies references indirectly by using a *Global Offset Table* (GOT), which allows code to be relocated simply by updating the GOT. Each executable and each DSO has its own GOT.

The compiler system will now produce PIC by default when compiling higher-level language files. All of the standard libraries are now provided as DSOs, and therefore contain PIC code; if you compile a program into non-PIC, you will be unable to use those DSOs. One of the few reasons to compile non-PIC is to build a device driver, which doesn’t rely on standard libraries; in this case, you should use the *–non_shared* option to the compiler driver to negate the default option, *–KPIC*. For convenience, the C library and math library are provided in non-shared format as well as in DSO format (although the non-shared versions are not installed by default). These libraries can be linked *–non_shared* with other non-PIC files.

When running Position-Independent Code, the global pointer is used to point to the *Global Offset Table*, so you can no longer use the *–G* option to store data in the global pointer region (that is, *–KPIC*, the default, implies *–G 0*). The compiler will ignore any user-specified *–G* number other than zero. For more information about this option, see the *ld*(1) manual page.

Source File Considerations

This section describes conventions for naming source files and including header files.

Source File Naming Conventions

Each compiler driver recognizes the type of an input file by the suffix assigned to the file name. Table 1-2 describes the possible file name suffixes.

Table 1-2 Driver Input File Suffixes

Suffix	Description
.s	Assembly source
.i	Preprocessed source code in the language of the processing driver
.c	C source
.f	Fortran 77 source
.p	Pascal source
.u	Ucode object file
.b	Ucode object library
.o	Object file
.a	Object library

If you purchased the C++ option, refer to the *C++ Programming Guide* for details about input files for the C++ driver, *CC*.

Examples

calc.p is an example of a Pascal program name.

When invoked as follows, *f77* assumes the file *tickle.i* contains Fortran statements (because the Fortran driver is specified). *f77* also assumes the file has already been preprocessed (because the suffix is *.i*), and therefore will not invoke the *cfe* preprocessor:

```
f77 -c tickle.i
```

Header Files

Header files, also called *include* files, contain information about the libraries they're associated with. They define such things as data structures, symbolic constants, and prototypes and parameters for the functions in the library. For example, the *stdio.h* header file describes, among other things, the data types of the parameters required by *printf()*. To use those definitions without having to type them into each of your source files, you can use the *#include* command to tell the macro preprocessor to include the complete text of the given header file in the current source file. Including header files in your source files allows you to specify such definitions conveniently and consistently in each source file that uses any of the library routines.

By convention, header file names have a *.h* suffix. Each programming language handles these files the same way, via the macro preprocessor.

Note: Do not put any code other than definitions in an include file, particularly if you intend to debug your program using *dbx*. The debugger recognizes an include file as only one line of source code, so source lines in an include file do not appear during debugging sessions. ♦

Specifying a Header File

The #include command tells the preprocessor to replace the #include line with the text of the indicated header file. The usual way to specify a header file is with the line:

```
#include <filename>
```

where *filename* is the name of the header file to be included. The angle brackets (**< >**) surrounding the file name tell the macro preprocessor to search for the specified file only in directories specified by command-line options and in the default header-file directory (*/usr/include*).

There is another specification format, in which the file name is given between double quotation marks:

```
#include "filename"
```

In this case, the macro preprocessor searches for the specified header file in the current directory first, then (if it doesn't find the requested file) goes on and searches in the other directories as in the angle-bracket specification.

Note: When you specify header files in your source files, the #*include* keyword should always start in column 1 (that is, all the way over to the left) to be recognized by the preprocessor.

Creating a Header File for Multiple Languages

A single header file can contain definitions for multiple languages; this setup allows you to use the same header file for all programs that use a given library, no matter what language those programs are in. To set up a shareable header file, create a *.h* file and enter the definitions for the various languages as follows:

```
#ifdef _LANGUAGE_C

C definitions

#endif

#ifdef _LANGUAGE_C_PLUS_PLUS

C++ definitions
```

```
#endif

#ifdef _LANGUAGE_FORTRAN

Fortran definitions

#endif

and so on for other language definitions
```

Note: To indicate C++ definitions you must use _LANGUAGE_C_PLUS_PLUS, not _LANGUAGE_C++. ♦

You can specify the various language definitions in any order, but you must specify _LANGUAGE_ before the language name.

Compiler Drivers

The driver commands, such as *cc*, *f77*, and *pc*, call subsystems that compile, optimize, assemble, and link-edit your programs. This section describes driver options that can be used with any of the drivers.

Default Behavior for Compiler Drivers

At compilation time, you can select one or more options that affect a variety of program development functions, including debugging, optimization, and profiling facilities. You can also specify the names assigned to output files. However, some options have default values that apply if you do not specify the option.

When you invoke a compiler driver with source files as arguments, the driver calls other commands that compile your source code into object code. It then optimizes the object code (if requested to do so) and links together the object files, the default libraries, and any other libraries you specify.

Example

Given a source file *foo.c*, the default name for the object file is *foo.o*. The default name for an executable file is *a.out*. So the following example compiles source files *foo.c* and *bar.c* with the default options:

```
cc foo.c bar.c
```

This example produces two object files (*foo.o* and *bar.o*), then links those together with the default C library *libc* to produce an executable called *a.out*.

Note: If you compile a single source directly to an executable, the compiler will not create an object file. ♦

General Options for Compiler Drivers

Table 1-3 describes some of the command-line options for IRIS-4D compiler drivers. Note that not all of the options work with every driver.

Note: Table 1-3 lists only the most frequently used options, not all available options. See the *cc*(1), *pc*(1), and *f77*(1) manual pages for a complete list of options. ♦

You can use the compiler system to generate profiled programs that, when executed, provide operational statistics. To perform this procedure, use the *–p* compiler option (for pc sampling information) and the *pixie* program (for profiles of basic block counts). Refer to Chapter 3, “Improving Program Performance,” for details.

In addition to the general options in Table 1-3, each driver also has options that you normally will not use. These options primarily aid compiler development work. For information about nonstandard driver options, consult the appropriate driver manual page.

Table 1-3 General Driver Options

Option	Purpose
–c	Prevents the link editor from linking your program after assembly code generation. This option forces the driver to produce a *.o* file after the assembler phase, and prevents the driver from producing an executable file.
–C	(C driver only) Used with the *–P* or *–E* option. Prevents the macro preprocessor from stripping comments. Use this option when you suspect the preprocessor is not producing the intended code and you want to examine the code with its comments. Note that *–C* is really an option to *cfe*; this option will be passed along to *cfe* if you specify it with *cc*.
–C	(Pascal and Fortran drivers only) Generates code that invokes range checking for subscripts during program execution.
–Dname[=*def*]	Defines a macro *name* as if you had specified a *#define* in your program. If you do not specify a definition with *=def*, *name* is set to 1.
–E	(C driver only) Runs only the macro preprocessor and sends results to the standard output. To retain comments, use the *–C* option as well. Use *–E* when you suspect the preprocessor is not producing the intended code.
–I*dirname*	Adds *dirname* to the list of directories to be searched for specified header files. These directories are always searched before the default directory, */usr/include*.
–mips1	Generates code using the instruction set of the MIPS R2000/R3000 RISC architecture. This is the default.
–mips2	Generates code using the MIPS II instruction set (MIPS I + R4000 specific extensions). Note that code compiled with -mips2 will not run on R2000/R3000 based machines.
–nocpp	Suppresses running of the macro preprocessor on the source files prior to processing.

Table 1-3 (continued) General Driver Options

Option	Purpose
–non_shared	Turns off the default option, *–KPIC*, to produce non-shared code that thus can be linked to only a few standard libraries (such as *libc.a* and *libm.a*) that are provided in non-shared format, in the directory */usr/lib/nonshared*. Should therefore usually be used only when building device drivers.
–nostdinc	Suppresses searching of */usr/include* for the specified header files.
–o *filename*	Names the result of the compilation *filename*. If an executable is being generated, it will be named *filename* rather than the default name, *a.out*. If a single source file is being compiled with *–c*, the object will be named *filename* (not, it should be noted, *filename.o*; if you want the object file name to end with *.o*, you should specify that in the argument to *–o*). Otherwise, this option will be ignored.
–P	Runs only the macro preprocessor on the files and puts the result of each file in a *.i* file. Specify both *–P* and *–C* to retain comments.
–S	Similar to *–c*, except that it produces assembly code in a *.s* file instead of object code in a *.o* file.
–U*name*	Overrides a definition of the macro *name* that you specified with the *–D* option, or that is defined automatically by the driver. Note that this option does not override a macro definition in a source file, only on the command line.
–v	Lists compiler phases as they are executed. Use this option to see the default options for each compiler phase along with the options you've specified.
–w	Suppresses warning messages.

Linking

The link editor, *ld*, combines one or more object files and libraries (in the order specified) into one executable file, performing relocation, external symbol resolutions, and all other required processing. Unless directed otherwise, the link editor names the executable file *a.out*.

This section summarizes the functions of the link editor. Also described here are how to link-edit a program manually (without using a compiler driver) and how to compile multilanguage programs. Refer to the *ld*(1) manual page for complete information on the link editor.

Invoking the Link Editor Manually

Usually the link editor is invoked by the compiler driver as the final step in compilation (as explained in "Compiler Drivers"). If you have object files produced by previous compilations that you want to link together, you can invoke the link editor using a compiler driver instead of calling *ld* directly; just pass the object-file names to the compiler driver in place of source-file names. If the original source files were in a single language, simply invoke the associated driver and specify the list of object files. (For information about linking together objects derived from several languages, see "Linking Multilanguage Programs.")

There are a few command-line options to *ld*, such as *–p*, which have different meanings when used as command-line options to *cc*; to pass such options to *ld* through an invocation of a compiler driver, use the *–Wl* option to the driver (see the manual page for details).

There are a few circumstances under which you need to invoke *ld* directly, such as when you're doing special linking not supported by compiler drivers (such as building an embedded system). But most of the time it's simplest just to call a compiler driver and let it invoke *ld* as necessary. Nonetheless, a summary of *ld* syntax is provided here in case you need it.

Syntax

```
ld options object1 [object2...objectn]
```

options One or more of the options listed in Table 1-4.

object Specifies the name of the object file to be link-edited.

Table 1-4 contains only a partial list of link editor options. Many options that apply only to creating shared objects are discussed in the next chapter. For complete information on options and libraries that affect link editor processing, refer to the *ld*(1) manual page.

Table 1-4 Link Editor Options

Option	Purpose
–kl*libname*	Similar to *–llibname*, but the library is a ucode library named *lib*libname.*b*.
–l*libname*	Specifies the name of a library, where *libname* is the library name. The link editor searches for a *lib*libname.*so* (and then *lib*libname.*a*) first in any directories specified by *–L dirname* options, and then in the standard directories: */lib*, */usr/lib*, and */usr/local/lib*.
–L *dirname*	Adds *dirname* to the list of directories to be searched for along with libraries specified by subsequent *–llibname* options.
--m	Produces a link editor memory map, listing input and output sections of the code, in System V format.
–M	Produces a link map in BSD format, listing the names of files to be loaded.
–nostdlib	This option must be accompanied by the *–L dirname* option. If the link editor does not find the library in *dirname*, then it does not search any of the standard library directories.

Table 1-4 (continued) Link Editor Options

Option	**Purpose**
–o *filename*	Specifies a name for your executable. If you do not specify *filename*, the link editor names the executable *a.out*.
–s	Strips symbol table information from the program object, reducing its size. This option is useful for linking routines that are frequently linked into other program objects.
–v	Prints the name of each file as it is processed by the link editor.
–Xsortbss	Sorts bss symbols (this is the default in C but not in Fortran).
–Xnobsschange	Overrides defaults, eliminating all global bss reordering.
–y*symname*	Reports all references to, and definitions of, the symbol *symname*. Useful for locating references to undefined symbols.

Example

The following command tells the linker to search for the DSO *libcurses.so* in the directory */lib*. If it does not find that DSO, the linker then looks for *libcurses.a* in */lib*; then for *libcurses.so* in */usr/lib*, then in the same directory for *libcurses.a*. If it hasn't found an appropriate library by then, it looks in */usr/local/lib* for *libcurses.a*. (Note that the linker will not look for DSOs in */usr/local/lib*, so don't put shared objects there.) If found in any of those places, the DSO or library is linked with the objects *foiled.o* and *again.o*:

```
ld foiled.o again.o -lcurses
```

Note: The *–G* option, which formerly allowed you to specify which data items should be stored in the global pointer region, is no longer useful. *–KPIC*, the default, implies *–G 0*, and the compiler will ignore any user attempts to specify otherwise. Compiling *–non_shared* (to avoid *–KPIC*) is primarily useful only for creating device drivers, in which case there is no direct linking step in which to specify a *–G* number. For more information, see the *cc* and *ld* manual pages. ♦

Linking Assembly Language Programs

The assembler driver *as1* does not run the link editor. To link-edit a program written in assembly language, use one of these procedures:

- Assemble and link-edit using one of the other driver commands (*cc*, for example). The *.s* suffix of the assembly language source file causes the driver to invoke the assembler.
- Assemble the file using *as*; then link-edit the resulting object file with the *ld* command.

Specifying Libraries

The link editor *ld* processes its arguments from left to right as they appear on the command line. Arguments to *ld* can be DSOs, object files, or libraries.

When *ld* reads a DSO, it adds all the symbols from that DSO to a cumulative symbol table. If it encounters a symbol that's already in the symbol table, it does not change the symbol table entry; so if you define the same symbol in more than one DSO, only the first definition will be used.

When *ld* reads an archive, usually denoted by a file name ending in *.a*, it uses only the object files from that archive that can resolve currently unresolved symbol references. (When a symbol is referred to but not defined in any of the object files that have been loaded so far, it's called unresolved.) Once a library has been searched in this way, it is never searched again; so libraries should come after object files on the command line in order to resolve as many references as possible. Note that if a symbol is already in the cumulative symbol table from having been encountered in a DSO, its definition in any subsequent library will be ignored.

Libraries and DSOs can be specified either by explicitly stating a pathname or by use of the library search rules. To specify a library or DSO by path, simply include that path on the command line (relative to the current directory, or else absolute):

```
ld myprog.o /usr/lib/libc.so.1 mylib.so
```

Note: *libc.so.1* is the name of the standard C DSO, replacing the older *libc.a*. Similarly, *libX11.so.1* is the X11 DSO. Most other DSOs are simply named *name.so*, with no *.1* at the end. ♦

To use the linker's library search rules, specify the library with the *–llibname* option:

```
ld myprog.o –lmylib
```

When the *–lmylib* argument is processed, *ld* searches for a file called *libmylib.so*. If it can't find *libmylib.so* in a given directory, it tries to find *libmylib.a* there; if it can't find that either, it moves on to the next directory in its search order. The default search order is to look first in */lib*, then in */usr/lib*. After looking in both of those directories, *ld* will look in */usr/local/lib* for archives only (DSOs should not be installed in */usr/local/lib*). You can modify these defaults by specifying the *–L dir* and/or *–nostdlib* options. Directories specified by *–L dir* before the *–llibname* argument are searched in the order they appear on the command line, before the default directories are searched. If *–nostdlib* is specified, then *–L dir* must also be specified because the default directories will not be searched at all.

If *ld* is invoked from one of the compiler drivers, all *–L* and *–nostdlib* options are moved up on the command line so that they appear before any *–llibname* option. For example:

```
cc file1.o –lm –L mydir
```

invokes, at the linking stage of compilation:

```
ld –L mydir file1.o –lm
```

Note: There are three different kinds of files that contain object code files: non-shared libraries, PIC archives, and DSOs. Non-shared libraries are the old-fashioned kind of library, built using *ar* from *.o* files that were compiled with *–non_shared*. These archives must also be linked *–non_shared*. PIC archives are the default in IRIX 5.0, built using *ar* from *.o* files compiled with *–KPIC* (a default option); they can be linked with other PIC files. DSOs are built from PIC *.o* files by using *ld –shared*; see Chapter 2 for details. ♦

When compiling multilanguage programs, be sure to specify any required runtime libraries using the *–llibname* option. For a list of the libraries that a language uses, see the corresponding compiler driver manual page.

If the link editor tells you that a reference to a certain function is unresolved, check that function's manual page to find out which library the function is in. If it isn't in one of the standard libraries (which *ld* links in by default), you may need to specify the appropriate library on the command line. For an alternative method of finding out where a function is defined, see "Finding a symbol in an unknown library" on page 28.

Note: Simply including the header file associated with a library routine is not enough; you also must specify the library itself when linking (unless it's a standard library). There is no magical connection between header files and libraries; header files only give prototypes for library routines, not the library code itself. ♦

Examples

To link a sample program *foo.c* with the math DSO, *libm.so*, enter:

```
cc foo.c -lm
```

To specify the appropriate DSOs for a graphics program *foogl.c*, enter:

```
cc foogl.c -lgl -lX11
```

Linking to Dynamic Shared Objects

This section describes how to link your source files with previously built DSOs; for more information about how to build your own DSOs, see Chapter 2, "Dynamic Shared Objects."

Note: DSOs replace the older static shared libraries, which were named with the extension *_s.a*. The *_s.a* libraries are no longer shipped with IRIX; however, the runtime versions of those libraries, named with *_s* at the end (and no *.a*), are still present under IRIX 5.0 for backward compatibility with older executables that used static shared libraries. ♦

To build an executable that uses a DSO, call a compiler driver just as you would for a non-shared library. For instance,

```
cc needle.c –lthread
```

links the resulting object file (*needle.o*) with the previously built DSO *libthread.so* (and the standard C DSO, *libc.so.1*), if available. If there is no *libthread.so*, but there is a PIC archive named *libthread.a*, that archive will be used with *libc.so.1*, and you will still get dynamic (runtime) linking. Note that even *.a* libraries now contain Position-Independent Code by default, though it is also possible to build non-shared *.a* libraries that do not contain PIC.

Linking Multilanguage Programs

When the source language of the main program differs from that of a subprogram, use the following steps to link (refer to Figure 1-2):

1. Compile object files from the source files of each language separately by using the *–c* option. For example, if the source consists of a Fortran main program (*main.f*) and two files of C functions (*more.c* and *rest.c*), use the commands:

   ```
   cc –c more.c rest.c
   ```

   ```
   f77 –c main.f
   ```

   These commands produce the object files *main.o, more.o,* and *rest.o*.

2. Use the driver associated with the language of the main program to link the objects together:

   ```
   f77 main.o more.o rest.o
   ```

Note: The compiler drivers will supply the default set of libraries necessary to produce an executable from the source of the associated language, but when producing executables from source code in several languages, you may need to explicitly specify the default libraries for one or more of the languages used. For instructions on specifying libraries, see "Specifying Libraries." ♦

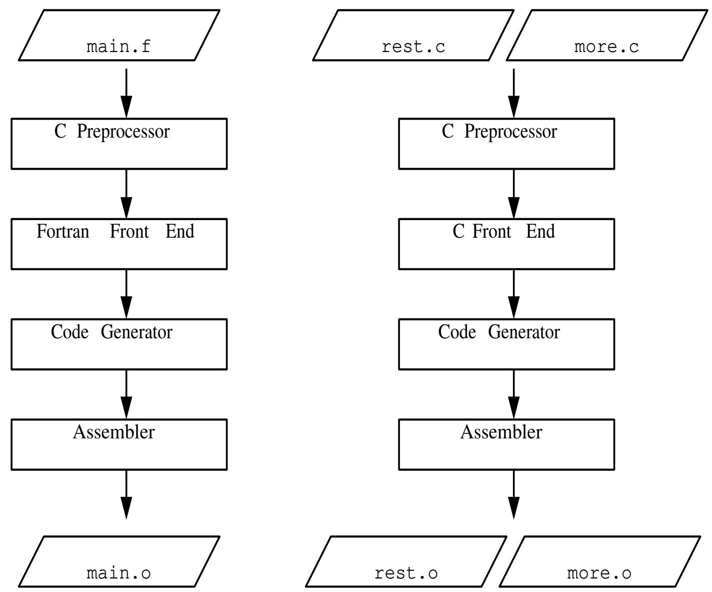

Figure 1-2 Compilation Control Flow for Multilanguage Programs

For specific details about compiling multilanguage programs, refer to the programming guides for the appropriate languages.

Debugging

The compiler system provides a debugging tool, *dbx*, which is explained in the *dbx User's Guide*. In addition, CASEVision/WorkShop™ contains debugging tools. For information about obtaining WorkShop for your computer, contact your dealer or sales representative.

Before using a debugging tool, you must use one of the standard driver options, listed in Table 1-5, to produce executables containing information that the debugger can use.

Table 1-5 Driver Options for Debugging

Option	Purpose
–g0	(Default) Produces a program object without debugging information. Reduces the size of the program object but retains optimizations. Use this option after you have finished debugging.
–g1	Specifies accurate, but limited, source-level debugging. This option performs most optimizations.
–g or –g2	Specifies full source-level debugging. These options suppress optimizations that might interfere with full debugging.
–g3	Specifies full, but inaccurate, debugging on fully optimized code. This level of debugger output can be confusing or misleading. Specify this option only for programs that malfunction after you optimize them.

Getting Information about Object Files

The following tools provide information on object files as indicated:

- *elfdump* lists the contents (including the symbol table and header information) of an ELF-format object file.
- *file* provides descriptive information on the general properties of the specified file.
- *nm* lists symbol table information.
- *odump* lists the contents of a COFF-format object file.
- *size* prints the size of each section of an object file (some such sections are named *text*, *data*, and *sbss*). The *a.out*(4) manual page describes the format of these sections.

Listing Selected Parts of Object Files and Libraries with *elfdump*

The *elfdump* tool lists headers, tables, and other selected parts of an ELF-format object file or archive file.

Syntax

`elfdump` *`options filename1`* `[`*`filename2..filenamen`*`]`

options One or more of the options listed in Table 1-6.

filename Specifies the name of one or more object files whose contents are to be dumped.

For more information, see the *elfdump*(1) manual page.

Table 1-6 *Elfdump* Options

Option	Dumps
–cr	Compact relocation information.
–Dc	Conflict list in Dynamic Shared Objects.
–Dg	Global Offset Table in Dynamic Shared Objects.
–Dl	Library list in Dynamic Shared Objects.
–Dt	String table entries of the dynamic symbol table in Dynamic Shared Objects.
–f	The file header.
–h	All section headers in the file.
–hash	Hash table entries.
–L	Dynamic section in Dynamic Shared Objects.
–o	Program header.
–r	Relocation information.
–reg	Register info.
–t	Symbol table entries.

Determining File Type with *file*

The *file* tool lists the properties of program source, text, object, and other files. This tool attempts to identify the contents of files using various heuristics. It is not exact and is occasionally fooled. For example, it often erroneously recognizes command files as C programs. For more information, see the *file*(1) manual page.

Syntax

`file` *`filename1`* `[`*`filename2..filenamen`*`]`

where each *filename* is the name of a file to be examined.

Example

Information given by *file* is self-explanatory for most kinds of files, but using *file* on object files and executables gives somewhat cryptic output. In this example, "*MSB*" indicates Most Significant Byte, also called Big-Endian; "dynamic executable" means the executable was linked with DSO libraries; and "(not stripped)" means the executable contains at least some symbol table information. "Dynamic lib" indicates a DSO.

```
file test.o a.out /lib/libc.so.1
test.o:         ELF 32-bit MSB relocatable MIPS - version 1
a.out:          ELF 32-bit MSB dynamic executable (not stripped) MIPS - version 1
/lib/libc.so.1:   ELF 32-bit MSB dynamic lib MIPS - version 1
```

Listing Symbol Table Information: *nm*

The *nm* tool lists symbol table information for object files and archive files.

Syntax

`nm` *`options filename1`* `[`*`filename2..filenamen`*`]`

options One or more of the options listed in Table 1-7.

filename Specifies the object files or archive files from which symbol table information is to be extracted. If you do not specify a file name, *nm* assumes the file is called *a.out*.

Table 1-7 Symbol Table Dump Options

Option	Purpose
–a	Prints debugging information. If used with *–B*, uses BSD ordering with System V formatting.
–A	Prints the listing in System V format (default).
–b	Prints the value field in octal.
–B	Prints the listing in BSD format.
–d	Prints the value field in decimal (the default for System V output).
–e	Prints only external and static variables.
–h	Suppresses printing of headers.
–n	Sorts external symbols by name for System V format. Sorts all symbols by value for Berkeley format (by name is the BSD default output).
–o	Prints value field in octal (System V output). Prints the file name immediately before each symbol name (BSD output).
–p	Lists symbols in the order they appear in the symbol table.
–r	Reverses the sort that you specified for external symbols with the *–n* and *–v* options.
–T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This option may make the listing easier to read.
–u	Prints only undefined symbols.
–v	Sorts external symbols by value (default for Berkeley format).
–V	Prints the version number of *nm*.
–x	Prints the value field in hexadecimal.

Table 1-8 defines the one-character codes shown in an *nm* listing. Refer to the example that follows the table for a sample listing.

Table 1-8 Character Code Meanings

Key	Description
a	Local absolute data
A	External absolute data
b	Local zeroed data
B	External zeroed data
C	Common data
d	Local initialized data
D	External initialized data
E	Small common data
G	External small initialized data
N	Nil storage class (avoids loading of unused external references)
r	Local read-only data
R	External read-only data
s	Local small zeroed data
S	External small zeroed data
t	Local text
T	External text
U	External undefined data
V	External small undefined data

Example

This example demonstrates how to obtain a symbol table listing. Consider the following program, *tnm.c*:

```
#include <stdio.h>
```

```
#include <math.h>
#define LIMIT 12
int unused_item = 14;
double mydata[LIMIT];

main()
{
    int i;
    for(i = 0; i < LIMIT; i++) {
        mydata[i] = sqrt((double)i);
    }
    return 0;
}
```

Compile the program into an object file by entering:

cc –c tnm.c

To obtain symbol table information for the object file *tnm.o* in BSD format, use the *nm –B* command:

```
nm -B tnm.o
0000000000 T main
0000000000 B mydata
0000000000 U sqrt
0000000000 D unused_item
00000000 N _bufendtab
```

To obtain symbol table information for the object file *tnm.o* in System V format use the *nm* command without any options:

```
nm tnm.o
Symbols from tnm.o:

[Index]   Value     Size    Class    Type         Section    Name

[0]      |         0|       |File    |ref=4       |Text      | tnm.c
[1]      |         0|       |Proc    |end=3 int   |Text      | main
[2]      |       116|       |End     |ref=1       |Text      | main
[3]      |         0|       |End     |ref=0       |Text      | tnm.c
[4]      |         0|       |File    |ref=6       |Text      | /usr/include/math.h
[5]      |         0|       |End     |ref=4       |Text      | /usr/include/math.h
[6]      |         0|       |Global  |            |Data      | unused_item
[7]      |         0|       |Global  |            |Bss       | mydata
[8]      |         0|       |Proc    |ref=1       |Text      | main
```

```
[9]    |        0|       |Proc     |            |Undefined| sqrt
[10]   |        0|       |Global   |            |Undefined| _gp_disp
```

Finding a symbol in an unknown library

When *ld* indicates that a symbol is undefined, you can use *nm* to figure out which DSO or library needs to be linked in by piping *nm*'s output through appropriate *grep*s.

Example

You're trying to compile a program, and *ld* tells you that you're trying to use an undefined symbol:

```
cc prog.c -lgl
ld:
Unresolved:
XGetPixel
```

But you don't know where XGetPixel is defined. So use *nm* to list the symbol tables for all of the available DSOs, and filter that output to find only the places where XGetPixel is mentioned. Then filter the result to find only the places where XGetPixel is actually defined, as indicated by the T character code.

```
nm -Bo /usr/lib/lib*.so* | grep XGetPixel | grep T
/usr/lib/libX11.so.1: 0f790ff8 T XGetPixel
```

Note: Some DSOs end in *.so*, while others end in *.so.1*, so we need to use multiple wildcards to get all of them. Also, note that this command line would have to be modified to look in PIC archives or non-shared libraries; as written it will only look in DSOs. ♦

So now we now that XGetPixel is defined in */usr/lib/libX11.so.1*, the X11 DSO; use the *–l* option to tell *cc* to link in that library, and *ld* won't complain any more.

```
cc prog.c -lgl -lX11
```

Listing Selected Parts of COFF Files with *odump*

The *odump* tool lists headers, tables, and other selected parts of a COFF-format object or archive file. It is provided with this release of IRIX for compatibility; use *elfdump* for ELF-format files.

Syntax

`odump` *`options filename1`* `[`*`filename2..filenamen`*`]`

options One or more of the options listed in Table 1-9.

filename Specifies the name of one or more object files whose contents are to be dumped.

For more information, see the *odump*(1) manual page.

Table 1-9 *Odump* Options

Option	Dumps
–a	Archive header of each object file in the specified archive library file.
–c	String table.
–d *number*	The section numbered *number*, or a range of sections starting with *number* and ending with the last section number available (or the number you specify with the *+d* auxiliary option).
+d *number*	All sections starting with the first section (or with the section specified with the *–d* option) and ending with the section numbered *number*.
–f	File header for each object file in the specified file.
–F	File descriptor table for each object file in the specified file.
–g	Global symbols in the symbol table of an archive library file.
–h	Section headers.
–i	Symbolic information header.
–l	Line number information.

Table 1-9 (continued) *Odump* Options

Option	Dumps
–*n name*	Information for section named *name* only. Use this option with the –*h*, –*s*, –*r*, –*l*, or –*t* option.
–o	Optional header for each object file.
–p	Suppresses the printing of headers.
–P	Procedure descriptor table.
–r	Relocation information.
–R	Relative file index table.
–s	Section contents.
–t	Symbol table entries.
–*t index*	Only the indexed symbol table entry. Use the +*t* option with the –*t* option to specify a range of table entries.
+*t index*	Symbol table entries in a range that ends with the indexed entry. The range begins with the first symbol table entry or with the section that you specify with the –*t* option.
–v	Information in symbolic rather than numeric representation. This option may be used with any *odump* option except –*s*.
–*z name, number*	Line number entry (or a range of entries starting at the specified number) for the named function.
+*z number*	Line number entries starting with the function name or line number specified by the –*z* option and ending with *number*.

Determining Section Sizes with *size*

The *size* tool prints information about the sections (such as *text*, *rdata*, and *sbss*) of the specified object or archive files.

Syntax

```
size options [filename1 filename2..filenamen]
```

options Specifies the format of the listing (see Table 1-10).

filename Specifies the object or archive files whose properties are to be listed. If you do not specify a file name, the default is *a.out*.

Table 1-10 *Size* Options

Option	Action
–A	Prints data section headers in System V format.
–B	Prints data section headers in Berkeley format.
–d	Prints sizes in decimal (default).
–F	Prints data on loadable segments.
–n	Prints symbol table, global pointer, and more.
–o	Prints sizes in octal.
–s	Follows shared libraries, adding them as they're encountered to the list of files to be *size*d.
–V	Prints the version of *size* that you are using.
–x	Prints sizes in hexadecimal.

Example

Below are examples of the *size* command and the listings they produce:

```
size -B -o test.o
        text   data  bss    rdata  sdata  sbss  decimal  hex
test.o  31250  2010  40470  550    210    50    31232    7a00

size -B -d test.o
        text   data  bss    rdata  sdata  sbss  decimal  hex
test.o  12968  1032  16696  360    136    40    31232    7a00
```

Using the Archiver to Create Libraries

An archive library is a file that includes the contents of one or more object (*.o*) files. When the link editor (*ld*) searches for a symbol in an archive library, it loads only the code from the object file where that symbol was defined (not the entire 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 around within the library
- Extracting individual objects from the library

The following section explains the syntax of the *ar* command and lists some examples of how to use it. See the *ar*(1) manual page for details.

Note: *ar* simply strings together whatever object files you tell it to archive; thus, it can be used to build either non-shared or PIC libraries, depending on how the included *.o* files were built in the first place. If you do create a non-shared library with *ar*, remember to link it *–non_shared* with your other code. For information about building DSOs and converting libraries to DSOs, see Chapter 2. ♦

Syntax

```
ar options [posObject] libName [object1...objectn]
```

options Specifies the action that the archiver is to take. Table 1-11, Table 1-12, and Table 1-13 list the available options. To specify more than one option, don't use a dash or put spaces between the options. For example, use *ar ts*, not *ar –t –s*.

posObject Specifies the name of an object within an archive library. It specifies the relative placement (either before or after *posObject*) of an object that is to be copied into the library or moved within the library. This parameter is required when the *a*, *b*, or *i* suboptions are specified with the *m* or *r* option. The last example in "Examples," shows the use of a *posObject* parameter.

libName Specifies the name of the archive library you are creating, updating, or extracting information from.

object Specifies the name(s) of the object file(s) to manipulate.

Archiver Options

When running the archiver, specify exactly one of the options *d*, *m*, *p*, *q*, *r*, *t*, or *x* (listed in Table 1-11). In addition, you can optionally specify any of the modifiers in Table 1-12, as well as any of the archiver suboptions listed in Table 1-13.

Table 1-11 Archiver Options

Option	Purpose
d	Deletes the specified objects from the archive.
m	Moves the specified files to the end of the archive. If you want to move the object to a specific position in the archive library, specify an *a*, *b*, or *i* suboption together with a *posObject* parameter.
p	Prints the specified objects in the archive on the standard output device (usually the terminal screen).
q	Adds the specified object files to the end of the archive. This option is similar to the *r* option (described below), but is faster and does not remove any older versions of the object files that may already be in the archive. Use the *q* option when creating a new library.
r	Adds the specified object files to the end of the archive file. If an object file with the same name already exists in the archive, the new object file will overwrite it. If you want to add an object at a specific position in the archive library, specify an *a*, *b*, or *i* suboption together with a *posObject* parameter. Use the *r* option when updating existing libraries.
t	Prints a table of contents on the standard output (usually the screen) for the specified object or archive file.
x	Copies the specified objects from the archive and places them in the current directory. Duplicate files are overwritten. The last modified date is the current date (unless you specify the *o* suboption, in which case the date stamp on the archive file is the last modified date). If no objects are specified, copies all the library objects into the current directory.

Table 1-12 Archiver Modifiers

Option	Purpose
c	Suppresses the warning message that the archiver issues when it discovers that the archive you specified does not already exist.
C	Makes an archive compatible with pre-SVR4 IRIX.
E	The default; creates an archive matching the specifications given by the SVR4 ABI.
l	Puts the archiver's temporary files in the current working directory. Ordinarily, the archiver puts those files in */tmp* (unless the STMDIR environment variable is set, in which case *ar* stores temporary files in the directory indicated by that variable). This option is useful when */tmp* (or STMDIR) is full.
s	Creates a symbol table in the archive. This modifier is rarely necessary since the archiver updates the symbol table of the archive library automatically. Options *m*, *p*, *q*, and *r*, in particular, create a symbol table by default and thus do not require *s* to be specified.
v	Lists descriptive information during the process of creating or modifying the archive. When specified with the *t* option, produces a verbose table of contents.

Table 1-13 Archiver Suboptions

Suboption	Use with Option	Purpose
a	*m* or *r*	Specifies that the object file being added should follow the already-archived object file specified by the *posObject* parameter on the command line.
b	*m* or *r*	Specifies that the object file precede the object file specified by the *posObject* parameter.
i	*m* or *r*	Same as *b*.
o	x	Forces the last modified date of the extracted object file to match that of the archive file.
u	r	Tells the archiver not to replace the existing object file in the archive if the last modified date indicates that the object file already in the archive is newer (more recently modified) than the one you're adding.

Note: The *a* and *b* suboptions are only useful if the same symbol is defined in two or more of the object files in the archive (in which case, the symbol table shows the first definition listed in the archive). Under other circumstances, order of object files in an archive is irrelevant (and the *a* and *b* suboptions are useless), since *ld* uses the archive symbol table rather than searching linearly through the file. ♦

Examples

Create a new library, *libtest.a*, and add object files to it by entering:

```
ar cq libtest.a mcount.o mon1.o string.o
```

The *c* option suppresses an archiver message during the creation process. The *q* option creates the library and puts *mcount.o, mon1.o,* and *string.o* into it.

An example of replacing an object file in an existing library:

```
ar r libtest.a mon1.o
```

The *r* option replaces *mon1.o* in the library *libtest.a*. If *mon1.o* does not already exist in the library *libtest.a*, it is added.

Note: If you specify the same file twice in an argument list of files to be added to an archive, that file will appear twice in the archive. ♦

To add a new file immediately before *mcount.o* in this library, enter:

```
ar rb mcount.o libtest.a new.o
```

The *r* option adds *new.o* to the library *libtest.a*. The *b* option followed by *mcount.o* as the *posObject* causes the archiver to place *new.o* immediately before *mcount.o* in the archive.