Chapter 2

Using the MIPSpro Compiler System

This chapter provides information about the MIPSpro compiler system, and describes the object file format and dynamic linking. Specifically, this chapter covers the topics listed below:

- "Selecting a Compiler" explains how to specify a n32-bit, 64-bit, or 32-bit compilation mode.
- "Object File Format and Dynamic Linking" discusses object files including executable and linking format, dynamic shared objects, and position-independent code.
- "Source File Considerations" explains source file naming conventions, and the procedure for including header files.
- "Using Precompiled Headers in C and C++" describes automatic and manual precompiled header processing.
- "Compiler Drivers" lists and explains general compiler-driver options.
- "Linking" explains how to link programs manually (using *ld* or a compiler) and how to compile multilanguage programs. It also covers Dynamic Shared Objects and how to link them into a program.
- "Debugging" describes the compiler-driver options for debugging.
- "Getting Information About Object Files" provides information on how to use the object file tools to analyze object files.
- "Using the Archiver to Create Libraries" explains how to use the archiver, *ar*.

For information about tools such as *prof* and *cord*, see Chapter 4, "Using the Performance Tools." For information about optimizing your program, see Chapter 5, "Optimizing Program Performance."

Selecting a Compiler

You can select a compilation mode to compile either n32-bit, 64-bit, or 32-bit objects by using command-line options and/or setting an environment variable. When determining the compilation mode to use, the system first checks command-line options. Next, it checks the environment variable.

Using Command-Line Options

You can use the following command line options to specify a compilation mode:

- If you specify the **–n32** option to the compiler, the system compiles the source files to n32-bit objects.
- If you specify the **–64** option on the command line, the system compiles the source files to 64-bit objects.
- If you specify the **–32** option to the compiler, the system compiles the source files to 32-bit objects.

Setting an Environment Variable

You can set an environment variable to specify the compilation mode to use. To set up your environment for n32-bit compilation, type:

```
setenv SGI_ABI -n32
```

To set up your environment for 64-bit compilation, type:

```
setenv SGI_ABI -64
```

To set up your environment for 32-bit compilation, type:

```
setenv SGI_ABI -32
```

If you don't specify **–32**, **–n32**, or **–64** on the command line, or don't set an environment variable, the system determines the compilation mode to use.

Object File Format and Dynamic Linking

This section describes how the compiler system

- uses “Executable and Linking Format” (ELF) for object files
- uses shared libraries called “Dynamic Shared Objects” (DSOs)
- creates “Position-Independent Code” (PIC), by default, to support dynamic linking

Executable and Linking Format

The compiler system produces ELF object files. ELF is the format specified by the System V Release 4 Applications Binary Interface (the SVR4 ABI). ELF provides support for Dynamic Shared Objects, described below.

Types of ELF object files are as follows:

- 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 run time, the run-time 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 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.

You can use this version of the compiler system to construct ABI-compliant executables that run on any operating system supporting the MIPS ABI. Be careful to avoid referencing symbols that are not defined as part of the MIPS ABI specification. For more information, see

- *System V Applications Binary Interface—Revised First Edition*. Prentice Hall, ISBN 0-13-880410-9
- *System V Application Binary Interface MIPS Processor Supplement*. Prentice Hall, ISBN 0-13-880170-3.

Dynamic Shared Objects

IRIX uses shared objects called *Dynamic Shared Objects*, or *DSOs*. 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 run time instead of at linking time, by the run-time 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. For more information on DSOs, see Chapter 3, "Using Dynamic Shared Objects."

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 the *MIPSpro Assembly Language Programmer's Guide* 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. For more information on DSOs, see Chapter 3, "Using Dynamic Shared Objects."

The compiler system produces PIC by default when compiling higher-level language files. All of the standard libraries are 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 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). You can link these libraries **–non_shared** with other non-PIC files.

Source File Considerations

This section describes conventions for naming source files and including header files. Topics covered include:

- "Source File Naming Conventions"
- "Header Files"
- "Using Precompiled Headers in C and C++"

Source File Naming Conventions

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

Table 2-1 Driver Input File Suffixes

Suffix	Description
.s	Assembly source
.i	Preprocessed source code in the language of the processing driver
.c	C source
.C, .c++, .CC, .cc, .CPP, .cpp, .CXX, .cxx	C++ source
.f, .F, .for, .FOR .f, .f90, .F90	Fortran 77 source Fortran 90 source
.p	Pascal source
.o	Object file
.a	Object library archive
.so	Dynamic shared object library

The following example compiles preprocessed source code:

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

The Fortran 77 compiler, *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 does not invoke the preprocessor.

Header Files

Header files, also called *include* files, contain information about the libraries with which they're associated. They define such things as data types, data structures, symbolic constants, and prototypes for functions exported by the library. To use those definitions without having to type them into each of your source files, you can use the #*include* directive to tell the macro preprocessor to include the complete text of the given header file in the current source file. When you include header files in your source files you can specify such definitions conveniently and consistently in each source file that uses any of the library routines.

By convention, header filenames have a *.h* suffix. Each programming language handles these files the same way, via the macro preprocessor. For example, the *stdio.h* header file describes, among other things, the data types of the parameters required by **printf()**.

For detailed information about standard header files and libraries, see the International Standard ISO/IEC. *Programming languages—C, 9899*. 1990. Also see "Using Typedefs" on page 162 for information about the *inttypes.h* header file.

Specifying a Header File

The #*include* directive 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 filename 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*).

Another specification format exists, in which the filename is given between double quotation marks. In this case, the macro preprocessor searches for the specified header file in the current directory first (that is, the directory containing the *including* file). Then, if the preprocessor doesn't find the requested file, it searches in the other directories as in the angle-bracket specification.

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
```

Note: You must specify _LANGUAGE_ before the language name. To indicate C++ definitions, you must use _LANGUAGE_C_PLUS_PLUS, not _LANGUAGE_C++.

You can specify the various language definitions in any order.

Using Precompiled Headers in C and C++

This section describes the precompiled header mechanism that is available with the n32-bit and 64-bit C and C++ compilers. For information on the 32-bit precompiled header mechanism for C++ available with the Smart Build utility, see the *C++ Programming Guide*.

This section contains the following topics:

- "About Precompiled Headers"
- "Automatic Precompiled Header Processing"
- "Other Ways to Control Precompiled Headers"
- "PCH Performance Issues"

About Precompiled Headers

The precompiled header (PCH) file mechanism is available through the compiler front end: *fec* and *fecc*. Use PCH to avoid recompiling a set of header files. This is particularly useful when your header files introduce many lines of code, and the primary source files that included them are relatively small.

In effect, *fec*/*fecc* takes a snapshot of the state of the compilation at a particular point and writes it to a file before completing the compilation. When you recompile the same source file or another file with the same set of header files, the PCH mechanism recognizes the snapshot point, verifies that the corresponding PCH file is usable, and reads it back in.

The PCH mechanism can give you a dramatic improvement in compile-time performance. The trade-off is that PCH files may take a lot of disk space.

Automatic Precompiled Header Processing

This section covers the following topics:

- PCH File Requirements
- Reusing PCH files
- Obsolete File Deletion Mechanism

You can enable the precompiled header processing by using the **–pch** option on the command line. With the PCH mechanism enabled, *fec/fecc* searches for a qualifying PCH file to read in and/or creates one for use on a subsequent compilation.

The PCH file contains a snapshot of all the code preceding the *header stop point*. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive. The header stop point can also be specified directly by inserting a **#pragma hdrstop**. For example, consider the following C++ code:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

In this case, the header stop point is `int i` (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of *xxx.h* and *yyy.h*. If the first non-preprocessor token or the **#pragma hdrstop** appears within a `#if` block, the header stop point is the outermost enclosing `#if`. For example, consider the following C++ code:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

In this case, the first token that does not belong to a preprocessing directive is again `int i`, but the header stop point is the start of the `#if` block containing the `int`. The PCH file reflects the inclusion of *xxx.h* and conditionally the definition of **YYY_H** and inclusion of *yyy.h*. The file does not contain the state produced `by #if TEST`.

PCH File Requirements

A PCH file is produced only if the header stop point and the code preceding it (generally the header files themselves) meet the following requirements:

- The header stop point must appear at file scope; it may not be within an unclosed scope established by a header file. For example, a PCH file is not created in the following case:

```
// xxx.h
class A {

// xxx.C
#include "xxx.h"
int i; };
```

- The header stop point can not be inside a declaration started within a header file, and it can not be part of a declaration list of a linkage specification. For example, a PCH file is not created in the following case:

  ```
  // yyy.h
  static

  // yyy.C
  #include "yyy.h"
  int i;
  ```

  In this case, the header stop point is `int i`, but since it is not the start of a new declaration, a PCH file is not created

- The header stop point can not be inside a `#if` block or a `#define` started within a header file.
- The processing preceding the header stop must not have produced any errors. (Note that warnings and other diagnostics are not reproduced when the PCH file is reused.)
- References to predefined macros **__DATE__** or **__TIME__** must not have appeared.
- Use of the `#line` preprocessing directive must not have appeared.
- `#pragma no_pch` must not have appeared.

Reusing PCH Files

When a precompiled header file is produced, in addition to the snapshot of the compiler state, it contains some information that can be checked to determine under what circumstances it can be reused. This information includes the following:

- The compiler version, including the date and time the compiler was built.
- The current directory (in other words, the directory in which the compilation is occurring).
- The command line options.
- The initial sequence of preprocessing directives from the primary source file, including `#include` directives.
- The date and time of the header files specified in `#include` directives.

This information comprises the PCH *prefix*. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether or not the latter is applicable to the current compilation.

For example, consider the following C++ code:

```
// a.C
#include "xxx.h"

...              // Start of code

// b.C
#include "xxx.h"

...              // Start of code
```

When you compiled *a.C* with the **–pch** option, the PCH file *a.pch* is created. When you compile *b.C* (or recompile *a.C*), the prefix section of *a.pch* is read in for comparison with the current source file. If the command line options are identical and *xxx.h* has not been modified, *fec/fecc* reads in the rest of *a.pch* rather than opening *xxx.h* and processing it line by line. This establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (in other words, the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with the following code:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If one PCH file exists for *xxx.h* and a second for *xxx.h* and *yyy.h*, the latter will be selected (assuming both are applicable to the current compilation). After the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by "pch." Unless **–pch_dir** is specified, the PCH file is created in the directory of the primary source file.

When a precompiled header file is created or used, a message similar to the following is issued:

```
"test.C": creating precompiled header file "test.pch"
```

Obsolete File Deletion Mechanism

In automatic mode (when **–pch** is used), *fec/fecc* considers a PCH file obsolete and deletes it under the following circumstances:

- The file is based on at least one out-of-date header file but is otherwise applicable for the current compilation.
- The file has the same base name as the source file being compiled (for example, *xxx.pch* and *xxx.C*) but is not applicable for the current compilation (for example, because of different command-line options).

You must manually clean up any other PCH file.

Support for PCH processing is not available when multiple source files are specified in a single compilation. If the command line includes a request for precompiled header processing and specifies more than one primary source file, an error is issued and the compilation is aborted.

Other Ways to Control Precompiled Headers

You can use the following ways to control and/or tune how precompiled headers are created and used:

- You can insert a `#pragma hdrstop` in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. Thus you can specify where the set of header files subject to precompilation ends. For example,

  ```
  #include "xxx.h"
  #include "yyy.h"
  #pragma hdrstop
  #include "zzz.h"
  ```

  In this case, the precompiled header file includes the processing state for *xxx.h* and *yyy.h* but not *zzz.h*. (This is useful if you decide that the information added by what follows the `#pragma hdrstop` does not justify the creation of another PCH file.)
- You can use a `#pragma no_pch` to suppress the precompiled header processing for a given source file.
- You can use the command-line option **–pch_dir** *directoryname* to specify the directory in which to search for and/or create a PCH file.

PCH Performance Issues

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, writing out a precompiled header file doesn't cost much, even if it does not end up being used, and if it is used it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more), and so you probably don't want many of them sitting around.

You can see that, despite the faster recompilations, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. The greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large precompiled header files can be minimized without giving up the advantage of a significant speedup in compilation times.

To take full advantage of header file precompilation, you should reorder the `#include` sections of your source files and/or group the `#include` directives within a commonly used header file.

The *fecc* source provides an example of how this can be done. A common idiom is the following:

```
#include "fe_common.h"
#pragma hdrstop
#include ...
```

In this example, *fe_common.h* pulls in (directly and indirectly) a few dozen header files. The `#pragma hdrstop` is inserted to get better sharing with fewer PCH files. The PCH file produced for *fe_common.h* is slightly over a megabyte in size. Another example, used by the source files involved in declaration processing, is the following:

```
#include "fe_common.h"
#include "decl_hdrs.h"
#pragma hdrstop
#include ...
```

decl_hdrs.h pulls in another dozen header files, and a second, somewhat larger, PCH file is created. In all, the fifty-odd source files of *fecc* share just six precompiled header files. If disk space is at a premium, you can decide to make *fe_common.h* pull in all the header files used. In that case, a single PCH file can be used in building *fecc*.

Different environments and different projects have different needs. You should, however, be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to your source code.

Compiler Drivers

The driver commands, such as *cc* and *f77* call subsystems that compile, optimize, assemble, and link your programs. This section describes:

- "Default Behavior for Compiler Drivers"
- "General Options for Compiler 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, profiling, and optimizing. You can also specify the names assigned to output files. Note that some options have default values that apply if you do not specify them.

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.

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*. 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 them 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 does not create an object file.

General Options for Compiler Drivers

The command-line options for MIPSpro compiler drivers are listed and explained in Table 2-2. The table lists only the most frequently used options, not all available options. See the appropriate compiler reference (manual) page for additional details.

In addition to the general options in Table 2-2, each driver has options that you typically won't use. These options primarily aid compiler development work. For information about nonstandard driver options, consult the appropriate driver reference page. Click the word cc to view the *cc*(1) and *CC*(1) reference pages.

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 *prof* command (for profiles of basic block counts). Refer to Chapter 4, "Using the Performance Tools," for details.

Table 2-2 General Driver Options

Option	Purpose
–32	Generates a 32-bit object.
–n32	Generates an n32-bit object.
–64	Generates a 64-bit object. This is the default.
–ansi	Strict ANSI/ISO C compilation mode. Preprocessing adds only standard predefined symbols to the name space, and standard include files declare only standard symbols.
–c	Prevents the linker from linking your program after code generation. This option forces the driver to produce a *.o* file after the back-end phase, and prevents the driver from producing an executable file.
–C	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. For C and C++ compiles only.

Table 2-2 (continued) General Driver Options

Option	Purpose
–cord	Runs the procedure rearranger, *cord*(1) on the resulting file after linking. Rearranging improves the paging and caching performance of the program's text. The output of *cord* is placed in *a.out*, by default, or a file specified by the **–o** option. If you don't specify **–feedback**, then *outfile.fb* is used as the default.
–cckr	K&R/Version7 C compatibility compilation mode. Preprocessing may add more predefined symbols to the name space than in **–ansi** mode. Compilation adheres to the K&R language semantics.
–D*name*[**=***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	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.
–elspec *filename*	Specifies an ELF layout specification file (specifies the layout of object files, programs, and shared objects). See *elspec*(5) for details.
–feedback	Use with the **–cord** option to specify feedback file(s). You can produce this file by using *prof* with its **–feedback** option from an execution of the instrumented program produced by *pixie*(1). Specify multiple feedback files with multiple **–feedback** options.
–fullwarn	Checks code and produces additional warnings that are normally suppressed. This option is recommended for all compiles during software development. You can turn off warnings selectively by using the **–woff** option.
–g[*num*]	Produces debugging information. The default is **–g0**: do not produce debugging information.
–G[*num*]	Specifies the maximum size, in bytes, of a data item that is accessed from the global pointer. The default is **–G8**.
–help	Lists the available compiler options.

Table 2-2 (continued) General Driver Options

Option	Purpose
–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*.
–KPIC	Generates position-independent code. This is the default and is required for programs linking with dynamic shared objects. Specify **–non_shared** if you don't want to generate PIC code.
–mips1	Generates code using the instruction set of the MIPS R2000/R3000 RISC architecture. This option implies **–32**.
–mips2	Generates code using the MIPS II instruction set (MIPS I + R4000 specific extensions). Note that code compiled with **–mips2** does not run on R2000/R3000-based machines. This option implies **–32**.
–mips3	Generates code using the full MIPS R4000 instruction set, including 64-bit code.
–mips4	Generates code using the MIPS R8000 instruction set.
–nocpp	Suppresses running of the macro preprocessor on the source files prior to processing.
–non_shared	Turns off the default option, **–KPIC**, to produce non-shared code. This code 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*. You should use this option 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 is named *filename* rather than the default name, *a.out*.

Table 2-2 (continued) General Driver Options

Option	Purpose
–O*num*	Specifies optimization options. **–O0** turns off optimizations. **–O1** turns on local optimizations that can be done quickly. **–O2** (**–O**) turns on global optimizations. This is the default. **–O3** turns on all optimizations. For more information, see Chapter 5, “Optimizing Program Performance.”
–OPT:*options*	Controls optimization options. For more information, see Chapter 5, “Optimizing Program Performance.”
–p	Sets up for profiling by periodically sampling the value of the program counter (only effects the loading).
–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.
–pch	Enables the precompiled header processing; *fec/fecc* searches for a qualifying PCH file to read in and/or creates one for use on a subsequent compilation.
–S	Similar to **–c**, except that it produces assembly code in a *.s* file instead of object code in a *.o* file. **–S** provides information about the code and comments about such things as software pipelining, the loops it works on, and the results.
–show	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.
–SWP:*options*	Controls software pipelining of inner loops. For more information, see Chapter 5, “Optimizing Program Performance.”
–TARG:*options*	Controls the target architecture and machine for which code is generated. For more information, see Chapter 5, “Optimizing Program Performance.”
–TENV:*options*	Controls the target environment assumed by the compiler. For more information, see Chapter 5, “Optimizing Program Performance.”

Table 2-2 (continued) General Driver Options

Option	Purpose
–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.
–woff *n*	Suppresses ANSI/ISO warning message number *n*. Suppress multiple warning numbers by using a comma-separated list **(–woff n1,n2**...), a range of warning numbers by using a hyphen-separated list (**–woff n1-n5**), or any combination thereof. You can suppress all warning messages via **–woff all.**
–xansi	Compilation follows an extended ANSI/ISO C language semantics, which is more lenient in terms of the forms of expressions it allows. Preprocessing combines predefined macros. This is the default C compilation mode.
–xgot	Compiles using a 32-bit offset in the Global Symbol Table (GOT). By default, addresses are loaded out of the GOT using a 16-bit offset from a context pointer. The default size of the GOT thus is limited to 64K bytes, or about 8K worth of symbols, 16K for 32-bit compiled objects. This can sometimes result in a "GOT overflow" message. Use of the **–xgot** option allows the GOT to contain up to 1G of entries; however, the resulting executable is somewhat larger and slower. **–xgot** is a synonym for **–TENV:large_GOT** (for information on this option, see "Controlling the Target Environment").

Note: To use 4.3 BSD extensions in C, compile using the **–xansi** or the **–D__EXTENSIONS__** option on the command line. For example:

```
cc prog.c -ansi -prototypes -fullwarn -lm -D__EXTENSIONS__
```

Linking

The linker, *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 linker names the executable file *a.out*.

This section summarizes the functions of the linker. It also covers how to link a program manually (without using a compiler driver) and how to compile multilanguage programs. Refer to the *ld*(1) reference page for complete information on the linker. Click the word ld to view this page.

Specifically, this section describes:

- "Invoking the Linker Manually"
- "Linking Assembly Language Programs"
- "Linking Libraries"
- "Linking to Previously Built Dynamic Shared Objects"
- "Linking Multilanguage Programs"

Invoking the Linker Manually

Usually the compiler invokes the linker as the final step in compilation (as explained in "Compiler Drivers"). If object files exist that were produced by previous compilations, and you want to link them, invoke the linker by using a compiler driver instead of calling **ld** directly. Just pass the object filenames to the compiler driver in place of source filenames. If the original source files are in a single language, simply invoke the associated driver and specify the list of object files. (For information about linking objects derived from several languages, see "Linking Multilanguage Programs.")

A few command-line options to *ld*, such as **–p**, 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 reference page for details).

Typically, the compiler driver invokes *ld* as necessary. Circumstances exist under which you may need to invoke *ld* directly, such as when you're building a shared object or doing special linking not supported by compiler drivers (such as building an embedded system). To build C++ shared objects, use the CC driver.

Linker Syntax

A summary of *ld* syntax follows.

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

options One or more of the options listed in Table 2-3.

object Specifies the name of the object file to be linked.

Table 2-3 contains only a partial list of linker options. Many options that apply only to creating shared objects are discussed in Chapter 3, "Using Dynamic Shared Objects." For complete information on options and libraries that affect linker processing, refer to the *ld*(1) reference page.

Table 2-3 Linker Options

Option	Purpose
–32	Links 32-bit programs and DSOs.
–n32	Links n32-bit programs and DSOs.
–64	Links 64-bit programs and DSOs.
–l*name*	Specifies the name of a library, where *lname* is the library name. The linker searches for *lname.so* (and then *lname.a*) first in any directories specified by **–L** *dirname* options, and then in the standard directories: */usr/lib*, */lib*, and */usr/local/lib*.
–L *dirname*	Adds *dirname* to the list of directories to be searched for (as well as libraries searched for) as specified by subsequent **–l***name* options.
–m	Produces a linker 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 linker does not find the library in *dirname* list, then it does not search any of the standard library directories.
–o *filename*	Specifies a name for your executable. If you do not specify *filename*, the linker names the executable *a.out*.
–s	Strips debugging information from the program object, reducing its size. This option is useful for linking routines that are frequently linked into other program objects, but may hamper debugging.
–v	Produces verbose linker output providing information about various linker passes.
–y*symname*	Reports all references to, and definitions of, the symbol *symname*. Useful for locating references to undefined symbols.

Linker Example

The following command tells the linker to search for the DSO *libcurses.so* in the directory */usr/lib*. If it does not find that DSO, the linker then looks for *libcurses.a* in */lib*.

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

If the linker doesn't find an appropriate library, it looks in */usr/local/lib* for *libcurses.a*. (Note that the linker does not look for DSOs in */usr/local/lib*, so don't put shared objects there.) If found in any of these places, the DSO or library is linked with the objects *foiled.o* and *again.o*; otherwise an error is generated.

Note: If the linker reports GOT overflow, GOT unreachable, or GP-related errors, see the *gp_overflow*(5) reference page, which describes the underlying causes of and possible solutions for overflowing the gp-relative area in the linker.

Linking Assembly Language Programs

The assembler driver (*as*) does not run the linker. To link a program written in assembly language, use one of these procedures:

- Assemble and link 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 the resulting object file with the *ld* command.

Linking Libraries

The linker 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. If you define the same symbol in more than one DSO, only the first definition is used.

When *ld* reads an archive, usually denoted by a filename 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. Therefore, 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 archive or DSO is ignored.

Specifying Libraries and DSOs

You can specify libraries and DSOs 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*, without a *.1* extension.

To use the linker's library search rules, specify the library with the **–lname** 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 */usr/lib*, then in */lib*.

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 **–lname** option. For example, consider the command:

```
cc file1.o –lm –L mydir
```

This command invokes, at the linking stage of compilation, the following:

```
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-style 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, built using *ar* from *.o* files compiled with **–KPIC** (the default option); they can be linked with other PIC files. DSOs are built from PIC *.o* files by using *ld –shared*; see Chapter 3 for details.

If the linker tells you that a reference to a certain function is unresolved, check that function's reference 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 an Unresolved Symbol With ld."

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). No automatic connection exists between header files and libraries; header files only give prototypes for library routines, not the library code itself.

Examples of Linking DSOs

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 Previously Built 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 3, "Using Dynamic Shared Objects."

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
```

This command 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 no *libthread.so* exists, but a PIC archive named *libthread.a* exists, that archive is used with *libc.so.1*, so you still get dynamic (run time) 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

The source language of the main program may differ from that of a subprogram. In this case, you can link multilanguage programs.

Follow the steps below to link multilanguage programs. (Refer to Figure 2-1 for an illustration of the process.)

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 compiler associated with the language of the main program to link the objects:

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

   The compiler drivers supply the default set of libraries necessary to produce an executable from the source of the associated language. However, when producing executables from source code in several languages, you may need to specify the default libraries explicitly for one or more of the languages used. For instructions on specifying libraries, see “Linking Libraries.”

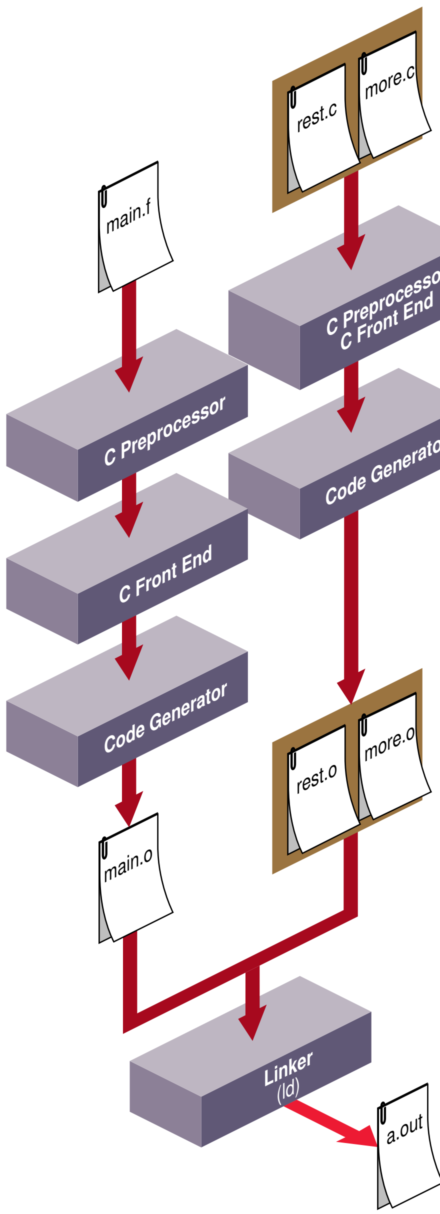

Figure 2-1 Compilation Control Flow for Multilanguage Programs

Note: Use caution when passing pointers and longs between languages as some languages use different type sizes and structures for data types.

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

Finding an Unresolved Symbol With *ld*

You can use *ld* to locate unresolved symbols. For example, suppose you're compiling a program, and *ld* tells you that you're using an unresolved symbol. However, you don't know where the unresolved symbol is referenced.

To find the unresolved symbol, enter:

`ld –y`*symbol file1... filen*

You can also enter:

`cc prog.o –Wl,–y`*symbol*

The output lists the source file that references *symbol*.

Debugging

The compiler system provides a debugging tool, *dbx* (described in detail 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 *dbx*, specify the **–g** driver option (see Table 2-4) to produce executables containing information that the debugger can use. Click the word dbx to view the dbx(1) reference page.

Table 2-4 Driver Options for Debugging

Option	Purpose
–g0	Produces a program object with a minimum of source-level debugging information. This is the default. Reduces the size of the program object but allows optimizations. Use this option with the **–O** option after you finish debugging.
–g, **–g2**	Produces additional debugging information for full symbolic debugging. This option overrides the optimization options (**–O***num*).
–g3	Produces additional debugging information for full symbolic debugging of fully optimized code. This option makes the debugger less accurate. You can use **–g3** with an optimization option (**–O***num*).

Getting Information About Object Files

The following tools provide information on object files:

- *dis* disassembles an object file into machine instructions.
- *dwarfdump* lists headers, tables, and other selected parts of a DWARF-format object file or archive file.
- *elfdump* lists the contents (including the symbol table and header information) of an ELF-format object file.
- *file* provides descriptive information on the properties of a file.
- *nm* lists symbol table information.
- *size* prints the size of each section of an object file (some such sections are named *text*, *data*, and *sbss*).
- *strip* removes symbol table and relocation bits.

Note that you can trace system call and scheduling activity by using the *par* command. For more information, see the *par*(1) reference page.

Disassembling Object Files with *dis*

The *dis* tool disassembles object files into machine instructions. You can disassemble an object, archive library, or executable file.

dis Syntax

The syntax for *dis* is:

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

options One or more of the options listed in Table 2-5.

filename Specifies the name of one or more files to disassemble.

dis Options

Table 2-5 lists *dis* options. For more information, see the *dis*(1) reference page. Click dis to view the page.

Table 2-5 *dis* Options

Option	Description
–b begin_addr	Starts disassembly at *begin_address*. You can specify the address as decimal, octal (with a leading **0**), or hexadecimal (with a leading **0x**).
–d section	Disassembles the named *section* as data, and prints the offset of the data from the beginning of the section.
–D *section*	Disassembles the named *section* as data, and prints the address of the data.
–e *end_address*	Stops disassembly at *end_address*. You can specify the address as decimal, octal (with a leading **0**), or hexadecimal (with a leading **0x**).
–F *function*	Disassembles the named *function* in each object file you specify on the command line.
–h	Substitutes the hardware register names for the software register names in the output.

Table 2-5 (continued) *dis* Options

Option	Description
–H	Removes the leading source line, and leaves the hex value and the instructions.
–i	Removes the leading source line and hexadecimal value of disassembly, and leaves only the instructions.
–I *directory*	Uses *directory* to help locate source code.
–l *string*	Disassembles the archive file specified by *string*.
–L	Looks up source labels for subsequent printing.
–o	Prints numbers in octal. The default is hexadecimal.
–s	Performs symbolic disassembly where possible. Prints (using C syntax) symbol names on the line following the instruction. Displays source code mixed with assembly code
–t *section*	Disassembles the named *section* as text.
–T	Specifies the trace flag for debugging the disassembler.
–V	Prints (on *stderr*) the version number of the disassembler being executed.
–w	Prints source code to the right of assembly code (produces wide output). Use this option with the **–s** option.
–x	Prints offsets in hexadecimal (the default).

Listing Parts of DWARF Object Files With *dwarfdump*

The *dwarfdump* tool provides debugging information from selected parts of DWARF symbolic information in an ELF object file. For more information on DWARF, see files in the *4Dgifts* directory.

dwarfdump Syntax

The syntax for *dwarfdump* is:

`dwarfdump` *options filename*

options	One or more of the options listed in Table 2-6.
filename	Specifies the name of the object file whose contents are to be dumped.

dwarfdump Options

Table 2-6 lists *dwarfdump* options. For more information, see the *dwarfdump*(1) reference page.

Table 2-6 *dwarfdump* Options

Option	Dumps
–a	All sections.
–b	The **.debug_abbrev** section.
–c	The .**debug_loc** section.
–d	Uses dense mode. Prints die information of the **.debug_info** section. Does not imply the **–i** option.
–e	Uses ellipsis mode. Uses the short names for DW_TAG_* and DW_ATTR_* in the output for the **.debug_info** section.
–f	The **.debug_frame** section.
–i	The **.debug_info** section.
–l	The **.debug_line** section.
–m	The **.debug_macinfo** section.
–o	The **.rel.debug_*** sections.
–p	The **.debug_pubnames** section.
–r	The **.debug_aranges** section.
–s	The **.debug_string** section.
–ta	The **.debug_static_funcs** and **.debug_static_vars** sections (same as **–tfv**).
–tf	The **.debug_static_funcs** section.
–tv	The **.debug_static_vars** section.

Table 2-6 (continued) *dwarfdump* Options

Option	Dumps
–u file	Sections to the named *file*.
–v	Prints detailed information (verbose mode).
–w	The **.debug_weaknames** section.
–y	The **.debug_types** section.

Listing Parts of ELF 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.

elfdump Syntax

The syntax for *dump* is:

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

options One or more of the options listed in Table 2-7.

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

elfump Options

Table 2-7 lists *elfdump* options. For more information, see the *elfdump*(1) reference page.

Table 2-7 *elfdump* Options

Option	Dumps
–a	Archive header of each member of the archive.
–A	Beginning address of a section.
–c	String table.
–C	Decoded C++ symbol names.

Table 2-7 (continued) *elfdump* Options

Option	Dumps
–cmt	The *.comment* sections.
–cnt	The *.content* sections.
–d	Range of sections.
–Dc	Conflict list in Dynamic Shared Objects.
–Dg	Global Offset Table in Dynamic Shared Objects.
–Dinfo	The *.MIPS.dclass* section.
–Dinst	The *.MIPS.inst* section.
–Dl	Library list in Dynamic Shared Objects.
–Dsym	The *.MIPS.sym* section.
–Dsymlib	Symbol library table (*.MIPS.symlib*).
–Dt	String table entries of the dynamic symbol table in Dynamic Shared Objects.
–e	Events sections.
–f	Each file header.
–F	Literal tables (*.lit4* and *.lit8* sections).
–g	Archive symbol table.
–G	The global pointer table information.
–h	All section headers in the file.
–hash	Hash table entries.
–i	The *.interp* section, which lists the path name of the program interpreter.
–info	Prints whether the object is marked quickstart, or is corded and is marked requickstart.
–l	The *.line* section.

Table 2-7 (continued) *elfdump* Options

Option	Dumps
–L	Dynamic linking information in Dynamic Shared Objects.
–n	The specified section (such as .MIPS.content, .dynamic, .got, MIPS.sym, .liblist, .conflict, .reginfo, and so forth).
–o	Each program execution header.
–op	Options section.
–p	Suppresses the printing of headings.
–r	Relocation information.
–R	Register information.
–reg	The *.reginfo* section.
–rpt	The run-time procedure table.
–s	Section contents.
–svr4	Information in SVR4-style format.
–t	Symbol table entries.
–T	Symbol table range.
–u	Updates a COFF file to an ELF file.
–V	Version information only.

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 often erroneously recognizes command files as C programs. For more information, see the *file*(1) reference page.

file Syntax

The syntax for *file* is:

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

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

file 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.

```
file test.o a.out /lib/libc.so.1
test.o:        ELF 64-bit MSB relocatable MIPS - version 1
a.out:         ELF 64-bit MSB executable MIPS - version 1
/lib/libc.so.1:   ELF 64-bit MSB dynamic lib MIPS - version 1
```

In this example, `MSB` indicates Most Significant Byte, also called Big-Endian; `relocatable` means the object contains relocation information that allows it to be linked with other objects to form an executable; `executable` means an executable file; and `dynamic lib` indicates a DSO.

Listing Symbol Table Information: *nm*

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

nm Syntax

The syntax for *nm* is:

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

options One or more of the options listed in Table 2-8.

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

nm Symbol Table Options

Table 2-8 lists *nm* symbol table options. For more information, see the *nm*(1) reference page.

Table 2-8 Symbol Table *nm* Options

Option	Purpose
–a	Prints debugging information.
–A	Prints the listing in System V format (default).
–b	Prints the value field in octal.
–B	Prints the listing in BSD format.
–C	Prints decoded C++ names.
–d	Prints the value field in decimal (the default for System V output).
–g	Prints globally visible names.
–h	Suppresses printing of headers.
–l	Produces a long listing.
–n	Sorts external symbols by name for System V format. Sorts all symbols by value for BSD format (by name is the BSD default output).
–o	Prints value field in octal (System V output). Prints the filename immediately before each symbol name (BSD output).
–p	Produces easily parsible, terse output similar to the BSD format.
–r	Prepends the name of the object file or archive to each output line.
–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 BSD format).
–V	Prints the version number of *nm*.
–x	Prints the value field in hexadecimal.

Table 2-9 defines the one-character codes shown in an *nm* listing. Refer to the example that follows the table for a sample listing.

Table 2-9 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 (unused external reference)
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

nm Example of Obtaining a Symbol Table Listing

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:

```
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 SVR4 format, use the *nm* command without any options:

```
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
```

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. The *elf*(4) reference page describes the format of these sections.

size Syntax

The syntax for *size* is:

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

options Specifies the format of the listing (see Table 2-10).

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

size Options

Table 2-10 lists *size* options. For more information, see the *size*(1) reference page.

Table 2-10 *size* Options

Option	Action
–A	Prints data section headers in System V format (default).
–B	Prints output in BSD-style format.
–d	Prints sizes in decimal (default).
–f	Prints data on allocatable sections including the size, permission flags, and the total of the loadable sizes.
–F	Prints data on loadable segments including the name and the total of the section sizes.
–n	Prints nonloadable and nonallocatable section sizes.
–o	Prints sizes in octal.
–svr4	Prints output in SVR4-style format.
–V	Prints the version of *size* that you are using.
–x	Prints sizes in hexadecimal.

size Example

An example of the *size* command and the listings produced follows.

size a.out

Section	Size	Physical Address	Virtual Address
.interp	21	268435856	268435856
.MIPS.options	104	268435880	268435880
.dynamic	464	268435984	268435984
.liblist	20	268436448	268436448
.MIPS.symlib	30	268436468	268436468
.msym	240	268436500	268436500
.dynstr	312	268436744	268436744
.dynsym	720	268437056	268437056
.hash	256	268437776	268437776
.MIPS.stubs	56	268438032	268438032
.text	460	268438088	268438088
.init	24	268438548	268438548
.data	17	268505088	268505088
.sdata	8	268505108	268505108
.got	112	268505120	268505120
.bss	36	268505232	268505232

Removing Symbol Table and Relocation Bits with *strip*

The *strip* tool removes symbol table and relocation bits that are attached to the assembler and loader. Use *strip* to save space after you debug a program. The effect of *strip* is the same as that of using the **-s** option to *ld*.

strip Syntax

The syntax for *strip* is:

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

options One or more of the options listed in Table 2-11.

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

For more information, see the *strip*(1) reference page.

Table 2-11 *strip* Options

Option	Description
–l	Strips line number information, and keeps the symbol table and debugging information.
–o *filename*	Puts the stripped information in the *filename* that you specify.
–V	Prints the version number of *strip*.
–x	Keeps symbol table information, but may strip debugging and line number information.

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 linker (*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
- Creating a symbol table for the linker to search symbols

The following section explains the syntax of the *ar* command and lists some examples of how to use it. See the *ar*(1) reference 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 3.

ar **Syntax**

The syntax for *ar* is:

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

options Specifies the action that the archiver is to take. Table 2-12 and Table 2-13 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 "ar 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.

ar Options

When running the archiver, specify exactly one of the options **d, m**, **p**, **q**, **r**, **t**, or **x** (listed in Table 2-12). In addition, you can optionally specify any of the modifiers in Table 2-13.

Table 2-12 Archiver Options

Option	Purpose
d	Deletes the specified objects from the archive.
f	Adds padding to the end of each object file archived, using the character **\n**. This enables the loader to have faster access to members in the archive while performing static linking. **Warning**: This option results in a permanent change in the size of object files.
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 overwrites it. 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, x copies all the library objects into the current directory.

Table 2-13 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.
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 **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.

ar Examples

To create a new library, *libtest.a*, and add object files to it, enter:

```
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.

To replace an object file in an existing library, enter:

```
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 appears twice in the archive.