Linker and Libraries Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817–1984–10
January 2005

Adobe PostScript

040830@9495

Contents

Tables

Figures

Preface

In the Solaris™ Operating System (Solaris OS), application developers can create applications and libraries by using the link-editor `ld`(1), and execute these objects with the aid of the runtime linker `ld.so.1`(1). This manual is for engineers who want to understand more fully the concepts involved in using the Solaris linkers.

About This Manual

This manual describes the operations of the Solaris link-editor and runtime linker. Special emphasis is placed on the generation and use of dynamic executables and shared objects because of their importance in a dynamic runtime environment.

Intended Audience

This manual is intended for a range of programmers who are interested in the Solaris linkers, from the curious beginner to the advanced user.

- Beginners learn the principle operations of the link-editor and runtime linker.
- Intermediate programmers learn to create, and use, efficient custom libraries.
- Advanced programmers, such as language-tools developers, learn how to interpret and generate object files.

Most programmers should not need to read this manual from cover to cover.

Organization

Chapter 1 provides an overview of the linking processes under the Solaris OS. This chapter is intended for all programmers.

Chapter 2 describes the functions of the link-editor. This chapter is intended for all programmers.

Chapter 3 describes the execution environment and program-controlled runtime binding of code and data. This chapter is intended for all programmers.

Chapter 4 provides definitions of shared objects, describes their mechanisms, and explains how to create and use them. This chapter is intended for all programmers.

Chapter 5 describes how to manage the evolution of an interface provided by a dynamic object. This chapter is intended for all programmers.

Chapter 6 describes interfaces for monitoring, and in some cases modifying, link-editor and runtime linker processing. This chapter is intended for advanced programmers.

Chapter 7 is a reference chapter on ELF files. This chapter is intended for advanced programmers.

Chapter 8 describes Thread-Local Storage. This chapter is intended for advanced programmers.

Chapter 9 describes the `mapfile` directives to the link-editor, which specify the layout of the output file. This chapter is intended for advanced programmers.

Appendix A provides an overview of the most commonly used link-editor options, and is intended for all programmers.

Appendix B provides naming conventions and guidelines for versioning shared objects, and is intended for all programmers.

Appendix C provides examples of how to use reserved dynamic string tokens to define dynamic dependencies, and is intended for all programmers.

Appendix D provides an overview of new features and updates that have been added to the link-editors and indicates to which release they were added.

Throughout this document, all command-line examples use `sh`(1) syntax. All programming examples are written in the C language.

Note – This Solaris release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems appear in the *Solaris 10 Hardware Compatibility List* at `http://www.sun.com/bigadmin/hcl`. This document cites any implementation differences between the platform types.

In this document the term “x86” refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the *Solaris 10 Hardware Compatibility List*.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.

TABLE P–1 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. Do *not* save the file. (Emphasis sometimes appears in bold online.)

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction to the Solaris Linkers

This manual describes the operations of the Solaris link-editor and runtime linker, together with the objects on which the link-editors operate. The basic operation of the Solaris linkers involves the combination of objects. This combination results in the symbolic references from one object being connected to the symbolic definitions within another object.

This manual expands the following areas:

Link-Editor
: The link-editor, `ld`(1), concatenates and interprets data from one or more input files. These files can be relocatable objects, shared objects, or archive libraries. From these input files, one output file is created. This file is either a relocatable object, an executable application, or a shared object. The link-editor is most commonly invoked as part of the compilation environment.

Runtime Linker
: The runtime linker, `ld.so.1`(1), processes dynamic executables and shared objects at runtime, binding the executable and shared objects together to create a runnable process.

Shared Objects
: Shared objects are one form of output from the link-edit phase. Shared objects are sometimes referred to as *Shared Libraries*. Shared objects are importance in creating a powerful, flexible runtime environment.

Object Files
: The Solaris linkers work with files that conform to the executable and linking format, otherwise referred to as ELF.

These areas, although separable into individual topics, have a great deal of overlap. While explaining each area, this document brings together the connecting principles.

Link-Editing

Link-editing takes a variety of input files, typically generated from compilers, assemblers, or ld(1). The link-editor concatenates and interprets the data within these input files to form a single output file. Although the link-editor provides numerous options, the output file that is produced is one of four basic types:

- *Relocatable object* – A concatenation of input relocatable objects that can be used in subsequent link-edit phases.
- *Static executable* – A concatenation of input relocatable objects that have all symbolic references resolved. This executable represents a ready-to-run process. See "Static Executables" on page 21.
- *Dynamic executable* – A concatenation of input relocatable objects that requires intervention by the runtime linker to produce a runnable process. A dynamic executable might still need symbolic references bound at runtime. Dynamic executables typically have one or more dependencies in the form of shared objects.
- *Shared object* – A concatenation of input relocatable objects that provide services that might be bound to a dynamic executable at runtime. The shared object can have dependencies on other shared objects.

These output files, and the key link-editor options used in their creation, are shown in Figure 1–1.

Dynamic executables and *shared objects* are often referred to jointly as *dynamic objects*. Dynamic objects are the main focus of this document.

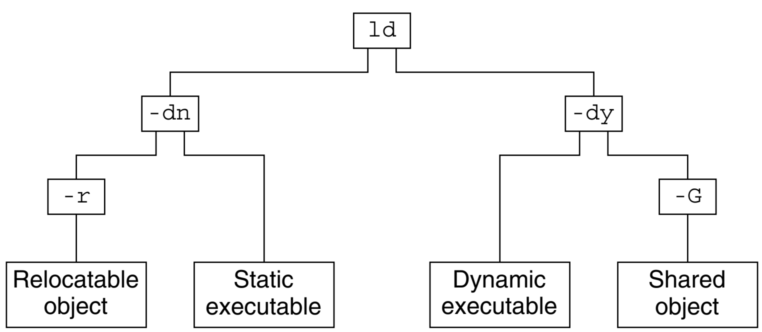


FIGURE 1–1 Static or Dynamic Link-Editing

Static Executables

The creation of static executables has been discouraged for many releases. In fact, 64–bit system archive libraries have never been provided. Because a static executable is built against system archive libraries, the executable contains system implementation details. This self-containment has a number of drawbacks.

- The executable is immune to the benefits of system patches delivered as shared objects. The executable therefore, must be rebuilt to take advantage of many system improvements.
- The ability of the executable to run on future releases can be compromised.
- The duplication of system implementation details negatively affects system performance.

With the Solaris 10 release, 32–bit system archive libraries are no longer provided. Without these libraries, specifically `libc.a`, the creation of static executables is no longer achievable without specialized system knowledge. Note, that the link-editors capability to process static linking options, and the processing of archive libraries, remains unchanged.

Runtime Linking

Runtime linking involves the binding of objects, usually generated from one or more previous link-edits, to generate a runnable process. During the generation of these objects by the link-editor, appropriate bookkeeping information is produced to represent the verified binding requirements. This information enables the runtime linker to load, relocate, and complete the binding process.

During process execution, the facilities of the runtime linker are made available. These facilities can be used to extend the process' address space by adding additional shared objects on demand. The two most common components involved in runtime linking are *dynamic executables* and *shared objects*.

Dynamic executables are applications that are executed under the control of a runtime linker. These applications usually have dependencies in the form of shared objects, which are located, and bound by the runtime linker to create a runnable process. Dynamic executables are the default output file generated by the link-editor.

Shared objects provide the key building-block to a dynamically linked system. A shared object is similar to a dynamic executable, however, shared objects have not yet been assigned a virtual address.

Dynamic executables usually have dependencies on one or more shared objects. Typically, one or more shared objects must be bound to the dynamic executable to produce a runnable process. Because shared objects can be used by many applications, aspects of their construction directly affect shareability, versioning, and performance.

Shared object processing by the link-editor or the runtime linker can be distinguished by the *environment* in which the shared object is used:

compilation environment
: Shared objects are processed by the link-editor to generate dynamic executables or other shared objects. The shared objects become dependencies of the output file being generated.

runtime environment
: Shared objects are processed by the runtime linker, together with a dynamic executable, to produce a runnable process.

Related Topics

Dynamic Linking

Dynamic linking is a term often used to embrace a number of linking concepts. Dynamic linking refers to those portions of the link-editing process that generate dynamic executables and shared objects. Dynamic linking also refers to the runtime linking of these objects to generate a runnable process. Dynamic linking enables multiple applications to use the code provided by a shared object by binding the application to the shared object at runtime.

By separating an application from the services of standard libraries, dynamic linking also increases the portability and extensibility of an application. This separation between the interface of a service and its implementation enables the system to evolve while maintaining application stability. Dynamic linking is a crucial factor in providing an *application binary interface* (ABI), and is the preferred compilation method for Solaris applications.

Application Binary Interfaces

Binary interfaces between system and application components are defined to enable the asynchronous evolution of these facilities. The Solaris linkers operate upon these interfaces to assemble applications for execution. Although all components handled by the Solaris linkers have binary interfaces, the whole set of interfaces provided by the system is referred to as the *Solaris ABI*.

The Solaris ABI is a technological descendent for work on ABIs that started with the *System V Application Binary Interface*. This work evolved with additions performed by SPARC International, Inc.® for SPARC processors, called the *SPARC Compliance Definition* (SCD).

32–Bit and 64–Bit Environments

The link-editors operate on 32–bit objects. On SPARCV9 systems, the link-editors are also capable of operating on 64–bit objects. On SPARC systems, the 64–bit link-editor is capable of generating 32–bit objects. In addition, the 32–bit link-editor is capable of generating 64–bit objects. In the latter case, the size of the generated object, not including the `.bss`, is restricted to 2 Gbytes.

No command-line option is required to distinguish a 32–bit link-edit or 64–bit link-edit. The link-editor uses the ELF class of the first relocatable object on the command-line to govern the mode in which to operate. Specialized link-edits, such as linking solely from a `mapfile` or an archive library, are uninfluenced by the command-line object. These link-edits default to a 32–bit mode. In these cases, a 64–bit link-edit can be enforced with the `-64` option. The mixing of 32–bit and 64–bit objects is not permitted.

The operations of the link-editors on 32–bit objects and 64–bit objects are identical. This document typically uses 32–bit examples. Cases where 64–bit processing differs from the 32–bit processing are highlighted.

For more information on 64–bit applications, refer to the *Solaris 64-bit Developer's Guide*.

Environment Variables

The link-editors support a number of environment variables that begin with the characters `LD_`, for example `LD_LIBRARY_PATH`. Each environment variable can exist in its generic form, or can be specified with a `_32` or `_64` suffix, for example `LD_LIBRARY_PATH_64`. This suffix makes the environment variable specific, respectively, to 32-bit or 64–bit processes. This suffix also overrides any generic, non-suffixed, version of the environment variable that might be in effect.

Note – Prior to the Solaris 10 release, the link-editors ignored environment variables that were specified without a value. Therefore, in the following example, the generic environment variable setting, `/opt/lib`, would have been used to search for the dependencies of the 32–bit application `prog`.

```
% LD_LIBRARY_PATH=/opt/lib  LD_LIBRARY_PATH_32=  prog
```

With the Solaris 10 release, environment variables specified without a value, that have a `_32` or `_64` suffix, are processed. These environment variables effectively cancel any associated generic environment variable setting. Thus in the previous example, `/opt/lib` will not be used to search for the dependencies of the 32–bit application `prog`.

Throughout this document, any reference to link-editor environment variables uses the generic, non-suffixed, variant. All supported environment variables are defined in `ld`(1) and `ld.so.1`(1).

Support Tools

The Solaris operating environment also provides several support tools and libraries. These tools provide for the analysis and inspection of these objects and the linking processes. These tools include `elfdump`(1), `lari`(1), `nm`(1), `dump`(1), `ldd`(1), `pvs`(1), `elf`(3ELF), and a linker debugging support library. Throughout this document, many discussions are augmented with examples of these tools.

CHAPTER 2

Link-Editor

The link-editing process creates an output file from one or more input files. Output file creation is directed by the options that are supplied to the link-editor and the input sections provided by the input files.

All files are represented in the *executable and linking format* (ELF). For a complete description of the ELF format see Chapter 7. For this introduction, two ELF structures are introduced, *sections* and *segments*.

Sections are the smallest indivisible units that can be processed within an ELF file. Segments are a collection of sections that represent the smallest individual units that can be mapped to a memory image by `exec`(2) or by the runtime linker `ld.so.1`(1).

Although many types of ELF section exist, sections all fall into two categories with respect to the link-editing phase:

- Sections that contain *program data*, whose interpretation is meaningful only to the application, such as the program instructions `.text` and the associated data `.data` and `.bss`.
- Sections that contain *link-editing information*, such as the symbol table information found from `.symtab` and `.strtab`, and relocation information such as `.rela.text`.

Basically, the link-editor concatenates the *program data* sections into the output file. The *link-editing information* sections are interpreted by the link-editor to modify other sections. The information sections are also used to generate new output information sections used in later processing of the output file.

The following simple breakdown of link-editor functionality introduces the topics that are covered in this chapter:

- The verification and consistency checking of all options provided.
- The concatenation of sections of the same characteristics from the input relocatable objects to form new sections within the output file. The concatenated sections can in turn be associated to output segments.

- The processing of symbol table information from both relocatable objects and shared objects to verify and unite references with definitions. The generation of a new symbol table, or tables, within the output file.
- The processing of relocation information from the input relocatable objects, and the application of this information to the output file by updating other input sections. In addition, output relocation sections might be generated for use by the runtime linker.
- The generation of *program headers* that describe all the segments that are created.
- The generation of dynamic linking information sections if necessary, which provide information such as shared object dependencies and symbol bindings to the runtime linker.

The process of concatenating like *sections* and associating *sections* to *segments* is carried out using default information within the link-editor. The default *section* and *segment* handling provided by the link-editor is usually sufficient for most link-edits. However, these defaults can be manipulated using the `-M` option with an associated `mapfile`. See Chapter 9.

Invoking the Link-Editor

You can either run the link-editor directly from the command line or have a compiler driver invoke the link-editor for you. In the following two sections the description of both methods are expanded. However, using the compiler driver is the preferred choice. The compilation environment is often the consequence of a complex and occasionally changing series of operations known only to compiler drivers.

Direct Invocation

When you invoke the link-editor directly, you have to supply every object file and library required to create the intended output. The link-editor makes no assumptions about the object modules or libraries that you meant to use in creating the output. For example, when you issue the command:

```
$ ld test.o
```

The link-editor creates a dynamic executable that is named `a.out` using only the input file `test.o`. For the `a.out` to be a useful executable, code for startup and exit processing should be included. This code can be language or operating system specific, and is usually provided through files supplied by the compiler drivers.

Additionally, you can also supply your own initialization code and termination code. This code must be encapsulated and be labeled correctly for the code to be correctly recognized and made available to the runtime linker. This encapsulation and labeling can also be provided through files supplied by the compiler drivers.

When creating runtime objects such as executables and shared objects, you should use a compiler driver to invoke the link-editor. Direct invocation of the link-editor is recommended only when creating intermediate relocatable objects when using the `-r` option.

Using a Compiler Driver

The conventional way to use the link-editor is through a language-specific compiler driver. You supply the compiler driver, `cc`(1), `CC`(1), and so forth, with the input files that make up your application. The compiler driver adds additional files and default libraries to complete the link-edit. These additional files can be seen by expanding the compilation invocation, for example:

```
$ cc -# -o prog main.o
/usr/ccs/bin/ld -dy /opt/COMPILER/crti.o /opt/COMPILER/crt1.o \
/usr/ccs/lib/values-Xt.o -o prog main.o \
-YP,/opt/COMPILER/lib:/usr/ccs/lib:/usr/lib -Qy -lc \
/opt/COMPILER/crtn.o
```

Note – The actual files included by your compiler driver and the mechanism used to display the link-editor invocation might differ.

Specifying the Link-Editor Options

Most options to the link-editor can be passed through the compiler driver command line. For the most part, the compiler and the link-editor options do not conflict. Where a conflict arises, the compiler drivers usually provide a command-line syntax that you can use to pass specific options to the link-editor. You can also provide options to the link-editor by setting the `LD_OPTIONS` environment variable.

```
$ LD_OPTIONS="-R /home/me/libs -L /home/me/libs" cc -o prog main.c -lfoo
```

The `-R` and `-L` options are interpreted by the link-editor. These options precede any command-line options that are received from the compiler driver.

The link-editor parses the entire option list for any invalid options or any options with invalid associated arguments. When either of these cases are found, a suitable error message is generated. If the error is deemed fatal, the link-edit terminates. In the following example, the illegal option `-X`, and the illegal argument to the `-z` option, are caught by the link-editor's checking.

```
$ ld -X -z sillydefs main.o
ld: illegal option -- X
ld: fatal: option -z has illegal argument 'sillydefs'
```

If an option that requires an associated argument is specified twice, the link-editor produces a suitable warning and continue with the link-edit.

```
$ ld -e foo ...... -e bar main.o
ld: warning: option -e appears more than once, first setting taken
```

The link-editor also checks the option list for any fatal inconsistencies.

```
$ ld -dy -a main.o
ld: fatal: option -dy and -a are incompatible
```

After processing all options, if no fatal error conditions have been detected, the link-editor proceeds to process the input files.

See Appendix A for the most commonly used link-editor options, and `ld`(1) for a complete description of all link-editor options.

Input File Processing

The link-editor reads input files in the order in which the files appear on the command line. Each file is opened and inspected to determine the files ELF type, and therefore determine how the file must be processed. The file types that apply as input for the link-edit are determined by the binding mode of the link-edit, either *static* or *dynamic*.

Under *static* mode, the link-editor accepts only relocatable objects or archive libraries as input files. Under *dynamic* mode, the link-editor also accepts shared objects.

Relocatable objects represent the most basic input file type to the link-editing process. The *program data* sections within these files are concatenated into the output file image being generated. The *link-edit information* sections are organized for later use. These sections do not become part of the output file image, as new sections are generated to take their places. Symbols are gathered into an internal symbol table for verification and resolution. This table is then used to create one or more symbol tables in the output image.

Although input files can be specified directly on the link-edit command-line, archive libraries and shared objects are commonly specified using the `-l` option. See "Linking With Additional Libraries" on page 31. During a link-edit, the interpretation of archive libraries and shared objects are quite different. The next two sections expand upon these differences.

Archive Processing

Archives are built using `ar`(1). Archives usually consist of a collection of relocatable objects with an archive symbol table. This symbol table provides an association of symbol definitions with the objects that supply these definitions. By default, the link-editor provides *selective* extraction of archive members. The link-editor uses unresolved symbolic references to select objects from the archive that are required to complete the binding process. You can also explicitly extract all members of an archive.

The link-editor extracts a relocatable object from an archive if:

- The archive member contains a symbol definition that satisfies a symbol reference, presently held in the link-editor's internal symbol table. This reference is sometimes referred to as an *undefined* symbol.
- The archive member contains a data symbol definition that satisfies a tentative symbol definition presently held in the link-editor's internal symbol table. An example is a `FORTRAN COMMON` block definition, which causes the extraction of a relocatable object that defines the same `DATA` symbol.
- The link-editors `-z allextract` is in effect. This option suspends selective archive extraction and causes all archive members to be extracted from the archive being processed.

Under selective archive extraction, a weak symbol reference does not extract an object from an archive unless the `-z weakextract` option is in effect. See "Simple Resolutions" on page 39 for more information.

Note – The options `-z weakextract`, `-z allextract`, and `-z defaultextract` enable you to toggle the archive extraction mechanism among multiple archives.

With selective archive extraction, the link-editor makes multiple passes through an archive. Relocatable objects are extracted as needed to satisfy the symbol information being accumulated in the link-editor internal symbol table. After the link-editor has made a complete pass through the archive without extracting any relocatable objects, the next input file is processed.

By extracting from the archive only the relocatable objects needed when the archive is encountered, the position of the archive on the command line can be significant. See "Position of an Archive on the Command Line" on page 32.

Note – Although the link-editor makes multiple passes through an archive to resolve symbols, this mechanism can be quite costly. Especially, for large archives that contain random organizations of relocatable objects. In these cases, you should use tools like `lorder`(1) and `tsort`(1) to order the relocatable objects within the archive. This ordering reduces the number of passes the link-editor must carry out.

Shared Object Processing

Shared objects are indivisible whole units that have been generated by a previous link-edit of one or more input files. When the link-editor processes a shared object, the entire contents of the shared object become a logical part of the resulting output file image. This logical inclusion means that all symbol entries defined in the shared object are made available to the link-editing process. The shared object is actually copied during process execution.

The shared object's program data sections and most of the link-editing information sections are unused by the link-editor. These sections are interpreted by the runtime linker when the shared object is bound to generate a runnable process. However, the occurrence of a shared object is remembered. Information is stored in the output file image to indicate that this object is a dependency that must be made available at runtime.

By default, all shared objects specified as part of a link-edit are recorded as dependencies in the object being built. This recording is made regardless of whether the object being built actually references symbols offered by the shared object. To minimize the overhead of runtime linking, only specify those dependencies that resolve symbol references from the object being built. The link-editor's debugging capabilities, and `ldd`(1) with the `-u` option, can be used to determine unused dependencies. Alternatively, the link-editor's `-z ignore` option can suppress the dependency recording of unused shared objects.

If a shared object has dependencies on other shared objects, these dependencies are also processed. This processing occurs after all command-line input files have been processed, to complete the symbol resolution process. However, the shared object names are not recorded as dependencies in the output file image being generated.

Although the position of a shared object on the command-line has less significance than archive processing, the position can have a global effect. Multiple symbols of the same name are allowed to occur between relocatable objects and shared objects, and between multiple shared objects. See "Symbol Resolution" on page 38.

The order of shared objects processed by the link-editor is maintained in the dependency information that is stored in the output file image. The runtime linker reads this information, and loads the specified shared objects in the same order. Therefore, the link-editor and the runtime linker select the first occurrence of a symbol of a multiply-defined series of symbols.

Note – Multiple symbol definitions, are reported in the load map output generated using the `-m` option.

Linking With Additional Libraries

Although the compiler drivers often ensure that appropriate libraries are specified to the link-editor, frequently you must supply your own. Shared objects and archives can be specified by explicitly naming the input files required to the link-editor. However, a more common and more flexible method involves using the link-editor's `-l` option.

Library Naming Conventions

By convention, shared objects are usually designated by the prefix `lib` and the suffix `.so`. Archives are designated by the prefix `lib` and the suffix `.a`. For example, `libc.so` is the shared object version of the standard C library that is made available to the compilation environment. `libc.a` is the library's archive version.

These conventions are recognized by the `-l` option of the link-editor. This option is commonly used to supply additional libraries to a link-edit. The following example directs the link-editor to search for `libfoo.so`. If the link-editor does not find `libfoo.so`, a search for `libfoo.a` is made before moving on to the next directory to be searched.

```
$ cc -o prog file1.c file2.c -lfoo
```

Note – A naming convention exists regarding the compilation environment and the runtime environment use of shared objects. The compilation environment uses the simple `.so` suffix, whereas the runtime environment commonly uses the suffix with an additional version number. See "Naming Conventions" on page 104 and "Coordination of Versioned Filenames" on page 147.

When link-editing in dynamic mode, you can choose to link with a mix of shared objects and archives. When link-editing in static mode, only archive libraries are acceptable for input.

In dynamic mode, when using the `-l` option, the link-editor first searches the given directory for a shared object that matches the specified name. If no match is found, the link-editor looks for an archive library in the same directory. In static mode, when using the `-l` option, only archive libraries are sought.

Linking With a Mix of Shared Objects and Archives

The library search mechanism in dynamic mode searches a given directory for a shared object, and then searches for an archive library. Finer control of the search is possible through the `-B` option.

By specifying the `-B dynamic` and `-B static` options on the command line, you can toggle the library search between shared objects or archives respectively. For example, to link an application with the archive `libfoo.a` and the shared object `libbar.so`, issue the following command:

```
$ cc -o prog main.o file1.c -Bstatic -lfoo -Bdynamic -lbar
```

The `-B static` and `-B dynamic` keywords are not exactly symmetrical. When you specify `-B static`, the link-editor does not accept shared objects as input until the next occurrence of `-B dynamic`. However, when you specify `-B dynamic`, the link-editor first looks for shared objects and then archive library's in any given directory.

The precise description of the previous example is that the link-editor first searches for `libfoo.a`, and then for `libbar.so`, and if that search fails, for `libbar.a`. Finally, the link-editor searches for `libc.so`, and if that search fails, `libc.a`.

Position of an Archive on the Command Line

The position of an archive on the command line can affect the output file being produced. The link-editor searches an archive only to resolve undefined or tentative external references that have previously been encountered. After this search is completed and any required members have been extracted, the link-editor moves onto the next input file on the command line.

Therefore by default, the archive is not available to resolve any new references from the input files that follow the archive on the command line. For example, the following command directs the link-editor to search `libfoo.a` only to resolve symbol references that have been obtained from `file1.c`. The `libfoo.a` archive is not available to resolve symbol references from `file2.c` or `file3.c`.

```
$ cc -o prog file1.c -Bstatic -lfoo file2.c file3.c -Bdynamic
```

Note – You should specify any archives at the end of the command line unless multiple-definition conflicts require you to do otherwise.

Interdependencies between archives can exist, such that the extraction of members from one archive must be resolved by extracting members from another archive. If these dependencies are cyclic, the archives must be specified repeatedly on the command line to satisfy previous references. For example:

```
$ cc -o prog .... -lA -lB -lC -lA -lB -lC -lA
```

The determination, and maintenance, of repeated archive specifications can be tedious. The `-z rescan` option makes this process simpler. Following all input file processing, this option causes the entire archive list to be reprocessed. This processing attempts to locate additional archive members that resolve symbol references. This archive rescanning continues until a pass over the archive list occurs in which no new members are extracted. The previous example could therefore be simplified to:

```
$ cc -o prog -z rescan .... -lA -lB -lC
```

Directories Searched by the Link-Editor

All previous examples assume the link-editor knows where to search for the libraries listed on the command line. By default, when linking 32–bit objects, the link-editor knows of only three standard directories in which to look for libraries, `/usr/ccs/lib`, followed by `/lib`, and finally `/usr/lib`. When linking 64–bit objects, only two standard directories are used, `/lib/64` followed by `/usr/lib/64`. All other directories to be searched must be added to the link-editor's search path explicitly.

You can change the link-editor search path by using a command-line option, or by using an environment variable.

Using a Command-Line Option

You can use the `-L` option to add a new path name to the library search path. This option alters the search path at the point the option is encountered on the command line. For example, the following command searches `path1`, followed by `/usr/ccs/lib`, `/lib`, and finally `/usr/lib`, to find `libfoo`. The command searches `path1` and then `path2`, followed by `/usr/ccs/lib`, `/lib`, and `/usr/lib`, to find `libbar`.

```
$ cc -o prog main.o -Lpath1 file1.c -lfoo file2.c -Lpath2 -lbar
```

Path names that are defined by using the `-L` option are used only by the link-editor. These path names are not recorded in the output file image being created. Therefore, these path names are not available for use by the runtime linker.

Note – You must specify `-L` if you want the link-editor to search for libraries in your current directory. You can use a period (.) to represent the current directory.

You can use the `-Y` option to change the default directories searched by the link-editor. The argument supplied with this option takes the form of a colon separated list of directories. For example, the following command searches for `libfoo` only in the directories `/opt/COMPILER/lib` and `/home/me/lib`.

```
$ cc -o prog main.c -YP,/opt/COMPILER/lib:/home/me/lib -lfoo
```

The directories that are specified by using the `-Y` option can be supplemented by using the `-L` option.

Using an Environment Variable

You can also use the environment variable `LD_LIBRARY_PATH`, which takes a colon-separated list of directories, to add to the link-editor's library search path. In its most general form, `LD_LIBRARY_PATH` takes two directory lists separated by a semicolon. These lists are searched before and after the lists supplied on the command line.

The following example shows the combined effect of setting LD_LIBRARY_PATH and calling the link-editor with several -L occurrences:

```
$ LD_LIBRARY_PATH=dir1:dir2;dir3
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

The effective search path is dir1:dir2:path1:path2... pathn:dir3:/usr/ccs/lib:/lib:/usr/lib.

If no semicolon is specified as part of the LD_LIBRARY_PATH definition, the specified directory list is interpreted *after* any -L options. In the following example, the effective search path is path1:path2... pathn:dir1:dir2:/usr/ccs/lib:/lib:/usr/lib.

```
$ LD_LIBRARY_PATH=dir1:dir2
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

Note – This environment variable can also be used to augment the search path of the runtime linker. See "Directories Searched by the Runtime Linker" on page 66. To prevent this environment variable from influencing the link-editor, use the -i option.

Directories Searched by the Runtime Linker

The runtime linker looks in two default locations for dependencies. When processing 32–bit objects, the default locations are /lib and /usr/lib. When processing 64–bit objects, the default locations are /lib/64 and /usr/lib/64. All other directories to be searched must be added to the runtime linker's search path explicitly.

When a dynamic executable or shared object is linked with additional shared objects, the shared objects are recorded as dependencies. These dependencies must be located during process execution by the runtime linker. When linking a dynamic object, one or more search paths can be recorded in the output file. These search paths are referred to as a *runpath*. The runtime linker uses the runpath of an object to locate the dependencies of that object.

Specialized objects can be built with the -z nodefaultlib option to suppress any search of the default location at runtime. Use of this option implies that all the dependencies of an object can be located using its runpaths. Without this option, no matter how you augment the runtime linker's search path, its last element is always the default location.

Note – The default search path can be administrated by using a runtime configuration file. See "Configuring the Default Search Paths" on page 69. However, the creator of an object should not rely on the existence of this file. You should always ensure that an object can locate its dependencies with only its runpaths or the default location.

You can use the `-R` option, which takes a colon-separated list of directories, to record a runpath in a dynamic executable or shared object. The following example records the runpath `/home/me/lib:/home/you/lib` in the dynamic executable `prog`.

```
$ cc -o prog main.c -R/home/me/lib:/home/you/lib -Lpath1 \
-Lpath2 file1.c file2.c -lfoo -lbar
```

The runtime linker uses these paths, followed by the default location, to obtain any shared object dependencies. In this case, this runpath is used to locate `libfoo.so.1` and `libbar.so.1`.

The link-editor accepts multiple `-R` options. These multiple specifications are concatenate together, separated by a colon. Thus, the previous example can also be expressed as follows.

```
$ cc -o prog main.c -R/home/me/lib -Lpath1 -R/home/you/lib \
-Lpath2 file1.c file2.c -lfoo -lbar
```

For objects that can be installed in various locations, the `$ORIGIN` dynamic string token provides a flexible means of recording a runpath. See "Locating Associated Dependencies" on page 329.

Note – A historic alternative to specifying the `-R` option is to set the environment variable `LD_RUN_PATH`, and make this available to the link-editor. The scope and function of `LD_RUN_PATH` and `-R` are identical, but when both are specified, `-R` supersedes `LD_RUN_PATH`.

Initialization and Termination Sections

Dynamic objects can supply code that provides for runtime initialization and termination processing. This code can be encapsulated in one of two section types, either an array of function pointers or a single code block. Each of these section types is built from a concatenation of like sections from the input relocatable objects.

The sections `.preinit_array`, `.init_array` and `.fini_array` provide arrays of runtime pre-initialization, initialization, and termination functions, respectively. When creating a dynamic object, the link-editor identifies these arrays with the `.dynamic` tag pairs `DT_PREINIT_[ARRAY/ARRAYSZ]`, `DT_INIT_[ARRAY/ARRAYSZ]`, and `DT_FINI_[ARRAY/ARRAYSZ]` accordingly. These tags identify the associated sections so they can be called by the runtime linker. A pre-initialization array is applicable to dynamic executables only.

The sections .init and .fini provide a runtime initialization and termination code block, respectively. However, the compiler drivers typically supply .init and .fini sections with files they add to the beginning and end of your input file list. These files have the effect of encapsulating the .init and .fini code into individual functions. These functions are identified by the reserved symbol names _init and _fini respectively. When creating a dynamic object, the link-editor identifies these symbols with the .dynamic tags DT_INIT and DT_FINI accordingly. These tags identify the associated sections so they can be called by the runtime linker.

For more information about the execution of initialization and termination code at runtime see "Initialization and Termination Routines" on page 79.

The registration of initialization and termination functions can be carried out directly by the link-editor by using the -z initarray and -z finiarray options. For example, the following command places the address of foo() in an .initarray element, and the address of bar() in a .finiarray element.

```
$ cat main.c
#include    <stdio.h>

void foo()
{
        (void) printf("initializing: foo()\n");
}

void bar()
{
        (void) printf("finalizing: bar()\n");
}

main()
{
        (void) printf("main()\n");
        return (0);
}

$ cc -o main -zinitarray=foo -zfiniarray=bar main.c
$ main
initializing: foo()
main()
finalizing: bar()
```

The creation of initialization and termination sections can be carried out directly using an assembler. However, most compilers offer special primitives to simplify their declaration. For example, the previous code example can be rewritten using the following #pragma definitions. These definitions result in a call to foo() being placed in an .init section, and a call to bar() being placed in a .fini section.

```
$ cat main.c
#include    <stdio.h>

#pragma init (foo)
#pragma fini (bar)
```

```
.......
$ cc -o main main.c
$ main
initializing: foo()
main()
finalizing: bar()
```

Initialization and termination code, spread throughout several relocatable objects, can result in different behavior when included in an archive library or shared object. The link-edit of an application that uses this archive might extract only a fraction of the objects contained in the archive. These objects might provide only a portion of the initialization and termination code spread throughout the members of the archive. At runtime, only this portion of code is executed. The same application built against the shared object will have all the accumulated initialization and termination code executed when the dependency is loaded at runtime.

To determine the order of executing initialization and termination code within a process at runtime is a complex issue that involves dependency analysis. Limit the content of initialization and termination code to simplify this analysis, and provide predictable runtime behavior. See "Initialization and Termination Order" on page 81 for more details.

Data initialization should be independent if the initialization code is involved with a dynamic object whose memory can be dumped using `dldump`(3C).

Symbol Processing

During input file processing, all *local* symbols from the input relocatable objects are passed through to the output file image. All global symbols are accumulated internally within the link-editor. Each *global* symbol supplied by a relocatable object is searched for within this internal symbol table. If a symbol with the same name has already been encountered from a previous input file, a symbol resolution process is called. This symbol resolution process determines which of the two entries are kept.

On completing input file processing, and providing no fatal symbol resolution errors have occurred, the link-editor determines if any unresolved symbol references remain. Unresolved symbol references can cause the link-edit to terminate.

Finally, the link-editor's internal symbol table is added to the symbol tables of the image being created.

The following sections expand upon symbol resolution and undefined symbol processing.

Symbol Resolution

Symbol resolution runs the entire spectrum, from simple and intuitive to complex and perplexing. Most resolutions are carried out silently by the link-editor. However, some relocations can be accompanied by warning diagnostics, while others can result in a fatal error condition.

The resolution of two symbols depends on their attributes, the type of file that provides the symbol, and the type of file being generated. For a complete description of symbol attributes, see "Symbol Table Section" on page 225. For the following discussions, however, three basic symbol types are identified:

- *Undefined* – Symbols that have been referenced in a file but have not been assigned a storage address.
- *Tentative* – Symbols that have been created within a file but have not yet been sized, or allocated in storage. These symbols appear as uninitialized C symbols, or FORTRAN COMMON blocks within the file.
- *Defined* – Symbols that have been created, and assigned storage addresses and space within the file.

In its simplest form, symbol resolution involves the use of a precedence relationship. This relationship has *defined* symbols dominate *tentative* symbols, which in turn dominate *undefined* symbols.

The following example of C code shows how these symbol types can be generated. Undefined symbols are prefixed with u_. Tentative symbols are prefixed with t_. Defined symbols are prefixed with d_.

```
$ cat main.c
extern int      u_bar;
extern int      u_foo();

int             t_bar;
int             d_bar = 1;

d_foo()
{
        return (u_foo(u_bar, t_bar, d_bar));
}
$ cc -o main.o -c main.c
$ nm -x main.o

[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[8]      |0x00000000|0x00000000|NOTY |GLOB |0x0  |UNDEF  |u_foo
[9]      |0x00000000|0x00000040|FUNC |GLOB |0x0  |2      |d_foo
[10]     |0x00000004|0x00000004|OBJT |GLOB |0x0  |COMMON |t_bar
[11]     |0x00000000|0x00000000|NOTY |GLOB |0x0  |UNDEF  |u_bar
[12]     |0x00000000|0x00000004|OBJT |GLOB |0x0  |3      |d_bar
```

Simple Resolutions

Simple symbol resolutions are by far the most common. In this case, two symbols with similar characteristics are detected, with one symbol taking precedence over the other. This symbol resolution is carried out silently by the link-editor. For example, with symbols of the same binding, a symbol reference from one file is bound to a defined, or tentative symbol definition, from another file. Or, a tentative symbol definition from one file is bound to a defined symbol definition from another file.

Symbols that undergo resolution can have either a global or weak binding. Weak bindings have lower precedence than global binding, so symbols with different bindings are resolved according to a slight alteration of the basic rules.

Weak symbols can usually be defined through the compiler, either individually or as aliases to global symbols. One mechanism uses a `#pragma` definition:

```
$ cat main.c
#pragma weak    bar
#pragma weak    foo = _foo

int             bar = 1;

_foo()
{
        return (bar);
}
$ cc -o main.o -c main.c
$ nm -x main.o
[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[7]      |0x00000000|0x00000004|OBJT |WEAK |0x0   |3      |bar
[8]      |0x00000000|0x00000028|FUNC |WEAK |0x0   |2      |foo
[9]      |0x00000000|0x00000028|FUNC |GLOB |0x0   |2      |_foo
```

Notice that the weak alias `foo` is assigned the same attributes as the global symbol `_foo`. This relationship is maintained by the link-editor and results in the symbols being assigned the same value in the output image. In symbol resolution, weak defined symbols are silently overridden by any global definition of the same name.

Another form of simple symbol resolution, interposition, occurs between relocatable objects and shared objects, or between multiple shared objects. In these cases, when a symbol is multiply-defined, the relocatable object, or the first definition between multiple shared objects, is silently taken by the link-editor. The relocatable object's definition, or the first shared object's definition, is said to *interpose* on all other definitions. This interposition can be used to override the functionality provided by another shared object.

The combination of weak symbols and interposition provide a useful programming technique. For example, the standard C library provides several services that you are allowed to redefine. However, ANSI C defines a set of standard services that must be present on the system. These services cannot be replaced in a strictly conforming program.

The function fread(3C), for example, is an ANSI C library function, whereas the system function read(2) is not. A conforming ANSI C program must be able to redefine read(2) and still use fread(3C) in a predictable way.

The problem here is that read(2) underlies the fread(3C) implementation in the standard C library. Therefore, a program that redefines read(2) might confuse the fread(3C) implementation. To guard against this occurrence, ANSI C states that an implementation cannot use a name that is not reserved for the implementation. Use the following #pragma directive to define just such a reserved name. Use this name to generate an alias for the function read(2).

```
#pragma weak read = _read
```

Thus, you can quite freely define your own read() function without compromising the fread(3C) implementation, which in turn is implemented to use the _read() function.

The link-editor has no difficulty with this redefinition of read(), either when linking against the shared object or archive version of the standard C library. In the former case, interposition takes its course. In the latter case, the fact that the C library's definition of read(2) is weak allows that definition to be quietly overridden.

Use the link-editor's -m option to write a list of all interposed symbol references, along with section load address information, to the standard output.

Complex Resolutions

Complex resolutions occur when two symbols of the same name are found with differing attributes. In these cases, the link-editor generates a warning message, while selecting the most appropriate symbol. This message indicates the symbol, the attributes that conflict, and the identity of the file from which the symbol definition is taken. In the following example, two files with a definition of the data item array have different size requirements.

```
$ cat foo.c
int array[1];

$ cat bar.c
int array[2] = { 1, 2 };

$ cc -dn -r -o temp.o foo.c bar.c
ld: warning: symbol 'array' has differing sizes:
        (file foo.o value=0x4; file bar.o value=0x8);
        bar.o definition taken
```

A similar diagnostic is produced if the symbol's alignment requirements differ. In both of these cases, the diagnostic can be suppressed by using the link-editor's -t option.

Another form of attribute difference is the symbol's type. In the following example, the symbol bar() has been defined as both a data item and a function.

```
$ cat foo.c
bar()
{
        return (0);
}
$ cc -o libfoo.so -G -K pic foo.c
$ cat main.c
int     bar = 1;

main()
{
        return (bar);
}
$ cc -o main main.c -L. -lfoo
ld: warning: symbol 'bar' has differing types:
        (file main.o type=OBJT; file ./libfoo.so type=FUNC);
        main.o definition taken
```

Note – Symbol types in this context are classifications that can be expressed in ELF. These symbol types are not related to the data types as employed by the programming language, except in the crudest fashion.

In cases like the previous example, the relocatable object definition is taken when the resolution occurs between a relocatable object and a shared object. Or, the first definition is taken when the resolution occurs between two shared objects. When such resolutions occur between symbols of weak or global binding, a warning is also produced.

Inconsistencies between symbol types are not suppressed by the link-editor's `-t` option.

Fatal Resolutions

Symbol conflicts that cannot be resolved result in a fatal error condition and an appropriate error message. This message indicates the symbol name together with the names of the files that provided the symbols. No output file is generated. Although the fatal condition is sufficient to terminate the link-edit, all input file processing is first completed. In this manner, all fatal resolution errors can be identified.

The most common fatal error condition exists when two relocatable objects both define non-weak symbols of the same name:

```
$ cat foo.c
int bar = 1;

$ cat bar.c
bar()
{
```

```
        return (0);
}

$ cc -dn -r -o temp.o foo.c bar.c
ld: fatal: symbol 'bar' is multiply-defined:
        (file foo.o and file bar.o);
ld: fatal: File processing errors. No output written to int.o
```

`foo.c` and `bar.c` have conflicting definitions for the symbol `bar`. Because the link-editor cannot determine which should dominate, the link-edit usually terminates with an error message. You can use the link-editor's `-z muldefs` option to suppress this error condition. This option allows the first symbol definition to be taken.

Undefined Symbols

After all of the input files have been read and all symbol resolution is complete, the link-editor searches the internal symbol table for any symbol references that have not been bound to symbol definitions. These symbol references are referred to as *undefined* symbols. The effect of these undefined symbols on the link-edit process can vary according to the type of output file being generated, and possibly the type of symbol.

Generating an Executable Output File

When generating an executable output file, the link-editor's default behavior is to terminate with an appropriate error message should any symbols remain undefined. A symbol remains undefined when a symbol reference in a relocatable object is never matched to a symbol definition:

```
$ cat main.c
extern int foo();

main()
{
        return (foo());
}
$ cc -o prog main.c
Undefined           first referenced
 symbol                 in file
foo                     main.o
ld: fatal: Symbol referencing errors. No output written to prog
```

Similarly, if a shared object is used to create a dynamic executable, and a symbol reference within the object remains unresolved, an undefined symbol error results.

```
$ cat foo.c
extern int bar;
foo()
{
        return (bar);
```

```
}

$ cc -o libfoo.so -G -K pic foo.c
$ cc -o prog main.c -L. -lfoo
Undefined           first referenced
 symbol                 in file
bar                     ./libfoo.so
ld: fatal: Symbol referencing errors. No output written to prog
```

To allow undefined symbols, as in the previous example, use the link-editor's `-z nodefs` option to suppress the default error condition.

Note – Take care when using the `-z nodefs` option. If an unavailable symbol reference is required during the execution of a process, a fatal runtime relocation error occurs. This error might be detected during the initial execution and testing of an application. However, more complex execution paths can result in this error condition taking much longer to detect, which can be time consuming and costly.

Symbols can also remain undefined when a symbol reference in a relocatable object is bound to a symbol definition in an implicitly defined shared object. For example, continuing with the files `main.c` and `foo.c` used in the previous example:

```
$ cat bar.c
int bar = 1;

$ cc -o libbar.so -R. -G -K pic bar.c -L. -lfoo
$ ldd libbar.so
        libfoo.so =>     ./libfoo.so

$ cc -o prog main.c -L. -lbar
Undefined           first referenced
 symbol                 in file
foo                     main.o  (symbol belongs to implicit \
                        dependency ./libfoo.so)
ld: fatal: Symbol referencing errors. No output written to prog
```

`prog` is built with an *explicit* reference to `libbar.so`. `libbar.so` has a dependency on `libfoo.so`. Therefore, an implicit reference to `libfoo.so` from `prog` is established.

Because `main.c` made a specific reference to the interface provided by `libfoo.so`, `prog` really has a dependency on `libfoo.so`. However, only explicit shared object dependencies are recorded in the output file being generated. Thus, `prog` fails to run if a new version of `libbar.so` is developed that no longer has a dependency on `libfoo.so`.

For this reason, bindings of this type are deemed fatal. The implicit reference must be made explicit by referencing the library directly during the link-edit of `prog`. The required reference is hinted at in the fatal error message that is shown in the preceding example.

Generating a Shared Object Output File

When the link-editor is generating a shared object output file, undefined symbols are allowed to remain at the end of the link-edit. This default behavior allows the shared object to import symbols from a dynamic executable that defines the shared object as a dependency.

The link-editor's `-z defs` option can be used to force a fatal error if any undefined symbols remain. This option is recommended when creating any shared objects. Shared objects that reference symbols from an application can use the `-z defs` option, together with defining the symbols by using an `extern mapfile` directive. See "Defining Additional Symbols" on page 46.

A self-contained shared object, in which all references to external symbols are satisfied by named dependencies, provides maximum flexibility. The shared object can be employed by many users without those users having to determine and establish dependencies to satisfy the shared object's requirements.

Weak Symbols

Weak symbol references that remain unresolved, do not result in a fatal error condition, no matter what output file type is being generated.

If a static executable is being generated, the symbol is converted to an absolute symbol with an assigned value of zero.

If a dynamic executable or shared object is being produced, the symbol is left as an undefined weak reference with an assigned value of zero. During process execution, the runtime linker searches for this symbol. If the runtime linker does not find a match, the reference is bound to an address of zero instead of generating a fatal relocation error.

Historically, these undefined weak referenced symbols have been employed as a mechanism to test for the existence of functionality. For example, the following C code fragment might have been used in the shared object `libfoo.so.1`:

```
#pragma weak    foo

extern  void    foo(char *);

void bar(char * path)
{
        void (* fptr)(char *);

        if ((fptr = foo) != 0)
                (* fptr)(path);
}
```

When building an application that references libfoo.so.1, the link-edit completes successfully regardless of whether a definition for the symbol foo is found. If during execution of the application the function address tests nonzero, the function is called. However, if the symbol definition is not found, the function address tests zero and therefore is not called.

Compilation systems view this address comparison technique as having undefined semantics, which can result in the test statement being removed under optimization. In addition, the runtime symbol binding mechanism places other restrictions on the use of this technique. These restrictions prevent a consistent model from being made available for all dynamic objects.

Note – Undefined weak references in this manner are discouraged. Instead, you should use dlsym(3C) with the RTLD_DEFAULT, or RTLD_PROBE handles as a means of testing for a symbol's existence. See "Testing for Functionality" on page 94.

Tentative Symbol Order Within the Output File

Contributions from input files usually appear in the output file in the order of their contribution. An exception occurs when processing tentative symbols and their associated storage. These symbols are not fully defined until their resolution is complete. If the resolution is to a *defined* symbol from a relocatable object, then the order of appearance of the symbol follows that of the definition.

If you need to control the ordering of a group of symbols, then any tentative definition should be redefined to a zero-initialized data item. For example, the following tentative definitions result in a reordering of the data items within the output file, as compared to the original order described in the source file foo.c:

```
$ cat foo.c
char A_array[0x10];
char B_array[0x20];
char C_array[0x30];

$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]    |0x00020754|0x00000010|OBJT |GLOB |0x0  |15  |A_array
[34]    |0x00020764|0x00000030|OBJT |GLOB |0x0  |15  |C_array
[42]    |0x00020794|0x00000020|OBJT |GLOB |0x0  |15  |B_array
```

By defining these symbols as initialized data items, the relative ordering of these symbols within the input file is carried over to the output file:

```
$ cat foo.c
char A_array[0x10] = { 0 };
char B_array[0x20] = { 0 };
char C_array[0x30] = { 0 };
```

```
$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]     |0x000206bc|0x00000010|OBJT |GLOB |0x0   |12   |A_array
[42]     |0x000206cc|0x00000020|OBJT |GLOB |0x0   |12   |B_array
[34]     |0x000206ec|0x00000030|OBJT |GLOB |0x0   |12   |C_array
```

Defining Additional Symbols

Besides the symbols provided from input files, you can supply additional symbol references or definitions to a link-edit. In the simplest form, symbol references can be generated using the link-editor's `-u` option. Greater flexibility is provided with the link-editor's `-M` option and an associated `mapfile`. This mapfile enables you to define symbol references and a variety of symbol definitions.

The `-u` option provides a mechanism for generating a symbol reference from the link-edit command line. This option can be used to perform a link-edit entirely from archives. This option can also provide additional flexibility in selecting the objects to extract from multiple archives. See section "Archive Processing" on page 28 for an overview of archive extraction.

For example, perhaps you want to generate a dynamic executable from the relocatable object `main.o`, which refers to the symbols `foo` and `bar`. You want to obtain the symbol definition `foo` from the relocatable object `foo.o` contained in `lib1.a`, and the symbol definition `bar` from the relocatable object `bar.o`, contained in `lib2.a`.

However, the archive `lib1.a` also contains a relocatable object that defines the symbol `bar`. This relocatable object is presumably of differing functionality to the relocatable object that is provided in `lib2.a`. To specify the required archive extraction, you can use the following link-edit:

```
$ cc -o prog -L. -u foo -l1 main.o -l2
```

The `-u` option generates a reference to the symbol `foo`. This reference causes extraction of the relocatable object `foo.o` from the archive `lib1.a`. The first reference to the symbol `bar` occurs in `main.o`, which is encountered after `lib1.a` has been processed. Therefore, the relocatable object `bar.o` is obtained from the archive `lib2.a`.

Note – This simple example assumes that the relocatable object `foo.o` from `lib1.a` does not directly or indirectly reference the symbol `bar`. If `lib1.a` does reference `bar`, then the relocatable object `bar.o` is also extracted from `lib1.a` during its processing. See "Archive Processing" on page 28 for a discussion of the link-editor's multi-pass processing of an archive.

A more extensive set of symbol definitions can be provided using the link-editor's `-M` option and an associated `mapfile`. The syntax for these `mapfile` entries is:

```
[ name ] {
        scope:
                symbol [ = [ type ] [ value ] [ size ] [ attribute ] ];
} [ dependency ];
```

name

A label for this set of symbol definitions, if present, identifies a *version definition* within the image. See Chapter 5.

scope

Indicates the visibility of the symbols' binding within the output file being generated. All symbols defined with a `mapfile` are treated as global in scope during the link-edit process. These symbols are resolved against any other symbols of the same name that are obtained from any of the input files. The following definitions, and aliases, define a symbols' visibility in the object being created:

`default` / `global`

Symbols of this scope remain visible to other external objects. References to such symbols from within the object are bound at runtime, thus allowing interposition to take place.

`protected` / `symbolic`

Symbols of this scope remain visible to other external objects. References to these symbols from within the object are bound at link-edit, thus preventing runtime interposition. This scope definition has the same affect as a symbol with `STV_PROTECTED` visibility. See Table 7–25.

`hidden` / `local`

Symbols of this scope are reduced to symbols with a local binding. Symbols of this scope are not visible to other external objects. This scope definition has the same affect as a symbol with `STV_HIDDEN` visibility. See Table 7–25.

`eliminate`

Symbols of this scope are `hidden`. Their symbol table entries are eliminated.

symbol

The name of the symbol required. If the name is not followed by one of the symbol attributes, *type*, *value*, *size* or `extern`, a symbol reference is created. This reference is exactly the same as would be generated using the `-u` option discussed earlier in this section. If the symbol name is followed by any symbol attributes, then a symbol definition is generated using the associated attributes.

When in `local` scope, this symbol name can be defined as the special *auto-reduction* directive "*". This directive demotes all global symbols, not explicitly defined as `global` in any `mapfile`, to a local binding within the dynamic object being generated.

type

Indicates the symbol type attribute. This attribute can be either `data`, `function`, or `COMMON`. The former two type attributes result in an absolute symbol definition. See "Symbol Table Section" on page 225. The latter type attribute results in a tentative symbol definition.

value
Indicates the value attribute. This attribute takes the form of `V`*number*.

size
Indicates the size attribute. This attribute takes the form of `S`*number*.

attribute
This keyword provides additional attributes for the symbol:

`EXTERN`
Indicates the symbol is defined externally to the object being created. This attribute can be used with the `DIRECT` or `NODIRECT` attributes to establish individual direct, or no-direct references. Undefined symbols that would be flagged with the `-z defs` option can also be suppressed with this option.

`DIRECT`
Indicates that this symbol should be directly bound to. This attribute can be used with the `EXTERN` attribute to control binding to an external symbol. See "Direct Binding" on page 72.

`NODIRECT`
Indicates that this symbol should not be directly bound to. This state applies to references from within the object being created and from external references. This attribute can be used with the `EXTERN` attribute to control binding to an external symbol. See "Direct Binding" on page 72.

`FILTER` *name*
Indicates that this symbol is a filter on the shared object *name*. See "Generating Standard Filters" on page 110. Filter symbols do not require any backing implementation to be provided from an input relocatable object. Therefore, use this directive together with defining the symbol's type, to create an absolute symbol table entry.

`AUXILIARY` *name*
Indicates that this symbol is an auxiliary filter on the shared object *name*. See "Generating Auxiliary Filters" on page 113.

dependency
Represents a *version definition* that is inherited by this definition. See Chapter 5.

If either a version definition or the auto-reduction directive is specified, then versioning information is recorded in the image created. If this image is an executable or shared object, then any symbol reduction is also applied.

If the image being created is a relocatable object, then by default, no symbol reduction is applied. In this case, any symbol reductions are recorded as part of the versioning information. These reductions are applied when the relocatable object is finally used to generate an executable or shared object. The link-editor's `-B reduce` option can be used to force symbol reduction when generating a relocatable object.

A more detailed description of the versioning information is provided in Chapter 5.

Note – To ensure interface definition stability, no wildcard expansion is provided for defining symbol names.

This section presents several examples of using the `mapfile` syntax.

The following example shows how three symbol references can be defined. These references are then used to extract members of an archive. Although this archive extraction can be achieved by specifying multiple `-u` options to the link-edit, this example also shows how the eventual scope of a symbol can be reduced to *local*.

```
$ cat foo.c
foo()
{
        (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
bar()
{
        (void) printf("bar: called from lib.a\n");
}
$ cat main.c
extern  void    foo(), bar();

main()
{
        foo();
        bar();
}
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
{
        local:
                foo;
                bar;
        global:
                main;
};
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ nm -x prog | egrep "main$|foo$|bar$"
[28]    |0x00010604|0x00000024|FUNC |LOCL |0x0  |7      |foo
[30]    |0x00010628|0x00000024|FUNC |LOCL |0x0  |7      |bar
[49]    |0x0001064c|0x00000024|FUNC |GLOB |0x0  |7      |main
```

The significance of reducing symbol scope from global to local is covered in more detail in the section "Reducing Symbol Scope" on page 51.

The following example shows how two absolute symbol definitions can be defined. These definitions are then used to resolve the references from the input file `main.c`.

```
$ cat main.c
extern  int     foo();
extern  int     bar;

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = FUNCTION V0x400;
                bar = DATA V0x800;
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 400 &bar = 800
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00000800|0x00000000|OBJT |GLOB |0x0  |ABS    |bar
[42]    |0x00000400|0x00000000|FUNC |GLOB |0x0  |ABS    |foo
```

When obtained from an input file, symbol definitions for functions or data items are usually associated with elements of data storage. A `mapfile` definition is insufficient to be able to construct this data storage, so these symbols must remain as absolute values.

However, a `mapfile` can also be used to define a `COMMON`, or tentative, symbol. Unlike other types of symbol definition, tentative symbols do not occupy storage within a file, but define storage that must be allocated at runtime. Therefore, symbol definitions of this kind can contribute to the storage allocation of the output file being generated.

A feature of tentative symbols that differs from other symbol types is that their *value* attribute indicates their alignment requirement. A `mapfile` definition can therefore be used to realign tentative definitions that are obtained from the input files of a link-edit.

The following example shows the definition of two tentative symbols. The symbol `foo` defines a new storage region whereas the symbol `bar` is actually used to change the alignment of the same tentative definition within the file `main.c`.

```
$ cat main.c
extern  int     foo;
int             bar[0x10];

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
```

```
                foo = COMMON V0x4 S0x200;
                bar = COMMON V0x100 S0x40;
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol 'bar' has differing alignments:
        (file mapfile value=0x100; file main.o value=0x4);
        largest value applied
$ prog
&foo = 20940
&bar = 20900
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00020900|0x00000040|OBJT |GLOB |0x0   |16     |bar
[42]    |0x00020940|0x00000200|OBJT |GLOB |0x0   |16     |foo
```

Note – This symbol resolution diagnostic can be suppressed by using the link-editor's `-t` option.

Reducing Symbol Scope

Symbol definitions that are defined to have local scope within a `mapfile` can be used to reduce the symbol's eventual binding. This mechanism removes the symbol's visibility to future link-edits that use the generated file as part of their input. In fact, this mechanism can provide for the precise definition of a file's interface, and so restrict the functionality made available to others.

For example, say you want to generate a simple shared object from the files `foo.c` and `bar.c`. The file `foo.c` contains the global symbol `foo`, which provides the service that you want to make available to others. The file `bar.c` contains the symbols `bar` and `str`, which provide the underlying implementation of the shared object. The creation of a simple shared object usually results in all three of these symbols having global scope.

```
$ cat foo.c
extern  const char *    bar();

const char * foo()
{
        return (bar());
}
$ cat bar.c
const char * str = "returned from bar.c";

const char * bar()
{
        return (str);
}
$ cc -o lib.so.1 -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[29]    |0x000104d0|0x00000004|OBJT |GLOB |0x0   |12     |str
```

```
[32]     |0x00000418|0x00000028|FUNC |GLOB |0x0   |6       |bar
[33]     |0x000003f0|0x00000028|FUNC |GLOB |0x0   |6       |foo
```

You can now use the functionality offered by this shared object as part of the link-edit of another application. References to the symbol `foo` are bound to the implementation provided by the shared object.

Because of their global binding, direct reference to the symbols `bar` and `str` is also possible. This visibility can have dangerous consequences, as you might later change the implementation that underlies the function `foo`. In so doing, you could unintentionally cause an existing application that had bound to `bar` or `str` to fail or misbehave.

Another consequence of the global binding of the symbols `bar` and `str` is that these symbols can be interposed upon by symbols of the same name. The interposition of symbols within shared objects is covered in section "Simple Resolutions" on page 39. This interposition can be intentional and be used as a means of circumventing the intended functionality offered by the shared object. On the other hand, this interposition can be unintentional, the result of the same common symbol name used for both the application and the shared object.

When developing the shared object, you can protect against this scenario by reducing the scope of the symbols `bar` and `str` to a local binding. In the following example, the symbols `bar` and `str` are no longer available as part of the shared object's interface. Thus, these symbols cannot be referenced, or interposed upon, by an external object. You have effectively defined an interface for the shared object. This interface can be managed while hiding the details of the underlying implementation.

```
$ cat mapfile
{
        local:
                bar;
                str;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[27]     |0x000003dc|0x00000028|FUNC |LOCL |0x0   |6       |bar
[28]     |0x00010494|0x00000004|OBJT |LOCL |0x0   |12      |str
[33]     |0x000003b4|0x00000028|FUNC |GLOB |0x0   |6       |foo
```

This symbol scope reduction has an additional performance advantage. The symbolic relocations against the symbols `bar` and `str` that would have been necessary at runtime are now reduced to relative relocations. See "When Relocations are Performed" on page 124 for details of symbolic relocation overhead.

As the number of symbols that are processed during a link-edit increases, definiting local scope reduction within a `mapfile` becomes harder to maintain. An alternative and more flexible mechanism enables you to define the shared object's interface in terms of the global symbols that should be maintained. Global symbol definitions allow the link-editor to reduce all other symbols to local binding. This mechanism is achieved using the special *auto-reduction* directive "*". For example, the previous `mapfile` definition can be rewritten to define `foo` as the only global symbol required in the output file generated:

```
$ cat mapfile
lib.so.1.1
{
        global:
                foo;
        local:
                *;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[30]    |0x00000370|0x00000028|FUNC |LOCL |0x0  |6      |bar
[31]    |0x00010428|0x00000004|OBJT |LOCL |0x0  |12     |str
[35]    |0x00000348|0x00000028|FUNC |GLOB |0x0  |6      |foo
```

This example also defines a version name, `lib.so.1.1`, as part of the `mapfile` directive. This version name establishes an internal version definition that defines the file's symbolic interface. The creation of a version definition is recommended. The definition forms the foundation of an internal versioning mechanism that can be used throughout the evolution of the file. See Chapter 5.

Note – If a version name is not supplied, the output file name is used to label the version definition. The versioning information that is created within the output file can be suppressed using the link-editor's `-z noversion` option.

Whenever a version name is specified, *all* global symbols must be assigned to a version definition. If any global symbols remain unassigned to a version definition, the link-editor generates a fatal error condition:

```
$ cat mapfile
lib.so.1.1 {
        global:
                foo;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
Undefined           first referenced
 symbol                 in file
str                     bar.o  (symbol has no version assigned)
bar                     bar.o  (symbol has no version assigned)
ld: fatal: Symbol referencing errors. No output written to lib.so.1
```

The `-B local` option can be used to assert the *auto-reduction* directive "*" from the command line. The previous example could be compiled successfully with:

```
$ cc -o lib.so.1 -M mapfile -B local -G foo.c bar.c
```

When generating an executable or shared object, any symbol reduction results in the recording of version definitions within the output image. When generating a relocatable object, the version definitions are created but the symbol reductions are not processed. The result is that the symbol entries for any symbol reductions still remain global. For example, using the previous `mapfile` with the auto-reduction directive and associated relocatable objects, an intermediate relocatable object is created with no symbol reduction.

```
$ cat mapfile
lib.so.1.1 {
        global:
                foo;
        local:
                *;
};
$ ld -o lib.o -M mapfile -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[17]    |0x00000000|0x00000004|OBJT |GLOB |0x0  |3     |str
[19]    |0x00000028|0x00000028|FUNC |GLOB |0x0  |1     |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1     |foo
```

The version definitions created within this image show that symbol reductions are required. When the relocatable object is used eventually to generate an executable or shared object, the symbol reductions occur. In other words, the link-editor reads and interprets symbol reduction information that is contained in the relocatable objects in the same manner as versioning data is processes from a `mapfile`.

Thus, the intermediate relocatable object produced in the previous example can now be used to generate a shared object:

```
$ ld -o lib.so.1 -G lib.o
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[22]    |0x000104a4|0x00000004|OBJT |LOCL |0x0  |14    |str
[24]    |0x000003dc|0x00000028|FUNC |LOCL |0x0  |8     |bar
[36]    |0x000003b4|0x00000028|FUNC |GLOB |0x0  |8     |foo
```

Symbol reduction at the point at which an executable or shared object is created is typically the most common requirement. However, symbol reductions can be forced to occur when creating a relocatable object by using the link-editor's `-B reduce` option.

```
$ ld -o lib.o -M mapfile -B reduce -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[15]    |0x00000000|0x00000004|OBJT |LOCL |0x0  |3     |str
[16]    |0x00000028|0x00000028|FUNC |LOCL |0x0  |1     |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1     |foo
```

Symbol Elimination

An extension to symbol reduction is the elimination of a symbol entry from an object's symbol table. Local symbols are only maintained in an object's `.symtab` symbol table. This entire table can be removed from the object by using the link-editor's `-s` option, or `strip`(1). On occasion, you might want to maintain the `.symtab` symbol table but remove selected local symbol definitions.

Symbol elimination can be carried out using the `mapfile` directive `eliminate`. As with the `local` directive, symbols can be individually defined. Or, the symbol name can be defined as the special *auto-elimination* directive "*". The following example shows the elimination of the symbol `bar` for the previous symbol reduction example.

```
$ cat mapfile
lib.so.1.1
{
        global:
                foo;
        local:
                str;
        eliminate:
                *;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[31]    |0x00010428|0x00000004|OBJT |LOCL |0x0  |12     |str
[35]    |0x00000348|0x00000028|FUNC |GLOB |0x0  |6      |foo
```

The `-B eliminate` option can be used to assert the *auto-elimination* directive "*" from the command line.

External Bindings

When a symbol reference from the object being created is satisfied by a definition within a shared object, the symbol remains undefined. The relocation information that is associated with the symbol provides for its lookup at runtime. The shared object that provided the definition typically becomes a dependency.

The runtime linker employs a default search model to locate this definition at runtime. Typically, each object is searched, starting with the dynamic executable, and progressing through each dependency in the same order in which the objects were loaded.

Objects can also be created to use direct bindings. With this technique, the relationship between the symbol reference and the object that provides the symbol definition is maintained within the object being created. The runtime linker uses this information to directly bind the reference to the object that defines the symbol, thus bypassing the default symbol search model. See "Direct Binding" on page 72.

String Table Compression

String tables are compressed by the link-editor by removing duplicate entries, and tail substrings. This compression can significantly reduce the size of any string tables. A compressed `.dynstr` table results in a smaller text segment and hence reduced runtime paging activity. Because of these benefits, string table compression is enabled by default.

Objects that contribute a very large number of symbols can increase the link-edit time due to the string table compression. To avoid this cost during development use the link-editors `-z nocompstrtab` option. Any string table compression performed during a link-edit can be displayed using the link-editors debugging tokens `-D strtab,detail`.

Generating the Output File

After input file processing and symbol resolution has completed with no fatal errors, the link-editor generates the output file. The link-editor first generates the additional sections necessary to complete the output file. These sections include the symbol tables, which contain local symbol definitions together with resolved global and weak symbol information, from all the input files.

Also included are any output relocation and dynamic information sections required by the runtime linker. After all the output section information has been established, the total output file size is calculated. The output file image is then created accordingly.

When creating a dynamic executable or shared object, two symbol tables are usually generated. The `.dynsym` table and its associated string table `.dynstr` contain register, global, weak, and section symbols. These sections become part of the `text` segment that is mapped as part of the process image at runtime. See `mmap`(2). This mapping enables the runtime linker to read these sections to perform any necessary relocations.

The `.symtab` table, and its associated string table `.strtab` contain all the symbols collected from the input file processing. These sections are not mapped as part of the process image. These sections can even be stripped from the image by using the link-editor's `-s` option, or after the link-edit by using `strip`(1).

During the generation of the symbol tables, reserved symbols are created. These symbols have special meaning to the linking process. These symbols should not be defined in your code.

`_etext`
: The first location after the text segment.

`_edata`
: The first location after initialized data.

_end
: The first location after all data.

_DYNAMIC
: The address of the .dynamic dynamic information section.

END
: The same as _end. The symbol has local scope and, together with _START_, provides a means of establishing an object's address range.

_GLOBAL_OFFSET_TABLE_
: The position-independent reference to a link-editor supplied table of addresses, the .got section. This table is constructed from position-independent data references that occur in objects that have been compiled with the -K pic option. See "Position-Independent Code" on page 118.

_PROCEDURE_LINKAGE_TABLE_
: The position-independent reference to a link-editor supplied table of addresses, the .plt section. This table is constructed from position-independent function references that occur in objects that have been compiled with the -K pic option. See "Position-Independent Code" on page 118.

START
: The first location within the text segment. The symbol has local scope and, together with _END_, provides a means of establishing an object's address range.

When generating an executable, the link-editor looks for additional symbols to define the executable's entry point. If a symbol was specified using the link-editor's -e option, that symbol is used. Otherwise the link-editor looks for the reserved symbol names _start, and then main. If none of these symbols exists, the first address of the text segment is used.

Identifying Hardware and Software Capabilities

The hardware and software capabilities of a relocatable object are typically recorded at compile time. The link-editor combines the capabilities of any input relocatable objects to create a final capabilities section for the output file. See "Hardware and Software Capabilities Section" on page 207.

In addition, capabilities can be defined when the link-editor creates an output file. These capabilities are identified using a mapfile and the link-editor's -M option. Capabilities that are defined using a mapfile can augment, or override, the capabilities supplied from input relocatable objects.

The following sections describe how capabilities can be defined using a mapfile.

Identifying Hardware Capabilities

The hardware capabilities of an object identify the hardware requirements of a platform necessary for the object to execute correctly. An example of this requirement might be the identification of code that requires the MMX or SSE features that are available on some x86 architectures.

Hardware capability requirements can be identified using the following mapfile syntax:

```
hwcap_1 = TOKEN | Vval [ OVERRIDE ];
```

The hwcap_1 declaration is qualified with one or more tokens, which are symbolic representations of hardware capabilities. In addition, or alternatively, a numeric value representing one of more capabilities can be supplied by prefixing the value with a V. For SPARC platforms, hardware capabilities are defined as AV_ values in sys/auxv_SPARC.h. For x86 platforms, hardware capabilities are defined as AV_ values in sys/auxv_386.h.

The following x86 example shows the declaration of MMX and SSE as hardware capabilities required by the object foo.so.1.

```
$ egrep "MMX|SSE" /usr/include/sys/auxv_386.h
#define AV_386_MMX    0x0040
#define AV_386_SSE    0x0800
$ cat mapfile
hwcap_1 = SSE MMX;
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
```

Relocatable objects can contain hardware capabilities values. The link-editor combines any hardware capabilities values from multiple input relocatable objects. The resulting CA_SUNW_HW_1 value is a bitwise-inclusive OR of the associated input values. By default, these values are combined with the hardware capabilities specified by a mapfile. The hardware capability requirements of the output file can be controlled explicitly from a mapfile by using the OVERRIDE keyword. An OVERRIDE keyword, together with a hardware capability value of 0, effectively removes any hardware capabilities requirement from the object being built.

Any hardware capability requirements defined by an object are validated by the runtime linker against the hardware capabilities that are available to the process. If any of the hardware capability requirements can not be satisfied, the object is not loaded at runtime. For example, if the SSE feature is not available to a process, ldd(1) indicates the following error.

```
$ ldd prog
        foo.so.1 =>     ./foo.so.1  - hardware capability unsupported: \
                        0x800 [ SSE ]
```

```
        libc.so.1 =>     /lib/libc.so.1
```

Dynamic objects that exploit different hardware capabilities can provide a flexible runtime environment using filters. See “Hardware Capability Specific Shared Objects” on page 325.

Identifying Software Capabilities

The software capabilities of an object identify characteristics of the software that might be important for debugging or monitoring processes. Presently, the only software capabilities that are recognized relate to frame pointer usage by the object. Objects can declare that their frame pointer use is known. This state is then qualified by declaring the frame pointer as being used or not.

Two flags defined in `sys/elf.h` represent the frame pointer state:

```
#define  SF1_SUNW_FPKNWN    0x001
#define  SF1_SUNW_FPUSED    0x002
```

These software capability requirements can be identified using the following `mapfile` syntax:

```
sfcap_1 = TOKEN | Vval [ OVERRIDE ];
```

The `sfcap_1` declaration can be qualified with the tokens `FPKNWN` and `FPUSED`. Or, alternatively with a numeric value that represents these states.

Relocatable objects can contain software capabilities values. The link-editor combines the software capabilities values from multiple input relocatable objects. The computation of a `CA_SUNW_SF_1` value from two input values is as follows.

TABLE 2–1 `CA_SUNW_SF_1` Flag Combination State Table

Input file 1	Input file 2		
	SF1_SUNW_FPKNWN **SF1_SUNW_FPUSED**	**SF1_SUNW_FPKNWN**	**<unknown>**
SF1_SUNW_FPKNWN **SF1_SUNW_FPUSED**	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED
SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN
<unknown>	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED	SF1_SUNW_FPKNWN	<unknown>

This computation is applied to each relocatable object value and `mapfile` value. The software capabilities of an object are unknown if no `.SUNW_cap` section exists, or if the section contains no `CA_SUNW_SF_1` value, or if neither the `SF1_SUNW_FPKNW` or `SF1_SUNW_FPUSED` flags are set.

By default, any software capabilities specified by a `mapfile` are processed using the same state model. The software capability requirements of the output file can be controlled explicitly from a `mapfile` by using the `OVERRIDE` keyword. An `OVERRIDE` keyword, together with a software capability value of 0, effectively removes any software capabilities requirement from the object being built.

Relocation Processing

After you have created the output file, all data sections from the input files are copied to the new image. Any relocations specified by the input files are applied to the output image. Any additional relocation information that must be generated is also written to the new image.

Relocation processing is normally uneventful, although error conditions might arise that are accompanied by specific error messages. Two conditions are worth more discussion. The first condition involves text relocations that result from position-dependent code. This condition is covered in more detail in "Position-Independent Code" on page 118. The second condition can arise from displacement relocations, which is described more fully in the next section.

Displacement Relocations

Error conditions might occur if displacement relocations are applied to a data item, which can be used in a copy relocation. The details of copy relocations are covered in "Copy Relocations" on page 125.

A displacement relocation remains valid when both the relocated offset and the relocation target remain separated by the same displacement. A copy relocation is where a global data item within a shared object is copied to the `.bss` of an executable. This copy preserves the executable's read-only text segment. If the copied data has a displacement relocation applied to it, or an external relocation is a displacement into the copied data, the displacement relocation becomes invalidated.

The areas to address in trying to catch these sorts of errors are:

- When generating a shared object, flag any potential copy relocatable data items that can be problematic if the copied data is involved in a displacement relocation. During construction of a shared object, the link-editor has no knowledge of what external references might be made to a data item. Thus, all that can be flagged are *potential* problems.
- When generating an executable, flag the creation of a copy relocation whose data is involved in a displacement relocation.

However, displacement relocations applied to a shared object might be completed during its creation at link-edit time. Therefore, a link-edit of an application that references this shared object has no knowledge of a displacement being in effect in any copy-relocated data.

To help diagnose these problem areas, the link-editor indicates the displacement relocation use of a dynamic object with one or more dynamic `DT_FLAGS_1` flags, as shown in Table 7–45. In addition, the link-editor's `-z verbose` option can be used to display suspicious relocations.

For example, say you create a shared object with a global data item, `bar[]`, to which a displacement relocation is applied. This item could be copy-relocated if referenced from a dynamic executable. The link-editor warns of this condition with:

```
$ cc -G -o libfoo.so.1 -z verbose -K pic foo.o
ld: warning: relocation warning: R_SPARC_DISP32: file foo.o: symbol foo: \
    displacement relocation to be applied to the symbol bar: at 0x194: \
    displacement relocation will be visible in output image
```

If you now create an application that references the data item `bar[]`, a copy relocation is created. This copy results in the displacement relocation being invalidated. Because the link-editor can explicitly discover this situation, an error message is generated regardless of the use of the `-z verbose` option.

```
$ cc -o prog prog.o -L. -lfoo
ld: warning: relocation error: R_SPARC_DISP32: file foo.so: symbol foo: \
    displacement relocation applied to the symbol bar at: 0x194: \
    the symbol bar is a copy relocated symbol
```

Note – `ldd`(1), when used with either the `-d` or `-r` options, uses the displacement dynamic flags to generate similar relocation warnings.

These error conditions can be avoided by ensuring that the symbol definition being relocated (offset) and the symbol target of the relocation are both local. Use static definitions or the link-editor's scoping technology. See "Reducing Symbol Scope" on page 51. Relocation problems of this type can be avoided by accessing data within shared objects by using functional interfaces.

Debugging Aids

A debugging library is provided with the Solaris linkers. This library enables you to trace the link-editing process in more detail. This library can help you understand, or debug, the link-edit of your own applications or libraries. Although the type of information displayed using this library is expected to remain constant, the exact format of the information might change slightly from release to release.

Some of the debugging output might be unfamiliar if you do not have an intimate knowledge of the ELF format. However, many aspects might be of general interest to you.

Debugging is enabled by using the `-D` option. All output that is produced is directed to the standard error. This option must be augmented with one or more tokens to indicate the type of debugging that is required. The tokens available can be displayed by typing `-D help` at the command line.

```
$ ld -Dhelp
debug:
debug:            For debugging the link-editing of an application:
debug:                  LD_OPTIONS=-Dtoken1,token2 cc -o prog ...
debug:            or,
debug:                  ld -Dtoken1,token2 -o prog ...
debug:            where placement of -D on the command line is significant
debug:            and options can be switched off by prepending with '!'.
debug:
debug:
debug: args       display input argument processing
debug: basic      provide basic trace information/warnings
debug: cap        display hardware/software capability processing
debug: detail     provide more information in conjunction with other options
debug: entry      display entrance criteria descriptors
debug: files      display input file processing (files and libraries)
debug: got        display GOT symbol information
debug: help       display this help message
debug: libs       display library search paths; detail flag shows actual
debug:              library lookup (-l) processing
debug: map        display map file processing
debug: move       display move section processing
debug: reloc      display relocation processing
debug: sections   display input section processing
debug: segments   display available output segments and address/offset
debug:              processing; detail flag shows associated sections
debug: statistics display processing statistics
debug: strtab     display information about string table compression; detail
debug:              shows layout of string tables
debug: support    display support library processing
debug: symbols    display symbol table processing; detail flag shows
debug:              internal symbol table addition and resolution
debug: tls        display TLS processing info
debug: unused     display unused/unreferenced files; detail flag shows
debug:              unused sections
debug: versions   display version processing
```

Note – This listing is an example that shows the options that are meaningful to the link-editor. The exact options might differ from release to release.

Most compiler drivers interpret the `-D` option during their preprocessing phase. Therefore, the `LD_OPTIONS` environment variable is a suitable mechanism for passing this option to the link-editor.

The following example shows how input files can be traced. This syntax can be especially useful in determining what libraries have been located, or what relocatable objects have been extracted from an archive.

```
$ LD_OPTIONS=-Dfiles cc -o prog main.o -L. -lfoo
............
debug: file=main.o  [ ET_REL ]
debug: file=./libfoo.a  [ archive ]
debug: file=./libfoo.a(foo.o)  [ ET_REL ]
debug: file=./libfoo.a  [ archive ] (again)
............
```

Here, the member `foo.o` is extracted from the archive library `libfoo.a` to satisfy the link-edit of `prog`. Notice that the archive is searched twice to verify that the extraction of `foo.o` did not warrant the extraction of additional relocatable objects. More than one "(again)" display indicates that the archive is a candidate for ordering using `lorder`(1) and `tsort`(1).

By using the `symbols` token, you can determine which symbol caused an archive member to be extracted, and which object made the initial symbol reference.

```
$ LD_OPTIONS=-Dsymbols cc -o prog main.o -L. -lfoo
............
debug: symbol table processing; input file=main.o  [ ET_REL ]
............
debug: symbol[7]=foo  (global); adding
debug:
debug: symbol table processing; input file=./libfoo.a  [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo  (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo(foo.o)  [ ET_REL ]
.............
```

The symbol `foo` is referenced by `main.o`, and is added to the link-editor's internal symbol table. This symbol reference causes the extraction of the relocatable object `foo.o` from the archive `libfoo.a`.

Note – This output has been simplified for this document.

By using the `detail` token together with the `symbols` token, the details of symbol resolution during input file processing can be observed.

```
$ LD_OPTIONS=-Dsymbols,detail cc -o prog main.o -L. -lfoo
............
debug: symbol table processing; input file=main.o  [ ET_REL ]
............
debug: symbol[7]=foo  (global); adding
debug:   entered  0x000000 0x000000 NOTY GLOB  UNDEF REF_REL_NEED
debug:
debug: symbol table processing; input file=./libfoo.a  [ archive ]
```

```
debug: archive[0]=bar
debug: archive[1]=foo  (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo.a(foo.o)  [ ET_REL ]
debug: symbol[1]=foo.c
.............
debug: symbol[7]=bar  (global); adding
debug:   entered  0x000000 0x000004 OBJT GLOB  3     REF_REL_NEED
debug: symbol[8]=foo  (global); resolving [7][0]
debug:       old  0x000000 0x000000 NOTY GLOB  UNDEF main.o
debug:       new  0x000000 0x000024 FUNC GLOB  2     ./libfoo.a(foo.o)
debug:  resolved  0x000000 0x000024 FUNC GLOB  2     REF_REL_NEED
............
```

The original undefined symbol `foo` from `main.o` has been overridden with the symbol definition from the extracted archive member `foo.o`. The detailed symbol information reflects the attributes of each symbol.

In the previous example, you can see that using some of the debugging tokens can produce a wealth of output. To monitor the activity around a subset of the input files, place the `-D` option directly in the link-edit command-line. This option can be toggled on and off. In the following example, the display of symbol processing is switched on only during the processing of the library `libbar`.

```
$ ld .... -o prog main.o -L. -Dsymbols -lbar -D!symbols ....
```

Note – To obtain the link-edit command line, you might have to expand the compilation line from any driver being used. See "Using a Compiler Driver" on page 27.

CHAPTER 3

Runtime Linker

As part of the initialization and execution of a *dynamic executable*, an *interpreter* is called to complete the binding of the application to its dependencies. In the Solaris OS, this interpreter is referred to as the runtime linker.

During the link-editing of a dynamic executable, a special `.interp` section, together with an associated program header, are created. This section contains a path name specifying the program's interpreter. The default name supplied by the link-editor is the name of the runtime linker: `/usr/lib/ld.so.1` for a 32–bit executable and `/usr/lib/64/ld.so.1` for a 64–bit executable.

Note – `ld.so.1` is a special case of a shared object. Here, a version number of 1 is used. However, later Solaris releases might provide higher version numbers.

During the process of executing a dynamic object, the kernel loads the file and reads the program header information. See "Program Header" on page 239. From this information, the kernel locates the name of the required interpreter. The kernel loads, and transfers control to this interpreter, passing sufficient information to enable the interpreter to continue executing the application.

In addition to initializing an application, the runtime linker provides services that enable the application to extend its address space. This process involves loading additional objects and binding to symbols provided by these objects.

The runtime linker:

- Analyzes the executable's dynamic information section (`.dynamic`) and determines what dependencies are required.
- Locates and loads these dependencies, analyzing their dynamic information sections to determine if any additional dependencies are required.
- Performs any necessary relocations to bind these objects in preparation for process execution.

- Calls any initialization functions provided by the dependencies.
- Passes control to the application.
- Can be called upon during the application's execution, to perform any delayed function binding.
- Can be called upon by the application to acquire additional objects with `dlopen`(3C), and bind to symbols within these objects with `dlsym`(3C).

Shared Object Dependencies

When the runtime linker creates the memory segments for a program, the dependencies tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the runtime linker generates a complete process image.

Note – Even when a shared object is referenced multiple times in the dependency list, the runtime linker connects the object only once to the process.

Locating Shared Object Dependencies

When linking a dynamic executable, one or more shared objects are explicitly referenced. These objects are recorded as dependencies within the dynamic executable.

The runtime linker uses this dependency information to locate, and load, the associated objects. These dependencies are processed in the same order as the dependencies were referenced during the link-edit of the executable.

Once all the dynamic executable's dependencies are loaded, each dependency is inspected, in the order the dependency is loaded, to locate any additional dependencies. This process continues until all dependencies are located and loaded. This technique results in a breadth-first ordering of all dependencies.

Directories Searched by the Runtime Linker

The runtime linker looks in two default locations for dependencies. When processing 32–bit objects, the default locations are `/lib` and `/usr/lib`. When processing 64–bit objects, the default locations are `/lib/64` and `/usr/lib/64`. Any dependency specified as a simple file name is prefixed with these default directory names. The resulting path name is used to locate the actual file.

The dependencies of a dynamic executable or shared object can be displayed using ldd(1). For example, the file /usr/bin/cat has the following dependencies:

```
$ ldd /usr/bin/cat
        libc.so.1 =>     /lib/libc.so.1
        libm.so.2 =>     /lib/libm.so.2
```

The file /usr/bin/cat has a dependency, or *needs*, the files libc.so.1 and libm.so.2.

The dependencies recorded in an object can be inspected using dump(1). Use this command to display the file's .dynamic section, and look for entries that have a NEEDED tag. In the following example, the dependency libm.so.2, displayed in the previous ldd(1) example, is not recorded in the file /usr/bin/cat. ldd(1) shows the *total* dependencies of the specified file, and libm.so.2 is actually a dependency of /lib/libc.so.1.

```
$ dump -Lvp /usr/bin/cat

/usr/bin/cat:
[INDEX] Tag      Value
[1]     NEEDED   libc.so.1
.........
```

In the previous dump(1) example, the dependencies are expressed as simple file names. In other words, there is no '/' in the name. The use of a simple file name requires the runtime linker to generate the path name from a set of rules. File names that contain an embedded '/', are used as provided.

The simple file name recording is the standard, most flexible mechanism of recording dependencies. The -h option of the link-editor records a simple name within the dependency. See "Naming Conventions" on page 104 and "Recording a Shared Object Name" on page 105.

Frequently, dependencies are distributed in directories other than /lib and /usr/lib, or /lib/64 and /usr/lib/64. If a dynamic executable or shared object needs to locate dependencies in another directory, the runtime linker must explicitly be told to search this directory.

You can specify additional search path, on a per-object basis, by recording a *runpath* during the link-edit of an object. See "Directories Searched by the Runtime Linker" on page 34 for details on recording this information.

Any runpath recording can be displayed using dump(1). Reference the .dynamic entry that has the RUNPATH tag. In the following example, prog has a dependency on libfoo.so.1. The runtime linker must search directories /home/me/lib and /home/you/lib before it looks in the default location.

```
$ dump -Lvp prog

prog:
[INDEX] Tag      Value
```

```
[1]     NEEDED   libfoo.so.1
[2]     NEEDED   libc.so.1
[3]     RUNPATH  /home/me/lib:/home/you/lib
.........
```

Another way to add to the runtime linker's search path is to set the environment variable LD_LIBRARY_PATH. This environment variable, which is analyzed once at process startup, can be set to a colon-separated list of directories. These directories are searched by the runtime linker before any runpath specification or default directory.

These environment variables are well suited to debugging purposes, such as forcing an application to bind to a local dependency. In the following example, the file prog from the previous example is bound to libfoo.so.1, found in the present working directory.

```
$ LD_LIBRARY_PATH=. prog
```

Although useful as a temporary mechanism of influencing the runtime linker's search path, the use of LD_LIBRARY_PATH is strongly discouraged in production software. Any dynamic executables that can reference this environment variable will have their search paths augmented. This augmentation can result in an overall degradation in performance. Also, as pointed out in "Using an Environment Variable" on page 33 and "Directories Searched by the Runtime Linker" on page 34, LD_LIBRARY_PATH affects the link-editor.

Environmental search paths can result in a 64–bit executable searching a path that contains a 32–bit library matching the name being looked for. Or, vice versa. The runtime linker rejects the mismatched 32–bit library and continues down its search path looking for a valid 64–bit match. If no match is found, an error message is generated. This can be observed in detail by setting the LD_DEBUG environment variable to include the *files* token. See "Debugging Library" on page 96.

```
$ LD_LIBRARY_PATH=/lib/64 LD_DEBUG=files /usr/bin/ls
...
00283: file=libc.so.1;  needed by /usr/bin/ls
00283:
00283: file=/lib/64/libc.so.1  rejected: ELF class mismatch: 32-bit/64-bit
00283:
00283: file=/lib/libc.so.1  [ ELF ]; generating link map
00283:     dynamic:  0xef631180  base:  0xef580000  size:      0xb8000
00283:     entry:    0xef5a1240  phdr:  0xef580034  phnum:          3
00283:      lmid:           0x0
00283:
00283: file=/lib/libc.so.1;  analyzing  [ RTLD_GLOBAL  RTLD_LAZY ]
...
```

If a dependency cannot be located, ldd(1) indicates that the object cannot be found. Any attempt to execute the application results in an appropriate error message from the runtime linker:

```
$ ldd prog
        libfoo.so.1 =>   (file not found)
        libc.so.1 =>     /lib/libc.so.1
```

```
        libm.so.2 =>     /lib/libm.so.2
$ prog
ld.so.1: prog: fatal: libfoo.so.1: open failed: No such file or directory
```

Configuring the Default Search Paths

The default search paths used by the runtime linker are `/lib` and `/usr/lib` for 32–bit application. For 64–bit applications, the default search paths are `/lib/64` and `/usr/lib/64`. These search paths can be administered using a runtime configuration file created by the `crle`(1) utility. This file is often a useful aid for establishing search paths for applications that have not been built with the correct runpaths.

A configuration file can be constructed in the default location `/var/ld/ld.config`, for 32–bit applications, or `/var/ld/64/ld.config`, for 64–bit applications. This file affects all applications of the respective type on a system. Configuration files can also be created in other locations, and the runtime linker's `LD_CONFIG` environment variable used to select these files. This latter method is useful for testing a configuration file before installing the file in the default location.

Dynamic String Tokens

The runtime linker allows for the expansion of various dynamic string tokens. These tokens are applicable for filter, runpath and dependency definitions.

- `$HWCAP` – Indicates a directory in which objects offering differing hardware capabilities can be located. See "Hardware Capability Specific Shared Objects" on page 325.
- `$ISALIST` – Expands to the native instruction sets executable on this platform. See "Instruction Set Specific Shared Objects" on page 327.
- `$ORIGIN` – Provides the directory location of the current object. See "Locating Associated Dependencies" on page 329.
- `$OSNAME` – Expands to the name of the operating system. See "System Specific Shared Objects" on page 329.
- `$OSREL` – Expands to the operating system release level. See "System Specific Shared Objects" on page 329.
- `$PLATFORM` – Expands to the processor type of the current machine. See "System Specific Shared Objects" on page 329.

Relocation Processing

After the runtime linker has loaded all the dependencies required by an application, the linker processes each object and performs all necessary relocations.

During the link-editing of an object, any relocation information supplied with the input relocatable objects is applied to the output file. However, when creating a dynamic executable or shared object, many of the relocations cannot be completed at link-edit time. These relocations require logical addresses that are known only when the objects are loaded into memory. In these cases, the link-editor generates new relocation records as part of the output file image. The runtime linker must then process these new relocation records.

For a more detailed description of the many relocation types, see "Relocation Types (Processor-Specific)" on page 215. Two basic types of relocation exist.

- Non-symbolic relocations
- Symbolic relocations

The relocation records for an object can be displayed by using `dump`(1). In the following example, the file `libbar.so.1` contains two relocation records that indicate that the *global offset table*, or `.got` section, must be updated.

```
$ dump -rvp libbar.so.1

libbar.so.1:

.rela.got:
Offset      Symndx                Type              Addend

0x10438     0                     R_SPARC_RELATIVE  0
0x1043c     foo                   R_SPARC_GLOB_DAT  0
```

The first relocation is a simple relative relocation that can be seen from its relocation type and the symbol index (`Symndx`) field being zero. This relocation needs to use the base address at which the object was loaded into memory to update the associated `.got` offset.

The second relocation requires the address of the symbol `foo`. To complete this relocation, the runtime linker must locate this symbol from either the dynamic executable or one of its dependencies.

Symbol Lookup

The runtime linker is responsible for searching for symbols that are required by objects at runtime. This symbol search is based upon the requesting object's symbol *search scope*, together with the symbol *visibility* offered by each object within the process.

These attributes can be applied as defaults at the time the object is loaded. These attributes can also be supplied as specific modes to dlopen(3C). In some cases, these attributes can be recorded within the object at the time the object is built.

Typically, users becomes familiar with the default search model that is applied to a dynamic executable and its dependencies, and to objects obtained through dlopen(3C). The former is outlined in the next section "Default Lookup" on page 71, and the latter, which is also able to exploit the various symbol lookup attributes, is discussed in "Symbol Lookup" on page 87.

An alternative model for symbol lookup is provided when a dynamic object employes direct bindings. This model directs the runtime linker to search for a symbol directly in the object that provided the symbol at link-edit time. See "Direct Binding" on page 72.

Default Lookup

A dynamic executable and all the dependencies loaded with the executable are assigned *world* search scope, and *global* symbol visibility. See "Symbol Lookup" on page 87. A symbol lookup for a dynamic executable or for any of the dependencies loaded with the executable, results in a search of each object. The runtime linker starts with the dynamic executable, and progresses through each dependency in the same order in which the objects were loaded.

As discussed in previous sections, ldd(1) lists the dependencies of a dynamic executable in the order in which the dependencies are loaded. For example, the shared object libbar.so.1 requires the address of symbol foo to complete its relocation. The dynamic executable prog specifies libbar.so.1 as one of its dependencies.

```
$ ldd prog
        libfoo.so.1 =>   /home/me/lib/libfoo.so.1
        libbar.so.1 =>   /home/me/lib/libbar.so.1
```

The runtime linker first looks for foo in the dynamic executable prog, then in the shared object /home/me/lib/libfoo.so.1, and finally in the shared object /home/me/lib/libbar.so.1.

Note – Symbol lookup can be an expensive operation, especially when the size of symbol names increases and the number of dependencies increases. This aspect of performance is discussed in more detail in "Performance Considerations" on page 115. See "Direct Binding" on page 72 for an alternative lookup model.

The default relocation processing model also provides for transitioning into a lazy loading environment. If a symbol can not be found in the presently loaded objects, any pending lazy loaded objects are processed in an attempt to locate the symbol. This loading compensates for objects that have not fully defined their dependencies. However, this compensation can undermine the advantages of a lazy loading.

Interposition

The runtime linker's default mechanism searches for a symbol first in the dynamic executable and then in each of the dependencies. This search means that the first occurrence of the required symbol satisfies the search. Therefore, if more than one instance of the same symbol exists, the first instance interposes on all others. See also "Shared Object Processing" on page 30.

Direct Binding

When creating an object to use direct bindings, the relationship between the referenced symbol and the dependency that provided the definition is recorded in the object. The runtime linker uses this information to search directly for the symbol in the associated object, rather than carry out the default symbol search model. Direct binding information can only be established to dependencies specified with the link-edit. Therefore, use of the `-z defs` option is recommended.

Direct bindings can be established with one of the following mechanisms.

- With the `-B direct` option. This option establishes direct bindings between the object being built and all its dependencies. The use of `-B direct` also enables lazy loading, which is equivalent to adding the option `-z lazyload` to the front of the link-edit command line. See "Lazy Loading of Dynamic Dependencies" on page 76.
- With the `-z direct` option. This option establishes direct bindings between the object being built and any dependencies that follow the option on the command line. This option can be used together with the `-z nodirect` option, to toggle the use of direct bindings between dependencies.
- With the `DIRECT` `mapfile` attribute. This attribute provides for directly binding individual symbols. The alternative attribute `NODIRECT`, can be used to prevent direct binding to individual symbols. See "Defining Additional Symbols" on page 46.

The direct binding model can significantly reduce the symbol lookup overhead within a dynamic process that has many symbolic relocations and many dependencies. This model also enables multiple symbols of the same name to be located from different objects that have been bound to directly.

Direct binding can circumvent the traditional use of interposition symbols because it bypasses the default search model. The default model ensures that all references to a symbol bind to one definition.

Interposition can still be achieved in a direct binding environment, on a per-object basis, if an object is identified as an interposer. Any object loaded using the environment variable `LD_PRELOAD` or created with the link-editor's `-z interpose` option, is identified as an interposer. When the runtime linker searches for a directly bound symbol, it first looks in any object identified as an interposer before it looks in the object that supplies the symbol definition.

Note – Direct bindings can be disabled at runtime by setting the environment variable `LD_NODIRECT` to a non-null value.

Some interfaces exist that offer alternative implementations of a default technology. These implementations also assume they are the only instance of that technology within a process. An example of this is the `malloc`(3C) family. Directly binding to interfaces within such a family should be avoided, as it is possible for more than one instance of the technology to be referenced by the same process. For example, one dependency within a process can directly bind against `libc.so.1`, while another dependency directly binds against `libmapmalloc.so.1`.

Objects that provide a single implementation for a process, should define the interfaces to that implementation using the `mapfile` directive `NODIRECT`. This directive insures no users directly bind to an implementation, but use the default symbol search model.

Note – `NODIRECT` `mapfile` directives can be combined with the command line options `-B direct` or `-z direct`. Symbols that are not explicitly defined `NODIRECT` will follow the command line directive.

When Relocations Are Performed

Relocations can be distinguish by when they are performed. This distinction arises due to the type of *reference* being made to the relocated offset, and is either:

- An immediate reference
- A lazy reference

An *immediate reference* refers to a relocation that must be determined immediately when an object is loaded. These references are typically to data items used by the object code, pointers to functions, and even calls to functions made from position-dependent shared objects. These relocations cannot provide the runtime linker with knowledge of when the relocated item is referenced. Therefore, all immediate relocations must be carried out when an object is loaded, and before the application gains, or regains, control.

A *lazy reference* refers to a relocation that can be determined as an object executes. These references are typically calls to global functions made from position-independent shared objects, or calls to external functions made from a dynamic executable. During the compilation and link-editing of any dynamic module that provide these references, the associated function calls become calls to a procedure linkage table entry. These entries make up the `.plt` section. Each procedure linkage table entry becomes a lazy reference with a relocation associated with it.

Procedure linkage table entries are constructed so that when they are first called, control is passed to the runtime linker. The runtime linker looks up the required symbol and rewrites information in the associated object so that any future calls to this procedure linkage table entry go directly to the function. This mechanism enables relocations of this type to be deferred until the first instance of a function is called. This process is sometimes referred to as *lazy* binding.

The runtime linker's default mode is to perform lazy binding whenever procedure linkage table relocations are provided. This default can be overridden by setting the environment variable `LD_BIND_NOW` to any non-null value. This environment variable setting causes the runtime linker to perform both immediate and lazy reference relocations when an object is loaded, and before the application gains, or regains, control. For example, setting the environment variable as follows means that all relocations within the file `prog` and within its dependencies, will be processed before control is transferred to the application.

```
$ LD_BIND_NOW=1 prog
```

Objects can also be accessed with `dlopen`(3C) with the mode defined as `RTLD_NOW`. Objects can also be built using the link-editor's `-z now` option to indicate that they require complete relocation processing at the time they are loaded. This relocation requirement is also propagated to any dependencies of the marked object at runtime.

Note – Although the preceding examples of immediate and lazy references are typical, the creation of procedure linkage table entries is ultimately controlled by the relocation information provided by the relocatable object files used as input to a link-edit. Relocation records such as `R_SPARC_WPLT30` and `R_386_PLT32` instruct the link-editor to create a procedure linkage table entry are common for position-independent code. However, as a dynamic executable has a fixed location, external function references that can be determined at link-edit time can be converted to procedure linkage table entries regardless of the original relocation records.

Relocation Errors

The most common relocation error occurs when a symbol cannot be found. This condition results in an appropriate runtime linker error message and the termination of the application. For example:

```
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libc.so.1 =>     /lib/libc.so.1
        libbar.so.1 =>   ./libbar.so.1
        libm.so.2 =>     /lib/libm.so.2
$ prog
ld.so.1: prog: fatal: relocation error: file ./libfoo.so.1: \
symbol bar: referenced symbol not found
```

The symbol `bar`, which is referenced in the file `libfoo.so.1`, cannot be located.

During the link-edit of a dynamic executable, any potential relocation errors of this sort are flagged as fatal undefined symbols. See "Generating an Executable Output File" on page 42 for examples. This runtime relocation error can occur if the link-edit of main used a different version of the shared object libbar.so.1 that contained a symbol definition for bar, or if the -z nodefs option was used as part of the link-edit.

If a relocation error of this type occurs because a symbol used as an immediate reference cannot be located, the error condition will occur immediately during process initialization. Because of the default mode of lazy binding, if a symbol used as a lazy reference cannot be found, the error condition will occur after the application has gained control. This latter case can take minutes or months, or might never occur, depending on the execution paths exercised throughout the code.

To guard against errors of this kind, the relocation requirements of any dynamic executable or shared object can be validated using ldd(1).

When the -d option is specified with ldd(1), all dependencies will be printed and all immediate reference relocations will be processed. If a reference cannot be resolved, a diagnostic message is produced. From the previous example this option would result in:

```
$ ldd -d prog
        libfoo.so.1 =>   ./libfoo.so.1
        libc.so.1 =>     /lib/libc.so.1
        libbar.so.1 =>   ./libbar.so.1
        libm.so.2 =>     /lib/libm.so.2
        symbol not found: bar           (./libfoo.so.1)
```

When the -r option is specified with ldd(1), all immediate *and* lazy reference relocations are processed. If either type of relocation cannot be resolved, a diagnostic message is produced.

Loading Additional Objects

The runtime linker provides an additional level of flexibility by enabling you to introduce new objects during process initialization.

The environment variable LD_PRELOAD can be initialized to a shared object or relocatable object file name, or a string of file names separated by white space. These objects are loaded after the dynamic executable and before any dependencies. These objects are assigned *world* search scope, and *global* symbol visibility.

```
$ LD_PRELOAD=./newstuff.so.1 prog
```

The dynamic executable prog is loaded, followed by the shared object newstuff.so.1, and then by the dependencies defined within prog.

The order in which these objects are processed can be displayed using ldd(1):

```
$ LD_PRELOAD=./newstuff.so.1 ldd prog
        ./newstuff.so.1 => ./newstuff.so
        libc.so.1 =>     /lib/libc.so.1
```

In another example the preloading is a little more complex and time consuming.

```
$ LD_PRELOAD="./foo.o ./bar.o" prog
```

The runtime linker first link-edits the relocatable objects `foo.o` and `bar.o` to generate a shared object that is maintained in memory. This memory image is then inserted between the dynamic executable and its dependencies in the same manner as the shared object `newstuff.so.1` was preloaded in the previous example. Again, the order in which these objects are processed can be displayed with ldd(1):

```
$ LD_PRELOAD="./foo.o ./bar.o" ldd prog
        ./foo.o =>       ./foo.o
        ./bar.o =>       ./bar.o
        libc.so.1 =>     /lib/libc.so.1
```

These mechanisms of inserting an object after a dynamic executable take the concept of interposition to another level. You can use these mechanisms to experiment with a new implementation of a function that resides in a standard shared object. If you preload an object containing this function, the object interposes on the original. Thus the old functionality can be completely hidden with the new preloaded version.

Another use of preloading is to augment a function that resides in a standard shared object. The intention is to interpose the new symbol on the original, enabling the new function to carry out some additional processing while calling through to the original function. This mechanism requires either a symbol alias that is to be associated with the original function or the ability to look up the original symbol's address.

Lazy Loading of Dynamic Dependencies

When a dynamic object is loaded into memory, the object is examined for any additional dependencies. By default, any dependencies that exist are immediately loaded. This cycle continues until the full dependency tree is exhausted. Finally, all inter-object data references that are specified by relocations, are resolved. These operations are performed regardless of whether the code in these dependencies is referenced by the application during its execution.

Under a lazy loading model, any dependencies that are labeled for lazy loading are loaded only when explicitly referenced. By taking advantage of a function call's lazy binding, the loading of a dependency is delayed until the function is first referenced. As a result, objects that are never referenced are never loaded.

A relocation reference can be immediate or lazy. Because immediate references must be resolved when an object is initialized, any dependency that satisfies this reference must be immediately loaded. Therefore, identifying such a dependency as lazy loadable has little effect. See "When Relocations Are Performed" on page 73. Immediate references between dynamic objects are generally discouraged.

Lazy loading is used by the link-editors reference to a debugging library, `liblddbg`. As debugging is only called upon infrequently, loading this library every time that the link-editor is invoked is unnecessary and expensive. By indicating that this library can be lazily loaded, the expense of processing the library can be moved to those invocations that ask for debugging output.

The alternate method of achieving a lazy loading model is to use `dlopen()` and `dlsym()` to load and bind to a dependency when needed. This model is ideal if the number of `dlsym()` references is small, or the dependency name or location is not known at link-edit time. For more complex interactions with known dependencies, coding to normal symbol references and designating the dependency to be lazily loaded is simpler.

An object is designated as lazily or normally loaded through the link-editor options `-z lazyload` and `-z nolazyload` respectfully. These options are position-dependent on the link-edit command line. Any dependency that is found following the option takes on the loading attribute specified by the option. By default, the `-z nolazyload` option is in effect.

The following simple program has a dependency on `libdebug.so.1`. The dynamic section (`.dynamic`), shows `libdebug.so.1` is marked for lazy loading. The symbol information section (`.SUNW_syminfo`), shows the symbol reference that triggers `libdebug.so.1` loading.

```
$ cc -o prog prog.c -L. -zlazyload -ldebug -znolazyload -R'$ORIGIN'
$ elfdump -d prog

Dynamic Section:  .dynamic
     index  tag           value
       [0]  POSFLAG_1     0x1           [ LAZY ]
       [1]  NEEDED        0x123         libdebug.so.1
       [2]  NEEDED        0x131         libc.so.1
       [3]  RUNPATH       0x13b         $ORIGIN
       ...
$ elfdump -y prog

Syminfo section: .SUNW_syminfo
     index  flgs        bound to        symbol
      ....
      [52]  DL      [1] libdebug.so.1   debug
```

The `POSFLAG_1` with the value of `LAZY` designates that the following `NEEDED` entry, `libdebug.so.1`, should be lazily loaded. As `libc.so.1` has no preceding `LAZY` flag, this library is loaded at the initial startup of the program.

The use of lazy loading can require a precise declaration of dependencies and runpaths through out the objects used by an application. For example, suppose two objects, libA.so and libB.so, both make reference to symbols in libX.so. libA.so declares libX.so as a dependency, but libB.so does not. Typically, when libA.so and libB.so are used together, libB.so can reference libX.so because libA.so made this dependency available. But, if libA.so declares libX.so to be lazy loaded, it is possible that libX.so might not be loaded when libB.so makes reference to this dependency. A similar failure can occur if libB.so declares libX.so as a dependency but fails to provide a runpath necessary to locate the dependency.

Regardless of lazy loading, dynamic objects should declare all their dependencies and how to locate the dependencies. With lazy loading, this dependency information becomes even more important.

Note – Lazy loading can be disabled at runtime by setting the environment variable LD_NOLAZYLOAD to a non-null value.

Providing an Alternative to dlopen()

Lazy loading can provide an alternative to dlopen(3C) and dlsym(3C) use. See "Runtime Linking Programming Interface" on page 84. For example, the following code from libfoo.so.1 verifies an object is loaded, and then calls interfaces provided by that object.

```
void foo()
{
    void * handle;

    if ((handle = dlopen("libbar.so.1", RTLD_LAZY)) != NULL) {
        int (* fptr)();

        if ((fptr = (int (*)())dlsym(handle, "bar1")) != NULL)
            (*fptr)(arg1);
        if ((fptr = (int (*)())dlsym(handle, "bar2")) != NULL)
            (*fptr)(arg2);
        ....
}
```

This code can be simplified if the object that supplies the required interfaces satisfies the following conditions.

- The object can be established as a dependency at link-edit time.
- The object is always available.

By exploiting lazy loading, the same deferred loading of libbar.so.1 can be achieved. In this case, the reference to the function bar1() results in lazy loading the associated dependency. In addition, the use of standard function calls provides for compiler, or lint(1) validation.

```
void foo()
{
    bar1(arg1);
    bar2(arg2);
    ....
}
$ cc -G -o libfoo.so.1 foo.c -L. -zlazyload -zdefs -lbar -R'$ORIGIN'
```

However, this model fails if the object that provides the required interfaces is not always available. In this case, the ability to test for the existence of the dependency, without having to know the dependencies name, is desirable. A means of testing for the availability of a dependency that satisfies a function reference is required.

`dlsym`(3C) with the `RTLD_PROBE` handle can be used to verify the existence, and loading of a dependency. For example, a reference to `bar1()` can verify that the lazy dependency that was established at link-edit time is available. This test can be used to control the reference to functions provided by the dependency in the same manner as `dlopen`(3C) had been used.

```
void foo()
{
    if (dlsym(RTLD_PROBE, "bar1")) {
        bar1(arg1);
        bar2(arg2);
        ....
}
```

This technique provides for safe deferred loading of recorded dependencies, together with standard function call use.

Note – The special handle `RTLD_DEFAULT` provides a mechanism that is similar to using `RTLD_PROBE`. However, the use of `RTLD_DEFAULT` can result in pending lazy loaded objects being processed in an attempt to locate a symbol that does not exist. This loading compensates for objects that have not fully defined their dependencies. However, this compensation can undermine the advantages of a lazy loading.

The use of the `-z defs` option to build any objects that employ lazy loading, is recommended.

Initialization and Termination Routines

Before transferring control to an application, the runtime linker processes any initialization sections found in the application and any loaded dependencies. The initialization sections `.preinit_array`, `.init_array`, and `.init` are created by the link-editor when a dynamic object is built.

The runtime linker executes functions whose addresses are contained in the `.preinit_array` and `.init_array` sections. These functions are executed in the same order in which their addresses appear in the array. The runtime linker executes an `.init` section as an individual function. If an object contains both `.init` and `.init_array` sections, the `.init` section is processed before the functions defined by the `.init_array` section for that object.

A dynamic executable can provide pre-initialization functions in a `.preinit_array` section. These functions are executed after the runtime linker has built the process image and performed relocations but before any other initialization functions. Pre-initialization functions are not permitted in shared objects.

Note – Any `.init` section within the dynamic executable is called from the application by the process startup mechanism supplied by the compiler driver. The `.init` section within the dynamic executable is called last, after all dependency initialization sections are executed.

Dynamic objects can also provide termination sections. The termination sections `.fini_array` and `.fini` are created by the link-editor when a dynamic object is built.

Any termination sections are passed to `atexit`(3C). These termination routines are called when the process calls `exit`(2), or when objects are removed from the running process with `dlclose`(3C).

The runtime linker executes functions whose addresses are contained in the `.fini_array` section. These functions are executed in the reverse order in which their addresses appear in the array. The runtime linker executes a `.fini` section as an individual function. If an object contains both `.fini` and `.fini_array` sections, the functions defined by the `.fini_array` section are processed before the `.fini` section for that object.

Note – Any `.fini` section within the dynamic executable is called from the application by the process termination mechanism supplied by the compiler driver. The `.fini` section of the dynamic executable is called first, before all dependency termination sections are executed.

For more information regarding the creation of initialization and termination sections by the link-editor see "Initialization and Termination Sections" on page 35.

Initialization and Termination Order

To determine the order of executing initialization and termination code within a process at runtime is a complex issue involving dependency analysis. This process has evolved substantially from the original inception of initialization and termination sections. This process attempts to fulfill the expectations of modern languages and current programming techniques. However, scenarios can exist, where user expectations are hard to meet. Understanding these scenarios, and limiting the content of initialization and termination code can provide both flexible and predictable runtime behavior.

Prior to the Solaris 2.6 release, dependency initialization routines were called in *reverse* load order, which is the reverse order of the dependencies displayed with `ldd`(1). Similarly, dependency termination routines were called in load order. However, as dependency hierarchies became more complex, this simple ordering approach became inadequate.

Starting with the Solaris 2.6 release, the runtime linker constructs a topologically sorted list of objects that have been loaded. This list is built from the dependency relationship expressed by each object, together with any symbol bindings that occur outside of the expressed dependencies.

Initialization sections are executed in the reverse topological order of the dependencies. If cyclic dependencies are found, the objects that form the cycle cannot be topologically sorted. The initialization sections of any cyclic dependencies are executed in their reverse load order. Similarly, termination routines are called in the topological order of dependencies and any cyclic dependencies are executed in their load order.

Use `ldd`(1) with the `-i` option to display the initialization order of an object's dependencies. For example, the following dynamic executable and its dependencies exhibit a cyclic dependency:

```
$ dump -Lv B.so.1 | grep NEEDED
[1]     NEEDED      C.so.1
$ dump -Lv C.so.1 | grep NEEDED
[1]     NEEDED      B.so.1
$ dump -Lv main | grep NEEDED
[1]     NEEDED      A.so.1
[2]     NEEDED      B.so.1
[3]     NEEDED      libc.so.1
$ ldd -i main
        A.so.1 =>        ./A.so.1
        B.so.1 =>        ./B.so.1
        libc.so.1 =>     /lib/libc.so.1
        C.so.1 =>        ./C.so.1
        libm.so.2 =>     /lib/libm.so.2

   cyclic dependencies detected, group[1]:
        ./libC.so.1
        ./libB.so.1
```

```
init object=/lib/libc.so.1
init object=./A.so.1
init object=./C.so.1 - cyclic group [1], referenced by:
     ./B.so.1
init object=./B.so.1 - cyclic group [1], referenced by:
     ./C.so.1
```

Caution – Prior to the Solaris 8 10/00 release, the environment variable `LD_BREADTH` could be set to a non-null value to force the runtime linker to execute initialization and termination sections in pre-Solaris 2.6 release order. This functionality has since been disabled, as the initialization dependencies of many applications have become complex and mandate topological sorting. Any `LD_BREADTH` setting is now silently ignored.

Initialization processing is repeated for any objects added to the running process with `dlopen`(3C). Termination processing is also carried out for any objects unloaded from the process as a result of a call to `dlclose`(3C).

Symbol bindings are incorporated as part of dependency analysis because many shared objects exist that do not express their dependencies accurately. The incorporation of symbol bindings with explicit dependencies can help produce a more accurate dependency relationship. However, although it is not recommended, some objects do not express all their dependencies. For these objects, symbol binding information can still be insufficient to determine the objects complete dependencies. The most common model of loading objects uses lazy binding. With this model, only *immediate reference* symbol bindings are processed before initialization processing. Symbol bindings from *lazy references* might still be pending, and can extend the dependency relationships so far established.

As the dependency analysis of an object can be incomplete, and as cyclic dependencies often exist, the runtime linker also provides for dynamic initialization. This initialization attempts to execute any initialization sections before any functions in the same object are called. During lazy symbol binding, the runtime linker determines whether the initialization sections of the object being bound to have been called. If not, the runtime linker calls them before returning from the symbol binding procedure.

Dynamic initialization can not be revealed with `ldd`(1). However, the exact sequence of initialization calls can be observed at runtime by setting the `LD_DEBUG` environment variable to include the token *basic*. See "Debugging Library" on page 96.

Dynamic initialization is only available when processing lazy references. Use of the environment variable `LD_BIND_NOW`, objects built with the `-z now` option, or objects referenced by `dlopen`(3C) with mode `RTLD_NOW`, circumvent any dynamic initialization.

Note – Objects that are pending initialization, and are referenced through dlopen(3C), will be initialized prior to returning control from this function.

The preceding sections describe the various techniques employed to execute initialization and termination sections in a manner that attempts to meet user expectations. However, coding style and link-editing practices should also be employed to simplify the initialization and termination relationships between dependencies. This simplification, helps keep initialization and termination processing predictable, and less prone to any side affects of unexpected dependency ordering.

Keep the content of initialization and termination sections to a minimum. Avoid global constructors by initializing objects at runtime. Reduce the dependency of initialization and termination code on other dependencies. Define the dependency requirements of all dynamic objects. See "Generating a Shared Object Output File" on page 44. Do not express dependencies that are not required. See "Shared Object Processing" on page 30. Avoid cyclic dependencies. Do not depend on the order of an initialization or termination sequence. The ordering of objects can be affected by both shared object and application development. See "Dependency Ordering" on page 108.

Security

Secure processes have some restrictions applied to the evaluation of their dependencies and runpaths to prevent malicious dependency substitution or symbol interposition.

The runtime linker categorizes a process as secure if the issetugid(2) system call returns true for the process.

For 32–bit objects, the default trusted directories that are known to the runtime linker are /lib/secure and /usr/lib/secure. For 64–bit objects, the default trusted directories are /lib/secure/64 and /usr/lib/secure/64. The utility crle(1) can be used to specify additional trusted directories applicable for secure applications. Administrators who use this technique should ensure that the target directories are suitably protected from malicious intrusion.

If an LD_LIBRARY_PATH family environment variable is in effect for a secure process, only the trusted directories specified by this variable are used to augment the runtime linker's search rules. See "Directories Searched by the Runtime Linker" on page 66.

In a secure process, any runpath specifications provided by the application or any of its dependencies is used, provided it is a full pathname, that is, the pathname starts with a '/').

In a secure process, the expansion of the $ORIGIN string is allowed only if it expands to a trusted directory. See "Security" on page 332.

In a secure process, LD_CONFIG is ignored. A secure process uses the default configuration file, if it exists. See crle(1).

In a secure process, LD_SIGNAL is ignored.

Additional objects can be loaded with a secure process using the LD_PRELOAD or LD_AUDIT environment variables. These objects must be specified as full path names or simple file names. Full path names are restricted to known trusted directories. Simple file names, in which no '/' appears in the name, are located subject to the search path restrictions previously described. Simple file names resolve only to known trusted directories.

In a secure process, any dependencies that consist of simple file names are processed using the path name restrictions previously described. Dependencies expressed as full or relative path names are used as is. Therefore, the developer of a secure process should ensure that the target directory referenced as a full or relative path name dependency is suitably protected from malicious intrusion.

When creating a secure process, do not use relative path names to express dependencies or to construct dlopen(3C) path names. This restriction should be applied to the application and to all dependencies.

Runtime Linking Programming Interface

Dependencies specified during the link-edit of an application are processed by the runtime linker during process initialization. In addition to this mechanism, the application can extend its address space during its execution by binding to additional objects. The application can request the same services of the runtime linker that are used to process the dependencies specified during the link-edit of the application.

This delayed object binding has several advantages:

- By processing an object when it is required rather than during the initialization of an application, startup time can be greatly reduced. In fact, the object might not be required if its services are not needed during a particular run of the application, such as for help or debugging information.
- The application can choose between several different objects, depending on the exact services required, such as for a networking protocol.
- Any objects added to the process address space during execution can be freed after use.

An application can use the following typical scenario to access an additional shared object.

- A shared object is located and added to the address space of a running application using dlopen(3C). Any dependencies that this shared object has are located and added at this time.
- The added shared object and its dependencies are relocated. Any initialization sections within these objects are called.
- The application locates symbols within the added objects using dlsym(3C). The application can then reference the data or call the functions defined by these new symbols.
- After the application has finished with the objects, the address space can be freed using dlclose(3C). Any termination sections within the objects being freed is called at this time.
- Any error conditions that occur as a result of using these runtime linker interface routines can be displayed using dlerror(3C).

The services of the runtime linker are defined in the header file dlfcn.h and are made available to an application by the shared object libc.so.1. In the following example, the file main.c can make reference to any of the dlopen(3C) family of routines, and the application prog can bind to these routines at runtime.

```
$ cc -o prog main.c
```

Note – In previous releases of Solaris, the dynamic linking interfaces were made available by the shared object libdl.so.1. This shared object remains available to support existing dependencies. However, the dynamic linking interfaces are now available from libc.so.1, and thus linking with -ldl is no longer necessary.

Loading Additional Objects

Additional objects can be added to a running process's address space using dlopen(3C). This function takes a path name and a binding mode as arguments, and returns a handle to the application. This handle can be used to locate symbols for use by the application using dlsym(3C).

If the path name is specified as a *simple* file name, one with no '/' in the name, then the runtime linker will use a set of rules to generate an appropriate path name. Path names that contain a '/' will be used as provided.

These search path rules are exactly the same as are used to locate any initial dependencies. See "Directories Searched by the Runtime Linker" on page 66. For example, if the file main.c contains the following code fragment:

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

main(int argc, char ** argv)
```

```
{
        void *  handle;
        .....

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                exit (1);
        }
        .....
```

then to locate the shared object `foo.so.1`, the runtime linker uses any `LD_LIBRARY_PATH` definition present at process initialization, followed by any runpath specified during the link-edit of `prog`. Finally, the runtime linker uses the default locations `/lib` and `/usr/lib` for 32–bit objects, or `/lib/64` and `/usr/lib/64` for 64–bit objects.

If the path name is specified as:

```
        if ((handle = dlopen("./foo.so.1", RTLD_LAZY)) == NULL) {
```

then the runtime linker searches for the file only in the current working directory of the process.

Note – Any shared object specified using `dlopen`(3C) should be referenced by its *versioned* file name. For more information on versioning, see "Coordination of Versioned Filenames" on page 147.

If the required object cannot be located, `dlopen`(3C) returns a `NULL` handle. In this case `dlerror`(3C) can be used to display the true reason for the failure. For example:

```
$ cc -o prog main.c
$ prog
dlopen: ld.so.1: prog: fatal: foo.so.1: open failed: No such \
file or directory
```

If the object being added by `dlopen`(3C) has dependencies on other objects, they too are brought into the process's address space. This process continues until all the dependencies of the specified object are loaded. This dependency tree is referred to as a *group*.

If the object specified by `dlopen`(3C), or any of its dependencies, are already part of the process image, then the objects are not processed any further. A valid handle is returned to the application. This mechanism prevents the same object from being loaded more than once, and enables an application to obtain a handle to itself. For example, if the previous `main.c` example contained the following `dlopen()` call:

```
        if ((handle = dlopen((const char *)0, RTLD_LAZY)) == NULL) {
```

then the handle returned from dlopen(3C) can be used to locate symbols within the application itself, within any of the dependencies loaded as part of the process's initialization, or within any objects added to the process's address space, using a dlopen(3C) that specified the RTLD_GLOBAL flag.

Relocation Processing

As described in Chapter 3, after locating and loading any objects, the runtime linker must process each object and perform any necessary relocations. Any objects brought into the process's address space with dlopen(3C) must also be relocated in the same manner.

For simple applications this process is straightforward. However, for users who have more complex applications with many dlopen(3C) calls involving many objects, possibly with common dependencies, this process can be quite important.

Relocations can be categorized according to when they occur. The default behavior of the runtime linker is to process all immediate reference relocations at initialization and all lazy references during process execution, a mechanism commonly referred to as lazy binding.

This same mechanism is applied to any objects added with dlopen(3C) when the mode is defined as RTLD_LAZY. An alternative is to require all relocations of an object to be performed immediately when the object is added. You can use a mode of RTLD_NOW, or record this requirement in the object when it is built using the link-editor's -z now option. This relocation requirement is propagated to any dependencies of the object being opened.

Relocations can also be categorized into non-symbolic and symbolic. The remainder of this section covers issues regarding symbolic relocations, regardless of when these relocations occur, with a focus on some of the subtleties of symbol lookup.

Symbol Lookup

If an object acquired by dlopen(3C) refers to a global symbol, the runtime linker must locate this symbol from the pool of objects that make up the process. In the absence of direct binding, a default symbol search model is applied to objects obtained by dlopen(3C). However, the mode of a dlopen(3C), combined with the attributes of the objects that make up the process, provide for alternative symbol search models.

Objects that required direct binding, although maintaining all the attributes described later, search for symbols directly in the associated dependency. See "Direct Binding" on page 72.

Two attributes of an object affect symbol lookup. The first is the requesting object's symbol *search scope*, and the second is the symbol *visibility* offered by each object within the process. An object's search scope can be:

`world`
: The object can look in any other global object within the process.

`group`
: The object can look only in an object of the same *group*. The dependency tree created from an object obtained with `dlopen`(3C), or from an object built using the link-editor's `-B group` option, forms a unique group.

The visibility of a symbol from an object can be:

`global`
: The object's symbols can be referenced from any object that has *world* search scope.

`local`
: The object's symbols can be referenced only from other objects that make up the same group.

By default, objects obtained with `dlopen`(3C) are assigned *world* symbol search scope, and *local* symbol visibility. The section, "Default Symbol Lookup Model" on page 88, uses this default model to illustrate typical object group interactions. The sections "Defining a Global Object" on page 91, "Isolating a Group" on page 92, and "Object Hierarchies" on page 92 show examples of using `dlopen`(3C) modes and file attributes to extend the default symbol lookup model.

Default Symbol Lookup Model

For each object added by `dlopen`(3C) the runtime linker first looks for the symbol in the dynamic executable. The runtime linker then looks in each of the objects provided during the initialization of the process. If the symbol is still not found, the runtime linker continues the search. The runtime linker next looks in the object acquired through the `dlopen`(3C) and in any of its dependencies.

The default symbol lookup model provides for transitioning into a lazy loading environment. If a symbol can not be found in the presently loaded objects, any pending lazy loaded objects are processed in an attempt to locate the symbol. This loading compensates for objects that have not fully defined their dependencies. However, this compensation can undermine the advantages of a lazy loading.

In the following example, the dynamic executable `prog` and the shared object `B.so.1` have the following dependencies.

```
$ ldd prog
        A.so.1 =>        ./A.so.1
$ ldd B.so.1
        C.so.1 =>        ./C.so.1
```

If prog acquires the shared object B.so.1 by dlopen(3C), then any symbol required to relocate the shared objects B.so.1 and C.so.1 will first be looked for in prog, followed by A.so.1, followed by B.so.1, and finally in C.so.1. In this simple case, think of the shared objects acquired through the dlopen(3C) as if they had been added to the end of the original link-edit of the application. For example, the objects referenced in the previous listing can be expressed diagrammatically as shown in the following figure.

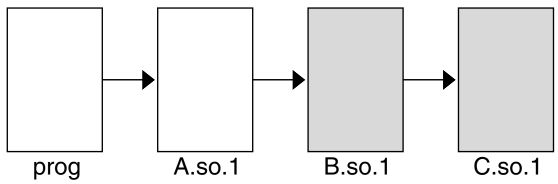


FIGURE 3–1 A Single dlopen() Request

Any symbol lookup required by the objects acquired from the dlopen(3C), shown as shaded blocks, proceeds from the dynamic executable prog through to the final shared object C.so.1.

This symbol lookup is established by the attributes assigned to the objects as they were loaded. Recall that the dynamic executable and all the dependencies loaded with it are assigned global symbol visibility, and that the new objects are assigned world symbol search scope. Therefore, the new objects are able to look for symbols in the original objects. The new objects also form a unique group in which each object has local symbol visibility. Therefore, each object within the group can look for symbols within the other group members.

These new objects do not affect the normal symbol lookup required by either the application or its initial object dependencies. For example, if A.so.1 requires a function relocation after the above dlopen(3C) has occurred, the runtime linker's normal search for the relocation symbol is to look in prog and then A.so.1. The runtime linker does not follow through and look in B.so.1 or C.so.1.

This symbol lookup is again a result of the attributes assigned to the objects as they were loaded. The world symbol search scope is assigned to the dynamic executable and all the dependencies loaded with it. This scope does not allow them to look for symbols in the new objects that only offer local symbol visibility.

These symbol search and symbol visibility attributes maintain associations between objects based on their introduction into the process address space, and on any dependency relationship between the objects. Assigning the objects associated with a given dlopen(3C) to a unique group ensures that only objects associated with the same dlopen(3C) are allowed to look up symbols within themselves and their related dependencies.

This concept of defining associations between objects becomes more clear in applications that carry out more than one dlopen(3C). For example, suppose the shared object D.so.1 has the following dependency:

```
$ ldd D.so.1
        E.so.1 =>         ./E.so.1
```

and the prog application used dlopen(3C) to load this shared object in addition to the shared object B.so.1. The following figure illustrates the symbol lookup releationship between the objects.

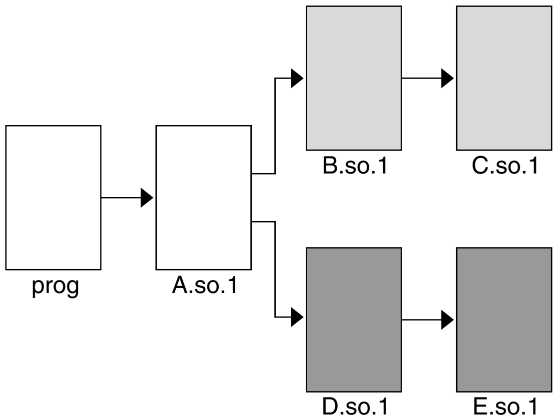


FIGURE 3–2 Multiple dlopen() Requests

Suppose that both B.so.1 and D.so.1 contain a definition for the symbol foo, and both C.so.1 and E.so.1 contain a relocation that requires this symbol. Because of the association of objects to a unique group, C.so.1 is bound to the definition in B.so.1, and E.so.1 is bound to the definition in D.so.1. This mechanism is intended to provide the most intuitive binding of objects obtained from multiple calls to dlopen(3C).

When objects are used in the scenarios that have so far been described, the order in which each dlopen(3C) occurs has no effect on the resulting symbol binding. However, when objects have common dependencies, the resultant bindings can be affected by the order in which the dlopen(3C) calls are made.

In the following example, the shared objects O.so.1 and P.so.1 have the same common dependency.

```
$ ldd O.so.1
        Z.so.1 =>        ./Z.so.1
$ ldd P.so.1
        Z.so.1 =>        ./Z.so.1
```

In this example, the prog application will dlopen(3C) each of these shared objects. Because the shared object Z.so.1 is a common dependency of both O.so.1 and P.so.1, Z.so.1 is assigned to both of the groups that are associated with the two dlopen(3C) calls. This relationship is shown in the following figure.

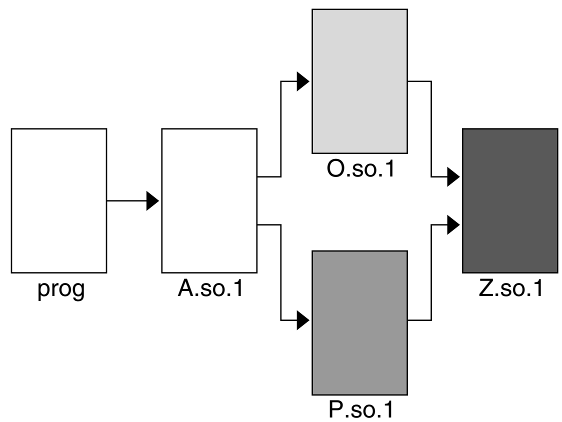


FIGURE 3–3 Multiple `dlopen()` Requests With A Common Dependency

`Z.so.1` is available for both `O.so.1` and `P.so.1` to look up symbols. More importantly, as far as `dlopen`(3C) ordering is concerned, `Z.so.1` is also be able to look up symbols in both `O.so.1` and `P.so.1`.

Therefore, if both `O.so.1` and `P.so.1` contain a definition for the symbol `foo`, which is required for a `Z.so.1` relocation, the actual binding that occurs is unpredictable because it is affected by the order of the `dlopen`(3C) calls. If the functionality of symbol `foo` differs between the two shared objects in which it is defined, the overall outcome of executing code within `Z.so.1` might vary depending on the application's `dlopen`(3C) ordering.

Defining a Global Object

The default assignment of local symbol visibility to the objects obtained by a `dlopen`(3C) can be promoted to global by augmenting the mode argument with the `RTLD_GLOBAL` flag. Under this mode, any objects obtained through a `dlopen`(3C) can be used by any other objects with world symbol search scope to locate symbols.

In addition, any object obtained by `dlopen`(3C) with the `RTLD_GLOBAL` flag is available for symbol lookup using `dlopen()` with a path name whose value is `0`.

Note – If a member of a group has local symbol visibility, and is referenced by another group requiring global symbol visibility, the object's visibility will become a concatenation of both local and global. This promotion of attributes remains even if the global group reference is later removed.

Isolating a Group

The default assignment of world symbol search scope to the objects obtained by a `dlopen`(3C) can be reduced to group by augmenting the mode argument with the `RTLD_GROUP` flag. Under this mode, any objects obtained through a `dlopen`(3C) will only be allowed to look for symbols within their own group.

Using the link-editor's `-B group` option, you can assign the group symbol search scope to objects when they are built.

Note – If a member of a group, has group search capability, and is referenced by another group requiring world search capability, the object's search capability will become a concatenation of both group and world. This promotion of attributes remains even if the world group reference is later removed.

Object Hierarchies

If an initial object, obtained from a `dlopen`(3C), was to use `dlopen`(3C) to open a secondary object, both objects would be assigned to a unique group. This situation can prevent either object from locating symbols from one another.

In some implementations the initial object has to export symbols for the relocation of the secondary object. This requirement can be satisfied by one of two mechanisms:

- Making the initial object an explicit dependency of the second object
- Use the `RTLD_PARENT` mode flag to `dlopen`(3C) the secondary object

If the initial object is an explicit dependency of the secondary object, the initial object is assigned to the secondary objects' group. The initial object is therefore able to provide symbols for the secondary objects' relocation.

If many objects can use `dlopen`(3C) to open the secondary object, and each of these initial objects must export the same symbols to satisfy the secondary objects' relocation, then the secondary object cannot be assigned an explicit dependency. In this case, the `dlopen`(3C) mode of the secondary object can be augmented with the `RTLD_PARENT` flag. This flag causes the propagation of the secondary objects' group to the initial object in the same manner as an explicit dependency would do.

There is one small difference between these two techniques. If you specify an explicit dependency, the dependency itself becomes part of the secondary objects' `dlopen`(3C) dependency tree, and thus becomes available for symbol lookup with `dlsym`(3C). If you obtain the secondary object with `RTLD_PARENT`, the initial object does not become available for symbol lookup with `dlsym`(3C).

When a secondary object is obtained by `dlopen`(3C) from an initial object with global symbol visibility, the `RTLD_PARENT` mode is both redundant and harmless. This case commonly occurs when `dlopen`(3C) is called from an application or from one of the dependencies of the application.

Obtaining New Symbols

A process can obtain the address of a specific symbol using dlsym(3C). This function takes a *handle* and a *symbol name*, and returns the address of the symbol to the caller. The handle directs the search for the symbol in the following manner:

- A handle can be returned from a dlopen(3C) of a named object. The handle enables symbols to be obtained from the named object and the objects that define its dependency tree. A handle returned using the mode RTLD_FIRST, enables symbols to be obtained only from the named object.
- A handle can be returned from a dlopen(3C) of a path name whose value is 0. The handle enables symbols to be obtained from the initiating object of the associated link-map and the objects that define its dependency tree. Typically, the initiating object is the dynamic executable. This handle also enables symbols to be obtained from any object obtained by a dlopen(3C) with the RTLD_GLOBAL mode, on the associated link-map. A handle returned using the mode RTLD_FIRST, enables symbols to be obtained only from the initiating object of the associated link-map.
- The special handle RTLD_DEFAULT, and RTLD_PROBE enable symbols to be obtained from the initiating object of the associated link-map and objects that define its dependency tree. This handle also enables symbols to be obtained from any object obtained by a dlopen(3C) that belongs to the same group as the caller. Use of RTLD_DEFAULT, or RTLD_PROBE follows the same model as used to resolve a symbolic relocation from the calling object.
- The special handle RTLD_NEXT enables symbols to be obtained from the next associated object on the callers link-map list.

In the following example, which is probably the most common, an application adds additional objects to its address space. The application then uses dlsym(3C) to locate function or data symbols. The application then uses these symbols to call upon services provided in these new objects. The file main.c contains the following code:

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

main()
{
        void *  handle;
        int *   dptr, (* fptr)();

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                exit (1);
        }

        if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
            ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
                (void) printf("dlsym: %s\n", dlerror());
                exit (1);
        }
```

```
        return ((*fptr)(*dptr));
}
```

The symbols foo and bar are searched for in the file foo.so.1, followed by any dependencies that are associated with this file. The function foo is then called with the single argument bar as part of the return() statement.

The application prog, built using the previous file main.c, contains the following dependencies.

```
$ ldd prog
        libc.so.1 =>     /lib/libc.so.1
```

If the file name specified in the dlopen(3C) had the value 0, the symbols foo and bar are searched for in prog, followed by /lib/libc.so.1.

The handle indicates the root at which to start a symbol search. From this root, the search mechanism follows the same model as described in “Symbol Lookup” on page 70.

If the required symbol cannot be located, dlsym(3C) returns a NULL value. In this case, dlerror(3C) can be used to indicate the true reason for the failure. In the following example, the application prog is unable to locate the symbol bar.

```
$ prog
dlsym: ld.so.1: main: fatal: bar: can't find symbol
```

Testing for Functionality

The special handles RTLD_DEFAULT, and RTLD_PROBE enable an application to test for the existence of another symbol. The symbol search follows the same model as used to relocate the calling object. See “Default Symbol Lookup Model” on page 88. For example, if the application prog contained the following code fragment:

```
        if ((fptr = (int (*)())dlsym(RTLD_DEFAULT, "foo")) != NULL)
                (*fptr)();
```

then foo is searched for in prog, followed by /lib/libc.so.1. If this code fragment was contained in the file B.so.1 from the example that is shown in Figure 3–1, then the search for foo continues into B.so.1 and then C.so.1.

This mechanism provides a robust and flexible alternative to the use of undefined weak references, as discussed in “Weak Symbols” on page 44.

Using Interposition

The special handle RTLD_NEXT enables an application to locate the next symbol in a symbol scope. For example, if the application prog contained the following code fragment:

```
        if ((fptr = (int (*)())dlsym(RTLD_NEXT, "foo")) == NULL) {
                (void) printf("dlsym: %s\n", dlerror());
                exit (1);
        }

        return ((*fptr)());
```

then `foo` is searched for in the shared objects associated with `prog`, which in this case is `/lib/libc.so.1`. If this code fragment was contained in the file `B.so.1` from the example that is shown in Figure 3–1, then `foo` is searched for in `C.so.1` only.

Use of `RTLD_NEXT` provides a means to exploit symbol interposition. For example, a function within an object can be interposed upon by a preceding object, which can then augment the processing of the original function. For example, the following code fragment is placed in the shared object `malloc.so.1`:

```
#include    <sys/types.h>
#include    <dlfcn.h>
#include    <stdio.h>

void *
malloc(size_t size)
{
        static void * (* fptr)() = 0;
        char             buffer[50];

        if (fptr == 0) {
                fptr = (void * (*)())dlsym(RTLD_NEXT, "malloc");
                if (fptr == NULL) {
                        (void) printf("dlopen: %s\n", dlerror());
                        return (0);
                }
        }

        (void) sprintf(buffer, "malloc: %#x bytes\n", size);
        (void) write(1, buffer, strlen(buffer));
        return ((*fptr)(size));
}
```

This shared object can be interposed before the system library `/lib/libc.so.1` where `malloc`(3C) usually resides. Any calls to `malloc()` are now interposed upon before the original function is called to complete the allocation:

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog file1.o file2.o ..... -R. malloc.so.1
$ prog
malloc: 0x32 bytes
malloc: 0x14 bytes
..........
```

Alternatively, this same interposition can be achieved using the following:

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog main.c
$ LD_PRELOAD=./malloc.so.1 prog
```

```
malloc: 0x32 bytes
malloc: 0x14 bytes
..........
```

Note – Users of any interposition technique must be careful to handle any possibility of recursion. The previous example formats the diagnostic message using `sprintf`(3C), instead of using `printf`(3C) directly, to avoid any recursion caused by `printf`(3C)'s possible use of `malloc`(3C).

The use of `RTLD_NEXT` within a dynamic executable or preloaded object, provides a predictable and useful interposition technique. Be careful when using this technique in a generic object dependency, as the actual load order of objects is not always predictable.

Debugging Aids

A debugging library and `mdb`(1) module are provided with the Solaris linkers. The debugging library enables you to trace the runtime linking process in more detail. The `mdb`(1) module enables interactive process debugging.

Debugging Library

This debugging library helps you understand, or debug, the execution of applications and dependencies. Although the type of information displayed using this library is expected to remain constant, the exact format of the information might change slightly from release to release.

Some of the debugging output might be unfamiliar to those who do not have an intimate knowledge of the runtime linker. However, many aspects might be of general interest to you.

Debugging is enabled by using the environment variable `LD_DEBUG`. All debugging output is prefixed with the process identifier and by default is directed to the standard error. This environment variable must be augmented with one or more tokens to indicate the type of debugging required.

The tokens available with this debugging option can be displayed by using `LD_DEBUG=help`. Any dynamic executable can be used to solicit this information, as the process itself terminates following the display of the information. For example:

```
$ LD_DEBUG=help prog
11693:
```

```
11693:           For debugging the runtime linking of an application:
11693:                  LD_DEBUG=token1,token2  prog
11693:           enables diagnostics to the stderr.  The additional
11693:           option:
11693:                  LD_DEBUG_OUTPUT=file
11693:           redirects the diagnostics to an output file created
11593:           using the specified name and the process id as a
11693:           suffix.  All diagnostics are prepended with the
11693:           process id.
11693:
11693:
11693: audit     display runtime link-audit processing
11693: basic     provide basic trace information/warnings
11693: bindings  display symbol binding; detail flag shows
11693:             absolute:relative addresses
11693: detail    provide more information in conjunction with other
11693:             options
11693: files     display input file processing (files and libraries)
11693: help      display this help message
11693: libs      display library search paths
11693: move      display move section processing
11693: reloc     display relocation processing
11693: symbols   display symbol table processing
11693: tls       display TLS processing info
11693: unused    display unused/unreferenced files
11693: versions  display version processing
```

This example shows the options meaningful to the runtime linker. The exact options might differ from release to release.

The environment variable `LD_DEBUG_OUTPUT` can be used to specify an output file for use instead of the standard error. The process identifier is added as a suffix to the output file.

Debugging of secure applications is not allowed.

One of the most useful debugging options is to display the symbol bindings that occur at runtime. The following example uses a very trivial dynamic executable that has a dependency on two local shared objects.

```
$ cat bar.c
int bar = 10;
$ cc -o bar.so.1 -K pic -G bar.c

$ cat foo.c
foo(int data)
{
        return (data);
}
$ cc -o foo.so.1 -K pic -G foo.c

$ cat main.c
extern  int     foo();
extern  int     bar;
```

```
main()
{
        return (foo(bar));
}
$ cc -o prog main.c -R/tmp:. foo.so.1 bar.so.1
```

The runtime symbol bindings can be displayed by setting `LD_DEBUG=bindings`:

```
$ LD_DEBUG=bindings prog
11753: .......
11753: binding file=prog to file=./bar.so.1: symbol bar
11753: .......
11753: transferring control: prog
11753: .......
11753: binding file=prog to file=./foo.so.1: symbol foo
11753: .......
```

The symbol `bar`, which is required by an immediate relocation, is bound *before* the application gains control. Whereas the symbol `foo`, which is required by a lazy relocation, is bound *after* the application gains control when the function is first called. This demonstrates the default mode of lazy binding. If the environment variable `LD_BIND_NOW` is set, all symbol bindings occur before the application gains control.

Setting `LD_DEBUG=bindings,detail`, provides additional information regarding the real and relative addresses of the actual binding locations.

When the runtime linker performs a function relocation, it rewrites data associated with the functions `.plt` so that any subsequent calls will go directly to the function. The environment variable `LD_BIND_NOT` can be set to any value to prevent this data update. By using this variable together with the debugging request for detailed bindings, you can get a complete runtime account of all function binding. The output from this combination can be excessive, in which case the performance of the application is degraded.

You can use `LD_DEBUG` to display the various search paths used. For example, the search path mechanism used to locate any dependencies can be displayed by setting `LD_DEBUG=libs`.

```
$ LD_DEBUG=libs prog
11775:
11775: find object=foo.so.1; searching
11775:  search path=/tmp:.  (RPATH from file prog)
11775:  trying path=/tmp/foo.so.1
11775:  trying path=./foo.so.1
11775:
11775: find object=bar.so.1; searching
11775:  search path=/tmp:.  (RPATH from file prog)
11775:  trying path=/tmp/bar.so.1
11775:  trying path=./bar.so.1
11775: .......
```

The runpath recorded in the application `prog` affects the search for the two dependencies `foo.so.1` and `bar.so.1`.

In a similar manner, the search paths of each symbol lookup can be displayed by setting `LD_DEBUG=symbols`. If this is combined with a `bindings` request, you can obtain a complete picture of the symbol relocation process.

```
$ LD_DEBUG=bindings,symbols
11782: .......
11782: symbol=bar;  lookup in file=./foo.so.1  [ ELF ]
11782: symbol=bar;  lookup in file=./bar.so.1  [ ELF ]
11782: binding file=prog to file=./bar.so.1: symbol bar
11782: .......
11782: transferring control: prog
11782: .......
11782: symbol=foo;  lookup in file=prog  [ ELF ]
11782: symbol=foo;  lookup in file=./foo.so.1  [ ELF ]
11782: binding file=prog to file=./foo.so.1: symbol foo
11782: .......
```

In the previous example, the symbol `bar` is not searched for in the application `prog`. This is due to an optimization used when processing copy relocations. See "Copy Relocations" on page 125 for more details of this relocation type.

Debugger Module

The debugger module provides a set of `dcmds` and `walkers` that can be loaded under `mdb`(1). This module can be used to inspect various internal data structures of the runtime linker. Much of this information requires familiarity with the internals of the runtime linker, and can change from release to release. However, some elements of these data structures reveal the basic components of a dynamically linked process and can aid general debugging.

The following examples show some simple scenarios of using `mdb`(1) with the runtime linker debugger module.

```
$ cat main.c
#include  <dlfnc.h>

int main()
{
        void *  handle;
        void (* fptr)();

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL)
                return (1);

        if ((fptr = (void (*)())dlsym(handle, "foo")) == NULL)
                return (1);

        (*fptr)();
        return (0);
}
$ cc -o main main.c -R.
```

If mdb(1) has not automatically loaded the debugger module, `ld.so`, explicitly do so. The capabilities of the debugger module can then be inspected.

```
$ mdb main
> ::load ld.so
> ::dmods -l ld.so

ld.so
-----------------------------------------------------------------
  dcmd Bind                  - Display a Binding descriptor
  dcmd Callers               - Display Rt_map CALLERS binding descriptors
  dcmd Depends               - Display Rt_map DEPENDS binding descriptors
  dcmd ElfDyn                - Display Elf_Dyn entry
  dcmd ElfEhdr               - Display Elf_Ehdr entry
  dcmd ElfPhdr               - Display Elf_Phdr entry
  dcmd Groups                - Display Rt_map GROUPS group handles
  dcmd GrpDesc               - Display a Group Descriptor
  dcmd GrpHdl                - Display a Group Handle
  dcmd Handles               - Display Rt_map HANDLES group descriptors
  dcmd List                  - Display entries in a List
  dcmd ListRtmap             - Display a List of Rt_Map's
  dcmd Lm_list               - Display ld.so.1 Lm_list structure
  dcmd Rt_map                - Display ld.so.1 Rt_map structure
  dcmd Rt_maps               - Display list of Rt_map structures
  walk List                  - Walk a List
  walk Rt_maps               - Walk a List of Rt_maps
> ::bp main
> :r
```

Each dynamic object within a process is expressed as a link-map, `Rt_map`, which is maintained on a link-map list. All link-maps for the process can be displayed with `Rt_maps`.

```
> ::Rt_maps
Link-map lists (dynlm_list): 0xffbfe0d0
----------------------------------------------
  Lm_list: 0xff3f6f60  (LM_ID_BASE)
  ----------------------------------------------
    Link-map*  ADDR()     NAME()
    ----------------------------------------------
    0xff3f9040 0x00010000 main
    0xff3f977c 0xff280000 /lib/libc.so.1
  ----------------------------------------------
  Lm_list: 0xff3f6f88  (LM_ID_LDSO)
  ----------------------------------------------
    0xff3f8cc0 0xff3c0000 /lib/ld.so.1
```

An individual link-map can be displayed with `Rt_map`.

```
> 0xff3f9040::Rt_map
Rt_map located at: 0xff3f9040
     NAME: main
     ADDR: 0x00010000         DYN: 0x000207bc
     NEXT: 0xff3f9460        PREV: 0x00000000
      FCT: 0xff3f6f18    TLSMODID:          0
```

```
     INIT: 0x00010710       FINI: 0x0001071c
   GROUPS: 0x00000000     HANDLES: 0x00000000
  DEPENDS: 0xff3f96e8     CALLERS: 0x00000000
    .....
```

The object's `.dynamic` section can be displayed with the `ElfDyn` `dcmd`. The following example shows the first 4 entries.

```
> 0x000207bc,4::ElfDyn
Elf_Dyn located at: 0x207bc
    0x207bc  NEEDED       0x0000010f
Elf_Dyn located at: 0x207c4
    0x207c4  NEEDED       0x00000124
Elf_Dyn located at: 0x207cc
    0x207cc  INIT         0x00010710
Elf_Dyn located at: 0x207d4
    0x207d4  FINI         0x0001071c
```

`mdb`(1) is also very useful for setting deferred break points. In this example, it might be useful to put a break point on the function `foo()`. However, until the `dlopen`(3C) of `foo.so.1` occurs, this symbol isn't known to the debugger. A deferred break point instructs the debugger to set a real breakpoint when the dynamic object is loaded.

```
> ::bp foo.so.1‘foo
> :c
> mdb: You’ve got symbols!
> mdb: stop at foo.so.1‘foo
mdb: target stopped at:
foo.so.1‘foo:   save      %sp, -0x68, %sp
```

At this point, new objects have been loaded:

```
> *ld.so‘lml_main::Rt_maps
Link-map*  ADDR()     NAME()
---------------------------------------------
0xff3f9040 0x00010000 main
0xff3f977c 0xff280000 /lib/libc.so.1
0xff3f9ca4 0xff380000 ./foo.so.1
0xff37006c 0xff260000 ./bar.so.1
```

The link-map for `foo.so.1` shows the handle returned by `dlopen`(3C). You can expand this structure using `Handles`.

```
> 0xff3f9ca4::Handles -v
HANDLES for ./foo.so.1
---------------------------------------------
  HANDLE: 0xff3f9f60 Alist[used 1: total 1]
    ---------------------------------------------
    Group Handle located at: 0xff3f9f28
    ---------------------------------------------
        owner:               ./foo.so.1
        flags: 0x00000000    [ 0 ]
       refcnt:          1    depends: 0xff3f9fa0 Alist[used 2: total 4]
        ---------------------------------------------
        Group Descriptor located at: 0xff3f9fac
```

```
   depend: 0xff3f9ca4    ./foo.so.1
    flags: 0x00000003    [ AVAIL-TO-DLSYM,ADD-DEPENDENCIES ]
---------------------------------------------
Group Descriptor located at: 0xff3f9fd8
   depend: 0xff37006c    ./bar.so.1
    flags: 0x00000003    [ AVAIL-TO-DLSYM,ADD-DEPENDENCIES ]
```

The dependencies of a handle are a list of link-maps that represent the objects of the handle that can satisfy a `dlsym`(3C) request. In this case, the dependencies are `foo.so.1` and `bar.so.1`.

Note – The above examples provide a basic guide to the debugger module capabilities, but the exact commands, usage, and output can change from release to release. Refer to usage and help information for the exact capabilities available on your system.

CHAPTER 4

Shared Objects

Shared objects are one form of output created by the link-editor and are generated by specifying the -G option. In the following example, the shared object libfoo.so.1 is generated from the input file foo.c.

```
$ cc -o libfoo.so.1 -G -K pic foo.c
```

A shared object is an *indivisible* unit that is generated from one or more relocatable objects. Shared objects can be bound with dynamic executables to form a runable process. As their name implies, shared objects can be shared by more than one application. Because of this potentially far-reaching effect, this chapter describes this form of link-editor output in greater depth than has been covered in previous chapters.

For a shared object to be bound to a dynamic executable or another shared object, it must first be available to the link-edit of the required output file. During this link-edit, any input shared objects are interpreted as if they had been added to the logical address space of the output file being produced. All the functionality of the shared object is made available to the output file.

Any input shared objects become dependencies of this output file. A small amount of bookkeeping information is maintained within the output file to describe these dependencies. The runtime linker interprets this information and completes the processing of these shared objects as part of creating a runable process.

The following sections expand upon the use of shared objects within the compilation and runtime environments. These environments are introduced in "Runtime Linking" on page 21.

Naming Conventions

Neither the link-editor nor the runtime linker interprets any file by virtue of its file name. All files are inspected to determine their ELF type (see "ELF Header" on page 182). This information enables the link-editor to deduce the processing requirements of the file. However, shared objects usually follow one of two naming conventions, depending on whether they are being used as part of the compilation environment or the runtime environment.

When used as part of the compilation environment, shared objects are read and processed by the link-editor. Although these shared objects can be specified by explicit file names as part of the command passed to the link-editor, the `-l` option is usually used to take advantage of the link-editor's library search capabilities. See "Shared Object Processing" on page 30.

A shared object that is applicable to this link-editor processing, should be designated with the prefix `lib` and the suffix `.so`. For example, `/lib/libc.so` is the shared object representation of the standard C library made available to the compilation environment. By convention, 64–bit shared objects are placed in a subdirectory of the `lib` directory called `64`. For example, the 64–bit counterpart of `/lib/libc.so.1`, is `/lib/64/libc.so.1`.

When used as part of the runtime environment, shared objects are read and processed by the runtime linker. To allow for change in the exported interface of the shared object over a series of software releases, provide the shared object as a *versioned* file name.

A versioned file name commonly takes the form of a `.so` suffix followed by a version number. For example, `/lib/libc.so.1` is the shared object representation of version *one* of the standard C library made available to the runtime environment.

If a shared object is never intended for use within a compilation environment, its name might drop the conventional `lib` prefix. Examples of shared objects that fall into this category are those used solely with `dlopen`(3C). A suffix of `.so` is still recommended to indicate the actual file type. In additions, a version number is strongly recommended to provide for the correct binding of the shared object across a series of software releases. Chapter 5 describes versioning in more detail.

Note – The shared object name used in a `dlopen`(3C) is usually represented as a *simple* file name, that has no '`/`' in the name. The runtime linker can then use a set of rules to locate the actual file. See "Loading Additional Objects" on page 75 for more details.

Recording a Shared Object Name

The recording of a dependency in a dynamic executable or shared object will, by default, be the file name of the associated shared object as it is referenced by the link-editor. For example, the following dynamic executables, that are built against the same shared object `libfoo.so`, result in different interpretations of the same dependency.

```
$ cc -o ../tmp/libfoo.so -G foo.o
$ cc -o prog main.o -L../tmp -lfoo
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so

$ cc -o prog main.o ../tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   ../tmp/libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   /usr/tmp/libfoo.so
```

As these examples show, this mechanism of recording dependencies can result in inconsistencies due to different compilation techniques. Also, the location of a shared object as referenced during the link-edit might differ from the eventual location of the shared object on an installed system. To provide a more consistent means of specifying dependencies, shared objects can record within themselves the file name by which they should be referenced at runtime.

During the link-edit of a shared object, its runtime name can be recorded within the shared object itself by using the `-h` option. In the following example, the shared object's runtime name `libfoo.so.1`, is recorded within the file itself. This identification is known as an *soname*.

```
$ cc -o ../tmp/libfoo.so -G -K pic -h libfoo.so.1 foo.c
```

The following example shows how the soname recording can be displayed using `dump`(1) and referring to the entry that has the `SONAME` tag.

```
$ dump -Lvp ../tmp/libfoo.so

../tmp/libfoo.so:
[INDEX] Tag      Value
[1]     SONAME   libfoo.so.1
.........
```

When the link-editor processes a shared object that contains an soname, this is the name that is recorded as a dependency within the output file being generated.

If this new version of `libfoo.so` is used during the creation of the dynamic executable `prog` from the previous example, all three methods of creating the executable result in the same dependency recording.

```
$ cc -o prog main.o -L../tmp -lfoo
$ dump -Lv prog | grep NEEDED
```

```
[1]     NEEDED   libfoo.so.1

$ cc -o prog main.o ../tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so.1

$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]     NEEDED   libfoo.so.1
```

In the previous examples, the `-h` option is used to specify a simple file name, that has no '/' in the name. This convention enables the runtime linker to use a set of rules to locate the actual file. See "Locating Shared Object Dependencies" on page 66 for more details.

Inclusion of Shared Objects in Archives

The mechanism of recording an soname within a shared object is essential if the shared object is ever processed from an archive library.

An archive can be built from one or more shared objects and then used to generate a dynamic executable or shared object. Shared objects can be extracted from the archive to satisfy the requirements of the link-edit. Unlike the processing of relocatable objects, which are concatenated to the output file being created, any shared objects extracted from the archive will be recorded as dependencies. See "Archive Processing" on page 28 for more details on the criteria for archive extraction.

The name of an archive member is constructed by the link-editor and is a concatenation of the archive name and the object within the archive. For example:

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ar -r libfoo.a libfoo.so.1
$ cc -o main main.o libfoo.a
$ dump -Lv main | grep NEEDED
[1]     NEEDED   libfoo.a(libfoo.so.1)
```

Because a file with this concatenated name is unlikely to exist at runtime, providing an soname within the shared object is the only means of generating a meaningful runtime file name for the dependency.

Note – The runtime linker does not extract objects from archives. Therefore, in the previous example, the required shared object dependencies must be extracted from the archive and made available to the runtime environment.

Recorded Name Conflicts

When shared objects are used to create a dynamic executable or another shared object, the link-editor performs several consistency checks. These checks ensure that any dependency names recorded in the output file are unique.

Conflicts in dependency names can occur if two shared objects used as input files to a link-edit both contain the same soname. For example:

```
$ cc -o libfoo.so -G -K pic -h libsame.so.1 foo.c
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c
$ cc -o prog main.o -L. -lfoo -lbar
ld: fatal: recording name conflict: file './libfoo.so' and \
    file './libbar.so' provide identical dependency names: libsame.so.1
ld: fatal: File processing errors. No output written to prog
```

A similar error condition occurs if the file name of a shared object that does not have a recorded soname matches the soname of another shared object used during the same link-edit.

If the runtime name of a shared object being generated matches one of its dependencies, the link-editor also reports a name conflict

```
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c -L. -lfoo
ld: fatal: recording name conflict: file './libfoo.so' and \
    -h option provide identical dependency names: libsame.so.1
ld: fatal: File processing errors. No output written to libbar.so
```

Shared Objects With Dependencies

Shared objects can have their own dependencies. The search rules used by the runtime linker to locate shared object dependencies are covered in "Directories Searched by the Runtime Linker" on page 66. If a shared object does not reside in one of the default search directories, then the runtime linker must explicitly be told where to look. For 32–bit objects, the default search directories are `/lib` and `/usr/lib`. For 64–bit objects, the default search directories are `/lib/64` and `/usr/lib/64`. The preferred mechanism of indicating the requirement of a non-default search path, is to record a runpath in the object that has the dependencies. A runpath can be recorded by using the link-editor's `-R` option.

In the following example, the shared object `libfoo.so` has a dependency on `libbar.so`, which is expected to reside in the directory `/home/me/lib` at runtime or, failing that, in the default location.

```
$ cc -o libbar.so -G -K pic bar.c
$ cc -o libfoo.so -G -K pic foo.c -R/home/me/lib -L. -lbar
$ dump -Lv libfoo.so

libfoo.so:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag       Value
[1]     NEEDED    libbar.so
```

```
[2]     RUNPATH  /home/me/lib
.........
```

The shared object is responsible for specifying any runpath required to locate its dependencies. Any runpath specified in the dynamic executable is only used to locate the dependencies of the dynamic executable. These runpaths are not used to locate any dependencies of the shared objects.

The environment variable `LD_LIBRARY_PATH` has a more global scope. Any path names specified using this variable are used by the runtime linker to search for any shared object dependencies. Although useful as a temporary mechanism that influences the runtime linker's search path, the use of this environment variable is strongly discouraged in production software. See "Directories Searched by the Runtime Linker" on page 66 for a more extensive discussion.

Dependency Ordering

When dynamic executables and shared objects have dependencies on the same common shared objects, the order in which the objects are processed can become less predictable.

For example, assume a shared object developer generates `libfoo.so.1` with the following dependencies:

```
$ ldd libfoo.so.1
        libA.so.1 =>     ./libA.so.1
        libB.so.1 =>     ./libB.so.1
        libC.so.1 =>     ./libC.so.1
```

If you create a dynamic executable `prog`, using this shared object, and define an explicit dependency on `libC.so.1`, the resulting shared object order will be:

```
$ cc -o prog main.c -R. -L. -lC -lfoo
$ ldd prog
        libC.so.1 =>     ./libC.so.1
        libfoo.so.1 =>   ./libfoo.so.1
        libA.so.1 =>     ./libA.so.1
        libB.so.1 =>     ./libB.so.1
```

Any requirement on the order of processing the shared object `libfoo.so.1` dependencies would be compromised by the construction of the dynamic executable `prog`.

Developers who place special emphasis on symbol interposition and `.init` section processing should be aware of this potential change in shared object processing order.

Shared Objects as Filters

Shared objects can be defined to act as *filters* for the interfaces they provide. This technique involves associating filter interfaces with an alternative shared object. The alternative shared object supplies symbol definitions at runtime, and is referred to as a *filtee*. A filtee is built in the same manner as any shared object is built.

Filtering provides a mechanism of abstracting the compilation environment from the runtime environment. During a link-edit, a symbol reference that binds to a filter interface, is resolved to the filters symbol definition. At runtime, a symbol reference that binds to a filter interface can be redirected to an alternative shared object. Individual interfaces within a shared object can be defined as filters, or a shared object can define all the interfaces it offers as filters.

Two forms of filtering exist.

Standard filtering
: This filtering requires only a symbol table entry for the interface being filtered. At runtime, the implementation of a filter symbol definition must be provided from a filtee.

  Interfaces are defined to act as standard filters by using the link-editor's `mapfile` directive `FILTER`, or by using the link-editor's `-F` flag. This `mapfile` directive or flag, is qualified with the name of one or more filtees that must supply the symbol definition at runtime.

  If a filtee cannot be processed at runtime, or a filter symbol cannot be located within the filtee, the filtee is skipped for this symbol lookup.

Auxiliary filtering
: This filtering provides a similar mechanism to standard filtering, except the filter provides a fall back implementation corresponding to the auxiliary filter interfaces. At runtime, the implementation of the symbol definition can be provided from a filtee.

  Interfaces are defined to act as auxiliary filters by using the link-editor's `mapfile` directive `AUXILIARY`, or by using the link-editor's `-f` flag. This `mapfile` directive or flag, is qualified with the name of one or more filtees that can supply the symbol definition at runtime.

  If a filtee cannot be processed at runtime, or an auxiliary filter symbol cannot be located within the filtee, the implementation of the symbol within the filter is used.

A shared object that defines all its interfaces to be filters by using the `-F`, or `-f` option, is either a standard or auxiliary filter. A shared object can provide both standard and auxiliary filtering by defining individual filter interfaces by using the `mapfile` directives `FILTER` or `AUXILIARY`.

Generating Standard Filters

To generate a standard filter, you first define a filtee on which the filtering is applied. The following example builds a filtee `filtee.so.1`, that supplies the symbols `foo` and `bar`.

```
$ cat filtee.c
char * bar = "defined in filtee";

char * foo()
{
        return("defined in filtee");
}
$ cc -o filtee.so.1 -G -K pic filtee.c
```

Standard filtering can be provided in one of two ways. To declare all of the interfaces offered by a shared object to be filters, the shared object is defined to be a filter by using the link-editor's `-F` flag. To declare individual interfaces of a shared object to be filters, the individual interfaces are defined to be filters by using a link-editor `mapfile` and the `FILTER` directive.

In the following example, the shared object `filter.so.1`, is defined to be a filter. This shared object offers the symbols `foo` and `bar`, and indicates it is a filter on the filtee `filtee.so.1`. In this example, the environment variable `LD_OPTIONS` is used to circumvent the compiler driver from interpreting the `-F` option.

```
$ cat filter.c
char * bar = 0;

char * foo()
{
    return (0);
}
$ LD_OPTIONS='-F filtee.so.1' \
cc -o filter.so.1 -G -K pic -h filter.so.1 -R. filter.c
$ elfdump -d filter.so.1 | egrep "SONAME|FILTER"
    [2]  SONAME           0xee     filter.so.1
    [3]  FILTER           0xfb     filtee.so.1
```

If the link-editor references the standard filter `filter.so.1`, to create a dynamic executable or shared object, it uses information from the symbol table of the filter to satisfy any symbol resolution. However, at runtime, any reference to the symbols of the filter result in the additional loading of the filtee `filtee.so.1`. The runtime linker uses the filtee to resolve any symbols defined by `filter.so.1`. If the filtee is not found, or a filter symbol is not found in the filtee, the filter is skipped for this symbol lookup.

For example, the following dynamic executable `prog`, references the symbols `foo` and `bar`, which are resolved during link-edit from the filter `filter.so.1`. The execution of `prog` results in `foo` and `bar` being obtained from the filtee `filtee.so.1`, *not* from the filter `filter.so.1`.

```
$ cat main.c
extern char * bar, * foo();
```

```
main()
{
        (void) printf("foo is %s: bar is %s\n", foo(), bar);
}
$ cc -o prog main.c -R. filter.so.1
$ prog
foo is defined in filtee: bar is defined in filtee
```

In the following example, the shared object `filter.so.2` defines one of its interfaces, `foo`, to be a filter on the filtee `filtee.so.1`.

Note – As no source code is supplied for `foo()`, the `mapfile` directive, `FUNCTION`, is used to insure a symbol table entry for `foo` is created.

```
$ cat filter.c
char * bar = "defined in filter";
$ cat mapfile
{
    global:
        foo = FUNCTION FILTER filtee.so.1;
};
$ cc -o filter.so.2 -G -K pic -h filter.so.2 -M mapfile -R. filter.c
$ elfdump -d filter.so.2 | egrep "SONAME|FILTER"
    [2]  SONAME           0xd8     filter.so.2
    [3]  SUNW_FILTER      0xfb     filtee.so.1
$ elfdump -y filter.so.2 | egrep "foo|bar"
    [1]  F    [3] filtee.so.1      foo
   [10]  D        <self>           bar
```

At runtime, any reference to the symbol `foo` of the filter, results in the additional loading of the filtee `filtee.so.1`. The runtime linker uses the filtee to resolve only the symbol `foo` defined by `filter.so.2`. Reference to the symbol `bar` always uses the symbol from `filter.so.2`, as no filtee processing is defined for this symbol.

For example, the following dynamic executable `prog`, references the symbols `foo` and `bar`, which are resolved during link-edit from the filter `filter.so.2`. The execution of `prog` results in `foo` being obtained from the filtee `filtee.so.1`, and `bar` being obtained from the filter `filter.so.2`.

```
$ cc -o prog main.c -R. filter.so.2
$ prog
foo is defined in filtee: bar is defined in filter
```

In these examples, the filtee `filtee.so.1` is uniquely associated to the filter, and is not available to satisfy symbol lookup from any other objects that might be loaded as a consequence of executing `prog`.

Standard filters provide a mechanism for defining a subset interface of an existing shared object, or an interface group spanning a number of existing shared objects. Standard filters also provide a means of redirecting an interface to its implementation. Several standard filters are used in the Solaris OS.

The `/usr/lib/libsys.so.1` filter provides a subset of the standard C library `/usr/lib/libc.so.1`. This subset represents the ABI-conforming functions and data items that reside in the C library that must be imported by a conforming application.

The `/lib/libxnet.so.1` filter uses multiple filtees. This library provides socket and XTI interfaces from `/lib/libsocket.so.1`, `/lib/libnsl.so.1`, and `/lib/libc.so.1`.

`libc.so.1` defines interface filters to the runtime linker. These interfaces provide an abstraction between the symbols referenced in a compilation environment from `libc.so.1`, and the actual implementation binding produced within the runtime environment to `ld.so.1`(1).

`libnsl.so.1` defines the standard filter `gethostname`(3C) against `libc.so.1`. Historically, both `libnsl.so.1` and `libc.so.1` have provided the same implementation for this symbol. By establishing `libnsl.so.1` as a filter, only one implementation of `gethostname()` need exist. The interfaces provided by `libnsl.so.1` also remain compatible with previous releases.

Because the code in a standard filter is never referenced at runtime, adding content to any functions defined as filters is redundant. Any filter code might require relocation, which would result in an unnecessary overhead when processing the filter at runtime. Functions are best defined as empty routines, or directly from a `mapfile`. See "Defining Additional Symbols" on page 46.

Note – The link-editor uses the ELF class of the first relocatable file that is processed to govern the class of object that is created. Use the link-editor's `-64` option to create a 64–bit filter solely from a `mapfile`.

When generating data symbols within a filter, always initialize the data items. The resulting data definition insures that references are established correctly from dynamic executables. Some of the more complex symbol resolutions carried out by the link-editor require knowledge of a symbol's attributes, including the symbol's size. Therefore, you should generate the symbols in the filter so that their attributes match those of the symbols in the filtee. This ensures that the link-editing process analyzes the filter in a manner compatible with the symbol definitions used at runtime. See "Symbol Resolution" on page 38.

Generating Auxiliary Filters

To generate an auxiliary filter, you first define a filtee on which the filtering is applied. The following example builds a filtee `filtee.so.1`, that supplies the symbol `foo`.

```
$ cat filtee.c
char * foo()
{
        return("defined in filtee");
}
$ cc -o filtee.so.1 -G -K pic filtee.c
```

Auxiliary filtering can be provided in one of two ways. To declare all of the interfaces offered by a shared object to be auxiliary filters, the shared object is defined to be an auxiliary filter using the link-editor's `-f` flag. To declare individual interfaces of a shared object to be auxiliary filters, the individual interfaces are defined to be auxiliary filters using a link-editor `mapfile` and the `AUXILIARY` directive.

In the following example, the shared object `filter.so.1` is defined to be an auxiliary filter. This shared object offers the symbols `foo` and `bar`, and indicates it is an auxiliary filter on the filtee `filtee.so.1`. In this example, the environment variable `LD_OPTIONS` is used to circumvent the compiler driver from interpreting the `-f` option.

```
$ cat filter.c
char * bar = "defined in filter";

char * foo()
{
        return ("defined in filter");
}
$ LD_OPTIONS='-f filtee.so.1' \
cc -o filter.so.1 -G -K pic -h filter.so.1 -R. filter.c
$ elfdump -d filter.so.1 | egrep "SONAME|AUXILIARY"
    [2]  SONAME           0xee     filter.so.1
    [3]  AUXILIARY        0xfb     filtee.so.1
```

If the link-editor references the auxiliary filter `filter.so.1`, to create a dynamic executable or shared object, it uses information from the symbol table of the filter to satisfy any symbol resolution. However, at runtime, any reference to the symbols of the filter result in a search for the filtee `filtee.so.1`. If this filtee is found, the runtime linker uses the filtee to resolve any symbols defined by `filter.so.1`. If the filtee is not found, or a symbol from the filter is not found in the filtee, then the original symbol within the filter is used.

For example, the following dynamic executable `prog`, references the symbols `foo` and `bar`, which are resolved during link-edit from the filter `filter.so.1`. The execution of `prog` results in `foo` being obtained from the filtee `filtee.so.1`, *not* from the filter `filter.so.1`. However, `bar` is obtained from the filter `filter.so.1`, as this symbol has no alternative definition in the filtee `filtee.so.1`.

```
$ cat main.c
extern char * bar, * foo();
```

```
main()
{
        (void) printf("foo is %s: bar is %s\n", foo(), bar);
}
$ cc -o prog main.c -R. filter.so.1
$ prog
foo is defined in filtee: bar is defined in filter
```

In the following example, the shared object `filter.so.2` defines one of its interfaces, `foo`, to be an auxiliary filter on the filtee `filtee.so.1`.

```
$ cat filter.c
char * bar = "defined in filter";

char * foo()
{
        return ("defined in filter");
}
$ cat mapfile
{
    global:
        foo = AUXILIARY filtee.so.1;
};
$ cc -o filter.so.2 -G -K pic -h filter.so.2 -M mapfile -R. filter.c
$ elfdump -d filter.so.2 | egrep "SONAME|AUXILIARY"
     [2]  SONAME           0xd8     filter.so.2
     [3]  SUNW_AUXILIARY   0xfb     filtee.so.1
$ elfdump -y filter.so.2 | egrep "foo|bar"
     [1]  A    [3] filtee.so.1      foo
    [10]  D        <self>           bar
```

At runtime, any reference to the symbol `foo` of the filter, results in a search for the filtee `filtee.so.1`. If the filtee is found, it is loaded and used to resolve the symbol `foo` defined by `filter.so.2`. If the filtee is not found, symbol `foo` defined by `filter.so.2` is used. Reference to the symbol `bar` always uses the symbol from `filter.so.2`, as no filtee processing is defined for this symbol.

For example, the following dynamic executable `prog`, references the symbols `foo` and `bar`, which are resolved during link-edit from the filter `filter.so.2`. If the filtee `filtee.so.1` exists, the execution of `prog` results in `foo` being obtained from the filtee `filtee.so.1`, and `bar` being obtained from the filter `filter.so.2`.

```
$ cc -o prog main.c -R. filter.so.2
$ prog
foo is defined in filtee: bar is defined in filter
```

If the filtee `filtee.so.1` does not exist, the execution of `prog` results in `foo` and `bar` being obtained from the filter `filter.so.2`.

```
$ prog
foo is defined in filter: bar is defined in filter
```

In these examples, the filtee `filtee.so.1` is uniquely associated to the filter, and is not available to satisfy symbol lookup from any other objects that might be loaded as a consequence of executing `prog`.

Auxiliary filters provide a mechanism for defining an alternative interface of an existing shared object. This mechanism is used in the Solaris OS to provide optimized functionality within hardware capability, and platform specific shared objects. See "Hardware Capability Specific Shared Objects" on page 325, "Instruction Set Specific Shared Objects" on page 327, and "System Specific Shared Objects" on page 329 for examples.

Note – The environment variable `LD_NOAUXFLTR` can be set to disable the runtime linkers auxiliary filter processing. Because auxiliary filters are frequently employed to provide platform specific optimizations, this option can be useful in evaluating filtee use and their performance impact.

Filtee Processing

The runtime linker's processing of a filter defers the loading of a filtee until a filter symbol is referenced. This implementation is analogous to the filter performing a `dlopen`(3C), using mode `RTLD_LOCAL`, on each of its filtees as they are required. This implementation accounts for differences in dependency reporting that can be produced by tools such as `ldd`(1).

The link-editor's `-z loadfltr` option can be used when creating a filter to cause the immediate processing of its filtees at runtime. In addition, the immediate processing of any filtees within a process can be triggered by setting the `LD_LOADFLTR` environment variable to any value.

Performance Considerations

A shared object can be used by multiple applications within the same system. The performance of a shared object affects the applications that use it and the system as a whole.

Although the actual code within a shared object will directly affect the performance of a running process, the performance issues focused upon here target the runtime processing of the shared object itself. The following sections investigate this processing in more detail by looking at aspects such as text size and purity, together with relocation overhead.

Analyzing Files

Various tools are available to analyze the contents of an ELF file. To display the size of a file use the `size`(1) command. For example:

```
$ size -x libfoo.so.1
59c + 10c + 20 = 0x6c8

$ size -xf libfoo.so.1
..... + 1c(.init) + ac(.text) + c(.fini) + 4(.rodata) + \
..... + 18(.data) + 20(.bss) .....
```

The first example indicates the size of the shared objects *text*, *data*, and *bss*, a categorization used in previous releases of the SunOS operating system.

The ELF format provides a finer granularity for expressing data within a file by organizing the data into *sections*. The second example displays the size of each of the file's loadable sections.

Sections are allocated to units known as *segments*, some of which describe how portions of a file are mapped into memory. See `mmap`(2). These loadable segments can be displayed by using the `dump`(1) command and examining the `LOAD` entries. For example:

```
$ dump -ov libfoo.so.1

libfoo.so.1:
 ***** PROGRAM EXECUTION HEADER *****
Type        Offset      Vaddr       Paddr
Filesz      Memsz       Flags       Align

LOAD        0x94        0x94        0x0
0x59c       0x59c       r-x         0x10000

LOAD        0x630       0x10630     0x0
0x10c       0x12c       rwx         0x10000
```

There are two loadable segments in the shared object `libfoo.so.1`, commonly referred to as the *text* and *data* segments. The text segment is mapped to allow reading and execution of its contents (`r-x`), whereas the data segment is mapped to also allow its contents to be modified (`rwx`). The memory size (`Memsz`) of the data segment differs from the file size (`Filesz`). This difference accounts for the `.bss` section, which is part of the data segment, and is dynamically created when the segment is loaded.

Programmers usually think of a file in terms of the symbols that define the functions and data elements within their code. These symbols can be displayed using `nm`(1). For example:

```
$ nm -x libfoo.so.1

[Index]   Value      Size      Type  Bind  Other Shndx   Name
.........
```

```
[39]    |0x00000538|0x00000000|FUNC |GLOB |0x0   |7       |_init
[40]    |0x00000588|0x00000034|FUNC |GLOB |0x0   |8       |foo
[41]    |0x00000600|0x00000000|FUNC |GLOB |0x0   |9       |_fini
[42]    |0x00010688|0x00000010|OBJT |GLOB |0x0   |13      |data
[43]    |0x0001073c|0x00000020|OBJT |GLOB |0x0   |16      |bss
.........
```

The section that contains a symbol can be determined by referencing the section index (`Shndx`) field from the symbol table and by using `dump`(1) to display the sections within the file. For example:

```
$ dump -hv libfoo.so.1

libfoo.so.1:
           **** SECTION HEADER TABLE ****
[No]    Type    Flags   Addr      Offset    Size      Name
.........
[7]     PBIT    -AI     0x538     0x538     0x1c      .init

[8]     PBIT    -AI     0x554     0x554     0xac      .text

[9]     PBIT    -AI     0x600     0x600     0xc       .fini
.........
[13]    PBIT    WA-     0x10688   0x688     0x18      .data

[16]    NOBI    WA-     0x1073c   0x73c     0x20      .bss
.........
```

The output from both the previous `nm`(1) and `dump`(1) examples shows the association of the functions `_init`, `foo`, and `_fini` to the sections `.init`, `.text` and `.fini`. These sections, because of their read-only nature, are part of the *text* segment.

Similarly, the data arrays `data`, and `bss` are associated with the sections `.data` and `.bss` respectively. These sections, because of their writable nature, are part of the *data* segment.

Note – The previous `dump`(1) display has been simplified for this example.

Underlying System

When an application is built using a shared object, the entire loadable contents of the object are mapped into the virtual address space of that process at runtime. Each process that uses a shared object starts by referencing a single copy of the shared object in memory.

Relocations within the shared object are processed to bind symbolic references to their appropriate definitions. This results in the calculation of true virtual addresses that could not be derived at the time the shared object was generated by the link-editor. These relocations usually result in updates to entries within the process's data segments.

The memory management scheme underlying the dynamic linking of shared objects shares memory among processes at the granularity of a page. Memory pages can be shared as long as they are not modified at runtime. If a process writes to a page of a shared object when writing a data item, or relocating a reference to a shared object, it generates a private copy of that page. This private copy will have no effect on other users of the shared object. However, this page has lost any benefit of sharing between other processes. Text pages that become modified in this manner are referred to as *impure*.

The segments of a shared object that are mapped into memory fall into two basic categories; the *text* segment, which is read-only, and the *data* segment, which is read-write. See "Analyzing Files" on page 116 on how to obtain this information from an ELF file. An overriding goal when developing a shared object is to maximize the text segment and minimize the data segment. This optimizes the amount of code sharing while reducing the amount of processing needed to initialize and use a shared object. The following sections present mechanisms that can help achieve this goal.

Lazy Loading of Dynamic Dependencies

You can defer the loading of a shared object dependency until the dependency is first referenced by establishing the object as lazy loadable. See "Lazy Loading of Dynamic Dependencies" on page 76.

For small applications a typical thread of execution can reference all the applications dependencies. The application loads all of its dependencies whether they are defined lazy loadable or not. However, under lazy loading, dependency processing can be deferred from process startup and spread throughout the process's execution.

For applications with many dependencies, lazy loading often results in some dependencies not being loaded at all. These dependencies are those not referenced for the particular thread of execution.

Position-Independent Code

The compiler can generate position-independent code under the `-K pic` option. Whereas the code within a dynamic executable is usually tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of a process. Because the code is not tied to a specific address, it will execute correctly without page modification at a different address in each process that uses it. This code creates programs that require the smallest amount of page modification at runtime.

When you use position-independent code, relocatable references are generated as an indirection that use data in the shared object's data segment. The text segment code remains read-only, and all relocation updates are applied to corresponding entries within the data segment. See "Global Offset Table (Processor-Specific)" on page 264 and "Procedure Linkage Table (Processor-Specific)" on page 265 for more details on the use of these two sections.

If a shared object is built from code that is not position-independent, the text segment will usually require a large number of relocations to be performed at runtime. Although the runtime linker is equipped to handle this, the system overhead this creates can cause serious performance degradation.

You can identify a shared object that requires relocations against its text segment. Use `dump`(1) and inspect the output for any `TEXTREL` entry. For example:

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ dump -Lv libfoo.so.1 | grep TEXTREL
[9]     TEXTREL  0
```

Note – The value of the `TEXTREL` entry is irrelevant. Its presence in a shared object indicates that text relocations exist.

To prevent the creation of a shared object that contains text relocations use the link-editor's `-z text` flag. This flag causes the link-editor to generate diagnostics indicating the source of any non-position-independent code used as input. Such code results in a failure to generate the intended shared object. For example:

```
$ cc -o libfoo.so.1 -z text -G -R. foo.c
Text relocation remains                         referenced
    against symbol                  offset      in file
foo                                 0x0         foo.o
bar                                 0x8         foo.o
ld: fatal: relocations remain against allocatable but \
non-writable sections
```

Two relocations are generated against the text segment because of the non-position-independent code generated from the file `foo.o`. Where possible, these diagnostics indicate any symbolic references that are required to carry out the relocations. In this case, the relocations are against the symbols `foo` and `bar`.

Another common cause of creating text relocations when generating a shared object is by including hand-written assembler code that has not been coded with the appropriate position-independent prototypes.

Note – You might want to experiment with some simple source files to determine coding sequences that enable position-independence. Use the compilers ability to generate intermediate assembler output.

SPARC: `-K pic` and `-K PIC` Options

For SPARC binaries, a subtle difference between the `-K pic` option and an alternative `-K PIC` option affects references to global offset table entries. See "Global Offset Table (Processor-Specific)" on page 264.

The global offset table is an array of pointers, the size of whose entries are constant for 32–bit (4–bytes) and 64–bit (8–bytes). The following code sequence makes reference to an entry under -K pic:

```
        ld    [%l7 + j], %o0    ! load &j into %o0
```

Where %l7 is the precomputed value of the symbol _GLOBAL_OFFSET_TABLE_ of the object making the reference.

This code sequence provides a 13–bit displacement constant for the global offset table entry. This displacement therefore provides for 2048 unique entries for 32–bit objects, and 1024 unique entries for 64–bit objects. If the creation of an object requires more than the available number of entries, the link-editor produces a fatal error:

```
$ cc -K pic -G -o lobfoo.so.1 a.o b.o ... z.o
ld: fatal: too many symbols require ‘small’ PIC references:
        have 2050, maximum 2048 -- recompile some modules -K PIC.
```

To overcome this error condition, compile some of the input relocatable objects with the -K PIC option. This option provides a 32–bit constant for the global offset table entry:

```
        sethi %hi(j), %g1
        or    %g1, %lo(j), %g1    ! get 32-bit constant GOT offset
        ld    [%l7 + %g1], %o0    ! load &j into %o0
```

You can investigate the global offset table requirements of an object using elfdump(1) with the -G option. You can also examine the processing of these entries during a link-edit using the link-editors debugging tokens -D got,detail.

Ideally, frequently accessed data items benefit from using the -K pic model. You can reference a single entry using both models. However, determining which relocatable objects should be compiled with either option can be time consuming, and the performance improvement realized small. A recompilation of all relocatable objects with the -K PIC option is typically easier.

Remove Unused Material

The inclusion of functions and data that are not used by the object being built, is wasteful. This material bloats the object, which can result in unnecessary relocation overhead and associated paging activity. References to unused dependencies are also wasteful. These references result in the unnecessary loading and processing of other shared objects.

Unused sections are displayed during a link-edit when using the link-editors debugging token -D unused. Sections identified as unused should be removed from the link-edit. Unused sections can be eliminated using the link-editors -z ignore option.

The link-editor identifies a section from a relocatable object as unused if:

- The section is allocatable
- No other sections bind to (relocate) to this section
- The section provides no global symbols

You can improve the link-editor's ability to eliminate sections by defining the shared object's external interfaces. By defining an interface, global symbols that are not defined as part of the interface are reduced to locals. Reduced symbols that are unreferenced from other objects, are now clearly identified as candidates for elimination.

Individual functions and data variables can be eliminated by the link-editor if these items are assigned to their own sections. This section refinement is achieved using compiler options such as `-xF`. Earlier compilers only provided for the assignment of functions to their own sections. Newer compilers have extended the `-xF` syntax to assign data variables to their own sections. Earlier compilers required C++ exception handling to be disabled when using `-xF`. This restriction has been dropped with later compilers.

If all allocatable sections from a relocatable object can be eliminated, the entire file is discarded from the link-edit.

In addition to input file elimination, the link-editor also identifies unused dependencies. A dependency is deemed unused if the dependency is not bound to by the object being produced. An object can be built with the `-z ignore` option to eliminate the recording of unused dependencies.

The `-z ignore` option applies only to the files that follow the option on the link-edit command line. The `-z ignore` option is cancelled with `-z record`.

Maximizing Shareability

As mentioned in "Underlying System" on page 117, only a shared object's text segment is shared by all processes that use the object. The object's data segment typically is not shared. Each process using a shared object, generates a private memory copy of its entire data segment as data items within the segment are written to. Reduce the data segment, either by moving data elements that are never written to the text segment, or by removing the data items completely.

The following sections describe several mechanisms that can be used to reduce the size of the data segment.

Move Read-Only Data to Text

Data elements that are read-only should be moved into the text segment using `const` declarations. For example, the following character string resides in the `.data` section, which is part of the writable data segment:

```
char * rdstr = "this is a read-only string";
```

In contrast, the following character string resides in the `.rodata` section, which is the read-only data section contained within the text segment:

```
const char * rdstr = "this is a read-only string";
```

Reducing the data segment by moving read-only elements into the text segment is admirable. However, moving data elements that require relocations can be counterproductive. For example, examine the following array of strings:

```
char * rdstrs[] = { "this is a read-only string",
                    "this is another read-only string" };
```

A better definition might seem to be:

```
const char * const rdstrs[] = { ..... };
```

This definition ensures that the strings and the array of pointers to these strings are placed in a `.rodata` section. Unfortunately, although the user perceives the array of addresses as read-only, these addresses must be relocated at runtime. This definition therefore results in the creation of text relocations. Representing it as:

```
const char * rdstrs[] = { ..... };
```

insures the array pointers are maintained in the writable data segment where they can be relocated. The array strings are maintained in the read-only text segment.

Note – Some compilers, when generating position-independent code, can detect read-only assignments that result in runtime relocations. These compilers arrange for placing such items in writable segments. For example, `.picdata`.

Collapse Multiply-Defined Data

Data can be reduced by collapsing multiply-defined data. A program with multiple occurrences of the same error messages can be better off by defining one global datum, and have all other instances reference this. For example:

```
const char * Errmsg = "prog: error encountered: %d";

foo()
{
        ......
        (void) fprintf(stderr, Errmsg, error);
        ......
```

The main candidates for this sort of data reduction are strings. String usage in a shared object can be investigated using `strings`(1). The following example generates a sorted list of the data strings within the file `libfoo.so.1`. Each entry in the list is prefixed with the number of occurrences of the string.

```
$ strings -10 libfoo.so.1 | sort | uniq -c | sort -rn
```

Use Automatic Variables

Permanent storage for data items can be removed entirely if the associated functionality can be designed to use automatic (stack) variables. Any removal of permanent storage usually results in a corresponding reduction in the number of runtime relocations required.

Allocate Buffers Dynamically

Large data buffers should usually be allocated dynamically rather than being defined using permanent storage. Often this results in an overall saving in memory, as only those buffers needed by the present invocation of an application are allocated. Dynamic allocation also provides greater flexibility by enabling the buffer's size to change without affecting compatibility.

Minimizing Paging Activity

Any process that accesses a new page causes a page fault, which is an expensive operation. Because shared objects can be used by many processes, any reduction in the number of page faults generated by accessing a shared object will benefit the process and the system as a whole.

Organizing frequently used routines and their data to an adjacent set of pages frequently improves performance because it improves the locality of reference. When a process calls one of these functions, the function might already be in memory because of its proximity to the other frequently used functions. Similarly, grouping interrelated functions improves locality of references. For example, if every call to the function `foo()` results in a call to the function `bar()`, place these functions on the same page. Tools like `cflow`(1), `tcov`(1), `prof`(1) and `gprof`(1) are useful in determining code coverage and profiling.

Isolate related functionality to its own shared object. The standard C library has historically been built containing many unrelated functions. Only rarely, for example, will any single executable use everything in this library. Because of widespread use, determining what set of functions are really the most frequently used is also somewhat difficult. In contrast, when designing a shared object from scratch, maintain only related functions within the shared object. This will improve locality of reference and has the side effect of reducing the object's overall size.

Relocations

In "Relocation Processing" on page 70, the mechanisms by which the runtime linker relocates dynamic executables and shared objects to create a runable process was covered. "Symbol Lookup" on page 70 and "When Relocations Are Performed" on page 73 categorized this relocation processing into two areas to simplify and help illustrate the mechanisms involved. These same two categorizations are also ideally suited for considering the performance impact of relocations.

Symbol Lookup

When the runtime linker needs to look up a symbol, by default it does so by searching in each object. The runtime linker starts with the dynamic executable, and progresses through each shared object in the same order that the objects are loaded. In many instances, the shared object that requires a symbolic relocation turns out to be the provider of the symbol definition.

In this situation, if the symbol used for this relocation is not required as part of the shared object's interface, then this symbol is a strong candidate for conversion to a *static* or *automatic* variable. A symbol reduction can also be applied to removed symbols from a shared objects interface. See "Reducing Symbol Scope" on page 51 for more details. By making these conversions, the link-editor incurs the expense of processing any symbolic relocation against these symbols during the shared object's creation.

The only global data items that should be visible from a shared object are those that contribute to its user interface. Historically this has been a hard goal to accomplish, because global data are often defined to allow reference from two or more functions located in different source files. By applying symbol reduction, unnecessary global symbols can be removed. See "Reducing Symbol Scope" on page 51. Any reduction in the number of global symbols exported from a shared object results in lower relocation costs and an overall performance improvement.

The use of direct bindings can also significantly reduce the symbol lookup overhead within a dynamic process that has many symbolic relocations and many dependencies. See "Direct Binding" on page 72.

When Relocations are Performed

All immediate reference relocations must be carried out during process initialization before the application gains control. However, any lazy reference relocations can be deferred until the first instance of a function being called. Immediate relocations typically result from data references. Therefore, reducing the number of data references also reduces the runtime initialization of a process.

Initialization relocation costs can also be deferred by converting data references into function references. For example, you can return data items by a functional interface. This conversion usually results in a perceived performance improvement because the initialization relocation costs are effectively spread throughout the process's execution. Some of the functional interfaces might never be called by a particular invocation of a process, thus removing their relocation overhead altogether.

The advantage of using a functional interface can be seen in the section, "Copy Relocations" on page 125. This section examines a special, and somewhat expensive, relocation mechanism employed between dynamic executables and shared objects. It also provides an example of how this relocation overhead can be avoided.

Combined Relocation Sections

Relocations by default are grouped by the sections against which they are to be applied. However, when an object is built with the `-z combreloc` option, all but the procedure linkage table relocations are placed into a single common section named `.SUNW_reloc`. See "Procedure Linkage Table (Processor-Specific)" on page 265.

Combining relocation records in this manner enables all `RELATIVE` relocations to be grouped together. All symbolic relocations are sorted by symbol name. The grouping of `RELATIVE` relocations permits optimized runtime processing using the `DT_RELACOUNT/DT_RELCOUNT .dynamic` entries. Sorted symbolic entries help reduce runtime symbol lookup.

Copy Relocations

Shared objects are usually built with position-independent code. References to external data items from code of this type employs indirect addressing through a set of tables. See "Position-Independent Code" on page 118 for more details. These tables are updated at runtime with the real address of the data items. These updated tables enable access to the data without the code itself being modified.

Dynamic executables, however, are generally not created from position-independent code. Any references to external data they make can seemingly only be achieved at runtime by modifying the code that makes the reference. Modifying a read-only text segment is to be avoided. The *copy* relocation technique can solve this reference.

Suppose the link-editor is used to create a dynamic executable, and a reference to a data item is found to reside in one of the dependent shared objects. Space is allocated in the dynamic executable's `.bss`, equivalent in size to the data item found in the shared object. This space is also assigned the same symbolic name as defined in the shared object. Along with this data allocation, the link-editor generates a special copy relocation record that will instruct the runtime linker to copy the data from the shared object to this allocated space within the dynamic executable.

Because the symbol assigned to this space is global, it is used to satisfy any references from any shared objects. The dynamic executable inherits the data item. Any other objects within the process that make reference to this item are bound to this copy. The original data from which the copy is made effectively becomes unused.

The following example of this mechanism uses an array of system error messages that is maintained within the standard C library. In previous SunOS operating system releases, the interface to this information was provided by two global variables, sys_errlist[], and sys_nerr. The first variable provided the array of error message strings, while the second conveyed the size of the array itself. These variables were commonly used within an application in the following manner:

```
$ cat foo.c
extern int      sys_nerr;
extern char *   sys_errlist[];

char *
error(int errnumb)
{
        if ((errnumb < 0) || (errnumb >= sys_nerr))
                return (0);
        return (sys_errlist[errnumb]);
}
```

The application uses the function error to provide a focal point to obtain the system error message associated with the number errnumb.

Examining a dynamic executable built using this code shows the implementation of the copy relocation in more detail:

```
$ cc -o prog main.c foo.c
$ nm -x prog | grep sys_
[36]  |0x00020910|0x00000260|OBJT |WEAK |0x0  |16 |sys_errlist
[37]  |0x0002090c|0x00000004|OBJT |WEAK |0x0  |16 |sys_nerr
$ dump -hv prog | grep bss
[16]    NOBI    WA-    0x20908   0x908    0x268   .bss
$ dump -rv prog

    **** RELOCATION INFORMATION ****

.rela.bss:
Offset      Symndx              Type             Addend

0x2090c     sys_nerr            R_SPARC_COPY      0
0x20910     sys_errlist         R_SPARC_COPY      0
..........
```

The link-editor has allocated space in the dynamic executable's .bss to receive the data represented by sys_errlist and sys_nerr. These data are copied from the C library by the runtime linker at process initialization. Thus, each application that uses these data gets a private copy of the data in its own data segment.

There are two drawbacks to this technique. First, each application pays a performance penalty for the overhead of copying the data at runtime. Second, the size of the data array sys_errlist has now become part of the C library's interface. Suppose the size of this array were to change, perhaps as new error messages are added. Any dynamic executables that reference this array have to undergo a new link-edit to be able to access any of the new error messages. Without this new link-edit, the allocated space within the dynamic executable is insufficient to hold the new data.

These drawbacks can be eliminated if the data required by a dynamic executable are provided by a functional interface. The ANSI C function `strerror`(3C) returns a pointer to the appropriate error string, based on the error number supplied to it. One implementation of this function might be:

```
$ cat strerror.c
static const char * sys_errlist[] = {
        "Error 0",
        "Not owner",
        "No such file or directory",
        ......
};
static const int sys_nerr =
        sizeof (sys_errlist) / sizeof (char *);

char *
strerror(int errnum)
{
        if ((errnum < 0) || (errnum >= sys_nerr))
                return (0);
        return ((char *)sys_errlist[errnum]);
}
```

The error routine in `foo.c` can now be simplified to use this functional interface. This simplification in turn removes any need to perform the original copy relocations at process initialization.

Additionally, because the data are now local to the shared object, the data are no longer part of its interface. The shared object therefore has the flexibility of changing the data without adversely effecting any dynamic executables that use it. Eliminating data items from a shared object's interface generally improves performance while making the shared object's interface and code easier to maintain.

`ldd`(1), when used with either the `-d` or `-r` options, can verify any copy relocations that exist within a dynamic executable.

For example, suppose the dynamic executable `prog` had originally been built against the shared object `libfoo.so.1` and the following two copy relocations had been recorded:

```
$ nm -x prog | grep _size_
[36]    |0x000207d8|0x40|OBJT |GLOB |15  |_size_gets_smaller
[39]    |0x00020818|0x40|OBJT |GLOB |15  |_size_gets_larger
$ dump -rv size | grep _size_
0x207d8     _size_gets_smaller    R_SPARC_COPY      0
0x20818     _size_gets_larger     R_SPARC_COPY      0
```

A new version of this shared object is supplied that contains different data sizes for these symbols:

```
$ nm -x libfoo.so.1 | grep _size_
[26]    |0x00010378|0x10|OBJT |GLOB |8   |_size_gets_smaller
[28]    |0x00010388|0x80|OBJT |GLOB |8   |_size_gets_larger
```

Running ldd(1) against the dynamic executable reveals:

```
$ ldd -d prog
    libfoo.so.1 =>   ./libfoo.so.1
    ...........
    copy relocation sizes differ: _size_gets_smaller
       (file prog size=40; file ./libfoo.so.1 size=10);
       ./libfoo.so.1 size used; possible insufficient data copied
    copy relocation sizes differ: _size_gets_larger
       (file prog size=40; file ./libfoo.so.1 size=80);
       ./prog size used; possible data truncation
```

ldd(1) shows that the dynamic executable will copy as much data as the shared object has to offer, but only accepts as much as its allocated space allows.

Copy relocations can be eliminated by building the application from position-independent code. See "Position-Independent Code" on page 118.

Using -B symbolic

The link-editor's -B symbolic option enables you to bind symbol references to their global definitions within a shared object. This option is historic, in that it was designed for use in creating the runtime linker itself.

Defining an object's interface and reducing non-public symbols to local is preferable to using the -B symbolic option. See "Reducing Symbol Scope" on page 51. Using -B symbolic can often result in some non-intuitive side effects.

If a symbolically bound symbol is interposed upon, then references to the symbol from outside of the symbolically bound object bind to the interposer. The object itself is already bound internally. Essentially, two symbols with the same name are now being referenced from within the process. A symbolically bound data symbol that results in a copy relocation creates the same interposition situation. See "Copy Relocations" on page 125.

Note – Symbolically bound shared objects are identified by the .dynamic flag DF_SYMBOLIC. This flag is informational only. The runtime linker processes symbol lookups from these objects in the same manner as any other object. Any symbolic binding is assumed to have been created at the link-edit phase.

Profiling Shared Objects

The runtime linker can generate profiling information for any shared objects that are processed during the running of an application. The runtime linker is responsible for binding shared objects to an application and is therefore able to intercept any *global* function bindings. These bindings take place through .plt entries. See "When Relocations Are Performed" on page 73 for details of this mechanism.

The LD_PROFILE environment variable specifies the name of a shared object to profile. You can analyze one shared object at a time using this environment variable. The setting of the environment variable can be used to analyze the use of the shared object by one or more applications. In the following example, the use of libc by the single invocation of the command ls(1) is analyzed:

```
$ LD_PROFILE=libc.so.1  ls -l
```

In the following example, the environment variable setting is recorded in a configuration file. This setting causes any application's use of libc to accumulate the analyzed information:

```
# crle  -e LD_PROFILE=libc.so.1
$ ls -l
$ make
$ ...
```

When profiling is enabled, a profile data file is created, if it does not already exist. The file is mapped by the runtime linker. In the above examples, this data file is /var/tmp/libc.so.1.profile. 64–bit libraries require an extended profile format and are written using the .profilex suffix. You can also specify an alternative directory to store the profile data using the LD_PROFILE_OUTPUT environment variable.

This profile data file is used to deposit profil(2) data and call count information related to the use of the specified shared object. This profiled data can be directly examined with gprof(1).

Note – gprof(1) is most commonly used to analyze the gmon.out profile data created by an executable that has been compiled with the -xpg option of cc(1). The runtime linker's profile analysis does not require any code to be compiled with this option. Applications whose dependent shared objects are being profiled should not make calls to profil(2), because this system call does not provide for multiple invocations within the same process. For the same reason, these applications must not be compiled with the -xpg option of cc(1). This compiler-generated mechanism of profiling is also built on top of profil(2).

One of the most powerful features of this profiling mechanism is to enable the analysis of a shared object as used by multiple applications. Frequently, profiling analysis is carried out using one or two applications. However, a shared object, by its very nature, can be used by a multitude of applications. Analyzing how these applications use the shared object can offer insights into where energy might be spent to improvement the overall performance of the shared object.

The following example shows a performance analysis of libc over a creation of several applications within a source hierarchy.

```
$ LD_PROFILE=libc.so.1 ; export LD_PROFILE
$ make
```

```
$ gprof -b /lib/libc.so.1 /var/tmp/libc.so.1.profile
.....

granularity: each sample hit covers 4 byte(s) ....

                                  called/total     parents
index  %time    self descendents  called+self    name      index
                                  called/total     children
.....
-----------------------------------------------
                0.33        0.00      52/29381     _gettxt [96]
                1.12        0.00     174/29381     _tzload [54]
               10.50        0.00    1634/29381     <external>
               16.14        0.00    2512/29381     _opendir [15]
              160.65        0.00   25009/29381     _endopen [3]
[2]     35.0  188.74        0.00   29381         _open [2]
-----------------------------------------------
.....
granularity: each sample hit covers 4 byte(s) ....

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 35.0     188.74   188.74    29381     6.42     6.42  _open [2]
 13.0     258.80    70.06    12094     5.79     5.79  _write [4]
  9.9     312.32    53.52    34303     1.56     1.56  _read [6]
  7.1     350.53    38.21     1177    32.46    32.46  _fork [9]
 ....
```

The special name <*external*> indicates a reference from outside of the address range of the shared object being profiled. Thus, in the above example, 1634 calls to the function `open`(2) within `libc` occurred from the dynamic executables, or from other shared objects, bound with `libc` while the profiling analysis was in progress.

Note – The profiling of shared objects is multithread safe, except in the case where one thread calls `fork`(2) while another thread is updating the profile data information. The use of `fork1`(2) removes this restriction.

CHAPTER 5

Application Binary Interfaces and Versioning

ELF objects processed by the link-editors provide many global symbols to which other objects can bind. These symbols describe the object's application binary interface (ABI). During the evolution of an object, this interface can change due to the addition or deletion of global symbols. In addition, the object's evolution can involve internal implementation changes.

Versioning refers to several techniques that can be applied to an object to indicate interface and implementation changes. These techniques provide for the object's controlled evolution while maintaining backward compatibility.

This chapter describes how an object's ABI can be defined and classifies how changes to this interface can affect backward compatibility. It also presents models by which interface and implementation changes can be incorporated into new releases of the object.

The focus of this chapter is on the runtime interfaces of dynamic executables and shared objects. The techniques used to describe and manage changes within these dynamic objects are presented in generic terms. A common set of naming conventions and versioning scenarios as applied to shared objects can be found in Appendix B.

Developers of dynamic objects must be aware of the ramifications of an interface change and understand how such changes can be managed, especially in regards to maintaining backward compatibility with previously shipped objects.

The global symbols made available by any dynamic object represent the object's public interface. Frequently, the number of global symbols remaining in an object at the end of a link-edit are more than you would like to make public. These global symbols result from the relationship required between relocatable objects used to create the object. They represent private interfaces within the object itself.

Before defining an object's binary interface, you should first determine those global symbols you wish to make publicly available from the object being created. These public symbols can be established using the link-editor's `-M` option and an associated `mapfile` as part of the final link-edit. This technique is introduced in "Reducing Symbol Scope" on page 51. This public interface establishes one or more version definitions within the object being created. These definitions form the foundation for the addition of new interfaces as the object evolves.

The following sections build upon this initial public interface. First though, you should understand how various changes to an interface can be categorized so that they can be managed appropriately.

Interface Compatibility

Many types of change can be made to an object. In their simplest terms, these changes can be categorized into one of two groups:

- *Compatible* updates. These updates are additive, in that all previously available interfaces remain intact.
- *Incompatible* updates. These updates have changed the existing interface in such a way that existing users of the interface can fail or behave incorrectly.

The following table categorizes some common object changes.

TABLE 5–1 Interface Compatibility Examples

Object Change	Update Type
The addition of a symbol	Compatible
The removal of a symbol	Incompatible
The addition of an argument to a non-`varargs`(3EXT) function	Incompatible
The removal of an argument from a function	Incompatible
The change of size, or content, of a data item to a function or as an external definition	Incompatible
A bug fix, or internal enhancement to a function, providing the semantic properties of the object remain unchanged	Compatible
A bug fix, or internal enhancement to a function when the semantic properties of the object change	Incompatible

Because of interposition, the addition of a symbol can constitute an incompatible update, such that the new symbol might conflict with an applications use of that symbol. However, this does seem rare in practice as source-level name space management is commonly used.

Compatible updates can be accommodated by maintaining version definitions internal to the object being generated. Incompatible updates can be accommodated by producing a new object with a new external versioned name. Both of these versioning techniques enable the selective binding of applications. They also enable verification of correct version binding at runtime. These two techniques are explored in more detail in the following sections.

Internal Versioning

A dynamic object can have one or more internal version definitions associated with it. Each version definition is commonly associated with one or more symbol names. A symbol name can only be associated with *one* version definition. However, a version definition can inherit the symbols from other version definitions. Thus, a structure exists to define one or more independent, or related, version definitions within the object being created. As new changes are made to the object, new version definitions can be added to express these changes.

There are two consequences of providing version definitions within a shared object:

- Dynamic objects that are built against a versioned shared object can record their dependency on the version definitions bound to. These version dependencies are verified at runtime to ensure that the appropriate interfaces, or functionality, are available for the correct execution of an application.
- Dynamic objects can select the version definitions of a shared object to bind to during their link-edit. This mechanism enables developers to control their dependency on a shared object to the interfaces, or functionality, that provide the most flexibility.

Creating a Version Definition

Version definitions commonly consist of an association of symbol names to a unique version name. These associations are established within a `mapfile` and supplied to the final link-edit of an object using the link-editor's `-M` option. This technique is introduced in the section "Reducing Symbol Scope" on page 51.

A version definition is established whenever a version name is specified as part of the `mapfile` directive. In the following example, two source files are combined, together with `mapfile` directives, to produce an object with a defined public interface:

```
$ cat foo.c
extern  const char * _foo1;

void foo1()
```

```
{
        (void) printf(_foo1);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";

$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x0001058c|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[35]    |0x00000454|0x00000034|FUNC |GLOB |0x0  |9    |foo1
```

The symbol `foo1` is the only global symbol defined to provide the shared object's public interface. The special auto-reduction directive "*" causes the reduction of all other global symbols to have local binding within the object being generated. This directive is introduced in "Defining Additional Symbols" on page 46. The associated version name, `SUNW_1.1`, causes the generation of a version definition. Thus, the shared object's public interface consists of the internal version definition `SUNW_1.1`, associated with the global symbol `foo1`.

Whenever a version definition, or the auto-reduction directive, are used to generate an object, a base version definition is also created. This base version is defined using the name of the file itself, and is used to associate any reserved symbols generated by the link-editor. See "Generating the Output File" on page 56 for a list of these reserved symbols.

The version definitions contained within an object can be displayed using `pvs`(1) with the `-d` option:

```
$ pvs -d libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
```

The object `libfoo.so.1` has an internal version definition named `SUNW_1.1`, together with a base version definition `libfoo.so.1`.

Note – The link-editor's `-z noversion` option allows symbol reduction to be directed by a `mapfile` but suppresses the creation of version definitions.

Starting with this initial version definition, the object can evolve by adding new interfaces and updated functionality. For example, a new function, `foo2`, together with its supporting data structures, can be added to the object by updating the source files `foo.c` and `data.c`:

```
$ cat foo.c
extern  const char * _foo1;
extern  const char * _foo2;

void foo1()
{
        (void) printf(_foo1);
}

void foo2()
{
        (void) printf(_foo2);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";
const char * _foo2 = "string used by foo2()\n";
```

A new version definition, `SUNW_1.2`, can be created to define a new interface representing the symbol `foo2`. In addition, this new interface can be defined to inherit the original version definition `SUNW_1.1`.

The creation of this new interface is important as it identifies the evolution of the object and enables users to verify and select the interfaces to which they bind. These concepts are covered in more detail in "Binding to a Version Definition" on page 138 and in "Specifying a Version Binding" on page 142.

The following example shows the `mapfile` directives that create these two interfaces.

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                  # Release X+1
        global:
                foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x00010644|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[34]    |0x00010648|0x00000004|OBJT |LOCL |0x0  |17   |_foo2
[36]    |0x000004bc|0x00000034|FUNC |GLOB |0x0  |9    |foo1
[37]    |0x000004f0|0x00000034|FUNC |GLOB |0x0  |9    |foo2
```

The symbols `foo1` and `foo2` are both defined to be part of the shared object's public interface. However, each of these symbols is assigned to a different version definition; `foo1` is assigned to `SUNW_1.1`, and `foo2` is assigned to `SUNW_1.2`.

These version definitions, their inheritance, and their symbol association can be displayed using `pvs`(1) together with the `-d`, `-v` and `-s` options:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1}:
                foo2;
                SUNW_1.2
```

The version definition SUNW_1.2 has a dependency on the version definition SUNW_1.1.

The inheritance of one version definition by another is a useful technique that reduces the version information that will eventually be recorded by any object that binds to a version dependency. Version inheritance is covered in more detail in the section "Binding to a Version Definition" on page 138.

Any internal version definition has an associated *version definition symbol* created. As shown in the previous pvs(1) example, these symbols are displayed when using the -v option.

Creating a Weak Version Definition

Internal changes to an object that do not require the introduction of a new interface definition can be defined by creating a *weak* version definition. Examples of such changes are bug fixes or performance improvements.

Such a version definition is empty, in that it has no global interface symbols associated with it.

For example, suppose the data file data.c, used in the previous examples, is updated to provide more detailed string definitions:

```
$ cat data.c
const char * _foo1 = "string used by function foo1()\n";
const char * _foo2 = "string used by function foo2()\n";
```

A weak version definition can be introduced to identify this change:

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};
```

```
SUNW_1.2 {                    # Release X+1
        global:
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;      # Release X+2

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:       {SUNW_1.2};
```

The empty version definition is signified by the weak label. These weak version definitions enable applications to verify the existence of a particular implementation by binding to the version definition associated with that functionality. The section "Binding to a Version Definition" on page 138 illustrates how these definitions can be used in more detail.

Defining Unrelated Interfaces

The previous examples show how new version definitions added to an object inherit any existing version definitions. You can also create version definitions that are unique and independent. In the following example, two new files, `bar1.c` and `bar2.c`, are added to the object `libfoo.so.1`. These files contribute two new symbols, `bar1` and `bar2`, respectively:

```
$ cat bar1.c
extern  void foo1();

void bar1()
{
        foo1();
}
$ cat bar2.c
extern  void foo2();

void bar2()
{
        foo2();
}
```

These two symbols are intended to define two new public interfaces. Neither of these new interfaces are related to each other. However, each expresses a dependency on the original `SUNW_1.2` interface.

The following `mapfile` definition creates this required association:

```
$ cat mapfile
SUNW_1.1 {                    # Release X
        global:
```

```
                foo1;
        local:
                *;
};

SUNW_1.2 {                      # Release X+1
        global:
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;        # Release X+2

SUNW_1.3a {                     # Release X+3
        global:
                bar1;
} SUNW_1.2;

SUNW_1.3b {                     # Release X+3
        global:
                bar2;
} SUNW_1.2;
```

Again, the version definitions created in `libfoo.so.1` when using this `mapfile`, and their related dependencies, can be inspected using `pvs`(1):

```
$ cc -o libfoo.so.1 -M mapfile -G foo.o bar1.o bar2.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:      {SUNW_1.2};
        SUNW_1.3a:              {SUNW_1.2};
        SUNW_1.3b:              {SUNW_1.2};
```

The following sections explore how these version definition recordings can be used to verify runtime binding requirements and control the binding of an object during its creation.

Binding to a Version Definition

When a dynamic executable or shared object is built against other shared objects, these dependencies are recorded in the resulting object. See "Shared Object Processing" on page 30 and "Recording a Shared Object Name" on page 105 for more details. If these shared object dependencies also contain version definitions, then an associated version dependency is recorded in the object being built.

The following example takes the data files from the previous section and generates a shared object suitable for a compile time environment. This shared object, `libfoo.so.1`, is used in the succeeding binding examples.

```
$ cc -o libfoo.so.1 -h libfoo.so.1 -M mapfile -G foo.o bar.o \
data.o
```

```
$ ln -s libfoo.so.1 libfoo.so
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1}:
                foo2;
                SUNW_1.2;
        SUNW_1.2.1 [WEAK]:       {SUNW_1.2}:
                SUNW_1.2.1;
        SUNW_1.3a:               {SUNW_1.2}:
                bar1;
                SUNW_1.3a;
        SUNW_1.3b:               {SUNW_1.2}:
                bar2;
                SUNW_1.3b
```

In effect, there are six public interfaces being offered by the shared object. Four of these interfaces, `SUNW_1.1`, `SUNW_1.2`, `SUNW_1.3a`, and `SUNW_1.3b`, define exported symbol names. One interface, `SUNW_1.2.1`, describes an internal implementation change to the shared object, and one interface, `libfoo.so.1`, defines several reserved labels. Dynamic objects created with this shared object as a dependency, record the version names of the interfaces the dynamic object binds to.

The following example creates an application that references symbols `foo1` and `foo2`. The versioning dependency information recorded in the application can be examined using `pvs`(1) with the `-r` option.

```
$ cat prog.c
extern void foo1();
extern void foo2();

main()
{
        foo1();
        foo2();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2, SUNW_1.2.1);
```

In this example, the application `prog` has bound to the two interfaces `SUNW_1.1` and `SUNW_1.2`. These interfaces provided the global symbols `foo1` and `foo2` respectively.

Because version definition SUNW_1.1 is defined within libfoo.so.1 as being inherited by the version definition SUNW_1.2, you also need to record the latter version dependency. This normalization of version definition dependencies reduces the amount of version information maintained within an object, and reduces the processing required at runtime.

Because the application prog was built against the shared object's implementation containing the weak version definition SUNW_1.2.1, this dependency is also recorded. Even though this version definition is defined to inherit the version definition SUNW_1.2, the version's weak nature precludes its normalization with SUNW_1.1, and results in a separate dependency recording.

Had there been multiple weak version definitions that inherited from each other, then these definitions will be normalized in the same manner as non-weak version definitions are.

Note – The recording of a version dependency can be suppressed by the link-editor's -z noversion option.

Having recorded these version definition dependencies, the runtime linker validates the existence of the required version definitions in the objects that are bound to when the application is executed. This validation can be displayed using ldd(1) with the -v option. For example, by running ldd(1) on the application prog, the version definition dependencies are shown to be found correctly in the shared object libfoo.so.1:

```
$ ldd -v prog

   find object=libfoo.so.1; required by prog
        libfoo.so.1 =>   ./libfoo.so.1
   find version=libfoo.so.1;
        libfoo.so.1 (SUNW_1.2) =>            ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          ./libfoo.so.1
   ....
```

Note – ldd(1) with the -v option implies *verbose* output. A recursive list of all dependencies, together with all versioning requirements, is generated.

If a non-weak version definition dependency cannot be found, a fatal error occurs during application initialization. Any weak version definition dependency that cannot be found is silently ignored. For example, if the application prog is run in an environment in which libfoo.so.1 only contains the version definition SUNW_1.1, then the following fatal error occurs:

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
```

```
        SUNW_1.1;
$ prog
ld.so.1: prog: fatal: libfoo.so.1: version 'SUNW_1.2' not \
found (required by file prog)
```

Had the application `prog` not recorded any version definition dependencies, the nonexistence of the required interface symbol `foo2` would have manifested itself some time during the execution of the application as a fatal relocation error. This relocation error might occur at process initialization, during process execution, or might not occur at all if the execution path of the application did not call the function `foo2`. See "Relocation Errors" on page 74.

Recording version definition dependencies provides an alternative and immediate indication of the availability of the interfaces required by the application.

If the application `prog` is run in an environment in which `libfoo.so.1` only contains the version definitions `SUNW_1.1` and `SUNW_1.2`, then all non-weak version definition requirements will be satisfied. The absence of the weak version definition `SUNW_1.2.1` is deemed nonfatal, and so no runtime error condition is generated. However, `ldd`(1) can be used to display all version definitions that cannot be found:

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1};
$ prog
string used by foo1()
string used by foo2()
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          (version not found)
        ...........
```

Note – If an object requires a version definition from a given dependency, and at runtime an implementation of that dependency is found that contains no version definition information, the version verification of the dependency will be silently ignored. This policy provides a level of backward compatibility as a transition from non-versioned to versioned shared objects occurs. `ldd`(1), however, can still be used to display any version requirement discrepancies. The environment variable `LD_NOVERSION` can be used to suppress all runtime versioning verification.

Verifying Versions in Additional Objects

Version definition symbols also provide a mechanism for verifying the version requirements of an object obtained by `dlopen`(3C). Any object added to the process's address space using this function will have no automatic version dependency verification carried out by the runtime linker. Thus, the caller of this function is responsible for verifying that any versioning requirements are met.

The presence of a required version definition can be verified by looking up the associated version definition symbol using `dlsym`(3C). The following example adds the shared object `libfoo.so.1` to a process using `dlopen`(3C), and verifies the interface `SUNW_1.2` is available.

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

main()
{
        void *       handle;
        const char * file = "libfoo.so.1";
        const char * vers = "SUNW_1.2";
        ....

        if ((handle = dlopen(file, (RTLD_LAZY | RTLD_FIRST))) == NULL) {
                (void) printf("dlopen: %s\n", dlerror());
                exit (1);
        }

        if (dlsym(handle, vers) == NULL) {
                (void) printf("fatal: %s: version '%s' not found\n",
                    file, vers);
                exit (1);
        }
        ....
```

Specifying a Version Binding

When creating a dynamic object against a shared object containing version definitions, you can instruct the link-editor to limit the binding to specific version definitions. Effectively, the link-editor enables you to control an object's binding to specific interfaces.

An object's binding requirements can be controlled using a *file control directive*. This directive is supplied using the link-editor's `-M` option and an associated `mapfile`. The following syntax for file control directives is available:

name `-` *version* `[` *version* `... ] [ $ADDVERS=`*version* `];`

- *name* – Represents the name of the shared object dependency. This name should match the shared object's compilation environment name as used by the link-editor. See "Library Naming Conventions" on page 31.
- *version* – Represents the version definition name within the shared object that should be made available for binding. Multiple version definitions can be specified.
- `$ADDVERS` – Allows for additional version definitions to be recorded.

This binding control can be useful in the following scenarios:

- When a shared object defines independent, unique versions. This versioning is possible when defining different standards interfaces. An object can be built with binding controls to ensure it only binds to a specific interface.

- When a shared object has been versioned over several software releases. An object can be built with binding controls to restrict its binding to the interfaces available in a previous software release. Thus, an object can run with an old release of the shared object dependency, after being built using the latest release of the shared object.

The following example illustrates the use of the version control mechanism. This example uses the shared object `libfoo.so.1` containing the following version interface definitions:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                foo2;
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
```

The version definitions `SUNW_1.1` and `SUNW_1.2` represent interfaces within `libfoo.so.1` that were made available in software `Release X` and `Release X+1` respectively.

An application can be built to bind only to the interfaces available in `Release X` by using the following version control `mapfile` directive:

```
$ cat mapfile
libfoo.so - SUNW_1.1;
```

For example, suppose you develop an application, `prog`, and want to ensure that the application can run on `Release X`. The application can then only use the interfaces available in that release. If the application mistakenly references the symbol `bar`, then the application is not compliant with the required interface. This condition is signalled by the link-editor as an undefined symbol error:

```
$ cat prog.c
extern void foo1();
extern void bar();

main()
{
        foo1();
        bar();
}
$ cc -o prog prog.c -M mapfile -L. -R. -lfoo
Undefined           first referenced
 symbol                 in file
bar                     prog.o  (symbol belongs to unavailable \
```

```
                                version ./libfoo.so (SUNW_1.2))
ld: fatal: Symbol referencing errors. No output written to prog
```

To be compliant with the SUNW_1.1 interface, you must remove the reference to bar. You can either rework the application to remove the requirement on bar, or add an implementation of bar to the creation of the application.

Note – By default, shared object dependencies encountered as part of a link-edit, are also verified against any file control directives. Use the environment variable LD_NOVERSION to suppress the version verification of any shared object dependencies.

Binding to Additional Version Definitions

To record more version dependencies than would be produced from the normal symbol binding of an object, use the $ADDVERS file control directive. This section describes scenarios where this additional binding might be useful.

From the previous libfoo.so.1 example, assume that in Release X+2, the version definition SUNW_1.1 is subdivided into two standard releases, STAND_A and STAND_B. To preserve compatibility, the SUNW_1.1 version definition must be maintained. In this example, this version definition is expressed as inheriting the two standard definitions:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:           {STAND_A, STAND_B}:
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
        STAND_A:
                foo1;
                STAND_A;
        STAND_B:
                foo2;
                STAND_B;
```

If the only requirement of application prog is the interface symbol foo1, the application will have a single dependency on the version definition STAND_A. This precludes running prog on a system where libfoo.so.1 is less than Release X+2. The version definition STAND_A did not exist in previous releases, even though the interface foo1 did.

The application prog can be built to align its requirement with previous releases by creating a dependency on SUNW_1.1:

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cat prog
extern void foo1();

main()
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.1);
```

This explicit dependency is sufficient to encapsulate the true dependency requirements. This dependency satisfies compatibility with older releases.

"Creating a Weak Version Definition" on page 136 described how weak version definitions can be used to mark an internal implementation change. These version definitions are well suited to indicate bug fixes and performance improvements made to an object. If the existence of a weak version is required, an explicit dependency on this version definition can be generated. The creation of such a dependency can be important when a bug fix, or performance improvement, is critical for the object to function correctly.

From the previous `libfoo.so.1` example, assume a bug fix is incorporated as the weak version definition `SUNW_1.2.1` in software `Release X+3`:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:           {STAND_A, STAND_B}:
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
        STAND_A:
                foo1;
                STAND_A;
        STAND_B:
                foo2;
                STAND_B;
        SUNW_1.2.1 [WEAK]:  {SUNW_1.2}:
                SUNW_1.2.1;
```

Normally, if an application is built against this shared object, the application records a weak dependency on the version definition `SUNW_1.2.1`. This dependency is informational only. This dependency does not cause termination of the application should the version definition not exist in the `libfoo.so.1` used at runtime.

The file control directive $ADDVERS can be used to generate an explicit dependency on a version definition. If this definition is weak, then this explicit reference also causes the version definition to be promoted to a strong dependency.

The application prog can be built to enforce the requirement that the SUNW_1.2.1 interface be available at runtime by using the following file control directive:

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.2.1;
$ cat prog
extern void foo1();

main()
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2.1);
```

prog has been built with an explicit dependency on the interface STAND_A. Because the version definition SUNW_1.2.1 is promoted to a strong version, it is also normalized with the dependency STAND_A. At runtime, if the version definition SUNW_1.2.1 cannot be found, a fatal error is generated.

Note – When working with a small number of dependencies, you can use the link-editor's -u option to explicitly bind to a version definition. Use this option to reference the version definition symbol. However, a symbol reference is nonselective. When working with multiple dependencies, that contain similarly named version definitions, this technique might be insufficient to create explicit bindings.

Version Stability

The various models for binding to versions within an object only remain intact if the individual version definitions remain constant over the life time of the object.

Once a version definition for an object has been created and made public, it must exist in subsequent releases of that object unchanged. Both the version name and the symbols associated with it must remain constant. For this reason, wildcard expansion of the symbol names defined within a version definition is not supported. The number of symbols matching the wildcard might differ over the course of an objects evolution.

Relocatable Objects

Version information can be recorded and used within dynamic objects. Relocatable objects can maintain versioning information in a similar manner. However, there are some subtle differences in how this information is used.

Any version definitions supplied to the link-edit of a relocatable object are recorded in the same format as they are when building dynamic executables or shared objects. However, by default, symbol reduction is not carried out on the object being created. Instead, when the relocatable object is finally used as input to the generation of a dynamic object, the version recording itself will be used to determine the symbol reductions to apply.

In addition, any version definitions found in relocatable objects are propagated to the dynamic object. For an example of version processing in relocatable objects, see "Reducing Symbol Scope" on page 51.

External Versioning

Runtime references to a shared object should always refer to the file's version file name. This is usually expressed as a file name with a version number suffix. When a shared object's interface changes in an incompatible manner, such that it will break old applications, a new shared object should be distributed with a new versioned file name. In addition, the original versioned file name must still be distributed to provide the interfaces required by the old applications.

You should provide shared objects as separate versioned file names within the runtime environment when building applications over a series of software releases. You can then guarantee that the interface against which the applications were built is available for them to bind during their execution.

The following section describes how to coordinate the binding of an interface between the compilation and runtime environments.

Coordination of Versioned Filenames

During a link-edit, the most common method to input shared objects is to use the `-l` option. This option uses the link-editor's library search mechanism to locate shared objects that are prefixed with `lib` and suffixed with `.so`.

However, at runtime, any shared object dependencies should exist in their *versioned* name form. Instead of maintaining two distinct shared objects that follow these naming conventions, create file system links between the two file names.

To make the runtime shared object `libfoo.so.1` available to the compilation environment, provide a symbolic link from the compilation file name to the runtime file name. For example:

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ln -s libfoo.so.1 libfoo.so
$ ls -l libfoo*
```

```
lrwxrwxrwx  1 usr grp          11 1991 libfoo.so -> libfoo.so.1
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
```

Either a symbolic link or hard link can be used. However, as a documentation and diagnostic aid, symbolic links are more useful.

The shared object `libfoo.so.1` has been generated for the runtime environment. Generating a symbolic link `libfoo.so`, has also enabled this file's use in a compilation environment. For example:

```
$ cc -o prog main.o -L. -lfoo
```

The link-editor processes the relocatable object `main.o` with the interface described by the shared object `libfoo.so.1`, which is found by following the symbolic link `libfoo.so`.

Over a series of software releases, new versions of this shared object can be distributed with changed interfaces. The compilation environment can be constructed to use the interface that is applicable by changing the symbolic link. For example:

```
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1993 libfoo.so -> libfoo.so.3
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
-rwxrwxr-x  1 usr grp        3237 1992 libfoo.so.2
-rwxrwxr-x  1 usr grp        3554 1993 libfoo.so.3
```

Three major versions of the shared object are available. Two of these shared objects, `libfoo.so.1` and `libfoo.so.2`, provide the dependencies for existing applications. `libfoo.so.3` offers the latest major release for creating and running new applications.

Using this symbolic link mechanism itself is insufficient to coordinate the correct binding of a shared object from its use in the compilation environment to its requirement in the runtime environment. As the example presently stands, the link-editor records in the dynamic executable `prog` the file name of the shared object it has processed. In this case, that file name is the compilation environment file name.

```
$ dump -Lv prog

prog:
 **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so
.........
```

When the application `prog` is executed, the runtime linker searches for the dependency `libfoo.so`. `prog` binds to the file to which this symbolic link is pointing.

To provide the correct runtime name to be recorded as a dependency, the shared object `libfoo.so.1` should be built with an soname definition. This definition identifies the shared object's runtime name. This name is used as the dependency name by any object that links against this shared object. This definition can be provided using the `-h` option during the link-edit of the shared object itself. For example:

```
$ cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 foo.c
$ ln -s libfoo.so.1 libfoo.so
$ cc -o prog main.o -L. -lfoo
$ dump -Lv prog

prog:
 **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so.1
.........
```

This symbolic link and the soname mechanism have established a robust coordination between the shared-object naming conventions of the compilation and runtime environment. The interface processed during the link-edit is accurately recorded in the output file generated. This recording ensures that the intended interface are furnished at runtime.

Caution – Creating a new externally versioned shared object is a major change. Be sure you understand the complete dependencies of any processes that use this shared object.

For example, an application might have dependencies on `libfoo.so.1` and an externally delivered object `libISV.so.1`. This latter object might also have a dependency on `libfoo.so.1`. If the application is redesigned to use the new interfaces in `libfoo.so.2` without any change to its use of the external object `libISV.so.1`, then both major versions of `libfoo.so` will be brought into the running process. Because the only reason to change the version of `libfoo.so` is to mark an incompatible change, having both versions of the object within a process can lead to incorrect symbol binding and hence undesirable interactions.

CHAPTER 6

Support Interfaces

The link-editors provide a number of support interfaces that enable the monitoring, and in some cases modification, of link-editor and runtime linker processing. These interfaces typically require a more advanced understanding of link-editing concepts than has been described in previous chapters. The following interfaces are described in this chapter:

- *ld-support* – "Link-Editor Support Interface" on page 151
- *rtld-audit* – "Runtime Linker Auditing Interface" on page 157
- *rtld-debugger* – "Runtime Linker Debugger Interface" on page 167

Link-Editor Support Interface

The link-editor performs many operations including the opening of files and the concatenation of sections from these files. Monitoring, and sometimes modifying, these operations can often be beneficial to components of a compilation system.

This section describes the ld-support interface for input file inspection, and to some degree, input file data modification of those files that compose a link-edit. Two applications that employ this interface are the link-editor itself, which uses it to process debugging information within relocatable objects, and the `make`(1S) utility, which uses it to save state information.

The ld-support interface is composed of a support library that offers one or more support interface routines. This library is loaded as part of the link-edit process, and any support routines found are called at various stages of link-editing.

You should be familiar with the `elf`(3ELF) structures and file format when using this interface.

Invoking the Support Interface

The link-editor accepts one or more support libraries provided by either the SGS_SUPPORT environment variable or with the link-editor's -S option. The environment variable consists of a colon separated list of support libraries:

```
$ SGS_SUPPORT=./support.so.1:libldstab.so.1 cc ...
```

The -S option specifies a single support library. Multiple -S options can be specified:

```
$ LD_OPTIONS="-S./support.so.1 -Slibldstab.so.1" cc ...
```

A support library is a shared object. The link-editor opens each support library, in the order they are specified, using dlopen(3C). If both the environment variable and -S option are encountered, then the support libraries specified with the environment variable are processed first. Each support library is then searched, using dlsym(3C), for any support interface routines. These support routines are then called at various stages of link-editing.

A support library must be consistent with the ELF class of the link-editor being invoked, either 32–bit or 64–bit. See "32–Bit and 64–Bit Environments" on page 152 for more details.

Note – By default, the Solaris support library libldstab.so.1 is used by the link-editor to process, and compact, compiler-generated debugging information supplied within input relocatable objects. This default processing is suppressed if you invoke the link-editor with any support libraries specified using the -S option. If the default processing of libldstab.so.1 is required in addition to your support library services, add libldstab.so.1 explicitly to the list of support libraries supplied to the link-editor.

32–Bit and 64–Bit Environments

As described in "32–Bit and 64–Bit Environments" on page 23, the 64–bit link-editor, ld(1), is capable of generating 32–bit objects and the 32–bit link-editor is capable of generating 64–bit objects. Each of these objects has an associated support interface defined.

The support interface for 64–bit objects is similar to that of 32–bit objects, but ends in a *64* suffix, for example ld_start() and ld_start64(). This convention allows both implementations of the support interface to reside in a single shared object libldstab.so.1 of each class, 32–bit and 64–bit.

The SGS_SUPPORT environment variable can be specified with a _32 or _64 suffix, and the link-editor options -z ld32 and -z ld64 can be used to define -S option requirements. These definitions will only be interpreted, respectively, by the 32–bit or 64–bit class of the link-editor. This enables both classes of support library to be specified when the class of the link-editor might not be known.

Support Interface Functions

All ld-support interfaces are defined in the header file `link.h`. All interface arguments are basic C types or ELF types. The ELF data types can be examined with the ELF access library `libelf`. See `elf`(3ELF) for a description of `libelf` contents. The following interface functions are provided by the ld-support interface, and are described in their expected order of use.

`ld_version()`
: This function provides the initial handshake between the link-editor and the support library.

```
uint_t ld_version(uint_t version);
```

The link-editor calls this interface with the highest version of the ld-support interface it is capable of supporting. The support library can verify that this version is sufficient for its use, and return the version it expects to use. This version is normally `LD_SUP_VCURRENT`.

If the support library does not provide this interface, the initial support level `LD_SUP_VERSION1` is assumed.

If the support library returns a version of zero, or a value greater than the ld-support interface the link-editor supports, the support library will not be used.

`ld_start()`
: This function is called after initial validation of the link-editor command line, and indicates the start of input file processing.

```
void ld_start(const char * name, const Elf32_Half type,
        const char * caller);

void ld_start64(const char * name, const Elf64_Half type,
        const char * caller);
```

name is the output file name being created. *type* is the output file type, which is either `ET_DYN`, `ET_REL`, or `ET_EXEC`, as defined in `sys/elf.h`. *caller* is the application calling the interface, which is normally `/usr/ccs/bin/ld`.

`ld_file()`
: This function is called for each input file before any processing of the files data is carried out.

```
void ld_file(const char * name, const Elf_Kind kind, int flags,
        Elf * elf);

void ld_file64(const char * name, const Elf_Kind kind, int flags,
        Elf * elf);
```

name is the input file about to be processed. *kind* indicates the input file type, which is either `ELF_K_AR`, or `ELF_K_ELF`, as defined in `libelf.h`. *flags* indicates how the link-editor obtained the file, and can be one or more of the following definitions:

- LD_SUP_DERIVED – The file name was not explicitly named on the command line. It was either derived from a -l expansion, or it identifies an extracted archive member.
- LD_SUP_EXTRACTED – The file was extracted from an archive.
- LD_SUP_INHERITED – The file was obtained as a dependency of a command-line shared object.

If no *flags* values are specified then the input file has been explicitly named on the command line. *elf* is a pointer to the file's ELF descriptor.

ld_input_section()

This function is called for each section of the input file. This function is called before the link-editor has determined whether the section should be propagated to the output file. This function differs from ld_section() processing, which is only called for sections that contribute to the output file.

```
void ld_input_section(const char * name, Elf32_Shdr ** shdr,
        Elf32_Word sndx, Elf_Data * data, Elf * elf, unit_t flags);

void ld_input_section64(const char * name, Elf64_Shdr ** shdr,
        Elf64_Word sndx, Elf_Data * data, Elf * elf, uint_t flags);
```

name is the input section name. *shdr* is a pointer to the associated section header. *sndx* is the section index within the input file. *data* is a pointer to the associated data buffer. *elf* is a pointer to the file's ELF descriptor. *flags* is reserved for future use.

Modification of the section header is permitted by reallocating a section header and reassigning the **shdr* to the new header. The link-editor uses the section header information that **shdr* points to upon return from ld_input_section() to process the section.

You can modify the data by reallocating the data and reassigning the Elf_Data buffer's d_buf pointer. Any modification to the data should ensure the correct setting of the Elf_Data buffer's d_size element. For input sections that become part of the output image, setting the d_size element to zero effectively removes the data from the output image.

The *flags* field points to a uint_t data field that is initially zero filled. No flags are currently assigned, although the ability to assign flags in future updates, by the link-editor or the support library, is provided.

ld_section()

This function is called for each section of the input file that will be propagated to the output file, but before any processing of the section data is carried out.

```
void ld_section(const char * name, Elf32_Shdr * shdr,
        Elf32_Word sndx, Elf_Data * data, Elf * elf);

void ld_section64(const char * name, Elf64_Shdr * shdr,
        Elf64_Word sndx, Elf_Data * data, Elf * elf);
```

name is the input section name. *shdr* is a pointer to the associated section header. *sndx* is the section index within the input file. *data* is a pointer to the associated data buffer. *elf* is a pointer to the files ELF descriptor.

You can modify the data by reallocating the data itself and reassigning the `Elf_Data` buffer's `d_buf` pointer. Any modification to the data should ensure the correct setting of the `Elf_Data` buffer's `d_size` element. For input sections that will become part of the output image, setting the `d_size` element to zero will effectively remove the data from the output image.

Note – Sections that are stripped using the link-editor's `-s` option, or discarded due to `SHT_SUNW_COMDAT` processing or `SHF_EXCLUDE` identification, are not reported to `ld_section()`. See "COMDAT Section" on page 205, and Table 7–14.

`ld_input_done()`
This function is called when input file processing is complete but before the output file is laid out.

```
void ld_input_done(uint_t flags);
```

The *flags* field points to a `uint_t` data field that is initially zero filled. No flags are currently assigned, although the ability to assign flags in future updates, by the link-editor or the support library, is provided.

`ld_atexit()`
This function is called when the link-edit is complete.

```
void ld_atexit(int status);
```

```
void ld_atexit64(int status);
```

status is the `exit`(2) code that will be returned by the link-editor and is either `EXIT_FAILURE` or `EXIT_SUCCESS`, as defined in `stdlib.h`.

Support Interface Example

The following example creates a support library that prints the section name of any relocatable object file processed as part of a 32–bit link-edit.

```
$ cat support.c
#include        <link.h>
#include        <stdio.h>

static int      indent = 0;

void
ld_start(const char * name, const Elf32_Half type,
    const char * caller)
```

```
{
        (void) printf("output image: %s\n", name);
}

void
ld_file(const char * name, const Elf_Kind kind, int flags,
    Elf * elf)
{
        if (flags & LD_SUP_EXTRACTED)
                indent = 4;
        else
                indent = 2;

        (void) printf("%*sfile: %s\n", indent, "", name);
}

void
ld_section(const char * name, Elf32_Shdr * shdr, Elf32_Word sndx,
    Elf_Data * data, Elf * elf)
{
        Elf32_Ehdr *    ehdr = elf32_getehdr(elf);

        if (ehdr->e_type == ET_REL)
                (void) printf("%*s   section [%ld]: %s\n", indent,
                    "", (long)sndx, name);
}
```

This support library is dependent upon `libelf` to provide the ELF access function `elf32_getehdr`(3ELF) that is used to determine the input file type. The support library is built using:

```
$ cc -o support.so.1 -G -K pic support.c -lelf -lc
```

The following example shows the section diagnostics resulting from the construction of a trivial application from a relocatable object and a local archive library. The invocation of the support library, in addition to default debugging information processing, is brought about by the `-S` option usage.

```
$ LD_OPTIONS="-S./support.so.1 -Slibldstab.so.1" \
cc -o prog main.c -L. -lfoo

output image: prog
  file: /opt/COMPILER/crti.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /opt/COMPILER/crt1.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /opt/COMPILER/values-xt.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: main.o
```

```
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: ./libfoo.a
    file: ./libfoo.a(foo.o)
       section [1]: .shstrtab
       section [2]: .text
       .......
  file: /lib/libc.so
  file: /opt/COMPILER/crtn.o
     section [1]: .shstrtab
     section [2]: .text
     .......
```

Note – The number of sections displayed in this example have been reduced to simplify the output. Also, the files included by the compiler driver can vary.

Runtime Linker Auditing Interface

The rtld-audit interface enables a process to access runtime linking information regarding itself. An example of using this mechanism is the runtime profiling of shared objects described in "Profiling Shared Objects" on page 128.

The rtld-audit interface is implemented as an audit library that offers one or more auditing interface routines. If this library is loaded as part of a process, the audit routines are called by the runtime linker at various stages of process execution. These interfaces enable the audit library to access:

- The search for dependencies. Search paths can be substituted by the audit library.
- Information regarding loaded objects.
- Symbol bindings that occur between loaded objects. These bindings can be altered by the audit library.
- Exploitation of the lazy binding mechanism provided by procedure linkage table entries to allow auditing of function calls and their return values. The arguments to a function and its return value can be modified by the audit library. See "Procedure Linkage Table (Processor-Specific)" on page 265.

Some of these facilities can be achieved by preloading specialized shared objects. However, a preloaded object exists within the same namespace as the objects of a process. This often restricts or complicates the implementation of the preloaded shared object. The rtld-audit interface offers the user a unique namespace in which to execute their audit libraries. This namespace ensures that the audit library does not intrude upon the normal bindings that occur within the process.

Establishing a Namespace

When the runtime linker binds a dynamic executable with its dependencies, it generates a linked list of *link-maps* to describe the process. The link-map structure, defined in /usr/include/sys/link.h, describes each object within the process. The symbol search mechanism required to bind objects of an application traverses this list of link-maps. This link-map list is said to provide the *namespace* for process symbol resolution.

The runtime linker itself is also described by a link-map. This link-map is maintained on a different list from that of the application objects. The runtime linker therefore resides in its own unique name space, which prevents any direct binding of the application to services within the runtime linker. An application can only call upon the public services of the runtime linker by the filter libc.so.1, or libdl.so.1.

The rtld-audit interface employs its own link-map list on which it maintains any audit libraries. The audit libraries are thus isolated from the symbol binding requirements of the application. Inspection of the application link-map list is possible with dlmopen(3C). When used with the RTLD_NOLOAD flag, dlmopen(3C) allows the audit library to query an object's existence without causing its loading.

Two identifiers are defined in /usr/include/link.h to define the application and runtime linker link-map lists:

```
#define LM_ID_BASE      0     /* application link-map list */
#define LM_ID_LDSO      1     /* runtime linker link-map list */
```

Each rtld-audit support library is assigned a unique new link-map identifier.

Creating an Audit Library

An audit library is built like any other shared object. Its unique namespace within a process requires some additional care.

- The library must provide all dependency requirements.
- The library should not use system interfaces that do not provide for multiple instances of the interface within a process.

If the audit library calls printf(3C), then the audit library must define a dependency on libc. See "Generating a Shared Object Output File" on page 44. Because the audit library has a unique namespace, symbol references cannot be satisfied by the libc present in the application being audited. If an audit library has a dependency on libc, then two versions of libc.so.1 are loaded into the process. One version satisfies the binding requirements of the application link-map list. The other version satisfies the binding requirements of the audit link-map list.

To ensure that audit libraries are built with all dependencies recorded, use the link-editors -z defs option.

Some system interfaces assume that they are the only instance of their implementation within a process, for example, signals and `malloc`(3C). Audit libraries should avoid using such interfaces, as doing so can inadvertently alter the behavior of the application.

Note – An audit library can allocate memory using `mapmalloc`(3MALLOC), as this allocation method can exist with any allocation scheme normally employed by the application.

Invoking the Auditing Interface

The rtld-audit interface is enabled by one of two means. Each method implies a scope to the objects that are audited.

- *Global* auditing is enabled using the environment variable `LD_AUDIT`. The audit libraries made available by this method are provided with information regarding all dynamic objects used by the process.
- *Local* auditing is enabled through dynamic entries recorded within an object at the time it was built. The audit libraries made available by this method are provided with information regarding those dynamic objects identified for auditing.

Either method of invocation consists of a string that contains a colon-separated list of shared objects that are loaded by `dlmopen`(3C). Each object is loaded onto its own audit link-map list. Each object is searched for audit routines using `dlsym`(3C). Audit routines that are found are called at various stages during the applications execution.

The rtld-audit interface enables multiple audit libraries to be supplied. Audit libraries that expect to be employed in this fashion should not alter the bindings that would normally be returned by the runtime linker. Alteration of these bindings can produce unexpected results from audit libraries that follow.

Secure applications can only obtain audit libraries from trusted directories. By default, the only trusted directories known to the runtime linker for 32–bit objects are`/lib/secure` and `/usr/lib/secure`. For 64–bit objects, the trusted directories are `/lib/secure/64` and `/usr/lib/secure/64`.

Recording Local Auditors

Local auditing requirements can be established when an object is built using the link-editor options `-p` or `-P`. For example, to audit `libfoo.so.1`, with the audit library `audit.so.1`, record the requirement at link-edit time using the `-p` option:

```
$ cc -G -o libfoo.so.1 -Wl,-paudit.so.1 -K pic foo.c
$ dump -Lv libfoo.so.1 | fgrep AUDIT
[3]    AUDIT      audit.so.1
```

At runtime, the existence of this audit identifier results in the audit library being loaded and information being passed to it regarding the identifying object.

With this mechanism alone, information such as searching for the identifying object occurs prior to the audit library being loaded. To provide as much auditing information as possible, the existence of an object requiring local auditing is propagated to users of that object. For example, if an application is built that depends on `libfoo.so.1`, then the application is identified to indicate its dependencies require auditing:

```
$ cc -o main main.c libfoo.so.1
$ dump -Lv main | fgrep AUDIT
[5]    DEPAUDIT   audit.so.1
```

The auditing enabled via this mechanism results in the audit library being passed information regarding *all* of the applications explicit dependencies. This dependency auditing can also be recorded directly when creating an object by using the link-editor's `-P` option:

```
$ cc -o main main.c -Wl,-Paudit.so.1
$ dump -Lv main | fgrep AUDIT
[5]    DEPAUDIT   audit.so.1
```

Note – Auditing can be disabled at runtime by setting the environment variable `LD_NOAUDIT` to a non-null value.

Audit Interface Functions

The following functions are provided by the rtld-audit interface and are described in their expected order of use.

Note – References to architecture, or object class specific interfaces are reduced to their generic name to simplify the discussions. For example, a reference to `la_symbind32()` and `la_symbind64()` is specified as `la_symbind()`.

`la_version()`

This function provides the initial handshake between the runtime linker and the audit library. This interface must be provided for the audit library to be loaded.

```
uint_t la_version(uint_t version);
```

The runtime linker calls this interface with the highest *version* of the rtld-audit interface it is capable of supporting. The audit library can verify that this version is sufficient for its use, and return the version it expects to use. This version is normally `LAV_CURRENT`, which is defined in `/usr/include/link.h`.

If the audit library return is zero, or a version greater than the rtld-audit interface the runtime linker supports, the audit library is discarded.

`la_activity()`

This function informs an auditor that link-map activity is occurring.

```
void la_activity(uintptr_t * cookie, uint_t flags);
```

cookie identifies the object heading the link-map. *flags* indicates the type of activity as defined in `/usr/include/link.h`:

- `LA_ACT_ADD` – Objects are being added to the link-map list.
- `LA_ACT_DELETE` – Objects are being deleted from the link-map list.
- `LA_ACT_CONSISTENT` – Object activity has been completed.

`la_objsearch()`

This function informs an auditor that an object is about to be searched for.

```
char * la_objsearch(const char * name, uintptr_t * cookie, uint_t flags);
```

name indicates the file or path name being searched for. *cookie* identifies the object initiating the search. *flags* identifies the origin and creation of *name* as defined in `/usr/include/link.h`:

- `LA_SER_ORIG` – This is the initial search name. Typically, this indicates the file name that is recorded as a `DT_NEEDED` entry, or the argument supplied to `dlmopen`(3C).
- `LA_SER_LIBPATH` – The path name has been created from a `LD_LIBRARY_PATH` component.
- `LA_SER_RUNPATH` – The path name has been created from a runpath component.
- `LA_SER_DEFAULT` – The path name has been created from a default search path component.
- `LA_SER_CONFIG` – The path component originated from a configuration file. See `crle`(1).
- `LA_SER_SECURE` – The path component is specific to secure objects.

The return value indicates the search path name that the runtime linker should continue to process. A value of zero indicates that this path should be ignored. An audit library that monitors search paths should return *name*.

`la_objopen()`

This function is called when a new object is loaded by the runtime linker.

```
uint_t la_objopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie);
```

lmp provides the link-map structure that describes the new object. *lmid* identifies the link-map list to which the object has been added. *cookie* provides a pointer to an identifier. This identifier is initialized to the objects *lmp*. This identifier can be modified by the audit library to better identify the object to other rtld-audit interface routines.

The `la_objopen()` function returns a value that indicates the symbol bindings of interest for this object. The return value is a mask of the following values that are defined in `/usr/include/link.h`:

- `LA_FLG_BINDTO` – Audit symbol bindings *to* this object.
- `LA_FLG_BINDFROM` – Audit symbol bindings *from* this object.

These values allow an auditor to select the objects it wants to monitor with `la_symbind()`. A return value of zero indicates that binding information is of no interest for this object.

For example, an auditor can monitor the bindings from `libfoo.so` to `libbar.so`. `la_objopen()` for `libfoo.so` should return `LA_FLG_BINDFROM`. `la_objopen()` for `libbar.so` should return `LA_FLG_BINDTO`.

An auditor can monitor all bindings between `libfoo.so` and `libbar.so`. `la_objopen()` for both objects should return `LA_FLG_BINDFROM` and `LA_FLG_BINDTO`.

An auditor can monitor all bindings to `libbar.so`. `la_objopen()` for `libbar.so` should return `LA_FLG_BINDTO`. All `la_objopen()` calls should return `LA_FLG_BINDFROM`.

`la_objfilter()`

This function is called when a filter loads a new filtee. See "Shared Objects as Filters" on page 109.

```
int la_objfilter(uintptr_t * fltrcook, const char * fltestr,
        uintptr_t * fltecook, uint_t flags);
```

fltrcook identifies the filter. *fltestr* points to the filtee string. *fltecook* identifies the filtee. *flags* is presently unused. `la_objfilter()` is called after `la_objopen()` for both the filter and filtee.

A return value of zero indicates that this filtee should be ignored. An audit library that monitors the use of filters should return a non-zero value.

`la_preinit()`

This function is called once after all objects have been loaded for the application, but before transfer of control to the application occurs.

```
void la_preinit(uintptr_t * cookie);
```

cookie identifies the primary object that started the process, normally the dynamic executable.

`la_symbind()`

This function is called when a binding occurs between two objects that have been tagged for binding notification from `la_objopen()`.

```
uintptr_t la_symbind32(Elf32_Sym * sym, uint_t ndx,
        uintptr_t * refcook, uintptr_t * defcook, uint_t * flags);
```

```
uintptr_t la_symbind64(Elf64_Sym * sym, uint_t ndx,
        uintptr_t * refcook, uintptr_t * defcook, uint_t * flags,
        const char * sym_name);
```

sym is a constructed symbol structure, whose `sym->st_value` indicates the address of the symbol definition being bound. See `/usr/include/sys/elf.h`. `la_symbind32()` adjusts the `sym->st_name` to point to the actual symbol name. `la_symbind64()` leaves `sym->st_name` to be the index into the bound objects string table.

ndx indicates the symbol index within the bound object's dynamic symbol table. *refcook* identifies the object making reference to this symbol. This identifier is the same identifier as passed to the `la_objopen()` function that returned `LA_FLG_BINDFROM`. *defcook* identifies the object defining this symbol. This identifier is the same as passed to the `la_objopen()` that returned `LA_FLG_BINDTO`.

flags points to a data item that can convey information regarding the binding. This data item can also be used to modify the continued auditing of procedure linkage table entries. This value is a mask of the symbol binding flags that are defined in `/usr/include/link.h`.

The following flags can be supplied to `la_symbind()`:

- `LA_SYMB_DLSYM` – The symbol binding occurred as a result of calling `dlsym`(3C).
- `LA_SYMB_ALTVALUE (LAV_VERSION2)` – An alternate value was returned for the symbol value by a previous call to `la_symbind()`.

If `la_pltenter()` or `la_pltexit()` functions exist, these functions are called after `la_symbind()` for procedure linkage table entries. These functions are called each time that the symbol is referenced. See also "Audit Interface Limitations" on page 166.

The following flags can be supplied from `la_symbind()` to alter this default behavior. These flags are applied as a bitwise-inclusive OR with the value pointed to by the *flags* argument.

- `LA_SYMB_NOPLTENTER` – Do *not* call the `la_pltenter()` function for this symbol.
- `LA_SYMB_NOPLTEXIT` – Do *not* call the `la_pltexit()` function for this symbol.

The return value indicates the address to which control should be passed following this call. An audit library that monitors symbol binding should return the value of `sym->st_value` so that control is passed to the bound symbol definition. An audit library can intentionally redirect a symbol binding by returning a different value.

sym_name, which is applicable for `la_symbind64()` only, contains the name of the symbol being processed. This name is available in the `sym->st_name` field for the 32–bit interface.

`la_pltenter()`

These functions are called on a SPARC, SPARCV9, and x86 system respectively. These functions are called when a procedure linkage table entry, between two objects that have been tagged for binding notification, is called.

```
uintptr_t la_sparcv8_pltenter(Elf32_Sym * sym, uint_t ndx,
        uintptr_t * refcook, uintptr_t * defcook,
        La_sparcv8_regs * regs, uint_t * flags);

uintptr_t la_sparcv9_pltenter(Elf64_Sym * sym, uint_t ndx,
        uintptr_t * refcook, uintptr_t * defcook,
        La_sparcv9_regs * regs, uint_t * flags,
        const char * sym_name);

uintptr_t la_i86_pltenter(Elf32_Sym * sym, uint_t ndx,
        uintptr_t * refcook, uintptr_t * defcook,
        La_i86_regs * regs, uint_t * flags);
```

sym, *ndx*, *refcook*, *defcook* and *sym_name* provide the same information as passed to `la_symbind()`.

regs points to the out registers on a SPARC system, and the stack and frame registers on a x86 system, as defined in `/usr/include/link.h`.

flags points to a data item that can convey information regarding the binding. This data item can also be used to modify the continued auditing of procedure linkage table entries. This data item is the same as pointed to by the *flags* from `la_symbind()`

flags points to a data item that can convey information regarding the binding. This data can be used to modify the continuing auditing of this procedure linkage table entry. This data item is the same as pointed to by the *flags* from `la_symbind()`.

The following flags can be supplied from `la_pltenter()` to alter the present auditing behavior. These flags are applied as a bitwise-inclusive OR with the value pointed to by the *flags* argument.

- `LA_SYMB_NOPLTENTER` – `la_pltenter()` is *not* be called again for this symbol.
- `LA_SYMB_NOPLTEXIT` – `la_pltexit()` is *not* be called for this symbol.

The return value indicates the address to which control should be passed following this call. An audit library that monitors symbol binding should return the value of `sym->st_value` so that control is passed to the bound symbol definition. An audit library can intentionally redirect a symbol binding by returning a different value.

`la_pltexit()`

This function is called when a procedure linkage table entry, between two objects that have been tagged for binding notification, returns. This function is called before control reaches the caller.

```
uintptr_t la_pltexit(Elf32_Sym * sym, uint_t ndx, uintptr_t * refcook,
        uintptr_t * defcook, uintptr_t retval);
```

```
uintptr_t la_pltexit64(Elf64_Sym * sym, uint_t ndx, uintptr_t * refcook,
        uintptr_t * defcook, uintptr_t retval, const char * sym_name);
```

sym, *ndx*, *refcook*, *defcook* and *sym_name* provide the same information as passed to `la_symbind()`. *retval* is the return code from the bound function. An audit library that monitors symbol binding should return *retval*. An audit library can intentionally return a different value.

Note – The `la_pltexit()` interface is experimental. See "Audit Interface Limitations" on page 166.

`la_objclose()`
This function is called after any termination code for an object has been executed and prior to the object being unloaded.

```
uint_t la_objclose(uintptr_t * cookie);
```

cookie identifies the object, and was obtained from a previous `la_objopen()`. Any return value is presently ignored.

Audit Interface Example

The following simple example creates an audit library that prints the name of each shared object dependency loaded by the dynamic executable `date`(1).

```
$ cat audit.c
#include        <link.h>
#include        <stdio.h>

uint_t
la_version(uint_t version)
{
        return (LAV_CURRENT);
}

uint_t
la_objopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie)
{
        if (lmid == LM_ID_BASE)
                (void) printf("file: %s loaded\n", lmp->l_name);
        return (0);
}
$ cc -o audit.so.1 -G -K pic -z defs audit.c -lmapmalloc -lc
$ LD_AUDIT=./audit.so.1 date
file: date loaded
file: /lib/libc.so.1 loaded
file: /lib/libm.so.2 loaded
```

```
file: /usr/lib/locale/en_US/en_US.so.2 loaded
Thur Aug  10 17:03:55 PST 2000
```

Audit Interface Demonstrations

A number of demonstration applications that use the rtld-audit interface are provided in the `SUNWosdem` package under `/usr/demo/link_audit`:

`sotruss`
: This demo provides tracing of procedure calls between the dynamic objects of a named application.

`whocalls`
: This demo provides a stack trace for a specified function whenever called by a named application.

`perfcnt`
: This demo traces the amount of time spent in each function for a named application.

`symbindrep`
: This demo reports all symbol bindings performed to load a named application.

`sotruss`(1) and `whocalls`(1) are included in the `SUNWtoo` package. `perfcnt` and `symbindrep` are example programs. They are not intended for use in a production environment.

Audit Interface Limitations

There are some limitations to the use of the `la_pltexit()` family. These limitations stem from the need to insert an extra stack frame between the caller and callee to provide a `la_pltexit()` return value. This requirement is not a problem when calling just the `la_pltenter()` routines, as. In this case, any intervening stack can be cleaned up prior to transferring control to the destination function.

Because of these limitations, `la_pltexit()` should be considered an experimental interface. When in doubt, avoid the use of the `la_pltexit()` routines.

Functions That Directly Inspect the Stack

A small number of functions exist that directly inspect the stack or make assumptions of its state. Some examples of these functions are the `setjmp`(3C) family, `vfork`(2), and any function that returns a structure, not a pointer to a structure. These functions are compromised by the extra stack created to support `la_pltexit()`.

The runtime linker cannot detect functions of this type, and thus the audit library creator is responsible for disabling `la_pltexit()` for such routines.

Runtime Linker Debugger Interface

The runtime linker performs many operations including the mapping of objects into memory and the binding of symbols. Debugging programs often need to access information that describes these runtime linker operations as part of analyzing an application. These debugging programs run as a separate process to the application they are analyzing.

This section describes the rtld-debugger interface for monitoring and modifying a dynamically linked application from another process. The architecture of this interface follows the model used in `libc_db`(3LIB).

When using the rtld-debugger interface, at least two processes are involved:

- One or more *target* processes. The target processes must be dynamically linked and use the runtime linker `/usr/lib/ld.so.1` for 32–bit processes, or `/usr/lib/64/ld.so.1` for 64–bit processes.
- A *controlling* process links with the rtld-debugger interface library and uses it to inspect the dynamic aspects of the target processes. A 64–bit controlling process can debug both 64–bit and 32–bit targets. However, a 32–bit controlling process is limited to 32–bit targets.

The most anticipated use of the rtld-debugger interface is when the controlling process is a debugger and its target is a dynamic executable.

The rtld-debugger interface enables the following activities with a target process:

- Initial rendezvous with the runtime linker.
- Notification of the loading and unloading of dynamic objects.
- Retrieval of information regarding any loaded objects.
- Stepping over procedure linkage table entries.
- Enabling object padding.

Interaction Between Controlling and Target Process

To be able to inspect and manipulate a target process, the rtld-debugger interface employs an *exported* interface, an *imported* interface, and *agents* for communicating between these interfaces.

The controlling process is linked with the rtld-debugger interface provided by `librtld_db.so.1`, and makes requests of the interface exported from this library. This interface is defined in `/usr/include/rtld_db.h`. In turn, `librtld_db.so.1` makes requests of the interface imported from the controlling process. This interaction allows the rtld-debugger interface to:

- Look up symbols in a target process.

- Read and write memory in the target process.

The imported interface consists of a number of proc_service routines that most debuggers already employ to analyze processes. These routines are described in "Debugger Import Interface" on page 177.

The rtld-debugger interface assumes that the process being analyzed is stopped when requests are made of the rtld-debugger interface. If this halt does not occur, data structures within the runtime linker of the target process might not be in a consistent state for examination.

The flow of information between librtld_db.so.1, the controlling process (debugger) and the target process (dynamic executable) is diagrammed in the following figure.

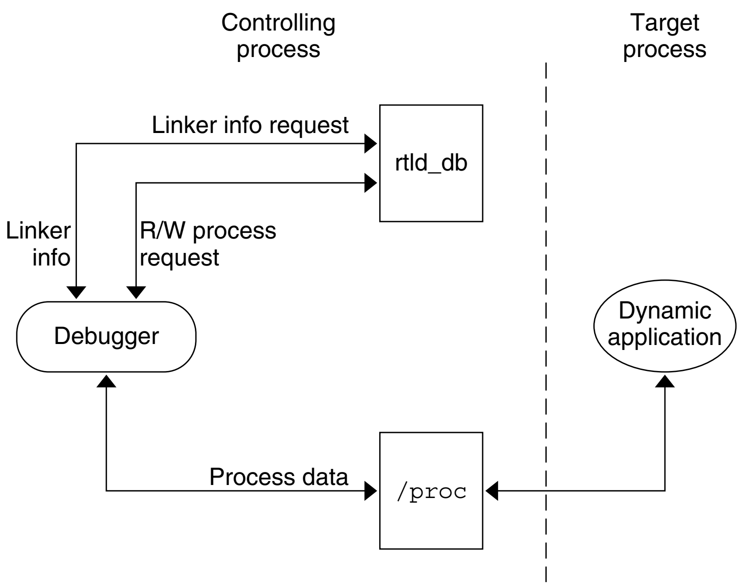


FIGURE 6–1 rtld-debugger Information Flow

Note – The rtld-debugger interface is dependent upon the proc_service interface, /usr/include/proc_service.h, which is considered experimental. The rtld-debugger interface might have to track changes in the proc_service interface as it evolves.

A sample implementation of a controlling process that uses the rtld-debugger interface is provided in the SUNWosdem package under /usr/demo/librtld_db. This debugger, rdb, provides an example of using the proc_service imported interface, and shows the required calling sequence for all librtld_db.so.1 exported interfaces. The following sections describe the rtld-debugger interfaces. More detailed information can be obtained by examining the sample debugger.

Debugger Interface Agents

An agent provides an opaque handle that can describe internal interface structures. The agent also provides a mechanism of communication between the exported and imported interfaces. The rtld-debugger interface is intended to be used by a debugger which can manipulate several processes at the same time, these agents are used to identify the process.

`struct ps_prochandle`
: Is an opaque structure that is created by the controlling process to identify the target process that is passed between the exported and imported interface.

`struct rd_agent`
: Is an opaque structure created by the rtld-debugger interface that identifies the target process that is passed between the exported and imported interface.

Debugger Exported Interface

This section describes the various interfaces exported by the /usr/lib/librtld_db.so.1 audit library. It is broken down into functional groups.

Agent Manipulation Interfaces

`rd_init()`
: This function establishes the rtld-debugger version requirements. The base *version* is defined as RD_VERSION1. The current *version* is always defined by RD_VERSION.

```
rd_err_e rd_init(int version);
```

Version RD_VERSION2, added in the Solaris 8 10/00 release, extends the rd_loadobj_t structure. See the rl_flags, rl_bend and rl_dynamic fields in "Scanning Loadable Objects" on page 171.

Version RD_VERSION3, added in the Solaris 8 01/01 release, extends the rd_plt_info_t structure. See the pi_baddr and pi_flags fields in "Procedure Linkage Table Skipping" on page 175.

If the version requirement of the controlling process is greater than the rtld-debugger interface available, then RD_NOCAPAB is returned.

`rd_new()`

This function creates a new exported interface agent.

```
rd_agent_t * rd_new(struct ps_prochandle * php);
```

php is a cookie created by the controlling process to identify the target process. This cookie is used by the imported interface offered by the controlling process to maintain context, and is opaque to the rtld-debugger interface.

`rd_reset()`

This function resets the information within the agent based off the same *ps_prochandle* structure given to `rd_new()`.

```
rd_err_e rd_reset(struct rd_agent * rdap);
```

This function is called when a target process is restarted.

`rd_delete()`

This function deletes an agent and frees any state associated with it.

```
void rd_delete(struct rd_agent * rdap);
```

Error Handling

The following error states can be returned by the rtld-debugger interface (defined in `rtld_db.h`):

```
typedef enum {
        RD_ERR,
        RD_OK,
        RD_NOCAPAB,
        RD_DBERR,
        RD_NOBASE,
        RD_NODYNAM,
        RD_NOMAPS
} rd_err_e;
```

The following interfaces can be used to gather the error information.

`rd_errstr()`

This function returns a descriptive error string describing the error code *rderr*.

```
char * rd_errstr(rd_err_e rderr);
```

`rd_log()`

This function turns logging on (1) or off (0).

```
void rd_log(const int onoff);
```

When logging is turned on, the imported interface function `ps_plog()` provided by the controlling process, is called with more detailed diagnostic information.

Scanning Loadable Objects

You can obtain information for each object maintained on the runtime linkers link-map is achieved by using the following structure, defined in `rtld_db.h`:

```
typedef struct rd_loadobj {
        psaddr_t        rl_nameaddr;
        unsigned        rl_flags;
        psaddr_t        rl_base;
        psaddr_t        rl_data_base;
        unsigned        rl_lmident;
        psaddr_t        rl_refnameaddr;
        psaddr_t        rl_plt_base;
        unsigned        rl_plt_size;
        psaddr_t        rl_bend;
        psaddr_t        rl_padstart;
        psaddr_t        rl_padend;
        psaddt_t        rl_dynamic;
} rd_loadobj_t;
```

Notice that all addresses given in this structure, including string pointers, are addresses in the target process and not in the address space of the controlling process itself.

`rl_nameaddr`
: A pointer to a string that contains the name of the dynamic object.

`rl_flags`
: With revision `RD_VERSION2`, dynamically loaded relocatable objects are identified with `RD_FLG_MEM_OBJECT`.

`rl_base`
: The base address of the dynamic object.

`rl_data_base`
: The base address of the data segment of the dynamic object.

`rl_lmident`
: The link-map identifier (see "Establishing a Namespace" on page 158).

`rl_refnameaddr`
: If the dynamic object is a standard filter, then this points to the name of the filtees.

`rl_plt_base`, `rl_plt_size`
: These elements are present for backward compatibility and are currently unused.

`rl_bend`
: The end address of the object (`text` + `data` + `bss`). With revision `RD_VERSION2`, a dynamically loaded relocatable object will cause this element to point to the end of the created object, which will include its section headers.

`rl_padstart`
: The base address of the padding before the dynamic object (refer to "Dynamic Object Padding" on page 176).

`rl_padend`
: The base address of the padding after the dynamic object (refer to "Dynamic Object Padding" on page 176).

`rl_dynamic`
: This field, added with `RD_VERSION2`, provides the base address of the object's dynamic section, which allows reference to such entries as `DT_CHECKSUM` (see Table 7–43).

The `rd_loadobj_iter()` routine uses this object data structure to access information from the runtime linker's link-map lists:

`rd_loadobj_iter()`
: This function iterates over all dynamic objects currently loaded in the target process.

```
typedef int rl_iter_f(const rd_loadobj_t *, void *);

rd_err_e rd_loadobj_iter(rd_agent_t * rap, rl_iter_f * cb,
        void * clnt_data);
```

On each iteration the imported function specified by *cb* is called. *clnt_data* can be used to pass data to the *cb* call. Information about each object is returned via a pointer to a volatile (stack allocated) `rd_loadobj_t` structure.

Return codes from the *cb* routine are examined by `rd_loadobj_iter()` and have the following meaning:

- 1 – continue processing link-maps.
- 0 – stop processing link-maps and return control to the controlling process.

`rd_loadobj_iter()` returns `RD_OK` on success. A return of `RD_NOMAPS` indicates the runtime linker has not yet loaded the initial link-maps.

Event Notification

A controlling process can track certain events that occur within the scope of the runtime linker that. These events are:

`RD_PREINIT`
: The runtime linker has loaded and relocated all the dynamic objects and is about to start calling the `.init` sections of each object loaded.

`RD_POSTINIT`
: The runtime linker has finished calling all of the `.init` sections and is about to transfer control to the primary executable.

`RD_DLACTIVITY`
: The runtime linker has been invoked to either load or unload a dynamic object.

These events can be monitored using the following interface, defined in `sys/link.h` and `rtld_db.h`:

```
typedef enum {
        RD_NONE = 0,
        RD_PREINIT,
        RD_POSTINIT,
        RD_DLACTIVITY
} rd_event_e;

/*
 * Ways that the event notification can take place:
 */
typedef enum {
        RD_NOTIFY_BPT,
        RD_NOTIFY_AUTOBPT,
        RD_NOTIFY_SYSCALL
} rd_notify_e;

/*
 * Information on ways that the event notification can take place:
 */
typedef struct rd_notify {
        rd_notify_e     type;
        union {
                psaddr_t        bptaddr;
                long            syscallno;
        } u;
} rd_notify_t;
```

The following functions track events:

`rd_event_enable()`
: This function enables (1) or disables (0) event monitoring.

```
rd_err_e rd_event_enable(struct rd_agent * rdap, int onoff);
```

Note – Presently, for performance reasons, the runtime linker ignores event disabling. The controlling process should not assume that a given break-point can not be reached because of the last call to this routine.

`rd_event_addr()`
: This function specifies how the controlling program is notified of a given event.

```
rd_err_e rd_event_addr(rd_agent_t * rdap, rd_event_e event,
        rd_notify_t * notify);
```

Depending on the event type, the notification of the controlling process takes place by calling a benign, cheap system call that is identified by `notify->u.syscallno`, or executing a break point at the address specified by `notify->u.bptaddr`. The controlling process is responsible for tracing the system call or place the actual break-point.

When an event has occurred, additional information can be obtained by this interface, defined in `rtld_db.h`:

```
typedef enum {
        RD_NOSTATE = 0,
        RD_CONSISTENT,
        RD_ADD,
        RD_DELETE
} rd_state_e;

typedef struct rd_event_msg {
        rd_event_e      type;
        union {
                rd_state_e      state;
        } u;
} rd_event_msg_t;
```

The `rd_state_e` values are:

`RD_NOSTATE`
: There is no additional state information available.

`RD_CONSISTANT`
: The link-maps are in a stable state and can be examined.

`RD_ADD`
: A dynamic object is in the process of being loaded and the link-maps are not in a stable state. They should not be examined until the `RD_CONSISTANT` state is reached.

`RD_DELETE`
: A dynamic object is in the process of being deleted and the link-maps are not in a stable state. They should not be examined until the `RD_CONSISTANT` state is reached.

The `rd_event_getmsg()` function is used to obtain this event state information.

`rd_event_getmsg()`
: This function provides additional information concerning an event.

```
rd_err_e rd_event_getmsg(struct rd_agent * rdap, rd_event_msg_t * msg);
```

The following table shows the possible state for each of the different event types.

RD_PREINIT	RD_POSTINIT	RD_DLACTIVITY
RD_NOSTATE	RD_NOSTATE	RD_CONSISTANT
		RD_ADD
		RD_DELETE

Procedure Linkage Table Skipping

The rtld-debugger interface enables a controlling process to skip over procedure linkage table entries. When a controlling process, such as a debugger, is asked to step into a function for the first time, the procedure linkage table processing, causes control to be passed to the runtime linker to search for the function definition.

The following interface enables a controlling process to step over the runtime linker's procedure linkage table processing. The controlling process can determine when a procedure linkage table entry is encountered based on external information provided in the ELF file.

Once a target process has stepped into a procedure linkage table entry, the process calls the `rd_plt_resolution()` interface:

`rd_plt_resolution()`

This function returns the resolution state of the current procedure linkage table entry and information on how to skip it.

```
rd_err_e rd_plt_resolution(rd_agent_t * rdap, paddr_t pc,
        lwpid_t lwpid, paddr_t plt_base, rd_plt_info_t * rpi);
```

pc represents the first instruction of the procedure linkage table entry. *lwpid* provides the `lwp` identifier and *plt_base* provides the base address of the procedure linkage table. These three variables provide information sufficient for various architectures to process the procedure linkage table.

rpi provides detailed information regarding the procedure linkage table entry as defined in the following data structure, defined in `rtld_db.h`:

```
typedef enum {
        RD_RESOLVE_NONE,
        RD_RESOLVE_STEP,
        RD_RESOLVE_TARGET,
        RD_RESOLVE_TARGET_STEP
} rd_skip_e;

typedef struct rd_plt_info {
        rd_skip_e       pi_skip_method;
        long            pi_nstep;
        psaddr_t        pi_target;
        psaddr_t        pi_baddr;
        unsigned int    pi_flags;
} rd_plt_info_t;

#define RD_FLG_PI_PLTBOUND     0x0001
```

The elements of the `rd_plt_info_t`structure are:

`pi_skip_method`

Identifies how the procedure linkage table entry can be traversed. This method is set to one of the `rd_skip_e` values.

pi_nstep
Identifies how many instructions to step over when RD_RESOLVE_STEP or RD_RESOLVE_TARGET_STEP are returned.

pi_target
Specifies the address at which to set a breakpoint when RD_RESOLVE_TARGET_STEP or RD_RESOLVE_TARGET are returned.

pi_baddr
The procedure linkage table destination address, added with RD_VERSION3. When the RD_FLG_PI_PLTBOUND flag of the pi_flags field is set, this element identifies the resolved (bound) destination address.

pi_flags
A flags field, added with RD_VERSION3. The flag RD_FLG_PI_PLTBOUND identifies the procedure linkage entry as having been resolved (bound) to its destination address, which is available in the pi_baddr field.

The following scenarios are possible from the rd_plt_info_t return values:

- The first call through this procedure linkage table must be resolved by the runtime linker. In this case, the rd_plt_info_t contains:

  ```
  {RD_RESOLVE_TARGET_STEP, M, <BREAK>, 0, 0}
  ```

  The controlling process sets a breakpoint at BREAK and continues the target process. When the breakpoint is reached, the procedure linkage table entry processing has finished. The controlling process can then step M instructions to the destination function. Notice that the bound address (pi_baddr) has not been set since this is the first call through a procedure linkage table entry.
- On the Nth time through this procedure linkage table, rd_plt_info_t contains:

  ```
  {RD_RESOLVE_STEP, M, 0, <BoundAddr>, RD_FLG_PI_PLTBOUND}
  ```

  The procedure linkage table entry has already been resolved and the controlling process can step M instructions to the destination function. The address that the procedure linkage table entry is bound to is <BoundAddr> and the RD_FLG_PI_PLTBOUND bit has been set in the flags field.

Dynamic Object Padding

The default behavior of the runtime linker relies on the operating system to load dynamic objects where they can be most efficiently referenced. Some controlling processes benefit from the existence of padding around the objects loaded into memory of the target process. This interface enables a controlling process to request this padding.

rd_objpad_enable()
This function enables or disables the padding of any subsequently loaded objects with the target process. Padding occurs on both sides of the loaded object.

```
rd_err_e rd_objpad_enable(struct rd_agent * rdap, size_t padsize);
```

padsize specifies the size of the padding, in bytes, to be preserved both before and after any objects loaded into memory. This padding is reserved as a memory mapping using mmap(2) with PROT_NONE permissions and the MAP_NORESERVE flag. Effectively, the runtime linker reserves areas of the virtual address space of the target process adjacent to any loaded objects. These areas can later be utilized by the controlling process.

A *padsize* of 0 disables any object padding for later objects.

Note – Reservations obtained using mmap(2) from /dev/zero with MAP_NORESERVE can be reported using the proc(1) facilities and by referring to the link-map information provided in rd_loadobj_t.

Debugger Import Interface

The imported interface that a controlling process must provide to librtld_db.so.1 is defined in /usr/include/proc_service.h. A sample implementation of these proc_service functions can be found in the rdb demonstration debugger. The rtld-debugger interface uses only a subset of the proc_service interfaces available. Future versions of the rtld-debugger interface might take advantage of additional proc_service interfaces without creating an incompatible change.

The following interfaces are currently being used by the rtld-debugger interface:

ps_pauxv()
: This function returns a pointer to a copy of the auxv vector.

```
ps_err_e ps_pauxv(const struct ps_prochandle * ph, auxv_t ** aux);
```

Because the auxv vector information is copied to an allocated structure, the pointer remains as long as the *ps_prochandle* is valid.

ps_pread()
: This function reads data from the target process.

```
ps_err_e ps_pread(const struct ps_prochandle * ph, paddr_t addr,
        char * buf, int size);
```

From address *addr* in the target process, *size* bytes are copied to *buf*.

ps_pwrite()
: This function writes data to the target process.

```
ps_err_e ps_pwrite(const struct ps_prochandle * ph, paddr_t addr,
        char * buf, int size);
```

size bytes from *buf* are copied into the target process at address *addr*.

`ps_plog()`
: This function is called with additional diagnostic information from the rtld-debugger interface.

```
void ps_plog(const char * fmt, ...);
```

The controlling process determines where, or if, to log this diagnostic information. The arguments to `ps_plog()` follow the `printf`(3C) format.

`ps_pglobal_lookup()`
: This function searches for the symbol in the target process.

```
ps_err_e ps_pglobal_lookup(const struct ps_prochandle * ph,
        const char * obj, const char * name, ulong_t * sym_addr);
```

The symbol named *name* is searched for within the object named *obj* within the target process *ph*. If the symbol is found, the symbol address is stored in *sym_addr*.

`ps_pglobal_sym()`
: This function searches for the symbol in the target process.

```
ps_err_e ps_pglobal_sym(const struct ps_prochandle * ph,
        const char * obj, const char * name, ps_sym_t * sym_desc);
```

The symbol named *name* is searched for within the object named *obj* within the target process *ph*. If the symbol is found, the symbol descriptor is stored in *sym_desc*.

In the event that the rtld-debugger interface needs to find symbols within the application or runtime linker prior to any link-map creation, the following reserved values for *obj* are available:

```
#define PS_OBJ_EXEC ((const char *)0x0)  /* application id */
#define PS_OBJ_LDSO ((const char *)0x1)  /* runtime linker id */
```

The controlling process can use the `procfs` file system for these objects, using the following pseudo code:

```
ioctl(.., PIOCNAUXV, ...)       - obtain AUX vectors
ldsoaddr = auxv[AT_BASE];
ldsofd = ioctl(..., PIOCOPENM, &ldsoaddr);

/* process elf information found in ldsofd ... */

execfd = ioctl(.., PIOCOPENM, 0);

/* process elf information found in execfd ... */
```

Once the file descriptors are found, the ELF files can be examined for their symbol information by the controlling program.

CHAPTER 7

Object File Format

This chapter describes the executable and linking format (ELF) of the object files produced by the assembler and link-editor. There are three main types of object files:

- A relocatable file holds sections containing code and data. These files are suitable to be linked with other object files to create executable files, shared object files, or another relocatable object.
- An executable file holds a program that is ready to execute. The file specifies how `exec`(2) creates a program's process image.
- A shared object file holds code and data suitable to be linked in two contexts. First, the link-editor can process this file with other relocatable and shared object files to create other object files. Second, the runtime linker combines this file with a dynamic executable file and other shared objects to create a process image.

The first section in this chapter, "File Format" on page 179, focuses on the format of object files and how that pertains to creating programs. The second section, "Dynamic Linking" on page 239, focuses on how the format pertains to loading programs.

Programs manipulate object files with the functions contained in the ELF access library, `libelf`. Refer to `elf`(3ELF) for a description of `libelf` contents. Sample source code that uses `libelf` is provided in the `SUNWosdem` package under the `/usr/demo/ELF` directory.

File Format

Object files participate in both program linking and program execution. For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities. The following figure shows an object file's organization.

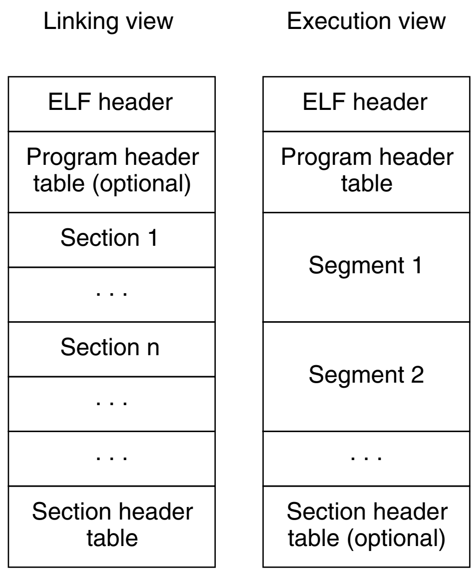


FIGURE 7–1 Object File Format

An ELF header resides at the beginning of an object file and holds a *road map* describing the file's organization.

Note – Only the ELF header has a fixed position in the file. The flexibility of the ELF format requires no specified order for header tables, sections or segments. However, this figure is typical of the layout used in Solaris.

Sections represent the smallest indivisible units that can be processed within an ELF file. *Segments* are a collection of sections that represent the smallest individual units that can be mapped to a memory image by `exec`(2) or by the runtime linker.

Sections hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of sections appear in the first part of this chapter. The second part of this chapter discusses segments and the program execution view of the file.

A program header table, if present, tells the system how to create a process image. Files used to generate a process image, executables and shared objects, must have a program header table; relocatable objects do not need such a table.

A section header table contains information describing the file's sections. Every section has an entry in the table. Each entry gives information such as the section name, the section size, and so forth. Files used in link-editing must have a section header table; other object files might or might not have one.

Data Representation

The object file format supports various processors with 8-bit bytes, 32–bit and 64–bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Table 7–1 and Table 7–2 list the 32–bit and 64–bit data types.

Object files represent some control data with a machine-independent format. making it possible to identify object files and interpret their contents in a common way. The remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created.

TABLE 7–1 ELF 32–Bit Data Types

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed integer
`Elf32_Word`	4	4	Unsigned integer
`unsigned char`	1	1	Unsigned small integer

TABLE 7–2 ELF 64–Bit Data Types

Name	Size	Alignment	Purpose
`Elf64_Addr`	8	8	Unsigned program address
`Elf64_Half`	2	2	Unsigned medium integer
`Elf64_Off`	8	8	Unsigned file offset
`Elf64_Sword`	4	4	Signed integer
`Elf64_Word`	4	4	Unsigned integer
`Elf64_Xword`	8	8	Unsigned long integer
`Elf64_Sxword`	8	8	Signed long integer
`unsigned char`	1	1	Unsigned small integer

All data structures that the object file format defines follow the natural size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, and so forth. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file, and a structure containing an `Elf64_Addr` member will be aligned on an 8–byte boundary.

Note – For portability, ELF uses no bit-fields.

ELF Header

Some object file control structures can grow because the ELF header contains their actual sizes. If the object file format changes, a program can encounter control structures that are larger or smaller than expected. Programs might therefore ignore extra information. The treatment of missing information depends on context and will be specified if and when extensions are defined.

The ELF header has the following structure, that is defined in `sys/elf.h`:

```
#define EI_NIDENT       16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

The elements of this structure are:

e_ident
: The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear in "ELF Identification" on page 185.

e_type
: Identifies the object file type, as listed in the following table.

TABLE 7–3 ELF File Identifiers

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

Although the core file contents are unspecified, type ET_CORE is reserved to mark the file. Values from ET_LOPROC through ET_HIPROC (inclusive) are reserved for processor-specific semantics. Other values are reserved and will be assigned to new object file types as necessary.

e_machine
: Specifies the required architecture for an individual file. Relevant architectures are listed in the following table.

TABLE 7–4 ELF Machines

Name	Value	Meaning
EM_NONE	0	No machine
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_SPARC32PLUS	18	Sun SPARC 32+
EM_SPARCV9	43	SPARC V9

Other values are reserved and will be assigned to new machines as necessary (see sys/elf.h). Processor-specific ELF names use the machine name to distinguish them. For example, the flags defined in Table 7–5 use the prefix EF_. A flag named WIDGET for the EM_XYZ machine would be called EF_XYZ_WIDGET.

`e_version`
Identifies the object file version, as listed in the following table.

TABLE 7–5 ELF Versions

Name	Value	Meaning
`EV_NONE`	`0`	Invalid version
`EV_CURRENT`	`>=1`	Current version

The value 1 signifies the original file format. The value of `EV_CURRENT` changes as necessary to reflect the current version number.

`e_entry`
The virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

`e_phoff`
The program header table's file offset in bytes. If the file has no program header table, this member holds zero.

`e_shoff`
The section header table's file offset in bytes. If the file has no section header table, this member holds zero.

`e_flags`
Processor-specific flags associated with the file. Flag names take the form `EF_machine_`*flag*. This member is presently zero for x86. The SPARC flags are listed in the following table.

TABLE 7–6 SPARC: ELF Flags

Name	Value	Meaning
`EF_SPARC_EXT_MASK`	`0xffff00`	Vendor Extension mask
`EF_SPARC_32PLUS`	`0x000100`	Generic V8+ features
`EF_SPARC_SUN_US1`	`0x000200`	Sun UltraSPARC™ 1 Extensions
`EF_SPARC_HAL_R1`	`0x000400`	HAL R1 Extensions
`EF_SPARC_SUN_US3`	`0x000800`	Sun UltraSPARC 3 Extensions
`EF_SPARCV9_MM`	`0x3`	Mask for Memory Model
`EF_SPARCV9_TSO`	`0x0`	Total Store Ordering
`EF_SPARCV9_PSO`	`0x1`	Partial Store Ordering
`EF_SPARCV9_RMO`	`0x2`	Relaxed Memory Ordering

`e_ehsize`
The ELF header's size in bytes.

e_phentsize
: The size in bytes of one entry in the file's program header table. All entries are the same size.

e_phnum
: The number of entries in the program header table. The product of e_phentsize and e_phnum gives the table's size in bytes. If a file has no program header table, e_phnum holds the value zero.

e_shentsize
: A section header's size in bytes. A section header is one entry in the section header table. All entries are the same size.

e_shnum
: The number of entries in the section header table. The product of e_shentsize and e_shnum gives the section header table's size in bytes. If a file has no section header table, e_shnum holds the value zero.

  If the number of sections is greater than or equal to SHN_LORESERVE (0xff00), this member has the value zero and the actual number of section header table entries is contained in the sh_size field of the section header at index 0. Otherwise, the sh_size member of the initial entry contains 0.

e_shstrndx
: The section header table index of the entry that is associated with the section name string table. If the file has no section name string table, this member holds the value SHN_UNDEF.

  If the section name string table section index is greater than or equal to SHN_LORESERVE (0xff00), this member has the value SHN_XINDEX (0xffff) and the actual index of the section name string table section is contained in thesh_link field of the section header at index 0. Otherwise, the sh_link member of the initial entry contains 0.

ELF Identification

ELF provides an object file framework to support multiple processors, multiple data encoding, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file. These bytes are independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header and an object file correspond to the e_ident member.

TABLE 7–7 ELF Identification Index

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_OSABI	7	Operating system/ABI identification
EI_ABIVERSION	8	ABI version
EI_PAD	9	Start of padding bytes
EI_NIDENT	16	Size of e_ident[]

These indexes access bytes that hold the values described below.

EI_MAG0 - EI_MAG3
A 4–byte *magic number*, identifying the file as an ELF object file, as listed in the following table.

TABLE 7–8 ELF Magic Number

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

EI_CLASS
Byte e_ident[EI_CLASS] identifies the file's class, or capacity, as listed in the following table.

TABLE 7–9 ELF File Class

Name	Value	Meaning
ELFCLASSNONE	0	Invalid class

TABLE 7–9 ELF File Class *(Continued)*

Name	Value	Meaning
ELFCLASS32	1	32–bit objects
ELFCLASS64	2	64–bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. The class of the file defines the basic types used by the data structures of the object file container itself. The data contained in object file sections can follow a different programming model.

Class ELFCLASS32 supports machines with files and virtual address spaces up to 4 gigabytes. It uses the basic types that are defined in Table 7–1.

Class ELFCLASS64 is reserved for 64–bit architectures such as SPARC. It uses the basic types that are defined in Table 7–2.

EI_DATA
Byte e_ident[EI_DATA] specifies the data encoding of the processor-specific data in the object file, as listed in the following table.

TABLE 7–10 ELF Data Encoding

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	See Figure 7–2.
ELFDATA2MSB	2	See Figure 7–3.

More information on these encodings appears in the section "Data Encoding" on page 188. Other values are reserved and will be assigned to new encodings as necessary.

EI_VERSION
Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT.

EI_OSABI
Byte e_ident[EI_OSABI] identifies the operating system and ABI to which the object is targeted. Some fields in other ELF structures have flags and values that have operating system or ABI specific meanings. The interpretation of those fields is determined by the value of this byte.

EI_ABIVERSION
Byte e_ident[EI_ABIVERSION] identifies the version of the ABI to which the object is targeted. This field is used to distinguish among incompatible versions of an ABI. The interpretation of this version number is dependent on the ABI

identified by the EI_OSABI field. If no values are specified for the EI_OSABI field for the processor, or no version values are specified for the ABI determined by a particular value of the EI_OSABI byte, the value 0 is used to indicate unspecified.

EI_PAD
This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and set to zero. Programs that read object files should ignore them.

Data Encoding

A file's data encoding specifies how to interpret the basic objects in a file. Class ELFCLASS32 files use objects that occupy 1, 2, and 4 bytes. Class ELFCLASS64 files use objects that occupy 1, 2, 4, and 8 bytes. Under the defined encodings, objects are represented as shown below. Byte numbers appear in the upper left corners.

Encoding ELFDATA2LSB specifies 2's complement values, with the least significant byte occupying the lowest address.

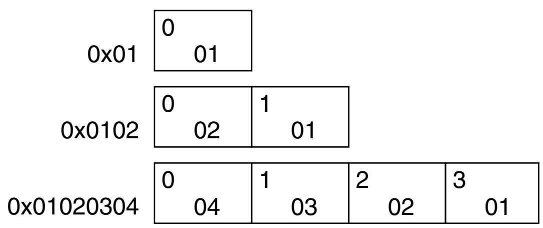


FIGURE 7–2 Data Encoding ELFDATA2LSB

Encoding ELFDATA2MSB specifies 2's complement values, with the most significant byte occupying the lowest address.

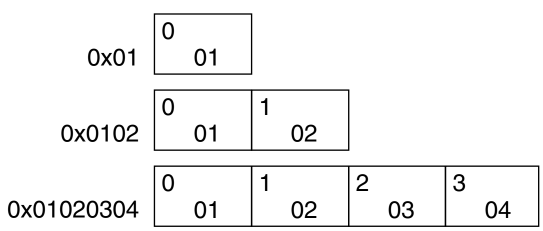


FIGURE 7–3 Data Encoding ELFDATA2MSB

Sections

An object file's section header table helps you locate all of the sections of the file. The section header table is an array of Elf32_Shdr or Elf64_Shdr structures, as described below. A section header table index is a subscript into this array. The ELF header's e_shoff member gives the byte offset from the beginning of the file to the section header table; e_shnum tells how many entries the section header table contains; e_shentsize gives the size in bytes of each entry.

If the number of sections is greater than or equal to SHN_LORESERVE (0xff00), e_shnum has the value SHN_UNDEF (0) and the actual number of section header table entries is contained in the sh_size field of the section header at index 0. Otherwise, the sh_size member of the initial entry contains 0.

Some section header table indexes are reserved in contexts where index size is restricted. For example, the st_shndx member of a symbol table entry and the e_shnum and e_shstrndx members of the ELF header. In such contexts, the reserved values do not represent actual sections in the object file. Also in such contexts, an escape value indicates that the actual section index is to be found elsewhere, in a larger field.

TABLE 7–11 ELF Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_HIPROC	0xff1f
SHN_LOOS	0xff20
SHN_LOSUNW	0xff3f
SHN_SUNW_IGNORE	0xff3f
SHN_HISUNW	0xff3f
SHN_HIOS	0xff3f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_XINDEX	0xffff
SHN_HIRESERVE	0xffff

Note – Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. That is, if the e_shnum member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5. The contents of the initial entry are specified later in this section.

SHN_UNDEF
: An undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol *defined* relative to section number SHN_UNDEF is an undefined symbol.

SHN_LORESERVE
: The lower boundary of the range of reserved indexes.

SHN_LOPROC - SHN_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics.

SHN_LOOS - SHN_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

SHN_BEFORE, SHN_AFTER
: Provide for initial and final section ordering in conjunction with the SHF_LINK_ORDER and SHF_ORDERED section flags, listed in Table 7–14.

SHN_ABS
: Absolute values for the corresponding reference. For example, symbols defined relative to section number SHN_ABS have absolute values and are not affected by relocation.

SHN_COMMON
: Symbols defined relative to this section are common symbols, such as FORTRAN COMMON or unallocated C external variables. These symbols are sometimes referred to as tentative.

SHN_XINDEX
: An escape value indicating that the actual section header index is too large to fit in the containing field. The header section index is found in another location specific to the structure where it appears.

SHN_HIRESERVE
: The upper boundary of the range of reserved indexes. The system reserves indexes between SHN_LORESERVE and SHN_HIRESERVE, inclusive. The values do not reference the section header table. The section header table does not contain entries for the reserved indexes.

Sections contain all information in an object file except the ELF header, the program header table, and the section header table. Moreover, the sections in object files satisfy several conditions:

- Every section in an object file has exactly one section header describing it. Section headers can exist that do not have a section.

- Each section occupies one contiguous, possibly empty, sequence of bytes within a file.
- Sections in a file cannot overlap. No byte in a file resides in more than one section.
- An object file can have inactive space. The various headers and the sections might not cover every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure, that is defined in `sys/elf.h`:

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;

typedef struct {
        Elf64_Word      sh_name;
        Elf64_Word      sh_type;
        Elf64_Xword     sh_flags;
        Elf64_Addr      sh_addr;
        Elf64_Off       sh_offset;
        Elf64_Xword     sh_size;
        Elf64_Word      sh_link;
        Elf64_Word      sh_info;
        Elf64_Xword     sh_addralign;
        Elf64_Xword     sh_entsize;
} Elf64_Shdr;
```

The elements of this structure are:

`sh_name`
: The name of the section. Its value is an index into the section header string table section giving the location of a null-terminated string. Section names and their descriptions are listed in Table 7–16.

`sh_type`
: Categorizes the section's contents and semantics. Section types and their descriptions are listed in Table 7–12.

`sh_flags`
: Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions are listed in Table 7–14.

`sh_addr`
: If the section is to appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

sh_offset
The byte offset from the beginning of the file to the first byte in the section. For a SHT_NOBITS section, this member indicates the conceptual offset in the file, as the section occupies no space in the file.

sh_size
The section's size in bytes. Unless the section type is SHT_NOBITS, the section occupies sh_size bytes in the file. A section of type SHT_NOBITS can have a nonzero size, but it occupies no space in the file.

sh_link
A section header table index link, whose interpretation depends on the section type. Table 7–15 describes the values.

sh_info
Extra information, whose interpretation depends on the section type. Table 7–15 describes the values.

sh_addralign
Some sections have address alignment constraints. For example, if a section holds a double-word, the system must ensure double-word alignment for the entire section. That is, the value of sh_addr must be congruent to 0, modulo the value of sh_addralign. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

sh_entsize
Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's sh_type member specifies the section's semantics, as shown in the following table.

TABLE 7–12 ELF Section Types, *sh_type*

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7

TABLE 7–12 ELF Section Types, *sh_type* *(Continued)*

Name	Value
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_INIT_ARRAY	14
SHT_FINI_ARRAY	15
SHT_PREINIT_ARRAY	16
SHT_GROUP	17
SHT_SYMTAB_SHNDX	18
SHT_LOOS	0x60000000
SHT_LOSUNW	0x6ffffff4
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_hwcap	0x6ffffff5
SHT_SUNW_SIGNATURE	0x6ffffff6
SHT_SUNW_ANNOTATE	0x6ffffff7
SHT_SUNW_DEBUGSTR	0x6ffffff8
SHT_SUNW_DEBUG	0x6ffffff9
SHT_SUNW_move	0x6ffffffa
SHT_SUNW_COMDAT	0x6ffffffb
SHT_SUNW_syminfo	0x6ffffffc
SHT_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
SHT_HISUNW	0x6fffffff
SHT_HIOS	0x6fffffff
SHT_LOPROC	0x70000000
SHT_SPARC_GOTDATA	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000

TABLE 7–12 ELF Section Types, *sh_type* *(Continued)*

Name	Value
SHT_HIUSER	0xffffffff

SHT_NULL
: Identifies the section header as inactive. This section header does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS
: Identifies information defined by the program, whose format and meaning are determined solely by the program.

SHT_SYMTAB, SHT_DYNSYM
: Identifies a symbol table. Typically, a SHT_SYMTAB section provides symbols for link-editing. As a complete symbol table, it can contain many symbols unnecessary for dynamic linking. Consequently, an object file can also contain a SHT_DYNSYM section, which holds a minimal set of dynamic linking symbols, to save space. See "Symbol Table Section" on page 225 for details.

SHT_STRTAB, SHT_DYNSTR
: Identifies a string table. An object file can have multiple string table sections. See "String Table Section" on page 224 for details.

SHT_RELA
: Identifies relocation entries with explicit addends, such as type Elf32_Rela for the 32–bit class of object files. An object file can have multiple relocation sections. See "Relocation Sections" on page 213 for details.

SHT_HASH
: Identifies a symbol hash table. A dynamically linked object file must contain a symbol hash table. Currently, an object file can have only one hash table, but this restriction might be relaxed in the future. See "Hash Table Section" on page 208 for details.

SHT_DYNAMIC
: Identifies information for dynamic linking. Currently, an object file can have only one dynamic section. See "Dynamic Section" on page 252 for details.

SHT_NOTE
: Identifies information that marks the file in some way. See "Note Section" on page 211 for details.

SHT_NOBITS
: Identifies a section that occupies no space in the file but otherwise resembles SHT_PROGBITS. Although this section contains no bytes, the sh_offset member contains the conceptual file offset.

SHT_REL
: Identifies relocation entries without explicit addends, such as type Elf32_Rel for the 32–bit class of object files. An object file can have multiple relocation sections. See "Relocation Sections" on page 213 for details.

SHT_SHLIB
: Identifies a reserved section which has unspecified semantics. Programs that contain a section of this type do not conform to the ABI.

SHT_INIT_ARRAY
: Identifies a section containing an array of pointers to initialization functions. Each pointer in the array is taken as a parameterless procedure with a void return. See "Initialization and Termination Sections" on page 35 for details.

SHT_FINI_ARRAY
: Identifies a section containing an array of pointers to termination functions. Each pointer in the array is taken as a parameterless procedure with a void return. See "Initialization and Termination Sections" on page 35 for details.

SHT_PREINIT_ARRAY
: Identifies a section containing an array of pointers to functions that are invoked before all other initialization functions. Each pointer in the array is taken as a parameterless procedure with a void return. See "Initialization and Termination Sections" on page 35 for details.

SHT_GROUP
: Identifies a section group. A section group identifies a set of related sections that must be treated as a unit by the link-editor. Sections of type SHT_GROUP can appear only in relocatable objects. See "Group Section" on page 205 for details.

SHT_SYMTAB_SHNDX
: Identifies a section containing extended section indexes, that is associated with a symbol table. If any section header indexes referenced by a symbol table, contain the escape value SHN_XINDEX, an associated SHT_SYMTAB_SHNDX is required.

  The SHT_SYMTAB_SHNDX section is an array of Elf32_Word values. There is one value for every entry in the associated symbol table entry. The values represent the section header indexes against which the symbol table entries are defined. Only if corresponding symbol table entry's st_shndx field contains the escape value SHN_XINDEX will the matching Elf32_Word hold the actual section header index. Otherwise, the entry must be SHN_UNDEF (0).

SHT_LOOS – SHT_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

SHT_LOSUNW – SHT_HISUNW
: Values in this inclusive range are reserved for Solaris semantics.

SHT_SUNW_cap
: Specifies hardware and software capability requirements. See "Hardware and Software Capabilities Section" on page 207 for details.

SHT_SUNW_SIGNATURE
: Identifies module verification signature.

SHT_SUNW_ANNOTATE
: The processing of an annotate section follows all of the default rules for processing a section. The only exception occurs if the annotate section is in non-allocatable

memory. If the section header flag SHF_ALLOC is not set, the link-editor silently ignores any relocations against this section that cannot be satisfied during the link-edit.

SHT_SUNW_DEBUGSTR, SHT_SUNW_DEBUG
Identifies debugging information. Sections of this type are stripped from the object using the link-editor's -s option, or after the link-edit using strip(1).

SHT_SUNW_move
Identifies data to handle partially initialized symbols. See "Move Section" on page 209 for details.

SHT_SUNW_COMDAT
Identifies a section that allows multiple copies of the same data to be reduced to a single copy. See "COMDAT Section" on page 205 for details.

SHT_SUNW_syminfo
Identifies additional symbol information. See "Syminfo Table Section" on page 233 for details.

SHT_SUNW_verdef
Identifies fine-grained versions defined by this file. See "Version Definition Section" on page 234 for details.

SHT_SUNW_verneed
Identifies fine-grained dependencies required by this file. See "Version Dependency Section" on page 237 for details.

SHT_SUNW_versym
Identifies a table describing the relationship of symbols to the version definitions offered by the file. See "Version Symbol Section" on page 236 for details.

SHT_LOPROC - SHT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHT_SPARC_GOTDATA
Identifies SPARC specific data, referenced using GOT-relative addressing. That is, offsets relative to the address assigned to the symbol _GLOBAL_OFFSET_TABLE_. For 64-bit SPARC, data in this section must be bound at link-edit time to locations within {+-} 2^32 bytes of the GOT address.

SHT_LOUSER
Specifies the lower boundary of the range of indexes reserved for application programs.

SHT_HIUSER
Specifies the upper boundary of the range of indexes reserved for application programs. Section types between SHT_LOUSER and SHT_HIUSER can be used by the application without conflicting with current or future system-defined section types.

Other section-type values are reserved. As mentioned before, the section header for index 0 (SHN_UNDEF) exists, even though the index marks undefined section references. The following table shows the values.

TABLE 7–13 ELF Section Header Table Entry: Index 0

Name	Value	Note
sh_name	0	No name
sh_type	SHT_NULL	Inactive
sh_flags	0	No flags
sh_addr	0	No address
sh_offset	0	No file offset
sh_size	0	No size
sh_link	SHN_UNDEF	No link information
sh_info	0	No auxiliary information
sh_addralign	0	No alignment
sh_entsize	0	No entries

A section header's `sh_flags` member holds 1-bit flags that describe the section's attributes:

TABLE 7–14 ELF Section Attribute Flags

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MERGE	0x10
SHF_STRINGS	0x20
SHF_INFO_LINK	0x40
SHF_LINK_ORDER	0x80
SHF_OS_NONCONFORMING	0x100
SHF_GROUP	0x200
SHF_TLS	0x400
SHF_MASKOS	0x0ff00000
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000

TABLE 7–14 ELF Section Attribute Flags *(Continued)*

Name	Value
SHF_MASKPROC	0xf0000000

If a flag bit is set in sh_flags, the attribute is *on* for the section. Otherwise, the attribute is *off* or does not apply. Undefined attributes are reserved and set to zero.

SHF_WRITE
: Identifies a section that should be writable during process execution.

SHF_ALLOC
: Identifies a section that occupies memory during process execution. Some control sections do not reside in the memory image of an object file. This attribute is off for those sections.

SHF_EXECINSTR
: Identifies a section that contains executable machine instructions.

SHF_MERGE
: Identifies a section containing data that can be merged to eliminate duplication. Unless the SHF_STRINGS flag is also set, the data elements in the section are of a uniform size. The size of each element is specified in the section header's sh_entsize field. If the SHF_STRINGS flag is also set, the data elements consist of null-terminated character strings. The size of each character is specified in the section header's sh_entsize field.

SHF_STRINGS
: Identifies a section that consists of null-terminated character strings. The size of each character is specified in the section header's sh_entsize field.

SHF_INFO_LINK
: This section headers sh_info field holds a section header table index.

SHF_LINK_ORDER
: This section adds special ordering requirements to the link-editor. The requirements apply if the sh_link field of this section's header references another section, the linked-to section. If this section is combined with other sections in the output file, the section appears in the same relative order with respect to those sections. Similarly the linked-to section appears with respect to sections the linked-to section is combined with.

  The special sh_link values SHN_BEFORE and SHN_AFTER (see Table 7–11) imply that the sorted section is to precede or follow, respectively, all other sections in the set being ordered. Input file link-line order is preserved if multiple sections in an ordered set have one of these special values.

  A typical use of this flag is to build a table that references text or data sections in address order.

In the absence of the sh_link ordering information, sections from a single input file combined within one section of the output file will be contiguous and have the same relative ordering as they did in the input file. The contributions from multiple input files appear in link-line order.

SHF_OS_NONCONFORMING
: This section requires special OS-specific processing beyond the standard linking rules to avoid incorrect behavior. If this section has either an sh_type value or contains sh_flags bits in the OS-specific ranges for those fields, and the link-editor does not recognize these values, then the link-editor will reject the object file containing this section with an error.

SHF_GROUP
: This section is a member, perhaps the only one, of a section group. The section must be referenced by a section of type SHT_GROUP. The SHF_GROUP flag can be set only for sections contained in relocatable objects. See "Group Section" on page 205 for details.

SHF_TLS
: This section holds thread-local storage, meaning that each separate execution flow has its own distinct instance of this data. See Chapter 8 for details.

SHF_MASKOS
: All bits included in this mask are reserved for operating system-specific semantics.

SHF_ORDERED
: This section requires ordering in relation to other sections of the same type. Ordered sections are combined within the section pointed to by the sh_link entry. The sh_link entry of an ordered section can point to itself.

If the sh_info entry of the ordered section is a valid section within the same input file, the ordered section will be sorted based on the relative ordering within the output file of the section pointed to by the sh_info entry.

The special sh_info values SHN_BEFORE and SHN_AFTER (see Table 7–11) imply that the sorted section is to precede or follow, respectively, all other sections in the set being ordered. Input file link-line order is preserved if multiple sections in an ordered set have one of these special values.

In the absence of the sh_info ordering information, sections from a single input file combined within one section of the output file will be contiguous and have the same relative ordering as they did in the input file. The contributions from multiple input files appear in link-line order.

SHF_EXCLUDE
: This section is excluded from input to the link-edit of an executable or shared object. This flag is ignored if the SHF_ALLOC flag is also set, or if relocations exist against the section.

SHF_MASKPROC
: All bits included in this mask are reserved for processor-specific semantics.

Two members in the section header, sh_link and sh_info, hold special information, depending on section type.

TABLE 7–15 ELF sh_link and sh_info Interpretation

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the associated string table.	0
SHT_HASH	The section header index of the associated symbol table.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies. See also Table 7–16 and "Relocation Sections" on page 213.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
SHT_GROUP	The section header index of the associated symbol table.	The symbol table index of an entry in the associated symbol table. The name of the specified symbol table entry provides a signature for the section group.
SHT_SYMTAB_SHNDX	The section header index of the associated symbol table.	0
SHT_SUNW_move	The section header index of the associated symbol table.	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	The section header index of the associated symbol table.	The section header index of the associated .dynamic section.
SHT_SUNW_verdef	The section header index of the associated string table.	The number of version definitions within the section.
SHT_SUNW_verneed	The section header index of the associated string table.	The number of version dependencies within the section.
SHT_SUNW_versym	The section header index of the associated symbol table.	0

Special Sections

Various sections hold program and control information. Sections in the following table are used by the system and have the indicated types and attributes.

TABLE 7–16 ELF Special Sections

Name	Type	Attribute
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	None
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	See "Global Offset Table (Processor-Specific)" on page 264
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.interp	SHT_PROGBITS	See "Program Interpreter" on page 251
.note	SHT_NOTE	None
.plt	SHT_PROGBITS	See "Procedure Linkage Table (Processor-Specific)" on page 265
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	None
.rel*name*	SHT_REL	See "Relocation Sections" on page 213
.rela*name*	SHT_RELA	See "Relocation Sections" on page 213
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	None
.strtab	SHT_STRTAB	See description below
.symtab	SHT_SYMTAB	See "Symbol Table Section" on page 225

TABLE 7–16 ELF Special Sections *(Continued)*

Name	Type	Attribute
.symtab_shndx	SHT_SYMTAB_SHNDX	See "Symbol Table Section" on page 225
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_hwcap	SHT_SUNW_hwcap	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_REL	SHF_ALLOC
	SHT_RELA	
.SUNW_syminfo	SHT_SUNW_syminfo	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef	SHF_ALLOC
	SHT_SUNW_verneed	
	SHT_SUNW_versym	

`.bss`
: Uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type `SHT_NOBITS`.

`.comment`
: Comment information, typically contributed by the components of the compilation system. This section can be manipulated by `mcs`(1).

`.data`, `.data1`
: Initialized data that contribute to the program's memory image.

`.dynamic`
: Dynamic linking information. See "Dynamic Section" on page 252 for details.

`.dynstr`
: Strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries.

`.dynsym`
Dynamic linking symbol table. See "Symbol Table Section" on page 225 for details.

`.fini`
Executable instructions that contribute to a single termination function for the executable or shared object containing the section. See "Initialization and Termination Routines" on page 79 for details.

`.fini_array`
An array of function pointers that contribute to a single termination array for the executable or shared object containing the section. See "Initialization and Termination Routines" on page 79 for details.

`.got`
The global offset table. See "Global Offset Table (Processor-Specific)" on page 264 for details.

`.hash`
Symbol hash table. See "Hash Table Section" on page 208 for details.

`.init`
Executable instructions that contribute to a single initialization function for the executable or shared object containing the section. See "Initialization and Termination Routines" on page 79 for details.

`.init_array`
An array of function pointers that contributes to a single initialization array for the executable or shared object containing the section. See "Initialization and Termination Routines" on page 79 for details.

`.interp`
The path name of a program interpreter. See "Program Interpreter" on page 251 for details.

`.note`
Information in the format described in "Note Section" on page 211.

`.plt`
The procedure linkage table. See "Procedure Linkage Table (Processor-Specific)" on page 265 for details.

`.preinit_array`
An array of function pointers that contribute to a single pre-initialization array for the executable or shared object containing the section. See "Initialization and Termination Routines" on page 79 for details.

`.rela`
Relocations that do not apply to a particular section. One use of this section is for register relocations. See "Register Symbols" on page 232 for details.

`.rel`*name*, `.rela`*name*
Relocation information, as "Relocation Sections" on page 213 describes. If the file has a loadable segment that includes relocation, the sections' attributes include the `SHF_ALLOC` bit. Otherwise, that bit is off. Conventionally, *name* is supplied by the

section to which the relocations apply. Thus, a relocation section for `.text` normally will have the name `.rel.text` or `.rela.text`.

`.rodata`, `.rodata1`
Read-only data that typically contribute to a non-writable segment in the process image. See “Program Header” on page 239 for details.

`.shstrtab`
Section names.

`.strtab`
Strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section’s attributes include the `SHF_ALLOC` bit. Otherwise, that bit is turned off.

`.symtab`
Symbol table, as “Symbol Table Section” on page 225 describes. If the file has a loadable segment that includes the symbol table, the section’s attributes include the `SHF_ALLOC` bit. Otherwise, that bit is turned off.

`.symtab_shndx`
This section holds the special symbol table section index array, as described by `.symtab`. The section’s attributes will include the `SHF_ALLOC` bit if the associated symbol table section does. Otherwise, that bit is turned off.

`.tbss`
This section holds uninitialized thread-local data that contribute to the program’s memory image. By definition, the system initializes the data with zeros when the data is instantiated for each new execution flow. The section occupies no file space, as indicated by the section type, `SHT_NOBITS`. See Chapter 8 for details.

`.tdata`, `.tdata1`
These sections hold initialized thread-local data that contribute to the program’s memory image. A copy of its contents is instantiated by the system for each new execution flow. See Chapter 8 for details.

`.text`
The *text* or executable instructions of a program.

`.SUNW_bss`
Partially initialized data for shared objects that contribute to the program’s memory image. The data is initialized at runtime. The section occupies no file space, as indicated by the section type `SHT_NOBITS`.

`.SUNW_heap`
The *heap* of a dynamic executable created from `dldump`(3C).

`.SUNW_cap`
Hardware and software capability requirements. See “Hardware and Software Capabilities Section” on page 207 for details.

`.SUNW_move`
Additional information for partially initialized data. See “Move Section” on page 209 for details.

.SUNW_reloc
: Relocation information, as "Relocation Sections" on page 213 describes. This section is a concatenation of relocation sections that provides better locality of reference of the individual relocation records. Only the offset of the relocation record itself is meaningful, thus the section sh_info value is zero.

.SUNW_syminfo
: Additional symbol table information. See "Syminfo Table Section" on page 233 for details.

.SUNW_version
: Versioning information. See "Versioning Sections" on page 234 for details.

Section names with a dot (.) prefix are reserved for the system, although applications can use these sections if their existing meanings are satisfactory. Applications can use names without the prefix to avoid conflicts with system sections. The object file format enables you to define sections not in the list above. An object file can have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for e_machine. For example, .Foo.psect is the psect section defined by the FOO architecture.

Existing extensions use their historical names

COMDAT Section

COMDAT sections are uniquely identified by their section name (sh_name). If the link-editor encounters multiple sections of type SHT_SUNW_COMDAT, with the same section name, the first section is retained and the rest discarded. Any relocations applied to a discarded SHT_SUNW_COMDAT section are ignored. Any symbols that are defined in a discarded section are removed.

Additionally, the link-editor supports the section naming convention used for section reordering when the compiler is invoked with the -xF option. If a function is placed in a section with the name .*sectname*%*funcname*, the final SHT_SUNW_COMDAT sections that are retained are coalesced into a section identified by .*sectname*. Using this method, the SHT_SUNW_COMDAT sections are placed into the .text, .data, or any other section as their final destination.

Group Section

Some sections occur in interrelated groups. For example, an out-of-line definition of an inline function might require additional information besides the section containing executable instructions. This additional information can be a read-only data section containing literals referenced, one or more debugging information sections, or other informational sections. Furthermore, there can be internal references among these sections. These references make no sense if one of the sections were removed, or one of the sections were replaced, by a duplicate from another object. Therefore, these groups are included, or these groups are omitted, from the linked object as a unit.

A section of type SHT_GROUP defines such a grouping of sections. The name of a symbol from one of the containing object's symbol tables provides a signature for the section group. The section header of the SHT_GROUP section specifies the identifying symbol entry. The sh_link member contains the section header index of the symbol table section that contains the entry. The sh_info member contains the symbol table index of the identifying entry. The sh_flags member of the section header contains 0. The name of the section (sh_name) is not specified.

The section data of a SHT_GROUP section is an array of Elf32_Word entries. The first entry is a flag word. The remaining entries are a sequence of section header indices.

The following flag is currently defined:

TABLE 7–17 ELF Group Section Flag

Name	Value
GRP_COMDAT	0x1

GRP_COMDAT
GRP_COMDAT is a COMDAT group. It can duplicate another COMDAT group in another object file, where duplication is defined as having the same group signature. In such cases, only one of the duplicate groups is retained by the link-editor. The members of the remaining groups are discarded.

The section header indices in the SHT_GROUP section, identify the sections that make up the group. These sections must have the SHF_GROUP flag set in their sh_flags section header member. If the link-editor decides to remove the section group, the link-editor removes all members of the group.

To facilitate removing a group without leaving dangling references and with only minimal processing of the symbol table, the following rules are followed:

- References to the sections comprising a group from sections outside of the group must be made through symbol table entries with STB_GLOBAL or STB_WEAK binding and section index SHN_UNDEF. If there is a definition of the same symbol in the object containing the references, it must have a separate symbol table entry from the references. Sections outside of the group can not reference symbols with STB_LOCAL binding for addresses contained in the group's sections, including symbols with type STT_SECTION.
- Non-symbol references to the sections comprising a group are not allowed from outside the group. For example, you cannot use a group member's section header index in an sh_link or sh_info member.

- A symbol table entry defined relative to one of the group's sections can be removed if the group members are discarded. This removal occurs if the symbol table entry is contained in a symbol table section that is not part of the group.

Hardware and Software Capabilities Section

A SHT_SUNW_cap section identifies the hardware and software capabilities of an object. This section contains an array of the following structures, that are defined in sys/link.h:

```
typedef struct {
        Elf32_Word      c_tag;
        union {
                Elf32_Word      c_val;
                Elf32_Addr      c_ptr;
        } c_un;
} Elf32_Cap;

typedef struct {
        Elf64_Xword     c_tag;
        union {
                Elf64_Xword     c_val;
                Elf64_Addr      c_ptr;
        } c_un;
} Elf64_Cap;
```

For each object with this type, c_tag controls the interpretation of c_un.

c_val
These objects represent integer values with various interpretations.

c_ptr
These objects represent program virtual addresses.

The following capabilities tags exist.

TABLE 7–18 ELF Capability Array Tags

Name	Value	c_un
CA_SUNW_NULL	0	Ignored
CA_SUNW_HW_1	1	c_val
CA_SUNW_SF_1	2	c_val

CA_SUNW_NULL
Marks the end of the capabilities array.

CA_SUNW_HW_1
Indicates hardware capability values. The c_val element contains a value that represents the associated hardware capabilities. On SPARC platforms, hardware

capabilities are defined in `sys/auxv_SPARC.h`. On x86 platforms, hardware capabilities are defined in `sys/auxv_386.h`.

`CA_SUNW_SF_1`
Indicates software capability values. The `c_val` element contains a value that represents the associated software capabilities that are defined in `sys/elf.h`.

Relocatable objects can contain a capabilities section. The link-editor combines any capabilities sections from multiple input relocatable objects into a single capabilities section. The link-editor also allows capabilities to be defined at the time an object is built. See "Identifying Hardware and Software Capabilities" on page 57.

A dynamic object that contains a capabilities section that contains hardware capabilities information, has a `PT_SUNWHWCAP` program header associated to the section. This program header allows the runtime linker to validate the object against the hardware capabilities that are available to the process.

Dynamic objects that exploit different hardware capabilities can provide a flexible runtime environment using filters. See "Hardware Capability Specific Shared Objects" on page 325.

Hash Table Section

A hash table consists of `Elf32_Word` or `Elf64_Word` objects that provide for symbol table access. The `SHT_HASH` section provides this hash table. The symbol table to which the hashing is associated is specified in the `sh_link` entry of the hash table's section header. Labels appear below to help explain the hash table organization, but they are not part of the specification.

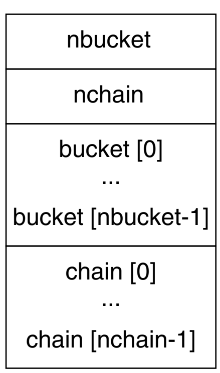


FIGURE 7–4 Symbol Hash Table

The `bucket` array contains `nbucket` entries, and the `chain` array contains `nchain` entries. Indexes start at 0. Both `bucket` and `chain` hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`, so symbol table indexes also select chain table entries.

A hashing function that accepts a symbol name, returns a value to compute a bucket index. Consequently, if the hashing function returns the value *x* for some name, bucket [*x*%nbucket] gives an index *y*. This is an index into both the symbol table and the chain table. If the symbol table entry is not the name desired, chain[*y*] gives the next symbol table entry with the same hash value.

The chain links can be followed until the selected symbol table entry holds the desired name, or the chain entry contains the value STN_UNDEF.

The hash function is as follows:

```
unsigned long
elf_Hash(const unsigned char *name)
{
    unsigned long h = 0, g;

        while (*name)
        {
             h = (h << 4) + *name++;
             if (g = h & 0xf0000000)
                  h ^= g >> 24;
                   h &= ~g;
        }
        return h;
}
```

Move Section

Typically, within ELF files, initialized data variables are maintained within the object file. If a data variable is very large, and contains only a small number of initialized (nonzero) elements, the entire variable is still maintained in the object file.

Objects that contain large partially initialized data variables, such as FORTRAN COMMON blocks, can result in a significant disk space overhead. The SHT_SUNW_move section provides a mechanism of compressing these data variables. This compression reduces the disk size of the associated object.

The SHT_SUNW_move section contains multiple entries of the type ELF32_Move or Elf64_Move. These entries allow data variables to be defined as tentative items (.bss), thus occupying no space in the object file but contributing to the object's memory image at runtime. The move records establish how the memory image is initialized with data to construct the complete data variable.

ELF32_Move and Elf64_Move entries are defined as follows:

```
typedef struct {
        Elf32_Lword       m_value;
        Elf32_Word        m_info;
        Elf32_Word        m_poffset;
        Elf32_Half        m_repeat;
```

```
        Elf32_Half        m_stride;
} Elf32_Move;

#define ELF32_M_SYM(info)       ((info)>>8)
#define ELF32_M_SIZE(info)      ((unsigned char)(info))
#define ELF32_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))

typedef struct {
        Elf64_Lword       m_value;
        Elf64_Xword       m_info;
        Elf64_Xword       m_poffset;
        Elf64_Half        m_repeat;
        Elf64_Half        m_stride;
} Elf64_Move;

#define ELF64_M_SYM(info)       ((info)>>8)
#define ELF64_M_SIZE(info)      ((unsigned char)(info))
#define ELF64_M_INFO(sym, size) (((sym)<<8)+(unsigned char)(size))
```

The elements of these structures are as follows:

`m_value`
: The initialization value, which is the value that is moved into the memory image.

`m_info`
: The symbol table index, with respect to which the initialization is applied, together with the size, in bytes, of the offset being initialized. The lower 8 bits of the member define the size, which can be 1, 2, 4 or 8. The upper bytes define the symbol index.

`m_poffset`
: The offset relative to the associated symbol to which the initialization is applied.

`m_repeat`
: A repetition count.

`m_stride`
: The stride count. This value indicates the number of units that should be skipped when performing a repetitive initialization. A unit is the size of an initialization object as defined by `m_info`. An `m_stride` value of 0 indicates that the initialization be performed contiguously for `m_repeat` units.

The following data definition would traditionally consume `0x8000` bytes within an object file:

```
typedef struct {
        int     one;
        char    two;
} Data

Data move[0x1000] = {
        {0, 0},        {1, '1'},     {0, 0},
        {0xf, 'F'},    {0xf, 'F'},   {0, 0},
        {0xe, 'E'},    {0, 0},       {0xe, 'E'}
};
```

Using an SHT_SUNW_move section the data item can be moved to the .bss section and initialized with the associated move entries:

```
$ elfdump -s data | fgrep move
      [17]  0x00020868 0x00008000  OBJT GLOB 0   .bss       move
$ elfdump -m data

Move Section: .SUNW_move
        offset  ndx     size    repeat  stride  value   with respect to
        0x8     0x17    4       1       0       0x1     move
        0xc     0x17    1       1       0       0x31    move
        0x18    0x17    4       2       2       0xf     move
        0x1c    0x17    1       2       8       0x46    move
        0x28    0x17    4       2       4       0xe     move
        0x2c    0x17    1       2       16      0x45    move
```

Move sections supplied from relocatable objects are concatenated and output in the object being created by the link-editor. However, the following conditions cause the link-editor to process the move entries and expand their contents into a traditional data item:

- The output file is a static executable.
- The size of the move entries is greater than the size of the symbol into which the move data would be expanded.
- The -z nopartial option is in effect.

Note Section

Sometimes a vendor or system engineer needs to mark an object file with special information that other programs will check for conformance, compatibility, and so forth. Sections of type SHT_NOTE and program header elements of type PT_NOTE can be used for this purpose.

The note information in sections and program header elements holds any number of entries, as shown in the following figure. For 64– and 32–bit objects, each entry is an array of 4-byte words in the format of the target processor. Labels are shown in Figure 7–6 to help explain note information organization, but they are not part of the specification.

FIGURE 7–5 Note Information

The elements of the structure are:

namesz and name
: The first namesz bytes in name contain a null-terminated character representation of the entry's owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as "XYZ Computer Company," as the identifier. If no name is present, namesz contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in namesz.

descsz and desc
: The first descsz bytes in desc hold the note descriptor. If no descriptor is present, descsz contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in descsz.

type
: Provides the interpretation of the descriptor. Each originator controls its own types. Multiple interpretations of a single type value can exist. A program must recognize both the name and the type to understand a descriptor. Types currently must be nonnegative.

The note segment shown in the following figure holds two entries.

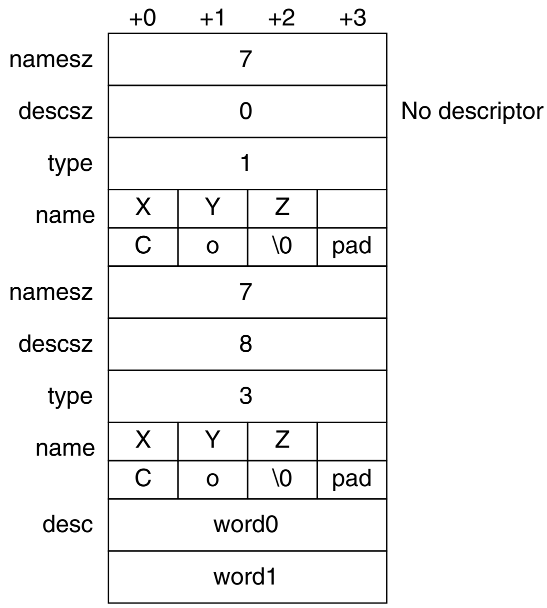


FIGURE 7–6 Example Note Segment

Note – The system reserves note information with no name (`namesz == 0`) and with a zero-length name (`name[0] == '\0'`) but currently defines no types. All other names must have at least one non-null character.

Relocation Sections

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Relocation entries are these data.

Relocation entries can have the following structure, that is defined in `sys/elf.h`:

```
typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr      r_offset;
```

```
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;

typedef struct {
        Elf64_Addr      r_offset;
        Elf64_Xword     r_info;
} Elf64_Rel;

typedef struct {
        Elf64_Addr      r_offset;
        Elf64_Xword     r_info;
        Elf64_Sxword    r_addend;
} Elf64_Rela;
```

The elements of this structure are:

`r_offset`
: This member gives the location at which to apply the relocation action. Different object files have slightly different interpretations for this member.

  For a relocatable file, the value indicates a section offset. The relocation section itself describes how to modify another section in the file. Relocation offsets designate a storage unit within the second section.

  For an executable or shared object, the value indicates the virtual address of the storage unit affected by the relocation. This information makes the relocation entries more useful for the runtime linker.

  Although the interpretation of the member changes for different object files to allow efficient access by the relevant programs, the meanings of the relocation types stay the same.

`r_info`
: This member gives both the symbol table index, with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry holds the symbol table index of the function being called. If the index is `STN_UNDEF`, the undefined symbol index, the relocation uses 0 as the symbol value.

  Relocation types are processor-specific. A relocation entry's relocation type or symbol table index is the result of applying `ELF32_R_TYPE` or `ELF32_R_SYM`, respectively, to the entry's `r_info` member:

```
#define ELF32_R_SYM(info)             ((info)>>8)
#define ELF32_R_TYPE(info)            ((unsigned char)(info))
#define ELF32_R_INFO(sym, type)       (((sym)<<8)+(unsigned char)(type))

#define ELF64_R_SYM(info)             ((info)>>32)
#define ELF64_R_TYPE(info)            ((Elf64_Word)(info))
#define ELF64_R_INFO(sym, type)       (((Elf64_Xword)(sym)<<32)+ \
                                        (Elf64_Xword)(type))
```

For Elf64_Rel and Elf64_Rela structures, the r_info field is further broken down into an 8–bit type identifier and a 24–bit type dependent data field:

```
#define ELF64_R_TYPE_DATA(info)       (((Elf64_Xword)(info)<<32)>>40)
#define ELF64_R_TYPE_ID(info)         (((Elf64_Xword)(info)<<56)>>56)
#define ELF64_R_TYPE_INFO(data, type) (((Elf64_Xword)(data)<<8)+ \
                                        (Elf64_Xword)(type))
```

r_addend
This member specifies a constant addend used to compute the value to be stored into the relocatable field.

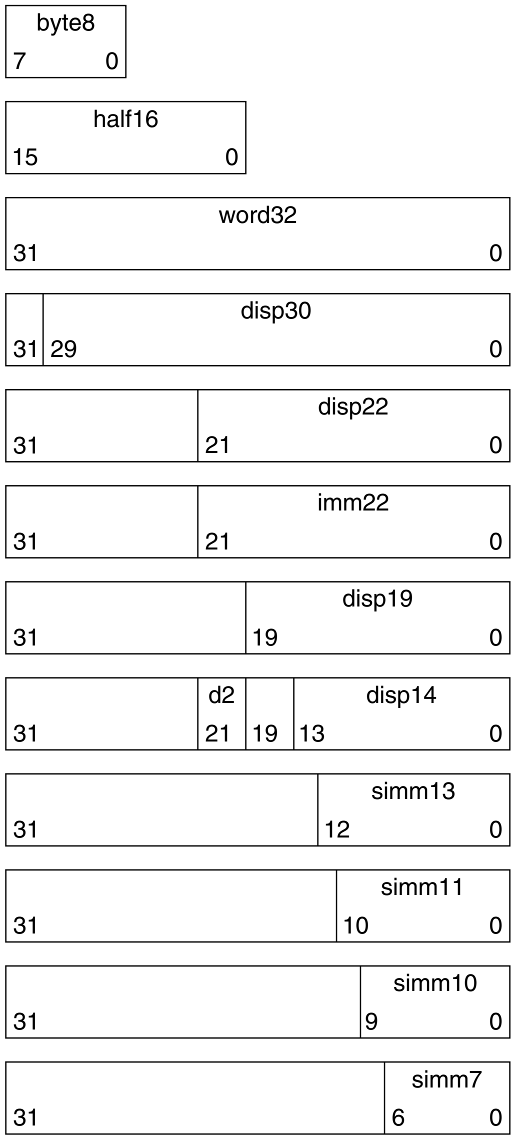
Rela entries contain an explicit addend. Entries of type Rel store an implicit addend in the location to be modified. 32–bit and 64–bit SPARC use only Elf32_Rela and Elf64_Rela relocation entries respectively. Thus, the r_addend member serves as the relocation addend. x86 uses only Elf32_Rel relocation entries. The field to be relocated holds the addend. In all cases, the addend and the computed result use the same byte order.

A relocation section can reference two other sections: a symbol table, identified by the sh_link section header entry, and a section to modify, identified by the sh_info section header entry. "Sections" on page 189 specifies these relationships. A sh_info entry is required when a relocation section exists in a relocatable object, but is optional for executables and shared objects. The relocation offset is sufficient to perform the relocation.

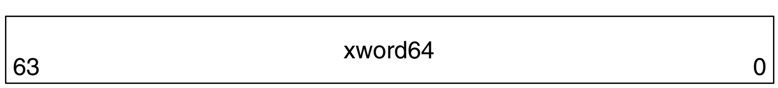
Relocation Types (Processor-Specific)

Relocation entries describe how to alter instruction and data fields in the following figures. Bit numbers appear in the lower box corners.

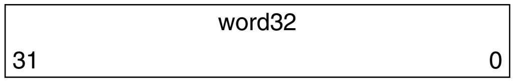
On the SPARC platform, relocation entries apply to bytes (byte8), half-words (half16), or words (the others).



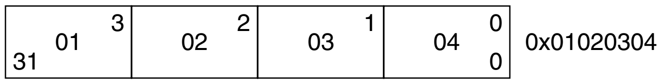
On 64–bit SPARC, relocations also apply to extended-words (`xword64`):



On x86, relocation entries apply to words (`word32`):



word32 specifies a 32–bit field occupying 4 bytes with an arbitrary byte alignment. These values use the same byte order as other word values in the x86 architecture:



In all cases, the r_offset value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values.

Calculations for the following relocation types assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link-editor merges one or more relocatable files to form the output. The link-editor first decides how to combine and locate the input files. The link-editor then updates the symbol values and performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions in the tables in this section use the following notation:

A The addend used to compute the value of the relocatable field.

B The base address at which a shared object is loaded into memory during execution. Generally, a shared object file is built with a base virtual address of 0. However, the execution address of the shared object is different. See "Program Header" on page 239.

G The offset into the global offset table at which the address of the relocation entry's symbol resides during execution. See "Global Offset Table (Processor-Specific)" on page 264.

GOT The address of the global offset table. See "Global Offset Table (Processor-Specific)" on page 264.

L The section offset or address of the procedure linkage table entry for a symbol. See "Procedure Linkage Table (Processor-Specific)" on page 265.

P The section offset or address of the storage unit being relocated, computed using r_offset.

S The value of the symbol whose index resides in the relocation entry.

SPARC: Relocation Types

Field names in the following table tell whether the relocation type checks for `overflow`. A calculated relocation value can be larger than the intended field, and a relocation type can verify (`V`) the value fits or truncate (`T`) the result. As an example, `V-simm13` means that the computed value can not have significant, nonzero bits outside the `simm13` field.

TABLE 7–19 SPARC: ELF Relocation Types

Name	Value	Field	Calculation
`R_SPARC_NONE`	0	None	None
`R_SPARC_8`	1	`V-byte8`	`S + A`
`R_SPARC_16`	2	`V-half16`	`S + A`
`R_SPARC_32`	3	`V-word32`	`S + A`
`R_SPARC_DISP8`	4	`V-byte8`	`S + A - P`
`R_SPARC_DISP16`	5	`V-half16`	`S + A - P`
`R_SPARC_DISP32`	6	`V-disp32`	`S + A - P`
`R_SPARC_WDISP30`	7	`V-disp30`	`(S + A - P) >> 2`
`R_SPARC_WDISP22`	8	`V-disp22`	`(S + A - P) >> 2`
`R_SPARC_HI22`	9	`T-imm22`	`(S + A) >> 10`
`R_SPARC_22`	10	`V-imm22`	`S + A`
`R_SPARC_13`	11	`V-simm13`	`S + A`
`R_SPARC_LO10`	12	`T-simm13`	`(S + A) & 0x3ff`
`R_SPARC_GOT10`	13	`T-simm13`	`G & 0x3ff`
`R_SPARC_GOT13`	14	`V-simm13`	`G`
`R_SPARC_GOT22`	15	`T-simm22`	`G >> 10`
`R_SPARC_PC10`	16	`T-simm13`	`(S + A - P) & 0x3ff`
`R_SPARC_PC22`	17	`V-disp22`	`(S + A - P) >> 10`
`R_SPARC_WPLT30`	18	`V-disp30`	`(L + A - P) >> 2`
`R_SPARC_COPY`	19	None	None
`R_SPARC_GLOB_DAT`	20	`V-word32`	`S + A`
`R_SPARC_JMP_SLOT`	21	None	See `R_SPARC_JMP_SLOT`
`R_SPARC_RELATIVE`	22	`V-word32`	`B + A`

TABLE 7–19 SPARC: ELF Relocation Types *(Continued)*

Name	Value	Field	Calculation
R_SPARC_UA32	23	V-word32	S + A
R_SPARC_PLT32	24	V-word32	L + A
R_SPARC_HIPLT22	25	T-imm22	(L + A) >> 10
R_SPARC_LOPLT10	26	T-simm13	(L + A) & 0x3ff
R_SPARC_PCPLT32	27	V-word32	L + A - P
R_SPARC_PCPLT22	28	V-disp22	(L + A - P) >> 10
R_SPARC_PCPLT10	29	V-simm13	(L + A - P) & 0x3ff
R_SPARC_10	30	V-simm10	S + A
R_SPARC_11	31	V-simm11	S + A
R_SPARC_HH22	34	V-imm22	(S + A) >> 42
R_SPARC_HM10	35	T-simm13	((S + A) >> 32) & 0x3ff
R_SPARC_LM22	36	T-imm22	(S + A) >> 10
R_SPARC_PC_HH22	37	V-imm22	(S + A - P) >> 42
R_SPARC_PC_HM10	38	T-simm13	((S + A - P) >> 32) & 0x3ff
R_SPARC_PC_LM22	39	T-imm22	(S + A - P) >> 10
R_SPARC_WDISP16	40	V-d2/disp14	(S + A - P) >> 2
R_SPARC_WDISP19	41	V-disp19	(S + A - P) >> 2
R_SPARC_7	43	V-imm7	S + A
R_SPARC_5	44	V-imm5	S + A
R_SPARC_6	45	V-imm6	S + A
R_SPARC_HIX22	48	V-imm22	((S + A) ^ 0xffffffffffffffff) >> 10
R_SPARC_LOX10	49	T-simm13	((S + A) & 0x3ff) | 0x1c00
R_SPARC_H44	50	V-imm22	(S + A) >> 22
R_SPARC_M44	51	T-imm10	((S + A) >> 12) & 0x3ff
R_SPARC_L44	52	T-imm13	(S + A) & 0xfff
R_SPARC_REGISTER	53	V-word32	S + A
R_SPARC_UA16	55	V-half16	S + A

TABLE 7–19 SPARC: ELF Relocation Types *(Continued)*

Name	Value	Field	Calculation
R_SPARC_GOTDATA_HIX22	80	T-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 42)
R_SPARC_GOTDATA_LOX22	81	T-imm13	((S + A - GOT) & 0x3ff) | (0x1c00 ^~ (((S + A - GOT) >> 50) & 0x1c00))
R_SPARC_GOTDATA_OP_HIX22	82	T-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 42)
R_SPARC_GOTDATA_OP_LOX22	83	T-imm13	((S + A - GOT) & 0x3ff) | (0x1c00 ^~ (((S + A - GOT) >> 50) & 0x1c00))
R_SPARC_GOTDATA_OP	84	Word32	special

Note – Additional relocations are available for thread-local storage references. These relocations are covered in Chapter 8.

Some relocation types have semantics beyond simple calculation:

`R_SPARC_GOT10`
: Resembles `R_SPARC_LO10`, except that the relocation refers to the address of the symbol's global offset table entry. Additionally, `R_SPARC_GOT10` instructs the link-editor to create a global offset table.

`R_SPARC_GOT13`
: Resembles `R_SPARC_13`, except that the relocation refers to the address of the symbol's global offset table entry. Additionally, `R_SPARC_GOT13` instructs the link-editor to create a global offset table.

`R_SPARC_GOT22`
: Resembles `R_SPARC_22`, except that the relocation refers to the address of the symbol's global offset table entry. Additionally, `R_SPARC_GOT22` instructs the link-editor to create a global offset table.

`R_SPARC_WPLT30`
: Resembles `R_SPARC_WDISP30`, except that the relocation refers to the address of the symbol's procedure linkage table entry. Additionally, `R_SPARC_WPLT30` instructs the link-editor to create a procedure linkage table.

R_SPARC_COPY
Created by the link-editor for dynamic executables to preserve a read-only text segment. The relocation offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the runtime linker copies data associated with the shared object's symbol to the location specified by the offset. See "Copy Relocations" on page 125.

R_SPARC_GLOB_DAT
Resembles R_SPARC_32, except that the relocation sets a global offset table entry to the address of the specified symbol. The special relocation type enables you to determine the correspondence between symbols and global offset table entries.

R_SPARC_JMP_SLOT
Created by the link-editor for dynamic objects to provide lazy binding. The relocation offset member gives the location of a procedure linkage table entry. The runtime linker modifies the procedure linkage table entry to transfer control to the designated symbol address.

R_SPARC_RELATIVE
Created by the link-editor for dynamic objects. The relocation offset member gives the location within a shared object that contains a value representing a relative address. The runtime linker computes the corresponding virtual address by adding the virtual address at which the shared object is loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_SPARC_UA32
Resembles R_SPARC_32, except that the relocation refers to an unaligned word. The word to be relocated must be treated as four separate bytes with arbitrary alignment, not as a word aligned according to the architecture requirements.

R_SPARC_LM22
Resembles R_SPARC_HI22, except that the relocation truncates rather than validates.

R_SPARC_PC_LM22
Resembles R_SPARC_PC22, except that the relocation truncates rather than validates.

R_SPARC_HIX22
Used with R_SPARC_LOX10 for executables that are confined to the uppermost 4 gigabytes of the 64–bit address space. Similar to R_SPARC_HI22, but supplies ones complement of linked value.

R_SPARC_LOX10
Used with R_SPARC_HIX22. Similar to R_SPARC_LO10, but always sets bits 10 through 12 of the linked value.

R_SPARC_L44
Used with the R_SPARC_H44 and R_SPARC_M44 relocation types to generate a 44-bit absolute addressing model.

R_SPARC_REGISTER
Used to initialize a register symbol. The relocation offset member contains the register number to be initialized. A corresponding register symbol must exist for this register. The symbol must be of type SHN_ABS.

R_SPARC_GOTDATA_OP_HIX22, R_SPARC_GOTDATA_OP_LOX22, and R_SPARC_GOTDATA_OP
These relocations provide for code transformations.

64-bit SPARC: Relocation Types

The following notation, used in relocation calculation, is unique to 64–bit SPARC.

O The secondary addend used to compute the value of the relocation field. This addend is extracted from the r_info field by applying the ELF64_R_TYPE_DATA macro.

The relocations listed in the following table extend, or alter, the relocations defined for 32–bit SPARC. See "SPARC: Relocation Types" on page 218.

TABLE 7–20 64-bit SPARC: ELF Relocation Types

Name	Value	Field	Calculation
R_SPARC_HI22	9	V-imm22	(S + A) >> 10
R_SPARC_GLOB_DAT	20	V-xword64	S + A
R_SPARC_RELATIVE	22	V-xword64	B + A
R_SPARC_64	32	V-xword64	S + A
R_SPARC_OLO10	33	V-simm13	((S + A) & 0x3ff) + O
R_SPARC_DISP64	46	V-xword64	S + A - P
R_SPARC_PLT64	47	V-xword64	L + A
R_SPARC_REGISTER	53	V-xword64	S + A
R_SPARC_UA64	54	V-xword64	S + A
R_SPARC_H34	85	V-imm22	(S + A) >> 12

The following relocation type has semantics beyond simple calculation:

R_SPARC_OLO10
Resembles R_SPARC_LO10, except that an extra offset is added to make full use of the 13-bit signed immediate field.

x86: Relocation Types

The relocations listed in the following table are defined for 32–bit x86.

TABLE 7–21 x86: ELF Relocation Types

Name	Value	Field	Calculation
R_386_NONE	0	None	None
R_386_32	1	word32	S + A
R_386_PC32	2	word32	S + A - P
R_386_GOT32	3	word32	G + A
R_386_PLT32	4	word32	L + A - P
R_386_COPY	5	None	None
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P
R_386_32PLT	11	word32	L + A

Note – Additional relocations are available for thread-local storage references. These relocations are covered in Chapter 8.

Some relocation types have semantics beyond simple calculation:

R_386_GOT32
: Computes the distance from the base of the global offset table to the symbol's global offset table entry. The relocation also instructs the link-editor to create a global offset table.

R_386_PLT32
: Computes the address of the symbol's procedure linkage table entry and instructs the link-editor to create a procedure linkage table.

R_386_COPY
: Created by the link-editor for dynamic executables to preserve a read-only text segment. The relocation offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the runtime linker copies data associated with the shared object's symbol to the location specified by the offset. See "Copy Relocations" on page 125.

R_386_GLOB_DAT
: Used to set a global offset table entry to the address of the specified symbol. The special relocation type enable you to determine the correspondence between symbols and global offset table entries.

R_386_JMP_SLOT
: Created by the link-editor for dynamic objects to provide lazy binding. The relocation offset member gives the location of a procedure linkage table entry. The runtime linker modifies the procedure linkage table entry to transfer control to the designated symbol address.

R_386_RELATIVE
: Created by the link-editor for dynamic objects. The relocation offset member gives the location within a shared object that contains a value representing a relative address. The runtime linker computes the corresponding virtual address by adding the virtual address at which the shared object is loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_386_GOTOFF
: Computes the difference between a symbol's value and the address of the global offset table. The relocation also instructs the link-editor to create the global offset table.

R_386_GOTPC
: Resembles R_386_PC32, except that it uses the address of the global offset table in its calculation. The symbol referenced in this relocation normally is _GLOBAL_OFFSET_TABLE_, which also instructs the link-editor to create the global offset table.

String Table Section

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. You reference a string as an index into the string table section.

The first byte, which is index zero, holds a null character. Likewise, a string table's last byte holds a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context.

An empty string table section is permitted. The section header's sh_size member contains zero. Nonzero indexes are invalid for an empty string table.

A section header's sh_name member holds an index into the section header string table section, as designated by the e_shstrndx member of the ELF header. The following figure shows a string table with 25 bytes and the strings associated with various indexes.

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

FIGURE 7–7 ELF String Table

The table below shows the strings of the string table shown in the preceding figure.

TABLE 7–22 ELF String Table Indexes

Index	String
`0`	None
`1`	`name`
`7`	`Variable`
`11`	`able`
`16`	`able`
`24`	*null string*

As the example shows, a string table index can refer to any byte in the section. A string can appear more than once. References to substrings can exist. A single string can be referenced multiple times. Unreferenced strings also are allowed.

Symbol Table Section

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. See Table 7–26.

A symbol table entry has the following format, that is defined in `sys/elf.h`:

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;

typedef struct {
```

```
        Elf64_Word      st_name;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf64_Half      st_shndx;
        Elf64_Addr      st_value;
        Elf64_Xword     st_size;
} Elf64_Sym;
```

The elements of this structure are:

`st_name`
: An index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is nonzero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

`st_value`
: The value of the associated symbol. Depending on the context, this can be an absolute value, an address, and so forth. See "Symbol Values" on page 231.

`st_size`
: Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

`st_info`
: The symbol's type and binding attributes. A list of the values and meanings appears in Table 7–23. The following code shows how to manipulate the values, that are defined in `sys/elf.h`:

```
#define ELF32_ST_BIND(info)          ((info) >> 4)
#define ELF32_ST_TYPE(info)          ((info) & 0xf)
#define ELF32_ST_INFO(bind, type)    (((bind)<<4)+((type)&0xf))

#define ELF64_ST_BIND(info)          ((info) >> 4)
#define ELF64_ST_TYPE(info)          ((info) & 0xf)
#define ELF64_ST_INFO(bind, type)    (((bind)<<4)+((type)&0xf))
```

`st_other`
: A symbol's visibility. A list of the values and meanings appears in Table 7–25. The following code shows how to manipulate the values for both 32–bit and 64–bit objects. Other bits contain 0 and have no defined meaning.

```
#define ELF32_ST_VISIBILITY(o)       ((o)&0x3)
#define ELF64_ST_VISIBILITY(o)       ((o)&0x3)
```

`st_shndx`
: Every symbol table entry is defined in relation to some section. This member holds the relevant section header table index. Some section indexes indicate special meanings. See Table 7–11.

  If this member contains `SHN_XINDEX`, then the actual section header index is too large to fit in this field. The actual value is contained in the associated section of type `SHT_SYMTAB_SHNDX`.

A symbol's binding, determined from its st_info field, determines the linkage visibility and behavior.

TABLE 7–23 ELF Symbol Binding, ELF32_ST_BIND and ELF64_ST_BIND

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL
: Local symbol. These symbols are not visible outside the object file containing their definition. Local symbols of the same name can exist in multiple files without interfering with each other.

STB_GLOBAL
: Global symbols. These symbols are visible to all object files being combined. One file's definition of a global symbol satisfies another file's undefined reference to the same global symbol.

STB_WEAK
: Weak symbols. These symbols resemble global symbols, but their definitions have lower precedence.

STB_LOOS - STB_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

STB_LOPROC - STB_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics.

Global symbols and weak symbols differ in two major ways:

- When the link-editor combines several relocatable object files, it does not allow multiple definitions of STB_GLOBAL symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name does not cause an error. The link-editor honors the global definition and ignores the weak ones.

  Similarly, if a common symbol exists, the appearance of a weak symbol with the same name does not cause an error. The link-editor uses the common definition and ignores the weak one. A common symbol has the st_shndx field holding SHN_COMMON. See "Symbol Resolution" on page 38.

- When the link-editor searches archive libraries it extracts archive members that contain definitions of undefined or tentative global symbols. The member's definition can be either a global or a weak symbol.

  The link-editor, by default, does not extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value. The use of `-z weakextract` overrides this default behavior. It enables weak references to cause the extraction of archive members.

Note – Weak symbols are intended primarily for use in system software. Their use in application programs is discouraged.

In each symbol table, all symbols with `STB_LOCAL` binding precede the weak and global symbols. As "Sections" on page 189 describes, a symbol table section's `sh_info` section header member holds the symbol table index for the first non-local symbol.

A symbol's type, determined from its `st_info` field, provides a general classification for the associated entity.

TABLE 7–24 ELF Symbol Types, `ELF32_ST_TYPE` and `ELF64_ST_TYPE`

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6
STT_LOOS	10
STT_HIOS	12
STT_LOPROC	13
STT_SPARC_REGISTER	13
STT_HIPROC	15

`STT_NOTYPE`
  The symbol type is not specified.

STT_OBJECT
: This symbol is associated with a data object, such as a variable, an array, and so forth.

STT_FUNC
: This symbol is associated with a function or other executable code.

STT_SECTION
: This symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.

STT_FILE
: Conventionally, the symbol's name gives the name of the source file that is associated with the object file. A file symbol has STB_LOCAL binding and a section index of SHN_ABS. This symbol, if present, precedes the other STB_LOCAL symbols for the file.

  Symbol index 1 of the SHT_SYMTAB is an STT_FILE symbol representing the object file. Conventionally, this symbol is followed by the files STT_SECTION symbols. These section symbols are then followed by any global symbols that have been reduced to locals.

STT_COMMON
: This symbol labels an uninitialized common block. This symbol is treated exactly the same as STT_OBJECT.

STT_TLS
: The symbol specifies a thread-local storage entity. When defined, this symbol gives the assigned offset for the symbol, not the actual address.

  Thread-local storage relocations can only reference symbols with type STT_TLS. A reference to a symbol of type STT_TLS from an allocatable section, can only be achieved by using special thread-local storage relocations. See Chapter 8 for details. A reference to a symbol of type STT_TLS from a non-allocatable section does not have this restriction.

STT_LOOS - STT_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

STT_LOPROC - STT_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics.

A symbol's visibility is determined from its st_other field. This visibility can be specified in a relocatable object. This visibility defines how that symbol can be accessed once the symbol has become part of an executable or shared object.

TABLE 7–25 ELF Symbol Visibility

Name	Value
STV_DEFAULT	0

TABLE 7–25 ELF Symbol Visibility *(Continued)*

Name	Value
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3

STV_DEFAULT
The visibility of symbols with the STV_DEFAULT attribute is as specified by the symbol's binding type. Global symbols and weak symbols are visible outside of their defining component, the executable file or shared object. Local symbols are hidden. Global symbols and weak symbols can also be preempted. These symbols can by interposed by definitions of the same name in another component.

STV_PROTECTED
A symbol that is defined in the current component is protected if the symbol is visible in other components, but cannot be preempted. Any reference to such a symbol from within the defining component must be resolved to the definition in that component. This resolution must occur, even if a symbol definition exists in another component that would interpose by the default rules. A symbol with STB_LOCAL binding will not have STV_PROTECTED visibility.

STV_HIDDEN
A symbol that is defined in the current component is hidden if its name is not visible to other components. Such a symbol is necessarily protected. This attribute is used to control the external interface of a component. An object named by such a symbol can still be referenced from another component if its address is passed outside.

A hidden symbol contained in a relocatable object is either removed or converted to STB_LOCAL binding when the object is included in an executable file or shared object.

STV_INTERNAL
This visibility attribute is currently reserved.

None of the visibility attributes affects the resolution of symbols within an executable or shared object during link-editing. Such resolution is controlled by the binding type. Once the link-editor has chosen its resolution, these attributes impose two requirements. Both requirements are based on the fact that references in the code being linked might have been optimized to take advantage of the attributes.

- First, all of the non-default visibility attributes, when applied to a symbol reference, imply that a definition to satisfy that reference must be provided within the current executable or shared object. If this type of symbol reference has no definition within the component being linked, then the reference must have STB_WEAK binding and is resolved to zero.

- Second, if any reference to or definition of a name is a symbol with a non-default visibility attribute, the visibility attribute must be propagated to the resolving symbol in the linked object. If different visibility attributes are specified for distinct references to or definitions of a symbol, the most constraining visibility attribute must be propagated to the resolving symbol in the linked object. The attributes, ordered from least to most constraining, are STV_PROTECTED, STV_HIDDEN and STV_INTERNAL.

If a symbol's value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well. References to the symbol continue to point to the same location in the program. Some special section index values give other semantics:

SHN_ABS
This symbol has an absolute value that does not change because of relocation.

SHN_COMMON
This symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's sh_addralign member. The link-editor allocates the storage for the symbol at an address that is a multiple of st_value. The symbol's size tells how many bytes are required.

SHN_UNDEF
This section table index means the symbol is undefined. When the link-editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be bound to the actual definition.

As mentioned above, the symbol table entry for index 0 (STN_UNDEF) is reserved. This entry holds the values listed in the following table.

TABLE 7–26 ELF Symbol Table Entry: Index 0

Name	Value	Note
st_name	0	No name
st_value	0	Zero value
st_size	0	No size
st_info	0	No type, local binding
st_other	0	
st_shndx	SHN_UNDEF	No section

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the st_value member.

- In relocatable files, st_value holds alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value holds a section offset for a defined symbol. st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable and shared object files, st_value holds a virtual address. To make these files' symbols more useful for the runtime linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allow efficient access by the appropriate programs.

Register Symbols

The SPARC architecture supports register symbols, which are symbols that initialize a global register. A symbol table entry for a register symbol contains the entries listed in the following table.

TABLE 7–27 SPARC: ELF Symbol Table Entry: Register Symbol

Field	Meaning
st_name	Index into string table of the name of the symbol, or 0 for a scratch register.
st_value	Register number. See the ABI manual for integer register assignments.
st_size	Unused (0).
st_info	Bind is typically STB_GLOBAL, type must be STT_SPARC_REGISTER.
st_other	Unused (0).
st_shndx	SHN_ABS if this object initializes this register symbol; SHN_UNDEF otherwise.

The register values defined for SPARC are listed in the following table.

TABLE 7–28 SPARC: ELF Register Numbers

Name	Value	Meaning
STO_SPARC_REGISTER_G2	0x2	%g2
STO_SPARC_REGISTER_G3	0x3	%g3

Absence of an entry for a particular global register means that the particular global register is not used at all by the object.

Syminfo Table Section

The syminfo section contains multiple entries of the type Elf32_Syminfo or Elf64_Syminfo. There is one entry in the .SUNW_syminfo section for every entry in the associated symbol table (sh_link).

If this section is present in an object, additional symbol information is to be found by taking the symbol index from the associated symbol table and using that to find the corresponding Elf32_Syminfo or Elf64_Syminfo entry in this section. The associated symbol table and the Syminfo table will always have the same number of entries.

Index 0 is used to store the current version of the Syminfo table, which is SYMINFO_CURRENT. Since symbol table entry 0 is always reserved for the UNDEF symbol table entry, this does not pose any conflicts.

An Syminfo entry has the following format, that is defined in sys/link.h:

```
typedef struct {
        Elf32_Half      si_boundto;
        Elf32_Half      si_flags;
} Elf32_Syminfo;

typedef struct {
        Elf64_Half      si_boundto;
        Elf64_Half      si_flags;
} Elf64_Syminfo;
```

The elements of this structure are:

si_boundto
: This index is to an entry in the .dynamic section, identified by the sh_info field, that augments the Syminfo flags. For example, a DT_NEEDED entry identifies a dynamic object associated with the Syminfo entry. The entries that follow are reserved values for si_boundto.

TABLE 7–29 ELF si_boundto Reserved Values

Name	Value	Meaning
SYMINFO_BT_SELF	0xffff	Symbol bound to self.
SYMINFO_BT_PARENT	0xfffe	Symbol bound to parent. The parent is the first object to cause this dynamic object to be loaded.
SYMINFO_BT_NONE	0xfffd	Symbol has no special symbol binding.

si_flags
: This bit-field can have flags set, as shown in the following table.

TABLE 7–30 ELF Syminfo Flags

Name	Value	Meaning
SYMINFO_FLG_DIRECT	0x01	Symbol reference has direct association to object containing definition.
SYMINFO_FLG_COPY	0x04	Symbol definition is the result of a copy-relocation.
SYMINFO_FLG_LAZYLOAD	0x08	Symbol reference is to an object that should be lazily loaded.
SYMINFO_FLG_DIRECTBIND	0x10	Symbol reference should be bound directly to the definition.
SYMINFO_FLG_NOEXTDIRECT	0x20	Do not allow external reference to directly bind to this symbol definition.

Versioning Sections

Objects created by the link-editor can contain two types of versioning information:

- *Version definitions* provide associations of global symbols and are implemented using sections of type `SHT_SUNW_verdef` and `SHT_SUNW_versym`.
- *Version dependencies* indicate the version definition requirements from other object dependencies and are implemented using sections of type `SHT_SUNW_verneed`.

The structures that form these sections are defined in `sys/link.h`. Sections that contain versioning information are named `.SUNW_version`.

Version Definition Section

This section is defined by the type `SHT_SUNW_verdef`. If this section exists, a `SHT_SUNW_versym` section must also exist. Using these two structures, an association of symbols-to-version definitions is maintained within the file. See "Creating a Version Definition" on page 133. Elements of this section have the following structure:

```
typedef struct {
        Elf32_Half      vd_version;
        Elf32_Half      vd_flags;
        Elf32_Half      vd_ndx;
        Elf32_Half      vd_cnt;
        Elf32_Word      vd_hash;
        Elf32_Word      vd_aux;
        Elf32_Word      vd_next;
} Elf32_Verdef;

typedef struct {
        Elf32_Word      vda_name;
```

```
        Elf32_Word      vda_next;
} Elf32_Verdaux;

typedef struct {
        Elf64_Half      vd_version;
        Elf64_Half      vd_flags;
        Elf64_Half      vd_ndx;
        Elf64_Half      vd_cnt;
        Elf64_Word      vd_hash;
        Elf64_Word      vd_aux;
        Elf64_Word      vd_next;
} Elf64_Verdef;

typedef struct {
        Elf64_Word      vda_name;
        Elf64_Word      vda_next;
} Elf64_Verdaux;
```

The elements of this structure are:

`vd_version`
This member identifies the version of the structure itself, as listed in the following table.

TABLE 7–31 ELF Version Definition Structure Versions

Name	Value	Meaning
`VER_DEF_NONE`	`0`	Invalid version.
`VER_DEF_CURRENT`	`>=1`	Current version.

The value 1 signifies the original section format. Extensions will create new versions with higher numbers. The value of `VER_DEF_CURRENT` changes as necessary to reflect the current version number.

`vd_flags`
This member holds version definition-specific information, as listed in the following table.

TABLE 7–32 ELF Version Definition Section Flags

Name	Value	Meaning
`VER_FLG_BASE`	`0x1`	Version definition of the file itself.
`VER_FLG_WEAK`	`0x2`	Weak version identifier.

The base version definition is always present when version definitions, or symbol auto-reduction, have been applied to the file. The base version provides a default version for the files reserved symbols. A weak version definition has no symbols associated with it. See “Creating a Weak Version Definition” on page 136.

`vd_ndx`
: The version index. Each version definition has a unique index that is used to associate `SHT_SUNW_versym` entries to the appropriate version definition.

`vd_cnt`
: The number of elements in the `Elf32_Verdaux` array.

`vd_hash`
: The hash value of the version definition name. This value is generated using the same hashing function described in "Hash Table Section" on page 208.

`vd_aux`
: The byte offset from the start of this `Elf32_Verdef` entry to the `Elf32_Verdaux` array of version definition names. The first element of the array must exist. It points to the version definition string this structure defines. Additional elements can be present. The number of elements is indicated by the `vd_cnt` value. These elements represent the dependencies of this version definition. Each of these dependencies will have its own version definition structure.

`vd_next`
: The byte offset from the start of this `Elf32_Verdef` structure to the next `Elf32_Verdef` entry.

`vda_name`
: The string table offset to a null-terminated string, giving the name of the version definition.

`vda_next`
: The byte offset from the start of this `Elf32_Verdaux` entry to the next `Elf32_Verdaux` entry.

Version Symbol Section

The version symbol section is defined by the type `SHT_SUNW_versym`, and consists of an array of elements having the following structure:

```
typedef Elf32_Half      Elf32_Versym;
typedef Elf64_Half      Elf64_Versym;
```

The number of elements of the array must equal the number of symbol table entries contained in the associated symbol table. This number is determined by the section's `sh_link` value. Each element of the array contains a single index that can have the values shown in the following table.

TABLE 7–33 ELF Version Dependency Indexes

Name	Value	Meaning
`VER_NDX_LOCAL`	0	Symbol has local scope.

TABLE 7–33 ELF Version Dependency Indexes *(Continued)*

Name	Value	Meaning
VER_NDX_GLOBAL	1	Symbol has global scope (assigned to base version definition).
	>1	Symbol has global scope (assigned to user-defined version definition).

Any index values greater than VER_NDX_GLOBAL must correspond to the vd_ndx value of an entry in the SHT_SUNW_verdef section. If no index values greater than VER_NDX_GLOBAL exist, then no SHT_SUNW_verdef section need be present.

Version Dependency Section

The version dependency section is defined by the type SHT_SUNW_verneed. This section complements the dynamic dependency requirements of the file by indicating the version definitions required from these dependencies. A recording is made in this section only if a dependency contains version definitions. Elements of this section have the following structure:

```
typedef struct {
        Elf32_Half      vn_version;
        Elf32_Half      vn_cnt;
        Elf32_Word      vn_file;
        Elf32_Word      vn_aux;
        Elf32_Word      vn_next;
} Elf32_Verneed;

typedef struct {
        Elf32_Word      vna_hash;
        Elf32_Half      vna_flags;
        Elf32_Half      vna_other;
        Elf32_Word      vna_name;
        Elf32_Word      vna_next;
} Elf32_Vernaux;

typedef struct {
        Elf64_Half      vn_version;
        Elf64_Half      vn_cnt;
        Elf64_Word      vn_file;
        Elf64_Word      vn_aux;
        Elf64_Word      vn_next;
} Elf64_Verneed;

typedef struct {
        Elf64_Word      vna_hash;
        Elf64_Half      vna_flags;
        Elf64_Half      vna_other;
        Elf64_Word      vna_name;
        Elf64_Word      vna_next;
```

```
} Elf64_Vernaux;
```

The elements of this structure are:

`vn_version`
This member identifies the version of the structure itself, as listed in the following table.

TABLE 7–34 ELF Version Dependency Structure Versions

Name	Value	Meaning
VER_NEED_NONE	0	Invalid version.
VER_NEED_CURRENT	>=1	Current version.

The value 1 signifies the original section format. Extensions will create new versions with higher numbers. The value of VER_NEED_CURRENT changes as necessary to reflect the current version number.

`vn_cnt`
The number of elements in the Elf32_Vernaux array.

`vn_file`
The string table offset to a null-terminated string, that provides the file name having a version dependency. This name matches one of the .dynamic dependencies found in the file. See "Dynamic Section" on page 252.

`vn_aux`
The byte offset, from the start of this Elf32_Verneed entry, to the Elf32_Vernaux array of version definitions required from the associated file dependency. There must exist at least one version dependency. Additional version dependencies can be present, the number being indicated by the vn_cnt value.

`vn_next`
The byte offset, from the start of this Elf32_Verneed entry, to the next Elf32_Verneed entry.

`vna_hash`
The hash value of the version dependency name. This value is generated using the same hashing function described in "Hash Table Section" on page 208.

`vna_flags`
Version dependency specific information, as listed in the following table.

TABLE 7–35 ELF Version Dependency Structure Flags

Name	Value	Meaning
VER_FLG_WEAK	0x2	Weak version identifier.

A weak version dependency indicates an original binding to a weak version definition.

vna_other
Presently unused.

vna_name
The string table offset to a null-terminated string, giving the name of the version dependency.

vna_next
The byte offset from the start of this Elf32_Vernaux entry to the next Elf32_Vernaux entry.

Dynamic Linking

This section describes the object file information and system actions that create running programs. Most information here applies to all systems. Information specific to one processor resides in sections marked accordingly.

Executable and shared object files statically represent application programs. To execute such programs, the system uses the files to create dynamic program representations, or process images. A process image has segments that contain its text, data, stack, and so on. The major subsections of this section are:

- "Program Header" on page 239 describes object file structures that are directly involved in program execution. The primary data structure, a program header table, locates segment images in the file and contains other information needed to create the memory image of the program.
- "Program Loading (Processor-Specific)" on page 245 describes the information used to load a program into memory.
- "Runtime Linker" on page 251 describes the information used to specify and resolve symbolic references among the object files of the process image.

Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information that the system needs to prepare the program for execution. An object file segment contains one or more sections, as described in "Segment Contents" on page 244.

Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's e_phentsize and e_phnum members..

A program header has the following structure, that is defined in sys/elf.h:

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;

typedef struct {
        Elf64_Word      p_type;
        Elf64_Word      p_flags;
        Elf64_Off       p_offset;
        Elf64_Addr      p_vaddr;
        Elf64_Addr      p_paddr;
        Elf64_Xword     p_filesz;
        Elf64_Xword     p_memsz;
        Elf64_Xword     p_align;
} Elf64_Phdr;
```

The elements of this structure are:

`p_type`
: The kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings are specified in Table 7–36.

`p_offset`
: The offset from the beginning of the file at which the first byte of the segment resides.

`p_vaddr`
: The virtual address at which the first byte of the segment resides in memory.

`p_paddr`
: The segment's physical address for systems in which physical addressing is relevant. Because the system ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

`p_filesz`
: The number of bytes in the file image of the segment, which can be zero.

`p_memsz`
: The number of bytes in the memory image of the segment, which can be zero.

`p_flags`
: Flags relevant to the segment. Type values and their meanings are specified in Table 7–37.

`p_align`
: Loadable process segments must have congruent values for `p_vaddr` and `p_offset`, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, `p_align` should be a positive, integral power of 2, and

p_vaddr should equal p_offset, modulo p_align. See “Program Loading (Processor-Specific)” on page 245.

Some entries describe process segments. Other entries give supplementary information and do not contribute to the process image. Segment entries can appear in any order, except as explicitly noted. Defined type values are listed in the following table.

TABLE 7–36 ELF Segment Types

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_TLS	7
PT_LOSUNW	0x6ffffffa
PT_SUNWBSS	0x6ffffffa
PT_SUNWSTACK	0x6ffffffb
PT_SUNWDTRACE	0x6ffffffc
PT_SUNWHWCAP	0x6ffffffd
PT_HISUNW	0x6fffffff
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL
Unused; other members' values are undefined. This type enables the program header table to contain ignored entries.

PT_LOAD
Specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the extra bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size can not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the p_vaddr member.

PT_DYNAMIC
Specifies dynamic linking information. See "Dynamic Section" on page 252.

PT_INTERP
Specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is mandatory for dynamic executable files and can occur in shared objects. It cannot occur more than once in a file. This type, if present, it must precede any loadable segment entry. See "Program Interpreter" on page 251 for details.

PT_NOTE
Specifies the location and size of auxiliary information. See "Note Section" on page 211 for details.

PT_SHLIB
Reserved but has unspecified semantics.

PT_PHDR
Specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type cannot occur more than once in a file. Moreover, it can occur only if the program header table is part of the memory image of the program. This type, if present, must precede any loadable segment entry. See "Program Interpreter" on page 251 for details.

PT_TLS
Specifies a thread-local storage template. See "Thread-Local Storage Section" on page 276 for details.

PT_LOSUNW - PT_HISUNW
Values in this inclusive range are reserved for Sun-specific semantics.

PT_SUNWBSS
The same attributes as a PT_LOAD element and used to describe a .SUNW_bss section.

PT_SUNWSTACK
Describes a process stack. Presently only one such element can exist, and only access permissions, as defined in the p_flags field, are meaningful.

PT_SUNWDTRACE
Reserved for internal use by dtrace(1M).

PT_SUNWHWCAP
Specifies hardware capability requirements. See "Hardware and Software Capabilities Section" on page 207 for details.

PT_LOPROC - PT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Note – Unless specifically required elsewhere, all program header segment types are optional. A file's program header table can contain only those elements relevant to its contents.

Base Address

Executable and shared object files have a base address, which is the lowest virtual address associated with the memory image of the program's object file. One use of the base address is to relocate the memory image of the program during dynamic linking.

An executable or shared object file's base address is calculated during execution from three values: the memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. The virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image. See "Program Loading (Processor-Specific)" on page 245.

To compute the base address, you determine the memory address associated with the lowest `p_vaddr` value for a `PT_LOAD` segment. You then obtain the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the kind of file being loaded into memory, the memory address might not match the `p_vaddr` values.

Segment Permissions

A program to be loaded by the system must have at least one loadable segment, although this is not required by the file format. When the system creates loadable segment memory images, it gives access permissions, as specified in the `p_flags` member. All bits included in the `PF_MASKPROC` mask are reserved for processor-specific semantics.

TABLE 7–37 ELF Segment Flags

Name	Value	Meaning
`PF_X`	`0x1`	Execute
`PF_W`	`0x2`	Write
`PF_R`	`0x4`	Read
`PF_MASKPROC`	`0xf0000000`	Unspecified

If a permission bit is 0, that bit's type of access is denied. Actual memory permissions depend on the memory management unit, which can vary from one system to another. Although all flag combinations are valid, the system can grant more access than requested. In no case, however, will a segment have write permission unless it is specified explicitly. The following table lists both the exact flag interpretation and the allowable flag interpretation.

TABLE 7–38 ELF Segment Permissions

Flags	Value	Exact	Allowable
None	0	All access denied	All access denied
PF_X	1	Execute only	Read, execute
PF_W	2	Write only	Read, write, execute
PF_W + PF_X	3	Write, execute	Read, write, execute
PF_R	4	Read only	Read, execute
PF_R + PF_X	5	Read, execute	Read, execute
PF_R + PF_W	6	Read, write	Read, write, execute
PF_R + PF_W + PF_X	7	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute, but not write permissions. Data segments normally have read, write, and execute permissions.

Segment Contents

An object file segment consists of one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The diagrams below illustrate segment contents in general terms. The order and membership of sections within a segment can vary.

Text segments contain read-only instructions and data. Data segments contain writable data and instructions. See Table 7–16 for a list of all special sections.

A PT_DYNAMIC program header element points at the .dynamic section. The .got and .plt sections also hold information related to position-independent code and dynamic linking.

The .plt can reside in a text or a data segment, depending on the processor. See "Global Offset Table (Processor-Specific)" on page 264 and "Procedure Linkage Table (Processor-Specific)" on page 265 for details.

The .bss section has the type SHT_NOBITS. Although it occupies no space in the file, it contributes to the segment's memory image. Normally, these uninitialized data reside at the end of the segment, thereby making p_memsz larger than p_filesz in the associated program header element.

Program Loading (Processor-Specific)

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When, and if, the system physically reads the file depends on the program's execution behavior, system load, and so forth.

A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore, delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for 32–bit segments are congruent modulo 64K (`0x10000`). Virtual addresses and file offsets for 64–bit segments are congruent modulo 1 megabyte (`0x100000`). By aligning segments to the maximum page size, the files are suitable for paging regardless of physical page size.

By default, 64–bit SPARC programs are linked with a starting address of `0x100000000`. The whole program is above 4 gigabytes, including its text, data, heap, stack, and shared object dependencies. This helps ensure that 64–bit programs are correct because the program will fault in the least significant 4 gigabytes of its address space if it truncates any of its pointers. While 64–bit programs are linked above 4 gigabytes, you can still link them below 4 gigabytes by using a `mapfile` and the `-M` option to the compiler or link-editor. See `/usr/lib/ld/sparcv9/map.below4G`.

The following figure presents the SPARC version of the executable file.

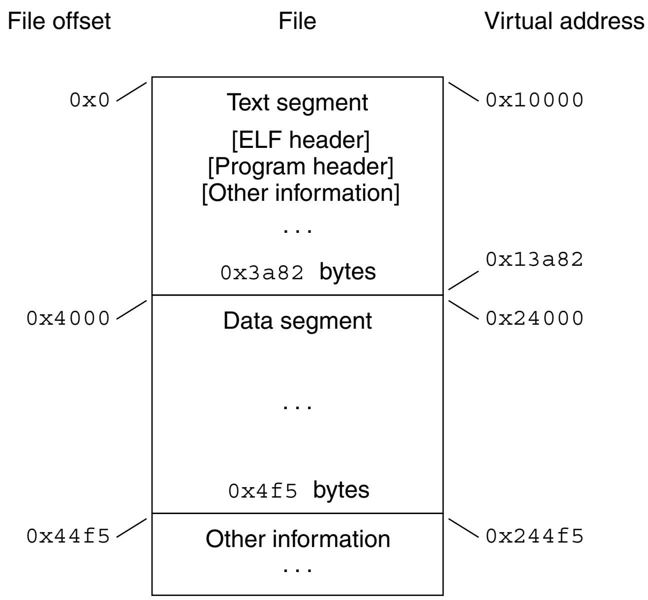


FIGURE 7–8 SPARC: Executable File (64K alignment)

The following table defines the loadable segment elements for the previous figure.

TABLE 7–39 SPARC: ELF Program Header Segments (64K alignment)

Member	Text	Data
`p_type`	`PT_LOAD`	`PT_LOAD`
`p_offset`	`0x0`	`0x4000`
`p_vaddr`	`0x10000`	`0x24000`
`p_paddr`	Unspecified	Unspecified
`p_filesize`	`0x3a82`	`0x4f5`
`p_memsz`	`0x3a82`	`0x10a4`
`p_flags`	`PF_R + PF_X`	`PF_R + PF_W + PF_X`
`p_align`	`0x10000`	`0x10000`

The following figure presents the x86 version of the executable file.

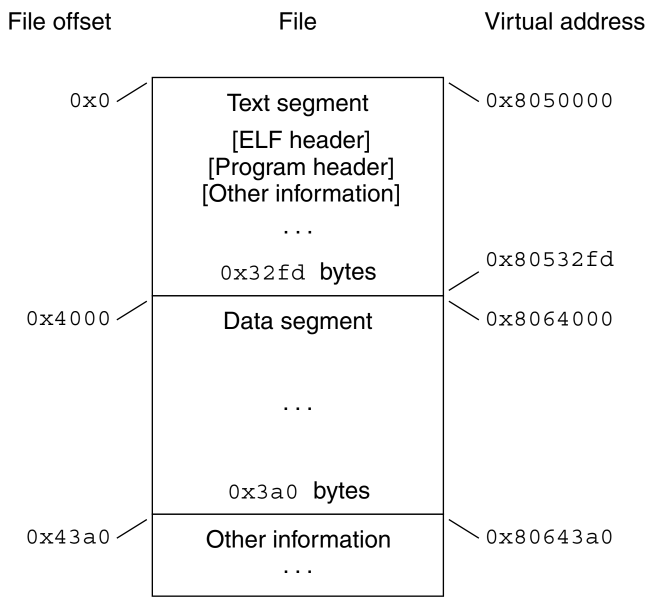


FIGURE 7–9 x86: Executable File (64K alignment)

The following table defines the loadable segment elements for the previous figure.

TABLE 7–40 x86: ELF Program Header Segments (64K alignment)

Member	Text	Data
`p_type`	`PT_LOAD`	`PT_LOAD`
`p_offset`	`0x0`	`0x4000`
`p_vaddr`	`0x8050000`	`0x8064000`
`p_paddr`	Unspecified	Unspecified
`p_filesize`	`0x32fd`	`0x3a0`
`p_memsz`	`0x32fd`	`0xdc4`
`p_flags`	`PF_R + PF_X`	`PF_R + PF_W + PF_X`
`p_align`	`0x10000`	`0x10000`

The example's file offsets and virtual addresses are congruent modulo the maximum page size for both text and data. Up to four file pages hold impure text or data depending on page size and file system block size.

- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.

- The first data page has a copy of the end of text.
- The last data page can contain file information not relevant to the running process. Logically, the system enforces the memory permissions as if each segment were complete and separate The segments addresses are adjusted to ensure that each logical page in the address space has a single set of permissions. In the examples above, the region of the file holding the end of text and the beginning of data is mapped twice: at one virtual address for text and at a different virtual address for data.

Note – The examples above reflect typical Solaris system binaries that have their text segments rounded.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. If a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file.

Impurities in the other three pages are not logically part of the process image. Whether the system expunges these impurities is unspecified. The memory image for this program is shown in the following figures, assuming 4 Kbyte (`0x1000`) pages. For simplicity, these figures illustrate only one page size.

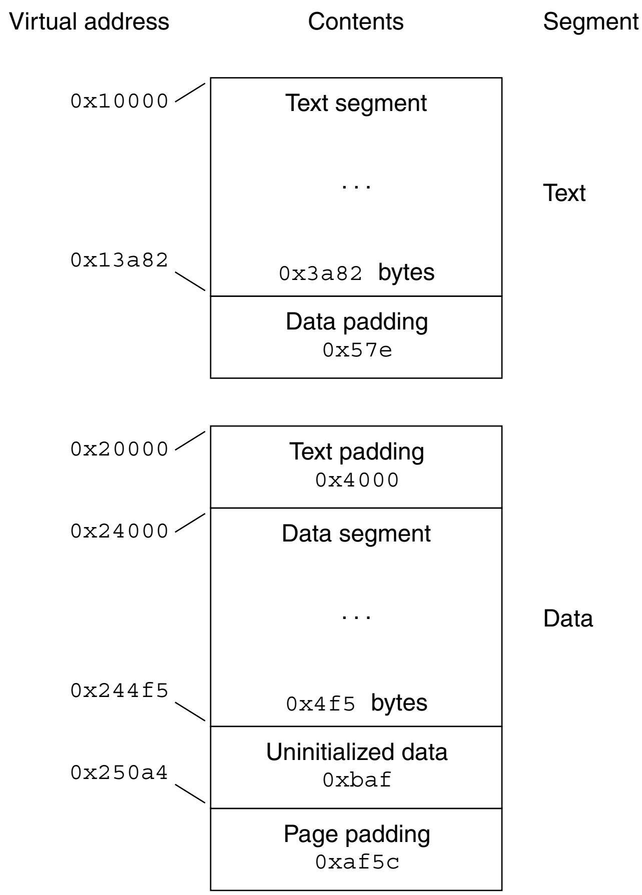


FIGURE 7–10 SPARC: Process Image Segments

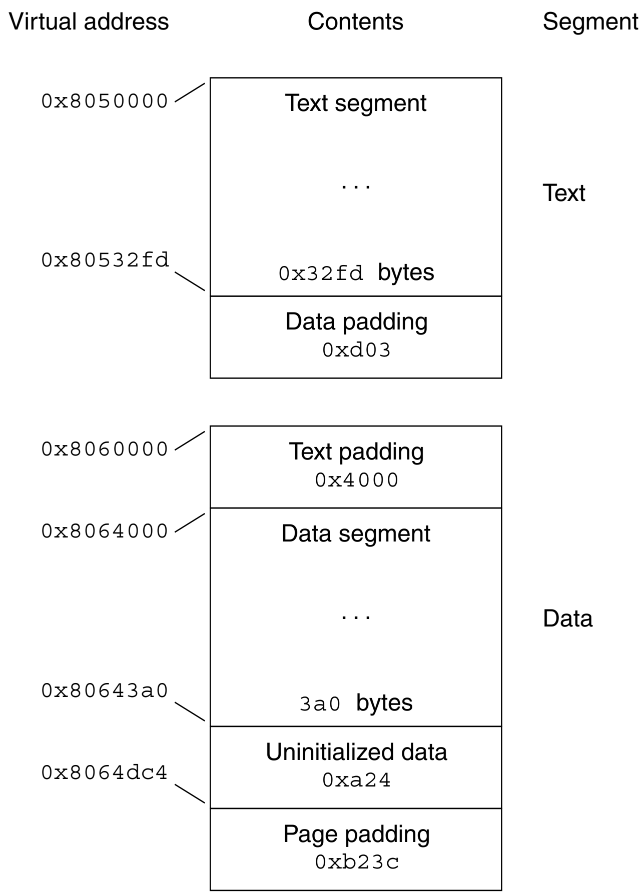


FIGURE 7–11 x86: Process Image Segments

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. For the process to execute correctly, the segments must reside at the virtual addresses used to create the executable file. The system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This code enables a segment's virtual address change from one process to another, without invalidating execution behavior.

Though the system chooses virtual addresses for individual processes, it maintains the relative positions of the segments. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file.

The following tables show possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The tables also include the base address computations.

TABLE 7–41 SPARC: ELF Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	`0x0`	`0x4000`	`0x0`
Process 1	`0xc0000000`	`0xc0024000`	`0xc0000000`
Process 2	`0xc0010000`	`0xc0034000`	`0xc0010000`
Process 3	`0xd0020000`	`0xd0024000`	`0xd0020000`
Process 4	`0xd0030000`	`0xd0034000`	`0xd0030000`

TABLE 7–42 x86: ELF Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	`0x0`	`0x4000`	`0x0`
Process 1	`0x8000000`	`0x8004000`	`0x80000000`
Process 2	`0x80081000`	`0x80085000`	`0x80081000`
Process 3	`0x900c0000`	`0x900c4000`	`0x900c0000`
Process 4	`0x900c6000`	`0x900ca000`	`0x900c6000`

Program Interpreter

A dynamic executable or shared object that initiates dynamic linking can have one `PT_INTERP` program header element. During `exec`(2), the system retrieves a path name from the `PT_INTERP` segment and creates the initial process image from the interpreter file's segments. The interpreter is responsible for receiving control from the system and providing an environment for the application program.

In the Solaris OS, the interpreter is known as the runtime linker, `ld.so.1`(1).

Runtime Linker

When creating a dynamic object that initiates dynamic linking, the link-editor adds a program header element of type `PT_INTERP` to an executable file. This element instructing the system to invoke the runtime linker as the program interpreter. `exec`(2) and the runtime linker cooperate to create the process image for the program.

The link-editor constructs various data for executable and shared object files that assist the runtime linker. These data reside in loadable segments, making them available during execution. These segments include:

- A .dynamic section with type SHT_DYNAMIC that holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The .got and .plt sections with type SHT_PROGBITS that hold two separate tables: the global offset table and the procedure linkage table. Sections below explain how the runtime linker uses and changes the tables to create memory images for object files.
- The .hash section with type SHT_HASH that holds a symbol hash table.

Shared objects can occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The runtime linker relocates the memory image, updating absolute addresses before the application gains control.

Dynamic Section

If an object file participates in dynamic linking, its program header table will have an element of type PT_DYNAMIC. This segment contains the .dynamic section. A special symbol, _DYNAMIC, labels the section, which contains an array of the following structures, that are defined in sys/link.h:

```
typedef struct {
        Elf32_Sword d_tag;
        union {
                Elf32_Word      d_val;
                Elf32_Addr      d_ptr;
                Elf32_Off       d_off;
        } d_un;
} Elf32_Dyn;

typedef struct {
        Elf64_Xword d_tag;
        union {
                Elf64_Xword     d_val;
                Elf64_Addr      d_ptr;
        } d_un;
} Elf64_Dyn;
```

For each object with this type, d_tag controls the interpretation of d_un.

d_val
: These objects represent integer values with various interpretations.

d_ptr
: These objects represent program virtual addresses. A file's virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the runtime linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do not contain relocation entries to *correct* addresses in the dynamic structure.

To make interpreting the contents of dynamic section entries simpler for tools, the value of each tag, except for those in two special compatibility ranges, will determine the interpretation of the d_un union. A tag whose value is an even number indicates a dynamic section entry that uses d_ptr. A tag whose value is an odd number indicates a dynamic section entry that uses d_val or that uses neither d_ptr nor d_val. Tags whose values are less than the special value DT_ENCODING and tags whose values fall between DT_HIOS and DT_LOPROC do not follow these rules.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked *mandatory*, then the dynamic linking array must have an entry of that type. Likewise, *optional* means an entry for the tag can appear but is not required.

TABLE 7–43 ELF Dynamic Array Tags

Name	Value	d_un	Executable	Shared Object
DT_NULL	0	Ignored	Mandatory	Mandatory
DT_NEEDED	1	d_val	Optional	Optional
DT_PLTRELSZ	2	d_val	Optional	Optional
DT_PLTGOT	3	d_ptr	Optional	Optional
DT_HASH	4	d_ptr	Mandatory	Mandatory
DT_STRTAB	5	d_ptr	Mandatory	Mandatory
DT_SYMTAB	6	d_ptr	Mandatory	Mandatory
DT_RELA	7	d_ptr	Mandatory	Optional
DT_RELASZ	8	d_val	Mandatory	Optional
DT_RELAENT	9	d_val	Mandatory	Optional
DT_STRSZ	10	d_val	Mandatory	Mandatory
DT_SYMENT	11	d_val	Mandatory	Mandatory
DT_INIT	12	d_ptr	Optional	Optional
DT_FINI	13	d_ptr	Optional	Optional
DT_SONAME	14	d_val	Ignored	Optional
DT_RPATH	15	d_val	Optional	Optional
DT_SYMBOLIC	16	Ignored	Ignored	Optional
DT_REL	17	d_ptr	Mandatory	Optional
DT_RELSZ	18	d_val	Mandatory	Optional
DT_RELENT	19	d_val	Mandatory	Optional

TABLE 7–43 ELF Dynamic Array Tags *(Continued)*

Name	Value	d_un	Executable	Shared Object
DT_PLTREL	20	d_val	Optional	Optional
DT_DEBUG	21	d_ptr	Optional	Ignored
DT_TEXTREL	22	Ignored	Optional	Optional
DT_JMPREL	23	d_ptr	Optional	Optional
DT_BIND_NOW	24	Ignored	Optional	Optional
DT_INIT_ARRAY	25	d_ptr	Optional	Optional
DT_FINI_ARRAY	26	d_ptr	Optional	Optional
DT_INIT_ARRAYSZ	27	d_val	Optional	Optional
DT_FINI_ARRAYSZ	28	d_val	Optional	Optional
DT_RUNPATH	29	d_val	Optional	Optional
DT_FLAGS	30	d_val	Optional	Optional
DT_ENCODING	32	Unspecified	Unspecified	Unspecified
DT_PREINIT_ARRAY	32	d_ptr	Optional	Ignored
DT_PREINIT_ARRAYSZ	33	d_val	Optional	Ignored
DT_LOOS	0x6000000d	Unspecified	Unspecified	Unspecified
DT_SUNW_RTLDINF	0x6000000e	d_ptr	Optional	Optional
DT_HIOS	0x6ffff000	Unspecified	Unspecified	Unspecified
DT_VALRNGLO	0x6ffffd00	Unspecified	Unspecified	Unspecified
DT_CHECKSUM	0x6ffffdf8	d_val	Optional	Optional
DT_PLTPADSZ	0x6ffffdf9	d_val	Optional	Optional
DT_MOVEENT	0x6ffffdfa	d_val	Optional	Optional
DT_MOVESZ	0x6ffffdfb	d_val	Optional	Optional
DT_FEATURE_1	0x6ffffdfc	d_val	Optional	Optional
DT_POSFLAG_1	0x6ffffdfd	d_val	Optional	Optional
DT_SYMINSZ	0x6ffffdfe	d_val	Optional	Optional
DT_SYMINENT	0x6ffffdff	d_val	Optional	Optional
DT_VALRNGHI	0x6ffffdff	Unspecified	Unspecified	Unspecified
DT_ADDRRNGLO	0x6ffffe00	Unspecified	Unspecified	Unspecified
DT_CONFIG	0x6ffffefa	d_ptr	Optional	Optional

TABLE 7–43 ELF Dynamic Array Tags *(Continued)*

Name	Value	d_un	Executable	Shared Object
DT_DEPAUDIT	0x6ffffefb	d_ptr	Optional	Optional
DT_AUDIT	0x6ffffefc	d_ptr	Optional	Optional
DT_PLTPAD	0x6ffffefd	d_ptr	Optional	Optional
DT_MOVETAB	0x6ffffefe	d_ptr	Optional	Optional
DT_SYMINFO	0x6ffffeff	d_ptr	Optional	Optional
DT_ADDRRNGHI	0x6ffffeff	Unspecified	Unspecified	Unspecified
DT_RELACOUNT	0x6ffffff9	d_val	Optional	Optional
DT_RELCOUNT	0x6ffffffa	d_val	Optional	Optional
DT_FLAGS_1	0x6ffffffb	d_val	Optional	Optional
DT_VERDEF	0x6ffffffc	d_ptr	Optional	Optional
DT_VERDEFNUM	0x6ffffffd	d_val	Optional	Optional
DT_VERNEED	0x6ffffffe	d_ptr	Optional	Optional
DT_VERNEEDNUM	0x6fffffff	d_val	Optional	Optional
DT_LOPROC	0x70000000	Unspecified	Unspecified	Unspecified
DT_SPARC_REGISTER	0x70000001	d_val	Optional	Optional
DT_AUXILIARY	0x7ffffffd	d_val	Unspecified	Optional
DT_USED	0x7ffffffe	d_val	Optional	Optional
DT_FILTER	0x7fffffff	d_val	Unspecified	Optional
DT_HIPROC	0x7fffffff	Unspecified	Unspecified	Unspecified

DT_NULL
Marks the end of the _DYNAMIC array.

DT_NEEDED
The DT_STRTAB string table offset of a null-terminated string, giving the name of a needed dependency. The dynamic array can contain multiple entries of this type. The relative order of these entries is significant, though their relation to entries of other types is not. See "Shared Object Dependencies" on page 66.

DT_PLTRELSZ
The total size, in bytes, of the relocation entries associated with the procedure linkage table. See "Procedure Linkage Table (Processor-Specific)" on page 265.

DT_PLTGOT
: An address associated with the procedure linkage table or the global offset table. See "Procedure Linkage Table (Processor-Specific)" on page 265 and "Global Offset Table (Processor-Specific)" on page 264.

DT_HASH
: The address of the symbol hash table. This table refers to the symbol table indicated by the DT_SYMTAB element. See "Hash Table Section" on page 208.

DT_STRTAB
: The address of the string table. Symbol names, dependency names, and other strings required by the runtime linker reside in this table. See "String Table Section" on page 224.

DT_SYMTAB
: The address of the symbol table. See "Symbol Table Section" on page 225.

DT_RELA
: The address of a relocation table. See "Relocation Sections" on page 213.

  An object file can have multiple relocation sections. When creating the relocation table for an executable or shared object file, the link-editor catenates those sections to form a single table. Although the sections can remain independent in the object file, the runtime linker sees a single table. When the runtime linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions.

  This element requires the DT_RELASZ and DT_RELAENT elements also be present. When relocation is mandatory for a file, either DT_RELA or DT_REL can occur.

DT_RELASZ
: The total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT
: The size, in bytes, of the DT_RELA relocation entry.

DT_STRSZ
: The total size, in bytes, of the DT_STRTAB string table.

DT_SYMENT
: The size, in bytes, of the DT_SYMTAB symbol entry.

DT_INIT
: The address of an initialization function. See "Initialization and Termination Sections" on page 35.

DT_FINI
: The address of a termination function. See "Initialization and Termination Sections" on page 35.

DT_SONAME
: The DT_STRTAB string table offset of a null-terminated string, identifying the name of the shared object. See "Recording a Shared Object Name" on page 105.

DT_RPATH
The DT_STRTAB string table offset of a null-terminated library search path string. This element's use has been superseded by DT_RUNPATH. See "Directories Searched by the Runtime Linker" on page 66.

DT_SYMBOLIC
Indicates the object contains symbolic bindings that were applied during its link-edit. This elements use has been superseded by the DF_SYMBOLIC flag. See "Using -B symbolic" on page 128.

DT_REL
Similar to DT_RELA, except its table has implicit addends. This element requires that the DT_RELSZ and DT_RELENT elements also be present.

DT_RELSZ
The total size, in bytes, of the DT_REL relocation table.

DT_RELENT
The size, in bytes, of the DT_REL relocation entry.

DT_PLTREL
Indicates the type of relocation entry to which the procedure linkage table refers, either DT_REL or DT_RELA. All relocations in a procedure linkage table must use the same relocation. See "Procedure Linkage Table (Processor-Specific)" on page 265. This element requires a DT_JMPREL element also be present.

DT_DEBUG
Used for debugging.

DT_TEXTREL
Indicates that one or more relocation entries might request modifications to a non-writable segment, and the runtime linker can prepare accordingly. This element's use has been superseded by the DF_TEXTREL flag. See "Position-Independent Code" on page 118.

DT_JMPREL
The address of relocation entries associated solely with the procedure linkage table. See "Procedure Linkage Table (Processor-Specific)" on page 265. Separating these relocation entries enables the runtime linker to ignore them when the object is loaded if lazy binding is enabled. This element requires the DT_PLTRELSZ and DT_PLTREL elements also be present.

DT_POSFLAG_1
Various state flags which are applied to the DT_ element immediately following. See Table 7–46.

DT_BIND_NOW
Indicates that all relocations for this object must be processed before returning control to the program. The presence of this entry takes precedence over a directive to use lazy binding when specified through the environment or via dlopen(3C). This element's use has been superseded by the DF_BIND_NOW flag. See "When Relocations Are Performed" on page 73.

DT_INIT_ARRAY
: The address of an array of pointers to initialization functions. This element requires that a DT_INIT_ARRAYSZ element also be present. See "Initialization and Termination Sections" on page 35.

DT_FINI_ARRAY
: The address of an array of pointers to termination functions. This element requires that a DT_FINI_ARRAYSZ element also be present. See "Initialization and Termination Sections" on page 35.

DT_INIT_ARRAYSZ
: The total size, in bytes, of the DT_INIT_ARRAY array.

DT_FINI_ARRAYSZ
: The total size, in bytes, of the DT_FINI_ARRAY array.

DT_RUNPATH
: The DT_STRTAB string table offset of a null-terminated library search path string. See "Directories Searched by the Runtime Linker" on page 66.

DT_FLAGS
: Flag values specific to this object. See Table 7–44.

DT_ENCODING
: Values greater than or equal to DT_ENCODING and less than or equal to DT_HIOS follow the rules for the interpretation of the d_un union.

DT_PREINIT_ARRAY
: The address of an array of pointers to pre-initialization functions. This element requires that a DT_PREINIT_ARRAYSZ element also be present. This array is processed only in an executable file. It is ignored if contained in a shared object. See "Initialization and Termination Sections" on page 35.

DT_PREINIT_ARRAYSZ
: The total size, in bytes, of the DT_PREINIT_ARRAY array.

DT_LOOS - DT_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics. All such values follow the rules for the interpretation of the d_un union.

DT_SUNW_RTLDINF
: Reserved for internal use by the runtime-linker.

DT_SYMINFO
: The address of the symbol information table. This element requires that the DT_SYMINENT and DT_SYMINSZ elements also be present. See "Syminfo Table Section" on page 233.

DT_SYMINENT
: The size, in bytes, of the DT_SYMINFO information entry.

DT_SYMINSZ
: The total size, in bytes, of the DT_SYMINFO table.

DT_VERDEF
The address of the version definition table. Elements within this table contain indexes into the string table DT_STRTAB. This element requires that the DT_VERDEFNUM element also be present. See "Version Definition Section" on page 234.

DT_VERDEFNUM
The number of entries in the DT_VERDEF table.

DT_VERNEED
The address of the version dependency table. Elements within this table contain indexes into the string table DT_STRTAB. This element requires that the DT_VERNEEDNUM element also be present. See "Version Dependency Section" on page 237.

DT_VERNEEDNUM
The number of entries in the DT_VERNEEDNUM table.

DT_RELACOUNT
Indicates the RELATIVE relocation count, that has been produced from the concatenation of all Elf32_Rela, or Elf64_Rela relocations. See "Combined Relocation Sections" on page 125.

DT_RELCOUNT
Indicates the RELATIVE relocation count, that has been produced from the concatenation of all Elf32_Rel relocations. See "Combined Relocation Sections" on page 125.

DT_AUXILIARY
The DT_STRTAB string table offset of a null-terminated string that names one or more auxiliary filtees. See "Generating Auxiliary Filters" on page 113.

DT_FILTER
The DT_STRTAB string table offset of a null-terminated string that names one or more standard filtees. See "Generating Standard Filters" on page 110.

DT_CHECKSUM
A simple checksum of selected sections of the object. See gelf_checksum(3ELF).

DT_MOVEENT
The size, in bytes, of the DT_MOVETAB move entries.

DT_MOVESZ
The total size, in bytes, of the DT_MOVETAB table.

DT_MOVETAB
The address of a move table. This element requires that the DT_MOVEENT and DT_MOVESZ elements also be present. See "Move Section" on page 209.

DT_CONFIG
The DT_STRTAB string table offset of a null-terminated string defining a configuration file. The configuration file is only meaningful in an executable, and is typically unique to this object. See "Configuring the Default Search Paths" on page 69.

DT_DEPAUDIT
: The DT_STRTAB string table offset of a null-terminated string defining one or more audit libraries. See "Runtime Linker Auditing Interface" on page 157.

DT_AUDIT
: The DT_STRTAB string table offset of a null-terminated string defining one or more audit libraries. See "Runtime Linker Auditing Interface" on page 157.

DT_FLAGS_1
: Flag values specific to this object. See Table 7–45.

DT_FEATURE_1
: Feature values specific to this object. See Table 7–47.

DT_VALRNGLO - DT_VALRNGHI
: Values in this inclusive range use the d_un.d_val field of the dynamic structure.

DT_ADDRRNGLO - DT_ADDRRNGHI
: Values in this inclusive range use the d_un.d_ptr field of the dynamic structure. If any adjustment is made to the ELF object after it has been built, these entries must be updated accordingly.

DT_SPARC_REGISTER
: The index of an STT_SPARC_REGISTER symbol within the DT_SYMTAB symbol table. There is one entry for every STT_SPARC_REGISTER symbol in the symbol table. See "Register Symbols" on page 232.

DT_LOPROC - DT_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics.

Except for the DT_NULL element at the end of the dynamic array and the relative order of DT_NEEDED and DT_POSFLAG_1 elements, entries can appear in any order. Tag values not appearing in the table are reserved.

TABLE 7–44 ELF Dynamic Flags, DT_FLAGS

Name	Value	Meaning
DF_ORIGIN	0x1	$ORIGIN processing required
DF_SYMBOLIC	0x2	Symbolic symbol resolution required
DF_TEXTREL	0x4	Text relocations exist
DF_BIND_NOW	0x8	Non-lazy binding required
DF_STATIC_TLS	0x10	Object uses static thread-local storage scheme

DF_ORIGIN
: Indicates that the object requires $ORIGIN processing. See "Locating Associated Dependencies" on page 329.

DF_SYMBOLIC
: Indicates that the object contains symbolic bindings that were applied during its link-edit. See "Using -B symbolic" on page 128.

DF_TEXTREL
Indicates that one or more relocation entries might request modifications to a non-writable segment, and the runtime linker can prepare accordingly. See "Position-Independent Code" on page 118.

DF_BIND_NOW
Indicates that all relocations for this object must be processed before returning control to the program. The presence of this entry takes precedence over a directive to use lazy binding when specified through the environment or via dlopen(3C). See "When Relocations Are Performed" on page 73.

DF_STATIC_TLS
Indicates that the object contains code using a static thread-local storage scheme. Static thread-local storage can not be used in objects that are dynamically loaded, either using dlopen(3C), or using lazy loading. Because of this restriction, the link-editor does not support the creation of a shared object that requires static thread-local storage.

TABLE 7–45 ELF Dynamic Flags, DT_FLAGS_1

Name	Value	Meaning
DF_1_NOW	0x1	Perform complete relocation processing.
DF_1_GLOBAL	0x2	Unused.
DF_1_GROUP	0x4	Indicate object is a member of a group.
DF_1_NODELETE	0x8	Object cannot be deleted from a process.
DF_1_LOADFLTR	0x10	Ensure immediate loading of filtees.
DF_1_INITFIRST	0x20	Objects' initialization occurs first.
DF_1_NOOPEN	0x40	Object can not be used with dlopen(3C).
DF_1_ORIGIN	0x80	$ORIGIN processing required.
DF_1_DIRECT	0x100	Direct bindings enabled.
DF_1_INTERPOSE	0x400	Object is an interposer.
DF_1_NODEFLIB	0x800	Ignore default library search path.
DF_1_NODUMP	0x1000	Object cannot be dumped with dldump(3C).
DF_1_CONFALT	0x2000	Object is a configuration alternative.
DF_1_ENDFILTEE	0x4000	Filtee terminates filter's search.
DF_1_DISPRELDNE	0x8000	Displacement relocation done.
DF_1_DISPRELPND	0x10000	Displacement relocation pending.
DF_1_NODIRECT	0x20000	Object contains non-direct bindings.

TABLE 7–45 ELF Dynamic Flags, DT_FLAGS_1 *(Continued)*

Name	Value	Meaning
DF_1_IGNMULDEF	0x40000	Internal use.
DF_1_NOKSYMS	0x80000	Internal use.

DF_1_NOW
Indicates that all relocations for this object must be processed before returning control to the program. The presence of this flag takes precedence over a directive to use lazy binding when specified through the environment or via dlopen(3C). See "When Relocations Are Performed" on page 73.

DF_1_GROUP
Indicates that the object is a member of a group. This flag is recorded in the object using the link-editor's -B group option. See "Object Hierarchies" on page 92.

DF_1_NODELETE
Indicates that the object cannot be deleted from a process. If the object is loaded in a process, either directly or as a dependency, with dlopen(3C), it cannot be unloaded with dlclose(3C). This flag is recorded in the object using the link-editor's -z nodelete option.

DF_1_LOADFLTR
Meaningful only for filters. Indicates that all associated filtees be processed immediately. This flag is recorded in the object using the link-editor's -z loadfltr option. See "Filtee Processing" on page 115.

DF_1_INITFIRST
Indicates that this object's initialization section be run before any other objects loaded with it. This flag is intended for specialized system libraries only, and is recorded in the object using the link-editor's -z initfirst option.

DF_1_NOOPEN
Indicates that the object cannot be added to a running process with dlopen(3C). This flag is recorded in the object using the link-editor's -z nodlopen option.

DF_1_ORIGIN
Indicates that the object requires $ORIGIN processing. See "Locating Associated Dependencies" on page 329.

DF_1_DIRECT
Indicates that the object should use direct binding information. See "Direct Binding" on page 72.

DF_1_INTERPOSE
Indicates that the objects symbol table is to interpose before all symbols except the primary load object, which is typically the executable. This flag is recorded with the link-editor's -z interpose option. See "Direct Binding" on page 72.

DF_1_NODEFLIB
Indicates that the search for dependencies of this object ignores any default library search paths. This flag is recorded in the object using the link-editor's `-z nodefaultlib` option. See "Directories Searched by the Runtime Linker" on page 34.

DF_1_NODUMP
Indicates that this object is not dumped by `dldump`(3C). Candidates for this option include objects with no relocations that might get included when generating alternative objects using `crle`(1). This flag is recorded in the object using the link-editor's `-z nodump` option.

DF_1_CONFALT
Identifies this object as a configuration alternative object generated by `crle`(1). This flag triggers the runtime linker to search for a configuration file `$ORIGIN/ld.config.`*app-name*.

DF_1_ENDFILTEE
Meaningful only for filtees. Terminates a filters search for any further filtees. This flag is recorded in the object using the link-editor's `-z endfiltee` option. See "Reducing Filtee Searches" on page 328.

DF_1_DISPRELDNE
Indicates that this object has displacement relocations applied. The displacement relocation records no longer exist within the object as they were discarded once the relocation was applied. See "Displacement Relocations" on page 60.

DF_1_DISPRELPND
Indicates that this object has displacement relocations pending. The displacement relocations exits within the object so they can be completed at runtime. See "Displacement Relocations" on page 60.

DF_1_NODIRECT
Indicates that this object contains symbols that can not be directly bound to. See "Defining Additional Symbols" on page 46.

DF_1_IGNMULDEF
Reserved for internal use by the kernel runtime-linker.

DF_1_NOKSYMS
Reserved for internal use by the kernel runtime-linker.

TABLE 7–46 ELF Dynamic Position Flags, `DT_POSFLAG_1`

Name	Value	Meaning
DF_P1_LAZYLOAD	0x1	Identify lazy loaded dependency.
DF_P1_GROUPPERM	0x2	Identify group dependency.

DF_P1_LAZYLOAD
Identifies the following `DT_NEEDED` entry as an object to be lazy loaded. This flag is recorded in the object using the link-editor's `-z lazyload` option. See "Lazy Loading of Dynamic Dependencies" on page 76.

DF_P1_GROUPPERM
Identifies the following DT_NEEDED entry as an object to be loaded as a group. This flag is recorded in the object using the link-editor's -z groupperm option. See "Isolating a Group" on page 92.

TABLE 7–47 ELF Dynamic Feature Flags, DT_FEATURE_1

Name	Value	Meaning
DTF_1_PARINIT	0x1	Partial initialization is required.
DTF_1_CONFEXP	0x2	A Configuration file is expected.

DTF_1_PARINIT
Indicates that the object requires partial initialization. See "Move Section" on page 209.

DTF_1_CONFEXP
Identifies this object as a configuration alternative object generated by crle(1). This flag triggers the runtime linker to search for a configuration file $ORIGIN/ld.config.*app-name*. This flag has the same affect as DF_1_CONFALT.

Global Offset Table (Processor-Specific)

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data. Addresses are therefore available without compromising the position-independence and shareability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values. This technique redirects position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries. After the system creates memory segments for a loadable object file, the runtime linker processes the relocation entries, some of which will be type R_SPARC_GLOB_DAT (for SPARC), or R_386_GLOB_DAT (for x86), referring to the global offset table.

The runtime linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link-editor creates an object file, the runtime linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address can appear in several tables. The runtime linker processes all the global offset table relocations before giving control to any code in the process image. This processing ensures that absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This symbol enables a program, such as the runtime linker, to find its own dynamic structure without having yet processed its relocation entries. This method is especially important for the runtime linker, because it must initialize itself without relying on other programs to relocate its memory image.

The system can choose different memory segment addresses for the same shared object in different programs. It can even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For SPARC and x86 processors, the symbol _GLOBAL_OFFSET_TABLE_ can be used to access the table. This symbol can reside in the middle of the .got section, allowing both negative and nonnegative subscripts into the array of addresses. The symbol type is an array of Elf32_Addr for 32–bit code, and an array of Elf64_Addr for 64–bit code:

```
extern  Elf32_Addr  _GLOBAL_OFFSET_TABLE_[];
extern  Elf64_Addr  _GLOBAL_OFFSET_TABLE_[];
```

Procedure Linkage Table (Processor-Specific)

The global offset table converts position-independent address calculations to absolute locations. Similarly the procedure linkage table converts position-independent function calls to absolute locations. The link-editor cannot resolve execution transfers such as function calls from one executable or shared object to another. So, the link-editor arranges to have the program transfer control to entries in the procedure linkage table. The runtime linker thus redirects the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables.

SPARC: 32–bit Procedure Linkage Table

For 32–bit SPARC dynamic objects, the procedure linkage table resides in private data. The runtime linker determines the absolute addresses of the destinations and modifies the procedure linkage table's memory image accordingly.

The first four procedure linkage table entries are reserved. The original contents of these entries are unspecified, despite the example shown in Table 7–48. Each entry in the table occupies 3 words (12 bytes), and the last table entry is followed by a nop instruction.

A relocation table is associated with the procedure linkage table. The DT_JMP_REL entry in the _DYNAMIC array gives the location of the first relocation entry. The relocation table has one entry, in the same sequence, for each non-reserved procedure linkage table entry. The relocation type of each of these entries is `R_SPARC_JMP_SLOT`. The relocation offset specifies the address of the first byte of the associated procedure linkage table entry. The symbol table index refers to the appropriate symbol.

To illustrate procedure linkage tables, Table 7–48 shows four entries: two of the four initial reserved entries, the third is a call to `name101`, and the fourth entry is a call to `name102`. The example assumes that the entry for `name102` is the table's last entry and shows the following `nop` instruction. The left column shows the instructions from the object file before dynamic linking. The right column demonstrates a possible way the runtime linker might fix the procedure linkage table entries.

TABLE 7–48 SPARC: Procedure Linkage Table Example

Object File	*Memory Segment*

`.PLT0:`
`    unimp`
`    unimp`
`    unimp`
`.PLT1:`
`    unimp`
`    unimp`
`    unimp`
	`.PLT0:`
	`    save    %sp, -64, %sp`
	`    call    runtime_linker`
	`    nop`
	`.PLT1:`
	`    .word   identification`
	`    unimp`
	`    unimp`

`.PLT101:`
`    sethi   (.-.PLT0), %g1`
`    ba,a    .PLT0`
`    nop`
`.PLT102:`
`    sethi   (.-.PLT0), %g1`
`    ba,a    .PLT0`
`    nop`
	`.PLT101:`
	`    nop`
	`    ba,a    name101`
	`    nop`
	`.PLT102:`
	`    sethi   (.-.PLT0), %g1`
	`    sethi   %hi(name102), %g1`
	`    jmpl    %g1+%lo(name102), %g0`

`    nop`	`    nop`

Following the steps below, the runtime linker and program jointly resolve the symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the runtime linker is not specified.

1. When first creating the memory image of the program, the runtime linker changes the initial procedure linkage table entries, making them transfer control to one of the runtime linker's own routines. The runtime linker also stores a word of identification information in the second entry. When the runtime linker receives control, it can examine this word to find which object called it.
2. All other procedure linkage table entries initially transfer to the first entry, letting the runtime linker to gain control at the first execution of each table entry. For example, the program calls `name101`, which transfers control to the label `.PLT101`.

3. The sethi instruction computes the distance between the current and the initial procedure linkage table entries, .PLT101 and .PLT0, respectively. This value occupies the most significant 22 bits of the %g1 register.
4. Next, the ba,a instruction jumps to .PLT0, establishing a stack frame and calls the runtime linker.
5. With the identification value, the runtime linker gets its data structures for the object, including the relocation table.
6. By shifting the %g1 value and dividing by the size of the procedure linkage table entries, the runtime linker calculates the index of the relocation entry for name101. Relocation entry 101 has type R_SPARC_JMP_SLOT, its offset specifies the address of .PLT101, and its symbol table index refers to name101. Thus, the runtime linker gets the symbol's real value, unwinds the stack, modifies the procedure linkage table entry, and transfers control to the desired destination.

The runtime linker does not have to create the instruction sequences under the memory segment column. If it does, some points deserve more explanation.

- To make the code re-entrant, the procedure linkage table's instructions are changed in a particular sequence. If the runtime linker is fixing a function's procedure linkage table entry and a signal arrives, the signal handling code must be able to call the original function with predictable and correct results.
- The runtime linker changes three words to convert an entry. The runtime linker can update only a single word atomically with regard to instruction execution. Therefore, re-entrancy is achieved by updating each word in reverse order. If a re-entrant function call occurs just prior to the last patch, the runtime linker gains control a second time. Although both invocations of the runtime linker modify the same procedure linkage table entry, their changes do not interfere with each other.
- The first sethi instruction of a procedure linkage table entry can fill the delay slot of the previous entry's jmpl instruction. Although the sethi changes the value of the %g1 register, the previous contents can be safely discarded.
- After conversion, the last procedure linkage table entry, .PLT102, needs a delay instruction for its jmpl. The required, trailing nop fills this delay slot.

Note – The different instruction sequences shown for .PLT101, and .PLT102 demonstrate how the update can be optimized for the associated destination.

The LD_BIND_NOW environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes R_SPARC_JMP_SLOT relocation entries (procedure linkage table entries) before transferring control to the program.

SPARC: 64–bit Procedure Linkage Table

For 64–bit SPARC dynamic objects, the procedure linkage table resides in private data. The runtime linker determines the absolute addresses of the destinations and modifies the procedure linkage table's memory image accordingly.

The first four procedure linkage table entries are reserved. The original contents of these entries are unspecified, despite the example shown in Table 7–49. Each of the first 32,768 entries in the table occupies 8 words (32 bytes), and must be aligned on a 32–byte boundary. The table as a whole must be aligned on a 256–byte boundary. If more than 32,768 entries are required, the remaining entries consist of 6 words (24 bytes) and 1 pointer (8 bytes). The instructions are collected together in blocks of 160 entries followed by 160 pointers. The last group of entries and pointers can contain less than 160 items. No padding is required.

Note – The numbers 32,768 and 160 are based on the limits of branch and load displacements respectively with the second rounded down to make the divisions between code and data fall on 256–byte boundaries so as to improve cache performance.

A relocation table is associated with the procedure linkage table. The `DT_JMP_REL` entry in the `_DYNAMIC` array gives the location of the first relocation entry. The relocation table has one entry, in the same sequence, for each non-reserved procedure linkage table entry. The relocation type of each of these entries is `R_SPARC_JMP_SLOT`. For the first 32,767 slots, the relocation offset specifies the address of the first byte of the associated procedure linkage table entry, the addend field is zero. The symbol table index refers to the appropriate symbol. For slots 32,768 and beyond, the relocation offset specifies the address of the first byte of the associated pointer. The addend field is the unrelocated value `-(.PLTN + 4)`. The symbol table index refers to the appropriate symbol.

To illustrate procedure linkage tables, Table 7–49 shows several entries. The first three show initial reserved entries. The following three show examples of the initial 32,768 entries together with possible resolved forms that might apply if the target address was +/- 2 Gbytes of the entry, within the lower 4 Gbytes of the address space, or anywhere respectively. The final two show examples of later entries, which consist of instruction and pointer pairs. The left column shows the instructions from the object file before dynamic linking. The right column demonstrates a possible way the runtime linker might fix the procedure linkage table entries.

TABLE 7–49 64-bit SPARC: Procedure Linkage Table Example

Object File	*Memory Segment*
.PLT0:	.PLT0:
unimp	save %sp, -176, %sp
unimp	sethi %hh(runtime_linker_0), %l0
unimp	sethi %lm(runtime_linker_0), %l1
unimp	or %l0, %hm(runtime_linker_0), %l0
unimp	sllx %l0, 32, %l0
unimp	or %l0, %l1, %l0
unimp	jmpl %l0+%lo(runtime_linker_0), %o1
unimp	mov %g1, %o0
.PLT1:	.PLT1:
unimp	save %sp, -176, %sp
unimp	sethi %hh(runtime_linker_1), %l0
unimp	sethi %lm(runtime_linker_1), %l1
unimp	or %l0, %hm(runtime_linker_1), %l0
unimp	sllx %l0, 32, %l0
unimp	or %l0, %l1, %l0
unimp	jmpl %l0+%lo(runtime_linker_0), %o1
unimp	mov %g1, %o0
.PLT2:	.PLT2:
unimp	.xword identification
.PLT101:	.PLT101:
sethi (.-.PLT0), %g1	nop
ba,a %xcc, .PLT1	mov %o7, %g1
nop	call name101
nop	mov %g1, %o7
nop; nop	nop; nop
nop; nop	nop; nop
.PLT102:	.PLT102:
sethi (.-.PLT0), %g1	nop
ba,a %xcc, .PLT1	sethi %hi(name102), %g1
nop	jmpl %g1+%lo(name102), %g0
nop	nop
nop; nop	nop; nop
nop; nop	nop; nop
.PLT103:	.PLT103:
sethi (.-.PLT0), %g1	nop
ba,a %xcc, .PLT1	sethi %hh(name103), %g1
nop	sethi %lm(name103), %g5
nop	or %hm(name103), %g1
nop	sllx %g1, 32, %g1
nop	or %g1, %g5, %g5
nop	jmpl %g5+%lo(name103), %g0
nop	nop

TABLE 7–49 64-bit SPARC: Procedure Linkage Table Example *(Continued)*

```
              Object File                               Memory Segment

.PLT32768:                         .PLT32768:
    mov     %o7, %g5                   <unchanged>
    call    .+8                        <unchanged>
    nop                                <unchanged>
    ldx     [%o7+.PLTP32768 -          <unchanged>
              (.PLT32768+4)], %g1
    jmpl    %o7+%g1, %g1               <unchanged>
    mov     %g5, %o7                   <unchanged>

    ...                                ...

.PLT32927:                         .PLT32927:
    mov     %o7, %g5                   <unchanged>
    call    .+8                        <unchanged>
    nop                                <unchanged>
    ldx     [%o7+.PLTP32927 -          <unchanged>
              (.PLT32927+4)], %g1
    jmpl    %o7+%g1, %g1               <unchanged>
    mov     %g5, %o7                   <unchanged>

.PLTP32768                         .PLTP32768
    .xword  .PLT0 -                    .xword  name32768 -
              (.PLT32768+4)                      (.PLT32768+4)
    ...                                ...

.PLTP32927                         .PLTP32927
    .xword  .PLT0 -                    .xword  name32927 -
              (.PLT32927+4)                  (.PLT32927+4)
```

Following the steps below, the runtime linker and program jointly resolve the symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the runtime linker is not specified.

1. When first creating the memory image of the program, the runtime linker changes the initial procedure linkage table entries, making them transfer control to one of the runtime linker’s own routines. The runtime linker also stores an extended word of identification information in the third entry. When the runtime linker receives control, it can examine this extended word to find which object called it.
2. All other procedure linkage table entries initially transfer to the first or second entry. Those entries establish a stack frame and call the runtime linker.
3. With the identification value, the runtime linker gets its data structures for the object, including the relocation table.
4. The runtime linker computes the index of the relocation entry for the table slot.

5. With the index information, the runtime linker gets the symbol's real value, unwinds the stack, modifies the procedure linkage table entry, and transfers control to the desired destination.

The runtime linker does not have to create the instruction sequences under the memory segment column, it might. If it does, some points deserve more explanation.

- To make the code re-entrant, the procedure linkage table's instructions are changed in a particular sequence. If the runtime linker is fixing a function's procedure linkage table entry and a signal arrives, the signal handling code must be able to call the original function with predictable and correct results.
- The runtime linker can change up to eight words to convert an entry. The runtime linker can update only a single word atomically with regard to instruction execution. Therefore, re-entrancy is achieved by first overwriting the `nop` instructions with their replacement instructions, and then patching the `ba,a`, and the `sethi` if using a 64–bit store. If a re-entrant function call occurs just prior to the last patch, the runtime linker gains control a second time. Although both invocations of the runtime linker modify the same procedure linkage table entry, their changes do not interfere with each other.
- If the initial `sethi` instruction is changed, it can only be replaced by a `nop`.

Changing the pointer as done for the second form of entry is done using a single atomic 64–bit store.

Note – The different instruction sequences shown for `.PLT101`, `.PLT102`, and `.PLT103` demonstrate how the update can be optimized for the associated destination.

The `LD_BIND_NOW` environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes `R_SPARC_JMP_SLOT` relocation entries (procedure linkage table entries) before transferring control to the program.

x86: 32–bit Procedure Linkage Table

For 32–bit x86 dynamic objects, the procedure linkage table resides in shared text but uses addresses in the private global offset table. The runtime linker determines the absolute addresses of the destinations and modifies the global offset table's memory image accordingly. The runtime linker thus redirects the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables.

TABLE 7–50 x86: Absolute Procedure Linkage Table Example

```
.PLT0:
    pushl   got_plus_4
    jmp     *got_plus_8
    nop;    nop
    nop;    nop
.PLT1:
    jmp     *name1_in_GOT
    pushl   $offset
    jmp     .PLT0@PC
.PLT2:
    jmp     *name2_in_GOT
    pushl   $offset
    jmp     .PLT0@PC
```

TABLE 7–51 x86: Position-Independent Procedure Linkage Table Example

```
.PLT0:
    pushl   4(%ebx)
    jmp     *8(%ebx)
    nop;    nop
    nop;    nop
.PLT1:
    jmp     *name1@GOT(%ebx)
    pushl   $offset
    jmp     .PLT0@PC
.PLT2:
    jmp     *name2@GOT(%ebx)
    pushl   $offset
    jmp     .PLT0@PC
```

Note – As the preceding examples show, the procedure linkage table instructions use different operand addressing modes for absolute code and for position-independent code. Nonetheless, their interfaces to the runtime linker are the same.

Following the steps below, the runtime linker and program cooperate to resolve the symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the runtime linker sets the second and third entries in the global offset table to special values. The steps below explain these values.
2. If the procedure linkage table is position-independent, the address of the global offset table must be in `%ebx`. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. So, the calling function must set the global offset table base register before it calls the procedure linkage table entry.

3. For example, the program calls name1, which transfers control to the label .PLT1.
4. The first instruction jumps to the address in the global offset table entry for name1. Initially, the global offset table holds the address of the following pushl instruction, not the real address of name1.
5. The program pushes a relocation offset (offset) on the stack. The relocation offset is a 32–bit, nonnegative byte offset into the relocation table. The designated relocation entry has the type R_386_JMP_SLOT, and its offset specifies the global offset table entry used in the previous jmp instruction. The relocation entry also contains a symbol table index, which the runtime linker uses to get the referenced symbol, name1.
6. After pushing the relocation offset, the program jumps to .PLT0, the first entry in the procedure linkage table. The pushl instruction pushes the value of the second global offset table entry (got_plus_4 or 4(%ebx)) on the stack, giving the runtime linker one word of identifying information. The program then jumps to the address in the third global offset table entry (got_plus_8 or 8(%ebx)), to jump to the runtime linker.
7. The runtime linker unwinds the stack, checks the designated relocation entry, gets the symbol's value, stores the actual address of name1 in its global offset entry table, and jumps to the destination.
8. Subsequent executions of the procedure linkage table entry transfer directly to name1, without calling the runtime linker again. The jmp instruction at .PLT1 jumps to name1 instead of falling through to the pushl instruction.

The LD_BIND_NOW environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes R_386_JMP_SLOT relocation entries (procedure linkage table entries) before transferring control to the program.

CHAPTER 8

Thread-Local Storage

The compilation environment supports the declaration of thread-local data. This data is sometime referred to as thread-specific, or thread-private data, but more typically by the acronym TLS. By declaring variables to be thread-local, the compiler automatically arranges for these variables to be allocated on a per-thread basis.

The built-in support for this feature serves three purposes:

- It provides a foundation upon which the POSIX interfaces for allocating thread specific data are built.
- It offers a more convenient and more efficient mechanism for direct use by applications and libraries.
- It allows compilers to allocate TLS as necessary when performing loop-parallelizing optimizations.

C/C++ Programming Interface

Variables are declared thread-local using the `__thread` keyword, as in the following examples:

```
__thread int i;
__thread char *p;
__thread struct state s;
```

During loop optimizations, the compiler can choose to create thread-local temporaries as needed.

Applicability
: The `__thread` keyword can be applied to any global, file-scoped static, or function-scoped static variable. It has no effect on automatic variables, which are always thread-local.

Initialization

In C++, a thread-local variable can not be initialized if the initialization requires a static constructor. Otherwise, a thread-local variable can be initialized to any value that would be legal for an ordinary static variable.

No variable, thread-local or otherwise, can be statically initialized to the address of a thread-local variable.

Binding

Thread-local variables can be declared and referenced externally, and they are subject to the same interposition rules as normal symbols.

Dynamic loading restrictions

A shared library can be dynamically loaded during process startup, or after process startup via lazy loading, filters, or `dlopen`(3C). A shared library containing a reference to a thread-local variable, can be loaded post-startup if every translation unit containing the reference is compiled with a dynamic TLS model.

Static TLS models generates faster code. However, code compiled to use this model cannot reference thread-local variables in post-startup dynamically loaded libraries. A dynamic TLS model is able to reference all TLS. These models are described in "Thread-Local Storage Access Models" on page 281.

Address-of operator

The address-of operator, `&`, can be applied to a thread-local variable. This operator is evaluated at runtime, and returns the address of the variable within the current thread. The address obtained by this operator can be used freely by any thread in the process as long as the thread that evaluated the address remains in existence. When a thread terminates, any pointers to thread-local variables in that thread become invalid.

When `dlsym`(3C) is used to obtain the address of a thread-local variable, the address returned is the address of the instance of that variable in the thread that called `dlsym()`.

Thread-Local Storage Section

Separate copies of thread-local data that have been allocated at compile-time, must be associated with individual threads of execution. To provide this data, TLS sections are used to specify the size and initial contents.

The compilation environment allocates TLS in sections that are identified with the `SHF_TLS` flag. These sections provide initialized TLS and uninitialized TLS based on how the storage is declared:

- An initialized thread-local variable is allocated in a `.tdata`, or `.tdata1` section. This initialization can require relocation.

- An uninitialized thread-local variable is defined as a COMMON symbol. The resulting allocation is made in a .tbss section.

The uninitialized section is allocated immediately following any initialized sections, subject to padding for proper alignment. Together, the combined sections form a TLS template that is used to allocate TLS whenever a new thread is created.

The initialized portion of this template is called the TLS initialization image. All relocations that are generated as a result of initialized thread-local variables are applied to this template. These relocated values, are then used when a new thread requires the initial values.

TLS symbols have the symbol type STT_TLS. These symbols are assigned offsets relative to the beginning of the TLS template. The actual virtual address that is associated with these symbols is irrelevant. The address refers only to the template, and not to the per-thread copy of each data item.

In dynamic executables and shared objects, the st_value field of a STT_TLS symbol contains the assigned offset for defined symbols, or zero for undefined symbols.

Several relocations are defined to support access to TLS. See "SPARC: Thread-Local Storage Relocation Types" on page 287 and "x86: Thread-Local Storage Relocation Types" on page 294. TLS relocations only reference symbols of type STT_TLS.

In dynamic executables and shared objects, a PT_TLS program entry describes a TLS template. This template has the following members:

TABLE 8–1 ELF PT_TLS Program Header Entry

Member	Value
p_offset	File offset of the TLS initialization image
p_vaddr	Virtual memory address of the TLS initialization image
p_paddr	Reserved
p_filesz	Size of the TLS initialization image
p_memsz	Total size of the TLS template
p_flags	PF_R
p_align	Alignment of the TLS template

Runtime Allocation of Thread-Local Storage

TLS is created at three occasions during the lifetime of a program:

- At program startup.
- When a new thread is created.
- When a thread references a TLS block for the first time after a shared library is loaded following program startup.

Thread-local data storage is layed out at runtime as illustrated in Figure 8–1.

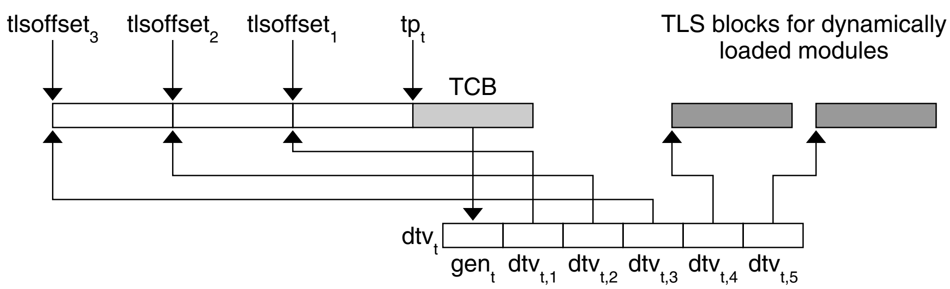


FIGURE 8–1 Runtime Storage Layout of Thread-Local Storage

Program Startup

At program startup, the runtime system creates TLS for the main thread.

First, the runtime linker logically combines the TLS templates for all loaded dynamic objects, including the dynamic executable, into a single static template. Each dynamic objects's TLS template is assigned an offset within the combined template, $\texttt{tlsoffset}_m$, as follows:

- $\texttt{tlsoffset}_1$ = round($\texttt{tlssize}_1$, $\texttt{align}_1$)
- $\texttt{tlsoffset}_{m+1}$ = round($\texttt{tlsoffset}_m$ + $\texttt{tlssize}_{m+1}$, $\texttt{align}_{m+1}$)

$\texttt{tlssize}_{m+1}$ and $\texttt{align}_{m+1}$ are the size and alignment, respectively, for the allocation template for dynamic object `m` (1 <= `m` <= `M`, where `M` is the total number of loaded dynamic objects). The `round(offset, align)` function returns an offset rounded up to the next multiple of *align*. The TLS template is placed immediately preceding the thread pointer $\texttt{tp}_t$. Accesses to the TLS data are based off of subtractions from $\texttt{tp}_t$.

Next, the runtime linker computes the total startup TLS allocation size, $tlssize_S$, which is equal to $tlsoffset_M$.

The runtime linker then constructs a linked list of initialization records. Each record in this list describes the TLS initialization image for one loaded dynamic object, and contains the following fields:

- A pointer to the TLS initialization image.
- The size of the TLS initialization image.
- The $tlsoffset_m$ of the object.
- A flag indicating whether the object uses a static TLS model.

The thread library uses this information to allocate storage for the initial thread. This storage is initialized, and a dynamic TLS vector for the initial thread is created.

Thread Creation

For the initial thread, and for each new thread created, the thread library allocates a new TLS block for each loaded dynamic object. Blocks can be allocated separately, or as a single contiguous block.

Each thread $_t$ has an associated thread pointer tp_t, which points to the thread control block, `TCB`. The thread pointer, `tp`, always contains the value of tp_t for the current running thread.

The thread library then creates a vector of pointers, dtv_t, for the current thread `t`. The first element of each vector contains a generation number gen_t, which is used to determine when the vector needs to be extended. See "Deferred Allocation of Thread-Local Storage Blocks" on page 280.

Each remaining element in the vector, $dtv_{t,m}$, is a pointer to the block reserved for the TLS belonging to dynamic object `m`.

For dynamically loaded, post-startup objects, the thread library defers the allocation of TLS blocks. This allocation occurs when the first reference is made to a TLS variable within the loaded object. All references to TLS defined in a post-startup, dynamically loaded object, must use a dynamic TLS model. For blocks whose allocation has been deferred, the pointer $dtv_{t,m}$ is set to an implementation-defined special value.

Note – The runtime linker can group TLS templates for all startup objects such that they share a single element in the vector, $dtv_{t,1}$. This does not affect the offset calculations described above or the creation of the list of initialization records. For the following sections, however, the value of `M`, the total number of objects, start with the value of 1.

The thread library then copies the initialization images to the corresponding locations within the new block of storage.

Post-Startup Dynamic Loading

When a shared library containing TLS is loaded following process startup, the runtime linker extends the list of initialization records to include the initialization template of new library. The new object is given an index of `m = M + 1`, and the counter `M` is incremented by one. However, the allocation of new TLS blocks is deferred until they are actually referenced.

When a library containing TLS is unloaded, the TLS blocks used by that library are freed.

Deferred Allocation of Thread-Local Storage Blocks

In a dynamic TLS model, when a thread `t` needs to access a TLS block for object `m`, the code updates the dtv_t and performs the initial allocation of the TLS block. The thread library provides the following interface to provide for dynamic TLS allocation:

```
typedef struct {
    unsigned long ti_moduleid;
    unsigned long ti_tlsoffset;
} TLS_index;

extern void * __tls_get_addr(TLS_index * ti);     (SPARC)
extern void * ___tls_get_addr(TLS_index * ti);    (x86)
```

Note – The SPARC and x86 definitions of this function have the same function signature. However, the x86 version does not use the default x86 calling convention of passing arguments on the stack. Instead, the x86 version passes its argument via the `%eax` register which is more efficient. To denote that this alternate calling method is used, the x86 function name has three leading underscores in its name.

Both versions of `tls_get_addr()` check the per-thread generation counter, gen_t, to determine whether the vector needs to be updated. If the vector dtv_t is out of date, the routine updates the vector, possibly reallocating it to make room for more entries. The routine then checks to see if the TLS block corresponding to $dtv_{t,m}$ has been allocated. If it has not been allocated, the routine allocates and initializes the block, using the information in the list of initialization records provided by the runtime linker. The pointer $dtv_{t,m}$ is set to point to the allocated block. The routine returns a pointer to the given offset within the block.

Thread-Local Storage Access Models

Each TLS reference follows one of the following access models. These models are listed from the most general, but least optimized, to the fastest, but most restrictive.

General Dynamic (GD) - dynamic TLS
: This model allows reference of all TLS variables, from either a shared object or a dynamic executable. This model also supports the deferred allocation of a TLS block when it is first referenced from a specific thread.

Local Dynamic (LD) - dynamic TLS of local symbols
: This model is a optimization of the *GD* model. If the compiler determines that a variable is bound locally, or protected, within the dynamic object being built, it instructs the link-editor to statically bind the dynamic `tlsoffset` and use this model. This provides a performance benefit over the *GD* model. Only one call to `tls_get_addr()` is required per function, to determine the address of $\texttt{dtv}_{0,m}$. The dynamic TLS offset, bound at link-edit time, is added to the $\texttt{dtv}_{0,m}$ address for each reference.

Initial Executable (IE) - static TLS with assigned offsets
: This model can only reference TLS variables which are available as part of the initial static TLS template. This template is composed of all TLS blocks available at process startup. In this model, the thread pointer-relative offset for a given variable *x* is stored in the global offset table entry for *x*. This model can not reference TLS variables from shared libraries loaded after initial process startup, such as via lazy loading, filters, or `dlopen`(3C). This model can not access TLS blocks which use deferred allocation.

Local Executable (LE)- static TLS
: This model can only reference TLS variables which are part of the TLS block of the dynamic executable itself. The link-editor calculates the thread pointer-relative offsets statically, without the need for dynamic relocations, or the extra reference to the global offset table. This model can not be used to reference variables outside of the dynamic executable.

The link-editor can transition code from the more general access models to the more optimized models, if it is determined appropriate to do so. This transitioning is achievable through the use of unique TLS relocations. These relocations, not only request updates be performed, but identify which TLS access model is being used.

Knowing the TLS access model, and the type of object being created, allows the link-editor to perform translations. For example, if a relocatable object using the *GD* access model is being linked into a dynamic executable, the link-editor can transition the references using the *IE* or *LE* access models, as appropriate. The relocations required for the model are then performed.

The following diagram illustrates the different access models, and when one model can be transitioned from one to the other.

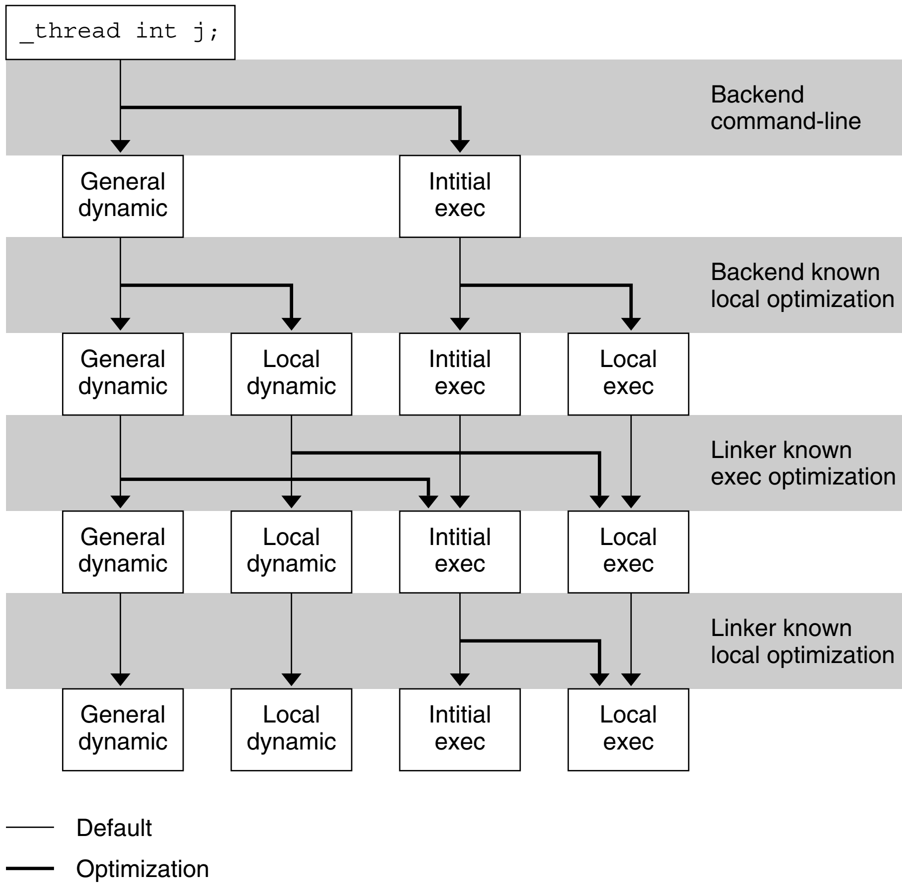


FIGURE 8–2 Thread-Local Storage Access Models and Transitions

SPARC: Thread-Local Variable Access

On SPARC, the following code sequence models are available for accessing thread-local variables.

SPARC: 32-bit and 64-bit General Dynamic (GD)

This code sequence is the most general, and can be included in both shared objects and dynamic executables. This code sequence can also reference an external TLS variable in either a shared object or dynamic executable.

TABLE 8–2 SPARC: 32-bit and 64-bit General Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`# %l7 - initialized to GOT pointer`		
`0x00 sethi %hi(@dtlndx(x)), %o0`	`R_SPARC_TLS_GD_HI22`	`x`
`0x04 add   %o0,%lo(@dtlndx(x)),%o0`	`R_SPARC_TLS_GD_LO10`	`x`
`0x08 add   %l7, %o0, %o0`	`R_SPARC_TLS_GD_ADD`	`x`
`0x0c call  x@TLSPLT`	`R_SPARC_TLS_GD_CALL`	`x`
`# %o0 - contains address of TLS variable`		

	Outstanding Relocations: 32-bit	Symbol
`GOT[n]`	`R_SPARC_TLS_DTPMOD32`	`x`
`GOT[n + 1]`	`R_SPARC_TLS_DTPOFF32`	`x`

	Outstanding Relocations: 64-bit	Symbol
`GOT[n]`	`R_SPARC_TLS_DTPMOD64`	`x`
`GOT[n + 1]`	`R_SPARC_TLS_DTPOFF64`	`x`

The `sethi`, and `add` instructions generate `R_SPARC_TLS_GD_HI22` and `R_SPARC_TLS_GD_LO10` relocations respectively. These relocations instruct the link-editor to allocate space in the global offset table to hold a `TLS_index` structure for variable `x`. The link-editor processes this relocation by substituting the `GOT`-relative offset for the new `GOT` entry.

Since the load object index and TLS block index for `x` are not known until runtime, the link-editor places the `R_SPARC_TLS_DTPMOD32` and `R_SPARC_TLS_DPTOFF32` relocations against the `GOT` for processing by the runtime linker.

The second `add` instruction causes the generation of the `R_SPARC_TLS_GD_ADD` relocation. This relocation is used only if the GD code sequence is changed to another sequence by the link-editor.

The `call` instruction generates the `R_SPARC_TLS_GD_CALL` relocation. This relocation instructs the link-editor to bind the call to the `__tls_get_addr()` function, and associates the `call` instruction with the GD code sequence.

Note – The add instruction must appear before the call instruction. It cannot be placed into the delay slot for the call. This is required as the code-transformations that can occur later require a known order.

The register used as the GOT-pointer for the add instruction tagged by the R_SPARC_TLS_GD_ADD relocation, must be the first register in the add instruction. This permits the link-editor to identify the GOT-pointer register during a code transformation.

SPARC: 32-bit and 64-bit Local Dynamic (LD)

This code sequence can be used in either a shared object or dynamic executable. This sequence is used when referencing a TLS variable bound within the same object as the reference. Because the dynamic tlsoffset can be bound at link-edit time, only one call to __tls_get_addr() is required per function call for all symbols referenced via the *LD* code sequence.

TABLE 8–3 SPARC: 32-bit and 64-bit Local Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
# %l7 - initialized to GOT pointer		
0x00 sethi %hi(@tmndx(x1)), %o0	R_SPARC_TLS_LDM_HI22	x1
0x04 add %o0,%lo(@tmndx(x1)),%o0	R_SPARC_TLS_LDM_LO10	x1
0x08 add %l7, %o0, %o0	R_SPARC_TLS_LDM_ADD	x1
0x0c call x@TLSPLT	R_SPARC_TLS_LDM_CALL	x1
# %o0 - contains address of TLS block of current object		
0x10 sethi %hi(@dtpoff(x1)), %l1	R_SPARC_TLS_LDO_HIX22	x1
0x14 xor %l1, %lo(@dtpoff(x1)), %l1	R_SPARC_TLS_LDO_LOX10	x1
0x18 add %o0, %l1, %l1	R_SPARC_TLS_LDO_ADD	x1
# %l1 - contains address of local TLS variable x1		
0x20 sethi %hi(@dtpoff(x2)), %l2	R_SPARC_TLS_LDO_HIX22	x2
0x24 xor %l2, %lo(@dtpoff(x2)), %l2	R_SPARC_TLS_LDO_LOX10	x2
0x28 add %o0, %l2, %l2	R_SPARC_TLS_LDO_ADD	x2
# %l2 - contains address of local TLS variable x2		

	Outstanding Relocations: 32-bit	Symbol
GOT[n]	R_SPARC_TLS_DTPMOD32	x1
GOT[n + 1]	<none>	

	Outstanding Relocations: 64-bit	Symbol
GOT[n]	R_SPARC_TLS_DTPMOD64	x1
GOT[n + 1]	<none>	

The first sethi and add instructions generate R_SPARC_TLS_LDM_HI22 and R_SPARC_TLS_LDM_LO10 relocations respectively. These relocations instruct the link-editor to allocate space in the global offset table to hold a TLS_index structure for the current object. The link-editor processes this relocation by substituting the GOT-relative offset for the new GOT entry.

Since the load object index is not known until runtime, a R_SPARC_TLS_DTPMOD32 relocation is created, and the ti_tlsoffset field of the TLS_index structure is zero filled.

The second add and the call instruction are tagged with the R_SPARC_TLS_LDM_ADD and R_SPARC_TLS_LDM_CALL relocations respectively.

The following sethi and xor instructions generate the R_SPARC_LDO_HIX22 and R_SPARC_TLS_LDO_LOX10 relocations, respectively. The TLS offset for each local symbol is known at link-edit time, therefore these values are filled in directly. The add instruction is tagged with the R_SPARC_TLS_LDO_ADD relocation.

When a procedure references more than one local symbol, the compiler generates code to obtain the base address of the TLS block once. This base address is then used to calculate the address of each symbol without a separate library call.

Note – The register containing the TLS object address in the add instruction tagged by the R_SPARC_TLS_LDO_ADD must be the first register in the instruction sequence. This permits the link-editor to identify the register during a code transformation.

SPARC: 32-bit Initial Executable (IE)

This code sequence can only be used in a dynamic executable. It can reference a TLS variable defined in either the executable or any shared libraries loaded at process startup. This model can not reference TLS variables from shared libraries loaded after process startup.

TABLE 8–4 SPARC: 32-bit Initial Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`# %l7 - initialized to GOT pointer, %g7 - thread pointer`		
`0x00 sethi %hi(@tpoff(x)), %o0`	`R_SPARC_TLS_IE_HI22`	`x`
`0x04 or    %o0,%lo(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_LO10`	`x`
`0x08 ld    [%l7 + %o0], %o0`	`R_SPARC_TLS_IE_LD`	`x`
`0x0c add   %g7, %o0, %o0`	`R_SPARC_TLS_IE_ADD`	`x`
`# %o0 - contains address of TLS variable`		

	Outstanding Relocations	Symbol
`GOT[n]`	`R_SPARC_TLS_TPOFF32`	`x`

The `sethi` and `or` instructions generate `R_SPARC_TLS_IE_HI22` and `R_SPARC_TLS_IE_LO10` relocations, respectively. These relocations instruct the link-editor to create space in the global offset table to store the static TLS offset for symbol `x`. A `R_SPARC_TLS_TPOFF32` relocation is left outstanding against the `GOT` for the runtime linker to fill in with the negative static TLS offset for symbol `x`. The `ld` and the `add` instructions are tagged with the `R_SPARC_TLS_IE_LD` and `R_SPARC_TLS_IE_ADD` relocations respectively.

Note – The register used as the `GOT`-pointer for the `add` instruction tagged by the `R_SPARC_TLS_IE_ADD` relocation must be the first register in the instruction. This permits the link-editor to identify the `GOT`-pointer register during a code transformation.

SPARC: 64-bit Initial Executable (IE)

This sequence is identical to the SPARC 32–bit sequence, except that an `ldx` instruction is used to load the 64–bit address instead of an `ld` instruction.

TABLE 8–5 SPARC: 64-bit Initial Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
# %l7 - initialized to GOT pointer, %g7 - thread pointer		
0x00 sethi %hi(@tpoff(x)), %o0	R_SPARC_TLS_IE_HI22	x
0x04 or %o0,%lo(@tpoff(x)),%o0	R_SPARC_TLS_IE_LO10	x
0x08 ldx [%l7 + %o0], %o0	R_SPARC_TLS_IE_LD	x
0x0c add %g7, %o0, %o0	R_SPARC_TLS_IE_ADD	x
# %o0 - contains address of TLS variable		

	Outstanding Relocations	Symbol
GOT[n]	R_SPARC_TLS_TPOFF64	x

SPARC: 32-bit and 64-bit Local Executable (LE)

This code sequence can only be used from within a dynamic executable to reference a TLS variable defined within the executable. If this is the case, the static `tlsoffset` is known at link-edit time and no runtime relocations are required.

TABLE 8–6 SPARC: 32-bit and 64-bit Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
# %g7 - thread pointer		
0x00 sethi %hix(@tpoff(x)), %o0	R_SPARC_TLS_LE_HIX22	x
0x04 xor %o0,%lo(@tpoff(x)),%o0	R_SPARC_TLS_LE_LOX10	x
0x08 add %g7, %o0, %o0	<none>	
# %o0 - contains address of TLS variable		

The `sethi` and `xor` instructions generate `R_SPARC_TLS_LE_HIX22` and `R_SPARC_TLS_LE_LOX10` relocations respectively. The link-editor binds these relocations directly to the static TLS offset for the symbol defined in the executable. No relocation processing is required at runtime.

SPARC: Thread-Local Storage Relocation Types

The TLS relocations listed in the following table are defined for SPARC. Descriptions in the table use the following notation:

@dtlndx(x)
: Allocates two contiguous entries in the global offset table to hold a TLS_index structure. This information is passed to __tls_get_addr(). The instruction referencing this entry is bound to the address of the first of the two GOT entries.

@tmndx(x)
: Allocates two contiguous entries in the global offset table to hold a TLS_index structure. This information is passed to __tls_get_addr(). The ti_tlsoffset field of this structure is set to 0, and the ti_moduleid is filled in at runtime. The call to __tls_get_addr() returns the starting offset of the dynamic TLS block.

@dtpoff(x)
: Calculates the tlsoffset relative to the TLS block.

@tpoff(x)
: Calculates the negative tlsoffset relative to the static TLS block. This value is added to the thread-pointer to calculate the TLS address.

@dtpmod(x)
: Calculates the object identifier of the object containing symbol S.

TABLE 8–7 SPARC: Thread-Local Storage Relocation Types

Name	Value	Field	Calculation
R_SPARC_TLS_GD_HI22	56	T-simm22	@dtlndx(S + A) >> 10
R_SPARC_TLS_GD_LO10	57	T-simm13	@dtlndx(S + A) & 0x3ff
R_SPARC_TLS_GD_ADD	58	None	See R_SPARC_TLS_GD_ADD
R_SPARC_TLS_GD_CALL	59	V-disp30	See R_SPARC_TLS_GD_CALL
R_SPARC_TLS_LDM_HI22	60	T-simm22	@tmndx(S + A) >> 10
R_SPARC_TLS_LDM_LO10	61	T-simm13	@tmndx(S + A) & 0x3ff
R_SPARC_TLS_LDM_ADD	62	None	See R_SPARC_TLS_LDM_ADD
R_SPARC_TLS_LDM_CALL	63	V-disp30	See R_SPARC_TLS_LDM_CALL
R_SPARC_TLS_LDO_HIX22	64	T-simm22	@dtpoff(S + A) >> 10
R_SPARC_TLS_LDO_LOX10	65	T-simm13	@dtpoff(S + A) & 0x3ff
R_SPARC_TLS_LDO_ADD	66	None	See R_SPARC_TLS_LDO_ADD
R_SPARC_TLS_IE_HI22	67	T-simm22	@got(@tpoff(S + A)) >> 10
R_SPARC_TLS_IE_LO10	68	T-simm13	@got(@tpoff(S + A)) & 0x3ff
R_SPARC_TLS_IE_LD	69	None	See R_SPARC_TLS_IE_LD
R_SPARC_TLS_IE_LDX	70	None	See R_SPARC_TLS_IE_LDX
R_SPARC_TLS_IE_ADD	71	None	See R_SPARC_TLS_IE_ADD

TABLE 8–7 SPARC: Thread-Local Storage Relocation Types *(Continued)*

Name	Value	Field	Calculation
R_SPARC_TLS_LE_HIX22	72	T-imm22	(@tpoff(S + A) ^0xffffffffffffffff) >> 10
R_SPARC_TLS_LE_LOX10	73	T-simm13	(@tpoff(S + A) & 0x3ff) | 0x1c00
R_SPARC_TLS_DTPMOD32	74	V-word32	@dtpmod(S + A)
R_SPARC_TLS_DTPMOD64	75	V-word64	@dtpmod(S + A)
R_SPARC_TLS_DTPOFF32	76	V-word32	@dtpoff(S + A)
R_SPARC_TLS_DTPOFF64	77	V-word64	@dtpoff(S + A)
R_SPARC_TLS_TPOFF32	78	V-word32	@tpoff(S + A)
R_SPARC_TLS_TPOFF64	79	V-word64	@tpoff(S + A)

Some relocation types have semantics beyond simple calculations:

`R_SPARC_TLS_GD_ADD`
: This relocation tags the `add` instruction of a `GD` code sequence. The register used for the `GOT`-pointer is the first register in the sequence. The instruction tagged by this relocation comes before the `call` instruction tagged by the `R_SPARC_TLS_GD_CALL` relocation. This is used to transition between TLS models at link-edit time.

`R_SPARC_TLS_GD_CALL`
: This relocation is handled as if it were a `R_SPARC_WPLT30` relocation referencing the `__tls_get_addr()` function. This relocation is part of a `GD` code sequence.

`R_SPARC_LDM_ADD`
: This relocation tags the first `add` instruction of a `LD` code sequence. The register used for the `GOT`-pointer is the first register in the sequence. The instruction tagged by this relocation comes before the `call` instruction tagged by the `R_SPARC_TLS_GD_CALL` relocation. This is used to transition between TLS models at link-edit time.

`R_SPARC_LDM_CALL`
: This relocation is handled as if it were a `R_SPARC_WPLT30` relocation referencing the `__tls_get_addr()` function. This relocation is part of a `LD` code sequence.

`R_SPARC_LDO_ADD`
: This relocation tags the final `add` instruction in a `LD` code sequence. The register which contains the object address computed in the initial part of the code sequence is the first register in this instruction. This permits the link-editor to identify this register for code transformations.

`R_SPARC_TLS_IE_LD`
: This relocation tags the `ld` instruction in the 32–bit `IE` code sequence. This is used to transition between TLS models at link-edit time.

R_SPARC_TLS_IE_LDX
: This relocation tags the ldx instruction in the 64–bit IE code sequence. This is used to transition between TLS models at link-edit time.

R_SPARC_TLS_IE_ADD
: This relocation tags the add instruction in the IE code sequence. The register that is used for the GOT-pointer is the first register in the sequence.

x86: Thread-Local Variable Access

On x86, the following code sequence models are available for accessing TLS

x86: General Dynamic (GD)

This code sequence is the most general, and can be included both in a shared objects and dynamic executables. This code sequence can reference an external TLS variable in either a shared object or dynamic executable.

TABLE 8–8 x86: General Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 leal x@tlsgd(,%ebx,1), %eax	R_386_TLS_GD	x
0x07 call x@tlsgdplt	R_386_TLS_GD_PLT	x
# %eax - contains address of TLS variable		

	Outstanding Relocations	Symbol
GOT[n]	R_386_TLS_DTPMOD32	x
GOT[n + 1]	R_386_TLS_DTPOFF32	

The leal instruction generates a R_386_TLS_GD relocation which instructs the link-editor to allocate space in the global offset table to hold a TLS_index structure for variable x. The link-editor processes this relocation by substituting the GOT-relative offset for the new GOT entry.

Since the load object index and TLS block index for x are not known until runtime, the link-editor places the R_386_TLS_DTPMOD32 and R_386_TLS_DTPOFF32 relocations against the GOT for processing by the runtime linker. The address of the generated GOT entry is loaded into register %eax for the call to ___tls_get_addr().

The call instruction causes the generation of the R_386_TLS_GD_PLT relocation. This instructs the link-editor to bind the call to the ___tls_get_addr() function and associates the call instruction with the GD code sequence.

The call instruction must immediately follow the leal instruction. This is a required to permit the code transformations.

x86: Local Dynamic (LD)

This code sequence can be used in either a shared object or dynamic executable. This sequence is used when referencing a TLS variable bound to the same object as the reference. Because the dynamic tlsoffset can be bound at link-edit time, only one call to ___tls_get_addr() is required per function call for all symbols which are referenced via the LD code sequence.

TABLE 8–9 x86: Local Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 leal x1@tlsldm(%ebx), %eax	R_386_TLS_LDM	x1
0x06 call x1@tlsldmplt	R_386_TLS_LDM_PLT	x1
# %eax - contains address of TLS block of current object		
0x10 leal x1@dtpoff(%eax), %edx	R_386_TLS_LDO_32	x1
# %edx - contains address of local TLS variable x1		
0x20 leal x2@dtpoff(%eax), %edx	R_386_TLS_LDO_32	x2
# %edx - contains address of local TLS variable x2		

	Outstanding Relocations	Symbol
GOT[n]	R_386_TLS_DTPMOD32	x
GOT[n + 1]	<none>	

The first leal instruction generates a R_386_TLS_LDM relocation that instructs the link-editor to allocate space in the global offset table to hold a TLS_index structure for the current object. The link-editor process this relocation by substituting the GOT-relative offset for the new linkage table entry.

Since the load object index is not known until runtime, aR_386_TLS_DTPMOD32 relocation is created, and the ti_tlsoffset field of the structure is zero filled. The call instruction is tagged with the R_386_TLS_LDM_PLT relocation.

The TLS offset for each local symbol is known at link-edit time so the link-editor fills these values in directly.

When a procedure references more than one local symbol, the compiler generates code to obtain the base address of the TLS block once. This base address is then used to calculate the address of each symbol without a separate library call.

x86: Initial Executable (IE)

There are two code-sequences for the IE model. One sequence is for position independent code which uses a GOT-pointer. The other sequence is for position dependent code which does not use a GOT-pointer. Both of these code sequences can only be used in a dynamic executable. These code sequences can reference a TLS variable defined in either the executable or any of the shared libraries loaded at process startup. This model can not reference TLS variables from shared libraries loaded after process startup.

TABLE 8–10 x86: Initial Executable, Position Independent, Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl  %gs:0, %eax`	`<none>`	
`0x06 addl  x@gotntpoff(%ebx), %eax`	`R_386_TLS_GOTIE`	`x`
`# %eax - contains address of TLS variable`		

	Outstanding Relocations	Symbol
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

The addl instruction generates a R_386_TLS_GOTIE relocation that instructs the link–editor to create space in the global offset table to store the static TLS offset for symbol x. A R_386_TLS_TPOFF relocation is left outstanding against the GOT table for the runtime linker to fill in with the static TLS offset for symbol x.

TABLE 8–11 x86: Initial Executable, Position Dependent, Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl  %gs:0, %eax`	`<none>`	
`0x06 addl  x@indntpoff, %eax`	`R_386_TLS_IE`	`x`
`# %eax - contains address of TLS variable`		

	Outstanding Relocations	Symbol
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

The addl instruction generates a R_386_TLS_IE relocation, that instructs the link-editor to create space in the global offset table to store the static TLS offset for symbol x. The main difference between this sequence and the position independent form, is that the instruction is bound directly to the GOT entry created, instead of via an offset off of the GOT-pointer register. A R_386_TLS_TPOFF relocation is left outstanding against the GOT for the runtime linker to fill in with the static TLS offset for symbol x.

The contents of variable x, rather then the address, can be loaded by embedding the offset directly into the memory reference as shown in the next two sequences.

TABLE 8–12 x86: Initial Executable, Position Independent, Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl  x@gotntpoff(%ebx), %eax`	`R_386_TLS_GOTIE`	`x`
`0x06 movl  %gs:(%eax), %eax`	`<none>`	
`# %eax - contains address of TLS variable`		

	Outstanding Relocations	Symbol
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

TABLE 8–13 x86: Initial Executable, Position Independent, Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl  x@indntpoff, %ecx`	`R_386_TLS_IE`	`x`
`0x06 movl  %gs:(%ecx), %eax`	`<none>`	
`# %eax - contains address of TLS variable`		

	Outstanding Relocations	Symbol
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

In the last sequence, if the %eax register is used instead of the %ecx above, the first instruction can be either 5 or 6 bytes long.

x86: Local Executable (LE)

This code sequence can only be used from within a dynamic executable and referencing a TLS variable defined within the executable. If this is the case the static tlsoffset it known at link-edit time and no runtime relocations are required.

TABLE 8–14 x86: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl %gs:0, %eax`	`<none>`	
`0x06 leal x@ntpoff(%eax), %eax`	`R_386_TLS_LE`	`x`
`# %eax - contains address of TLS variable`		

The `movl` instruction generates a`R_386_TLS_LE_32` relocation. The link-editor binds this relocation directly to the static TLS offset for the symbol defined in the executable. No processing is required at runtime.

The contents of variable `x`, rather then the address, can be accessed with the same relocation by using the following instruction sequence.

TABLE 8–15 x86: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl %gs:0, %eax`	`<none>`	
`0x06 movl x@ntpoff(%eax), %eax`	`R_386_TLS_LE`	`x`
`# %eax - contains address of TLS variable`		

If instead of computing the address of the variable we want to load from it or store in it the following sequence can be used. Note that in this case we use the `x@ntpoff` expression not as an immediate value, but instead as an absolute address.

TABLE 8–16 x86: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`0x00 movl %gs:x@ntpoff, %eax`	`R_386_TLS_LE`	`x`
`# %eax - contains address of TLS variable`		

x86: Thread-Local Storage Relocation Types

The TLS relocations listed in the following table are defined for x86. Descriptions in the table use the following notation:

`@tlsgd(x)`
: Allocates two contiguous entries in the `GOT` to hold a `TLS_index` structure. This structure is passed to `___tls_get_addr()`. The instruction referencing this entry will be bound to the first of the two `GOT` entries.

@tlsgdplt(x)
: This relocation is handled as if it were a R_386_PLT32 relocation referencing the ___tls_get_addr() function.

@tlsldm(x)
: Allocates two contiguous entries in the GOT to hold a TLS_index structure. This structure is passed to the ___tls_get_addr(). Theti_tlsoffset field of the TLS_index is set to 0, and the ti_moduleid is filled in at runtime. The call to ___tls_get_addr() returns the starting offset of the dynamic TLS block.

@gotntpoff(x)
: Allocates a entry in the GOT, and initializes it with the negative tlsoffset relative to the static TLS block. This is performed at runtime via the R_386_TLS_TPOFF relocation.

@indntpoff(x)
: This expression is similar to @gotntpoff, but used in position dependent code. @gotntpoff resolves to a GOT slot address relative to the start of the GOT in the movl or addl instructions. @indntpoff resolves to the absolute GOT slot address.

@ntpoff(x)
: Calculates the negative tlsoffset relative to the static TLS block.

@dtpoff(x)
: Calculates the tlsoffset relative to the TLS block. The value is used as an immediate value of an addend and is not associated with a specific register.

@dtpmod(x)
: Calculates the object identifier of the object containing symbol S.

TABLE 8–17 x86: Thread-Local Storage Relocation Types

Name	Value	Field	Calculation
R_386_TLS_GD_PLT	12	Word32	@tlsgdplt
R_386_TLS_LDM_PLT	13	Word32	@tlsldmplt
R_386_TLS_TPOFF	14	Word32	@ntpoff(S)
R_386_TLS_IE	15	Word32	@indntpoff(S)
R_386_TLS_GOTIE	16	Word32	@gotntpoff(S)
R_386_TLS_LE	17	Word32	@ntpoff(S)
R_386_TLS_GD	18	Word32	@tlsgd(S)
R_386_TLS_LDM	19	Word32	@tlsldm(S)
R_386_TLS_LDO_32	32	Word32	@dtpoff(S)
R_386_TLS_DTPMOD32	35	Word32	@dtpmod(S)
R_386_TLS_DTPOFF32	36	Word32	@dtpoff(S)

CHAPTER 9

Mapfile Option

The link-editor automatically and intelligently maps input sections from relocatable objects to segments in the output file being created. The `-M` option with an associated `mapfile` enables you to change the default mapping provided by the link-editor. In addition, new segments can be created, attributes modified, and symbol versioning information can be supplied with the `mapfile`.

Note – When using a `mapfile` option, you can easily create an output file that does not execute. The link-editor knows how to produce a correct output file without the use of the `mapfile` option.

Sample `mapfiles` provided on the system reside in the `/usr/lib/ld` directory.

Mapfile Structure and Syntax

You can enter four basic types of directives into a `mapfile`:

- Segment declarations.
- Mapping directives.
- Section-to-segment ordering.
- Size-symbol declarations.
- File control directives.

Each directive can span more than one line and can have any amount of white space, including new lines, as long as that white space is followed by a semicolon.

Typically, segment declarations are followed by mapping directives. You declare a segment and then define the criteria by which a section becomes part of that segment. If you enter a mapping directive or size-symbol declaration without first declaring the segment to which you are mapping, except for built-in segments, the segment is given default attributes. Such segment is an *implicitly* declared segment.

Size-symbol declarations and file control directives can appear anywhere in a `mapfile`.

The following sections describe each directive type. For all syntax discussions, the following notations apply:

- All entries in constant width, all colons, semicolons, equal signs, and at (@) signs are typed in literally.
- All entries in *italics* are substitutable.
- { ... }* means "zero or more."
- { ... }+ means "one or more."
- [...] means "optional."
- `section_names` and `segment_names` follow the same rules as C identifiers, where a period (.) is treated as a letter. For example, `.bss` is a legal name.
- `section_names`, `segment_names`, `file_names`, and `symbol_names` are case sensitive. Everything else is not case sensitive.
- Spaces, or new-lines, can appear anywhere except before a number or in the middle of a name or value.
- Comments beginning with # and ending at a newline can appear anywhere that a space can appear.

Segment Declarations

A segment declaration creates a new segment in the output file, or changes the attribute values of an existing segment. An existing segment is one that you previously defined or one of the four built-in segments described immediately following.

A segment declaration has the following syntax:

```
segment_name = {segment_attribute_value}*;
```

For each `segment_name`, you can specify any number of `segment_attribute_values` in any order, each separated by a space. Only one attribute value is allowed for each segment attribute. The segment attributes and their valid values are as shown in the following table.

TABLE 9–1 Mapfile Segment Attributes

Attribute	Value
`segment_type`	`LOAD | NOTE | STACK`
`segment_flags`	`? [E] [N] [O] [R] [W] [X]`
`virtual_address`	`V`*number*
`physical_address`	`P`*number*
`length`	`L`*number*
`rounding`	`R`*number*
`alignment`	`A`*number*

There are four built-in segments with the following default attribute values:

- `text` – `LOAD`, `?RX`, no `virtual_address`, `physical_address`, or `length` specified, `alignment` values set to defaults per CPU type.
- `data` – `LOAD`, `?RWX`, no `virtual_address`, `physical_address`, or `length` specified, `alignment` values set to defaults per CPU type.
- `bss` – disabled, `LOAD`, `?RWX`, no `virtual_address`, `physical_address`, or `length` specified, `alignment` values set to defaults per CPU type.
- `note` – `NOTE`.

By default, the `bss` segment is disabled. Any sections of type `SHT_NOBITS`, which are its sole input, are captured in the `data` segment. See Table 7–12 for a full description of `SHT_NOBITS` sections. The simplest `bss` declaration:

```
bss =;
```

is sufficient to enable the creation of a `bss` segment. Any `SHT_NOBITS` sections is captured by this segment, rather than captured in the `data` segment. In its simplest form, this segment is aligned using the same defaults as applied to any other segment. The declaration can also provide additional segment attributes that both enable the segment creation and assign the specified attributes.

The link-editor behaves as if these segments are declared before your `mapfile` is read in. See "Mapfile Option Defaults" on page 306.

Note the following when entering segment declarations:

- A number can be hexadecimal, decimal, or octal, following the same rules as in the C language.
- No space is allowed between the `V`, `P`, `L`, `R`, or `A` and the number.
- The `segment_type` value can be either `LOAD`, `NOTE` or `STACK`. If unspecified it defaults to `LOAD`.

- The segment_flags values are R for readable, W for writable, X for executable, and O for order. No spaces are allowed between the question mark (?) and the individual flags that make up the segment_flags value.
- The segment_flags value for a LOAD segment defaults to RWX.
- NOTE segments cannot be assigned any segment attribute value other than a segment_type.
- One segment_type of value STACK is permitted. Only the access requirements of the segment, selected from the segment_flags, can be specified.
- Implicitly declared segments default to segment_type value LOAD, segment_flags value RWX, a default virtual_address, physical_address, and alignment value, and have no length limit.

Note – The link-editor calculates the addresses and length of the current segment based on the previous segment's attribute values.

- LOAD segments can have an explicitly specified virtual_address value or physical_address value, as well as a maximum segment length value.
- If a segment has a segment_flags value of ? with nothing following, the value defaults to not readable, not writable, and not executable.
- The alignment value is used in calculating the virtual address of the beginning of the segment. This alignment only affects the segment for which it is specified. Other segments still have the default alignment unless their alignments are also changed.
- If any of the virtual_address, physical_address, or length attribute values are not set, the link-editor calculates these values as it creates the output file.
- If an alignment value is not specified for a segment, it is set to the built-in default. This default differs from one CPU to another and might even differ between software revisions.
- If both a virtual_address and an alignment value are specified for a segment, the virtual_address value takes priority.
- If a virtual_address value is specified for a segment, the alignment field in the program header contains the default alignment value.
- If the rounding value is set for a segment, that segment's virtual address will be rounded to the next address that conforms to the value given. This value only effects the segments that it is specified for. If no value is given, no rounding is performed.

Note – If a `virtual_address` value is specified, the segment is placed at that virtual address. For the system kernel, this method creates a correct result. For files that start through `exec`(2), this method creates an incorrect output file because the segments do not have correct offsets relative to their page boundaries.

The `?E` flag allows the creation of an empty segment. This empty segment has no sections associated with it. This segment can only be specified for executables, and must be of type `LOAD` with a specified size and alignment. Multiple segment definitions of this type are permitted.

The `?N` flag enables you control whether the ELF header, and any program headers are included as part of the first loadable segment. By default, the ELF header and program headers are included with the first segment. The information in these headers is used within the mapped image, typically by the runtime linker. The use of the `?N` option causes the virtual address calculations for the image to start at the first section of the first segment.

The `?O` flag enables you control the order of sections in the output file. This flag is intended for use in conjunction with the `-xF` option to the compilers. When a file is compiled with the `-xF` option, each function in that file is placed in a separate section with the same attributes as the `.text` section. These sections are called `.text%`*function_name*.

For example, a file containing three functions, `main()`, `foo()` and `bar()`, when compiled with the `-xF` option, yields a relocatable object file with text for the three functions being placed in sections called `.text%main`, `.text%foo`, and `.text%bar`. Because the `-xF` option forces one function per section, the use of the `?O` flag to control the order of sections in effect controls the order of functions.

Consider the following user-defined `mapfile`:

```
text = LOAD ?RXO;
text: .text%foo;
text: .text%bar;
text: .text%main;
```

The first declaration associates the `?O` flag with the default text segment.

If the order of function definitions in the source file is `main`, `foo`, and `bar`, then the final executable contains functions in the order `foo`, `bar`, and `main`.

For static functions with the same name, the file names must also be used. The `?O` flag forces the ordering of sections as requested in the `mapfile`. For example, if a static function `bar()` exists in files `a.o` and `b.o`, and function `bar()` from file `a.o` is to be placed before function `bar()` from file `b.o`, then the `mapfile` entries should read:

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

Although the syntax allows for the entry:

```
text: .text%bar: a.o b.o;
```

this entry does not guarantee that function `bar()` from file `a.o` is placed before function `bar()` from file `b.o`. The second format is not recommended as the results are not reliable.

Mapping Directives

A mapping directive instructs the link-editor how to map input sections to output segments. Basically, you name the segment that you are mapping to and indicate what the attributes of a section must be in order to map into the named segment. The set of `section_attribute_values` that a section must have to map into a specific segment is called the *entrance criteria* for that segment. In order to be placed in a specified segment of the output file, a section must meet the entrance criteria for a segment exactly.

A mapping directive has the following syntax:

```
segment_name : {section_attribute_value}* [: {file_name}+];
```

For a `segment_name`, you specify any number of `section_attribute_values` in any order, each separated by a space. At most, one section attribute value is allowed for each section attribute. You can also specify that the section must come from a certain `.o` file through a `file_name` declaration. The section attributes and their valid values are shown in the following table.

TABLE 9–2 Section Attributes

Section Attribute	Value
`section_name`	Any valid section name
`section_type`	`$PROGBITS`
	`$SYMTAB`
	`$STRTAB`
	`$REL`
	`$RELA`
	`$NOTE`
	`$NOBITS`
`section_flags`	`? [[!]A] [[!]W] [[!]X]`

Note the following points when entering mapping directives:

- You must choose at most one `section_type` from the `section_types` listed above. The `section_types` listed above are built-in types. For more information on `section_types`, see “Sections” on page 189.
- The `section_flags` values are `A` for allocatable, `W` for writable, or `X` for executable. If an individual flag is preceded by an exclamation mark (`!`), the link-editor checks that the flag is not set. No spaces are allowed between the question mark, exclamation marks, and the individual flags that make up the `section_flags` value.
- `file_name` can be any legal file name, of the form `*filename`, or of the form `archive_name(component_name)`, for example, `/lib/libc.a(printf.o)`. The link-editor does not check the syntax of file names.
- If a `file_name` is of the form `*filename`, the link-editor simulates a `basename`(1) on the file from the command line and uses it to match against the specified `file` name. In other words, the `filename` from the `mapfile` only needs to match the last part of the file name from the command line. See “Mapping Example” on page 304.
- If you use the `-l` option during a link-edit, and the library after the `-l` option is in the current directory, you must precede the library with `./`, or the entire path name, in the `mapfile` in order to create a match.
- More than one directive line can appear for a particular output segment. For example, the following set of directives is legal:

  ```
  S1 : $PROGBITS;
  S1 : $NOBITS;
  ```

  Entering more than one mapping directive line for a segment is the only way to specify multiple values of a section attribute.
- A section can match more than one entrance criteria. In this case, the first segment encountered in the `mapfile` with that entrance criteria is used. For example, if a `mapfile` reads:

  ```
  S1 : $PROGBITS;
  S2 : $PROGBITS;
  ```

  the `$PROGBITS` sections are mapped to segment `S1`.

Section-Within-Segment Ordering

By using the following notation you can specify the order that sections are placed within a segment:

```
segment_name | section_name1;
segment_name | section_name2;
segment_name | section_name3;
```

The sections that are named in the above form are placed before any unnamed sections, and in the order they are listed in the `mapfile`.

Size-Symbol Declarations

Size-symbol declarations enable you to define a new global-absolute symbol that represents the size, in bytes, of the specified segment. This symbol can be referenced in your object files. A size-symbol declaration has the following syntax:

```
segment_name @ symbol_name;
```

`symbol_name` can be any legal C identifier. The link-editor does not check the syntax of the `symbol_name`.

File Control Directives

File control directives enable you to specify which version definitions within shared objects are to be made available during a link-edit. The file control definition has the following syntax:

```
shared_object_name - version_name [ version_name ... ];
```

`version_name` is a version definition name contained within the specified `shared_object_name`.

Mapping Example

The following example is a user-defined `mapfile`. The numbers on the left are included in the example for tutorial purposes. Only the information to the right of the numbers actually appears in the `mapfile`.

EXAMPLE 9–1 User-Defined Mapfile

```
1.  elephant : .data : peanuts.o *popcorn.o;
2.  monkey : $PROGBITS ?AX;
3.  monkey : .data;
4.  monkey = LOAD V0x80000000 L0x4000;
5.  donkey : .data;
6.  donkey = ?RX A0x1000;
7.  text = V0x80008000;
```

Four separate segments are manipulated in this example. The implicitly declared segment `elephant` (line 1) receives all of the `.data` sections from the files `peanuts.o` and `popcorn.o`. Notice that `*popcorn.o` matches any `popcorn.o` file that can be supplied to the link-edit. The file need not be in the current directory. On the other hand, if `/var/tmp/peanuts.o` was supplied to the link-edit, it does not match `peanuts.o` because it is not preceded by an `*`.

The implicitly declared segment `monkey` (line 2) receives all sections that are both `$PROGBITS` and allocatable-executable (`?AX`), as well as all sections not already in the segment `elephant` with the name `.data` (line 3). The `.data` sections entering the `monkey` segment need not be `$PROGBITS` or allocatable-executable because the `section_type` and `section_flags` values are entered on a separate line from the `section_name` value.

An "and" relationship exists between attributes on the same line as illustrated by `$PROGBITS` "and" `?AX` on line 2. An "or" relationship exists between attributes for the same segment that span more than one line, as illustrated by `$PROGBITS ?AX` on line 2 "or" `.data` on line 3.

The `monkey` segment is implicitly declared in line 2 with `segment_type` value `LOAD`, `segment_flags` value `RWX`, and no `virtual_address`, `physical_address`, `length` or `alignment` values specified (defaults are used). In line 4 the `segment_type` value of `monkey` is set to `LOAD`. Because the `segment_type` attribute value does not change, no warning is issued. The `virtual_address` value is set to `0x80000000` and the maximum `length` value to `0x4000`.

Line 5 implicitly declares the `donkey` segment. The entrance criteria are designed to route all `.data` sections to this segment. Actually, no sections fall into this segment because the entrance criteria for `monkey` in line 3 capture all of these sections. In line 6, the `segment_flags` value is set to `?RX` and the `alignment` value is set to `0x1000`. Because both of these attribute values changed, a warning is issued.

Line 7 sets the `virtual_address` value of the text segment to `0x80008000`.

The example of a user-defined `mapfile` is designed to cause warnings for illustration purposes. If you want to change the order of the directives to avoid warnings, use the following example:

```
1.  elephant : .data : peanuts.o *popcorn.o;
4.  monkey = LOAD V0x80000000 L0x4000;
2.  monkey : $PROGBITS ?AX;
3.  monkey : .data;
6.  donkey = ?RX A0x1000;
5.  donkey : .data;
7.  text = V0x80008000;
```

The following `mapfile` example uses the segment-within-section ordering:

```
text = LOAD ?RXN V0xf0004000;
text | .text;
text | .rodata;
text : $PROGBITS ?A!W;
data = LOAD ?RWX R0x1000;
```

The text and data segments are manipulated in this example. Line 1 declares the text segment to have a virtual_address of 0xf0004000 and to *not* include the ELF header or any program headers as part of this segment's address calculations. Lines 2 and 3 turn on section-within-segment ordering and specify that the .text and .rodata sections are the first two sections in this segment. The result is that the .text section have a virtual address of 0xf0004000, and the .rodata section immediately follows that address.

Any other $PROGBITS section that makes up the text segment follows the .rodata section. Line 5 declares the data segment and specifies that its virtual address must begin on a 0x1000 byte boundary. The first section that constitutes the data segment also resides on a 0x1000 byte boundary within the file image.

Mapfile Option Defaults

The link-editor defines four built-in segments (text, data, bss and note) with default segment_attribute_values and corresponding default mapping directives. Even though the link-editor does not use an actual mapfile to provide the defaults, the model of a default mapfile helps illustrate what happens when the link-editor encounters your mapfile.

The following example shows how a mapfile would appear for the link-editor defaults. The link-editor begins execution behaving as if the mapfile has already been read in. Then the link-editor reads your mapfile and either augments or makes changes to the defaults.

```
text = LOAD ?RX;
text : ?A!W;
data = LOAD ?RWX;
data : ?AW;
note = NOTE;
note : $NOTE;
```

As each segment declaration in your mapfile is read in, it is compared to the existing list of segment declarations as follows:

1. If the segment does not already exist in the mapfile but another with the same segment-type value exists, the segment is added before all of the existing segments of the same segment_type.
2. If none of the segments in the existing mapfile has the same segment_type value as the segment just read in, then the segment is added by segment_type value to maintain the following order:

   INTERP

   LOAD

   DYNAMIC

NOTE

3. If the segment is of segment_type LOAD and you have defined a virtual_address value for this LOADable segment, the segment is placed before any LOADable segments without a defined virtual_address value or with a higher virtual_address value, but after any segments with a virtual_address value that is lower.

As each mapping directive in a mapfile is read in, the directive is added after any other mapping directives that you already specified for the same segment but before the default mapping directives for that segment.

Internal Map Structure

One of the most important data structures in the ELF-based link-editor is the map structure. A default map structure, corresponding to the model default mapfile, is used by the link-editor. Any user mapfile augments or overrides certain values in the default map structure.

A typical although somewhat simplified map structure is illustrated in Figure 9–1. The “Entrance Criteria” boxes correspond to the information in the default mapping directives. The “Segment Attribute Descriptors” boxes correspond to the information in the default segment declarations. The “Output Section Descriptors” boxes give the detailed attributes of the sections that fall under each segment. The sections themselves are shown in circles.

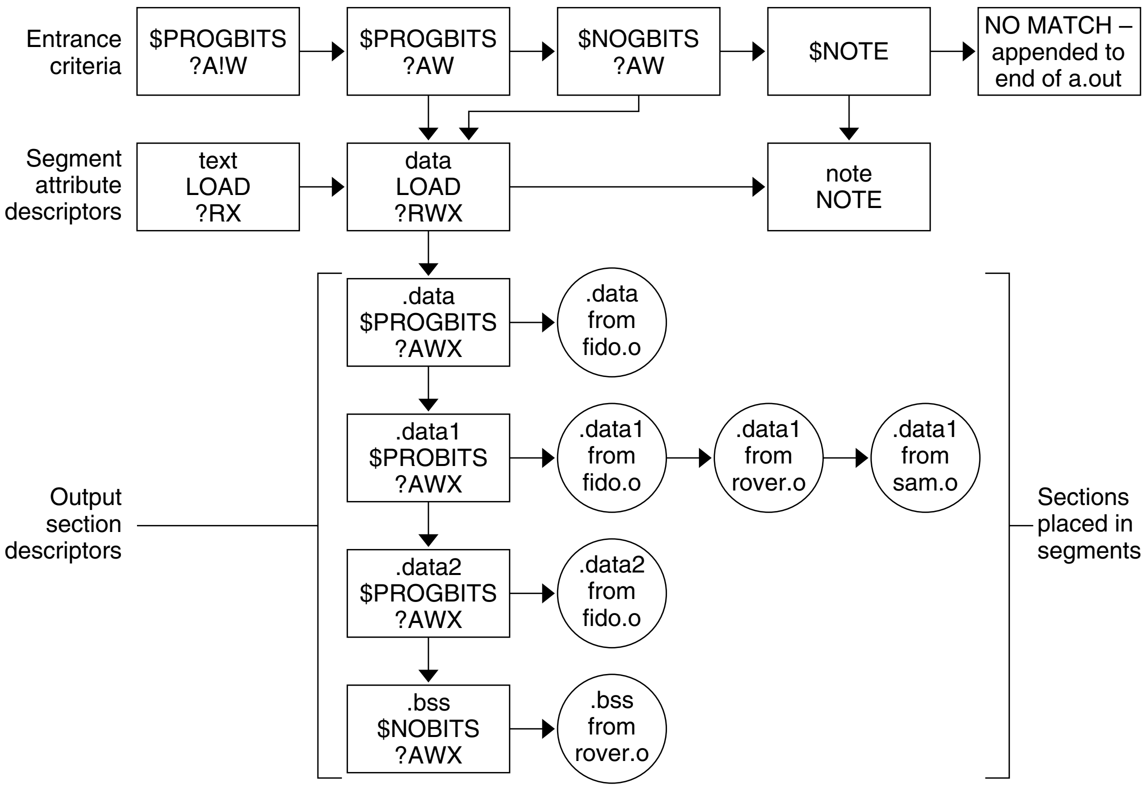


FIGURE 9–1 Simple Map Structure

The link-editor performs the following steps when mapping sections to segments:

1. When a section is read in, the link-editor checks the list of Entrance Criteria looking for a match. All specified criteria must be matched.

   In Figure 9–1, a section that falls into the `text` segment must have a `section_type` value of `$PROGBITS` and have a `section_flags` value of `?A!W`. It need not have the name `.text` since no name is specified in the Entrance Criteria. The section can be either `X` or `!X` in the `section_flags` value because nothing was specified for the execute bit in the Entrance Criteria.

   If no Entrance Criteria match is found, the section is placed at the end of the output file after all other segments. No program header entry is created for this information.

2. When the section falls into a segment, the link-editor checks the list of existing Output Section Descriptors in that segment as follows:

   If the section attribute values match those of an existing Output Section Descriptor exactly, the section is placed at the end of the list of sections associated with that Output Section Descriptor.

For instance, a section with a section_name value of .data1, a section_type value of $PROGBITS, and a section_flags value of ?AWX falls into the second Entrance Criteria box in Figure 9–1, placing it in the data segment. The section matches the second Output Section Descriptor box exactly (.data1, $PROGBITS, ?AWX) and is added to the end of the list associated with that box. The .data1 sections from fido.o, rover.o, and sam.o illustrate this point.

If no matching Output Section Descriptor is found but other Output Section Descriptors of the same section_type exist, a new Output Section Descriptor is created with the same attribute values as the section and that section is associated with the new Output Section Descriptor. The Output Section Descriptor and the section are placed after the last Output Section Descriptor of the same section type. The .data2 section in Figure 9–1 was placed in this manner.

If no other Output Section Descriptors of the indicated section type exist, a new Output Section Descriptor is created and the section is placed in that section.

Note – If the input section has a user-defined section type value between SHT_LOUSER and SHT_HIUSER, it is treated as a $PROGBITS section. No method exists for naming this section_type value in the mapfile, but these sections can be redirected using the other attribute value specifications (section_flags, section_name) in the entrance criteria.

---

3. If a segment contains no sections after all of the command line object files and libraries are read in, no program header entry is produced for that segment.

Note – Input sections of type $SYMTAB, $STRTAB, $REL, and $RELA are used internally by the link-editor. Directives that refer to these section types can only map output sections produced by the link-editor to segments.

APPENDIX **A**

Link-Editor Quick Reference

The following sections provide a simple overview, or *cheat sheet*, of the most commonly used link-editor scenarios. See "Link-Editing" on page 20 for an introduction to the kinds of output modules generated by the link-editor.

The examples provided show the link-editor options as supplied to a compiler driver, this being the most common mechanism of invoking the link-editor. In these examples we use `cc`(1). See "Using a Compiler Driver" on page 27.

The link-editor places no meaning on the name of any input file. Each file is opened and inspected to determine the type of processing it requires. See "Input File Processing" on page 28.

Shared objects that follow a naming convention of `lib`*x*`.so`, and archive libraries that follow a naming convention of `lib`*x*`.a`, can be input using the `-l` option. See "Library Naming Conventions" on page 31. This provides additional flexibility in allowing search paths to be specified using the `-L` option. See "Directories Searched by the Link-Editor" on page 33.

The link-editor basically operates in one of two modes, *static* or *dynamic*.

Static Mode

Static mode is selected when the `-d n` option is used, and enables you to create relocatable objects and static executables. Under this mode, only relocatable objects and archive libraries are acceptable forms of input. Use of the `-l` option results in a search for archive libraries.

Creating a Relocatable Object

- To create a relocatable object use the -d n and -r options:

```
$ cc -dn -r -o temp.o file1.o file2.o file3.o .....
```

Creating a Static Executable

The use of static executables is limited. See "Static Executables" on page 21. Static executables usually contain platform-specific implementation details that restricts the ability of the executable to be run on an alternative platform. Many implementations of Solaris libraries depend on dynamic linking capabilities, such as dlopen(3C) and dlsym(3C). See "Loading Additional Objects" on page 75. These capabilities are not available to static executables.

- To create a static executable use the -d n option *without* the -r option:

```
$ cc -dn -o prog file1.o file2.o file3.o .....
```

The -a option is available to indicate the creation of a static executable. The use of -d n *without* a -r implies -a.

Dynamic Mode

Dynamic mode is the default mode of operation for the link-editor. It can be enforced by specifying the -d y option, but is implied when not using the -d n option.

Under this mode, relocatable objects, shared objects and archive libraries are acceptable forms of input. Use of the -l option results in a directory search, where each directory is searched for a shared object. If no shared object is found, the same directory is then searched for an archive library. A search only for archive libraries can be enforced by using the -B static option. See "Linking With a Mix of Shared Objects and Archives" on page 31.

Creating a Shared Object

- To create a shared object use the -G option. -d y is optional as it is implied by default.
- Input relocatable objects should be built from position-independent code. For example, the C compiler generates position-independent code under the -K pic option. See "Position-Independent Code" on page 118. Use the -z text option to enforce this requirement.

- Avoid including unused relocatable objects. Or, use the -z ignore option, which instructs the link-editor to eliminate unreferenced ELF sections. See "Remove Unused Material" on page 120.
- If the shared object is intended for external use, make sure it uses no application registers. Not using application registers provides the external user freedom to use these registers without fear of compromising the shared object's implementation. For example, the SPARC C compiler does not use application registers under the -xregs=*no%appl* option.
- Establish the shared objects public interface by defining the global symbols that should be visible from the shared object, and reducing any other global symbols to local scope. This definition is provided by the -M option together with an associated mapfile. See Appendix B.
- Use a versioned name for the shared object to allow for future upgrades. See "Coordination of Versioned Filenames" on page 147.
- Self-contained shared objects offer maximum flexibility. They are produced when the object expresses all dependency needs. Use the -z defs to enforce this self containment. See "Generating a Shared Object Output File" on page 44.
- Avoid unneeded dependencies. Use ldd with the -u option to detect and remove unneeded dependencies. See "Shared Object Processing" on page 30. Or, use the -z ignore option, which instructs the link-editor to record dependencies only to objects that are referenced.
- If the shared object being generated has dependencies on other shared objects, indicate they should be lazily loaded using the -z lazyload option. See "Lazy Loading of Dynamic Dependencies" on page 76.
- If the shared object being generated has dependencies on other shared objects, and these dependencies do not reside in the default search locations, record their path name in the output file using the -R option. See "Shared Objects With Dependencies" on page 107.
- Optimize relocation processing by combining relocation sections into a single .SUNW_reloc section. Use the -z combreloc option.
- If interposing symbols are not used on this object or its dependencies, establish direct binding information with -B direct. See "Direct Binding" on page 72.

The following example combines the above points:

```
$ cc -c -o foo.o -K pic -xregs=no%appl foo.c
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
-z combreloc -z ignore -R /home/lib foo.o -L. -lbar -lc
```

- If the shared object being generated is used as input to another link-edit, record within it the shared object's runtime name using the -h option. See "Recording a Shared Object Name" on page 105.
- Make the shared object available to the compilation environment by creating a file system link to a non-versioned shared object name. See "Coordination of Versioned Filenames" on page 147.

The following example combines the above points:

```
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
-z combreloc -z ignore -R /home/lib -h libfoo.so.1 foo.o -L. -lbar -lc
$ ln -s libfoo.so.1 libfoo.so
```

- Consider the performance implications of the shared object: Maximize shareability, as described in "Maximizing Shareability" on page 121: Minimize paging activity, as described in "Minimizing Paging Activity" on page 123: Reduce relocation overhead, especially by minimizing symbolic relocations, as described in "Reducing Symbol Scope" on page 51: Allow access to data via functional interfaces, as described in "Copy Relocations" on page 125.

Creating a Dynamic Executable

- To create a dynamic executable don't use the `-G`, or `-d n` options.
- Indicate that the dependencies of the dynamic executable should be lazily loaded using the `-z lazyload` option. See "Lazy Loading of Dynamic Dependencies" on page 76.
- Avoid unneeded dependencies. Use `ldd` with the `-u` option to detect and remove unneeded dependencies. See "Shared Object Processing" on page 30. Or, use the `-z ignore` option, which instructs the link-editor to record dependencies only to objects that are referenced.
- If the dependencies of the dynamic executable do not reside in the default search locations, record their path name in the output file using the `-R` option. See "Directories Searched by the Runtime Linker" on page 34.
- Establish direct binding information using `-B direct`. See "Direct Binding" on page 72.

The following example combines the above points:

```
$ cc -o prog -R /home/lib -z ignore -z lazyload -B direct -L. \
-lfoo file1.o file2.o file3.o .....
```

APPENDIX B

Versioning Quick Reference

ELF objects make available global symbols to which other objects can bind. Some of these global symbols can be identified as providing the object's *public interface*. Other symbols are part of the object's internal implementation and are not intended for external use. An object's interface can evolve from one software release to another. The ability to identify this evolution is desirable.

In addition, identifying the *internal implementation* changes of an object from one software release to another might be desirable.

Both interface and implementation identifications can be recorded within an object by establishing internal *version definitions*. See Chapter 5 for a more complete introduction to the concept of internal versioning.

Shared objects are prime candidates for internal versioning. This technique defines their evolution, provides for interface validation during runtime processing (see "Binding to a Version Definition" on page 138), and provides for the selective binding of applications (see "Specifying a Version Binding" on page 142). Shared objects are used as the examples throughout this appendix.

The following sections provide a simple overview, or *cheat sheet*, of the internal versioning mechanism provided by the link-editors as applied to shared objects. The examples recommend conventions and mechanisms for versioning shared objects, from their initial construction through several common update scenarios.

Naming Conventions

A shared object follows a naming convention that includes a *major* number file suffix. See "Naming Conventions" on page 104. Within this shared object, one or more *version definitions* can be created. Each version definition corresponds to one of the following categories:

- It defines an industry-standard interface (for example, the *System V Application Binary Interface*).
- It defines a vendor-specific public interface.
- It defines a vendor-specific private interface.
- It defines a vendor-specific change to the internal implementation of the object.

The following version definition naming conventions help indicate which of these categories the definition represents.

The first three of these categories indicate interface definitions. These definitions consist of an association of the global symbol names that make up the interface, with a version definition name. See "Creating a Version Definition" on page 133. Interface changes within a shared object are often referred to as minor revisions. Therefore, version definitions of this type are suffixed with a minor version number, which is based on the file names major version number suffix.

The last category indicates a change having occurred within the object. This definition consists of a version definition acting as a label and has no symbol name associated with it. This definition is referred to as being a weak version definition. See "Creating a Weak Version Definition" on page 136. Implementation changes within a shared object are often referred to as micro revisions. Therefore, version definitions of this type are suffixed with a micro version number based on the previous minor number to which the internal changes have been applied.

Any industry standard interface should use a version definition name that reflects the standard. Any vendor interfaces should use a version definition name unique to that vendor. The company's stock symbol is often appropriate.

Private version definitions indicate symbols that have restricted or uncommitted use, and should have the word "private" clearly visible.

All version definitions result in the creation of associated version symbol names. The use of unique names and the minor/micro suffix convention reduces the chance of symbol collision within the object being built.

The following version definition examples show the possible use of these naming conventions:

`SVABI.1`
: Defines the *System V Application Binary Interface* standards interface.

`SUNW_1.1`
: Defines a Solaris public interface.

`SUNWprivate_1.1`
: Defines a Solaris private interface.

`SUNW_1.1.1`
: Defines a Solaris internal implementation change.

Defining a Shared Object's Interface

When establishing a shared object's interface, you should first determine which global symbols provided by the shared object can be associated to one of the three interface version definition categories:

- Industry standard interface symbols conventionally are defined in publicly available header files and associated manual pages supplied by the vendor, and are also documented in recognized standards literature.
- Vendor public interface symbols conventionally are defined in publicly available header files and associated manual pages supplied by the vendor.
- Vendor private interface symbols can have little or no public definition.

By defining these interfaces, a vendor is indicating the commitment level of each interface of the shared object. Industry standard and vendor public interfaces remain stable from release to release. You are free to bind to these interfaces safe in the knowledge that your application will continue to function correctly from release to release.

Industry-standard interfaces might be available on systems provided by other vendors. You can achieve a higher level of binary compatibility by restricting your applications to use these interfaces.

Vendor public interfaces might not be available on systems provided by other vendors. However, these interfaces remain stable during the evolution of the system on which they are provided.

Vendor private interfaces are very unstable, and can change, or even be deleted, from release to release. These interfaces provide for uncommitted or experimental functionality, or are intended to provide access for vendor-specific applications only. If you want to achieve any level of binary compatibility, you should avoid using these interfaces.

Any global symbols that do not fall into one of the above categories should be reduced to local scope so that they are no longer visible for binding. See "Reducing Symbol Scope" on page 51.

Versioning a Shared Object

Having determined a shared object's available interfaces, the associated version definitions are created using a `mapfile` and the link-editor's `-M` option. See "Defining Additional Symbols" on page 46 for an introduction to this `mapfile` syntax.

The following example defines a vendor public interface in the shared object libfoo.so.1:

```
$ cat mapfile
SUNW_1.1 {                  # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
$ cc -G -o libfoo.so.1 -h libfoo.so.1 -z text -M mapfile foo.c
```

The global symbols foo1 and foo2 are assigned to the shared object's public interface SUNW_1.1. Any other global symbols supplied from the input files are reduced to local by the auto-reduction directive "*". See "Reducing Symbol Scope" on page 51.

Note – Each version definition mapfile entry should be accompanied by a comment reflecting the release or date of the update. This information helps coordinate multiple updates of a shared object, possibly by different developers, into one version definition suitable for delivery of the shared object as part of a software release.

Versioning an Existing (Non-versioned) Shared Object

Versioning an existing, non-versioned shared object requires extra care. The shared object delivered in a previous software release has made available all its global symbols for others to bind with. Although you can determine the shared object's intended interfaces, others might have discovered and bound to other symbols. Therefore, the removal of any symbols might result in an application's failure on delivery of the new versioned shared object.

The internal versioning of an existing, non-versioned shared object can be achieved if the interfaces can be determined, and applied, without breaking any existing applications. The runtime linker's debugging capabilities can be useful to help verify the binding requirements of various applications. See "Debugging Library" on page 96. However, this determination of existing binding requirements assumes that all users of the shared object are known.

If the binding requirements of an existing, non-versioned shared object cannot be determined, then you should create a new shared object file using a new versioned name. See "Coordination of Versioned Filenames" on page 147. In addition to this new shared object, the original shared object must also be delivered so as to satisfy the dependencies of any existing applications.

If the implementation of the original shared object is to be frozen, then maintaining and delivering the shared object binary might be sufficient. If, however, the original shared object might require updating then an alternative source tree from which to generate the shared object can be more applicable. Updating might be necessary through patches, or because its implementation must evolve to remain compatible with new platforms.

Updating a Versioned Shared Object

The only changes that can be made to a shared object that can be absorbed by internal versioning are compatible changes. See "Interface Compatibility" on page 132. Any incompatible changes require producing a new shared object with a new external versioned name. See "Coordination of Versioned Filenames" on page 147.

Compatible updates that can be accommodated by internal versioning fall into three basic categories:

- Adding new symbols
- Creating new interfaces from existing symbols
- Internal implementation changes

The first two categories are achieved by associating an interface version definition with the appropriate symbols. The latter is achieved by creating a weak version definition that has no associated symbols.

Adding New Symbols

Any compatible new release of a shared object that contains new global symbols should assign these symbols to a new version definition. This new version definition should inherit the previous version definition.

The following `mapfile` example assigns the new symbol `foo3` to the new interface version definition `SUNW_1.2`. This new interface inherits the original interface `SUNW_1.1`.

```
$ cat mapfile
SUNW_1.2 {                  # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1 {                  # Release X.
        global:
                foo2;
```

```
                foo1;
        local:
                *;
};
```

The inheritance of version definitions reduces the amount of version information that must be recorded in any user of the shared object.

Internal Implementation Changes

Any compatible new release of the shared object that consists of an update to the implementation of the object, for example, a bug fix or performance improvement, should be accompanied by a *weak* version definition. This new version definition should inherit the latest version definition present at the time the update occurred.

The following `mapfile` example generates a weak version definition `SUNW_1.1.1`. This new interface indicates that the internal changes were made to the implementation offered by the previous interface `SUNW_1.1`.

```
$ cat mapfile
SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.

SUNW_1.1 {                   # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
```

New Symbols and Internal Implementation Changes

If both internal changes and the addition of a new interface have occurred during the same release, both a weak version and an interface version definition should be created. The following example shows the addition of a version definition `SUNW_1.2` and an interface change `SUNW_1.1.1`, which are added during the same release cycle. Both interfaces inherit the original interface `SUNW_1.1`.

```
$ cat mapfile
SUNW_1.2 {                   # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.

SUNW_1.1 {                   # Release X.
```

```
        global:
                foo2;
                foo1;
        local:
                *;
};
```

Note – The comments for the SUNW_1.1 and SUNW_1.1.1 version definitions indicate that they have both been applied to the same release.

Migrating Symbols to a Standard Interface

Occasionally, symbols offered by a vendor's interface become absorbed into a new industry standard. When creating a new standard interface, make sure to maintain the original interface definitions provided by the shared object. Create intermediate version definitions on which the new standard, and original interface definitions, can be built.

The following mapfile example shows the addition of a new industry standard interface STAND.1. This interface contains the new symbol foo4 and the existing symbols foo3 and foo1, which were originally offered through the interfaces SUNW_1.2 and SUNW_1.1 respectively.

```
$ cat mapfile
STAND.1 {                       # Release X+2.
        global:
                foo4;
} STAND.0.1 STAND.0.2;

SUNW_1.2 {                      # Release X+1.
        global:
                SUNW_1.2;
} STAND.0.1 SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;        # Release X+1.

SUNW_1.1 {                      # Release X.
        global:
                foo2;
        local:
                *;
} STAND.0.2;
                                # Subversion - providing for
STAND.0.1 {                     # SUNW_1.2 and STAND.1 interfaces.
        global:
                foo3;
};
                                # Subversion - providing for
STAND.0.2 {                     # SUNW_1.1 and STAND.1 interfaces.
```

```
        global:
                foo1;
};
```

The symbols foo3 and foo1 are pulled into their own intermediate interface definitions, which are used to create the original and new interface definitions.

The new definition of the SUNW_1.2 interface has referenced its own version definition symbol. Without this reference, the SUNW_1.2 interface would have contained no immediate symbol references and hence would be categorized as a weak version definition.

When migrating symbol definitions to a standards interface, any original interface definitions must continue to represent the same symbol list. This requirement can be validated using pvs(1). The following example shows the symbol list of the SUNW_1.2 interface as it existed in the software release X+1.

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
                foo3;
        SUNW_1.1:
                foo2;
                foo1;
```

Although the introduction of the new standards interface in software release X+2 has changed the interface version definitions available, the list of symbols provided by each of the original interfaces remains constant. The following example shows that interface SUNW_1.2 still provides symbols foo1, foo2 and foo3.

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
        STAND.0.1:
                foo3;
        SUNW_1.1:
                foo2;
        STAND.0.2:
                foo1;
```

An application might only reference one of the new subversions. In this case, any attempt to run the application on a previous release results in a runtime versioning error. See “Binding to a Version Definition” on page 138.

An application’s version binding can be promoted by directly referencing an existing version name. See “Binding to Additional Version Definitions” on page 144. For example, if an application only references the symbol foo1 from the shared object libfoo.so.1, then its version reference is to STAND.0.2. To enable this application to be run on previous releases, the version binding can be promoted to SUNW_1.1 using a version control mapfile directive.

```
$ cat prog.c
extern void foo1();

main()
```

```
{
        foo1();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (STAND.0.2);

$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.1);
```

In practice, you rarely have to promote a version binding in this manner. The introduction of new standards binary interfaces is rare, and most applications reference many symbols from an interface family.

APPENDIX C

Establishing Dependencies with Dynamic String Tokens

A dynamic object can establish dependencies explicitly or through filters. Each of these mechanisms can be augmented with a *runpath*, which directs the runtime linker to search for and load the required dependency. String names used to record filters, dependencies and runpath information can be augmented with the reserved dynamic string tokens:

- `$HWCAP`
- `$ISALIST`
- `$OSNAME`, `$OSREL` and `$PLATFORM`
- `$ORIGIN`

The following sections provide examples of how each of these tokens can be employed.

Hardware Capability Specific Shared Objects

The dynamic token `$HWCAP` can be used to specify a directory in which hardware capability specific shared objects exist. This token is available for filters and dependencies. As this token can expand to multiple objects, its use with dependencies is controlled. Dependencies obtained with `dlopen`(3C), can use this token with the mode `RTLD_FIRST`. Explicit dependencies that use this token will load the first appropriate dependency found.

The pathname specification must consist of a full pathname terminated with the `$HWCAP` token. Shared objects that exist in the directory that is specified with the `$HWCAP` token are inspected at runtime. These objects should indicate their hardware capability requirements. See "Identifying Hardware and Software Capabilities" on page 57. Each object is validated against the hardware capabilities that are available to the process. Those objects that are applicable for use with the process, are sorted in descending order of their hardware capability values. These sorted filtees are used to resolve symbols that are defined within the filter.

Filtees within the hardware capabilities directory have no naming restrictions. The following example shows how the auxiliary filter `libfoo.so.1` can be designed to access hardware capability filtees.

```
$ LD_OPTIONS='-f /opt/ISV/lib/hwcap/$HWCAP' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
  [1]    SONAME    libfoo.so.1
  [2]    AUXILIARY /opt/ISV/lib/hwcap/$HWCAP
$ elfdump -H /opt/ISV/lib/hwcap/*

/opt/ISV/lib/hwcap/filtee.so.3:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/hwcap/filtee.so.1:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/hwcap/filtee.so.2:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

If the filter `libfoo.so.1` is processed on a platform where the `MMX` and `SSE` capabilities are available, the following filtee search order occurs.

```
$ cc -o prog prog.c -R. -lfoo
$ LD_DEBUG=symbols prog
.....
debug: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
debug: symbol=foo;  lookup in file=hwcap/filtee.so.2  [ ELF ]
debug: symbol=foo;  lookup in file=hwcap/filtee.so.1  [ ELF ]
.....
```

Note that the capability value for `filtee.so.2` is greater than the capability value for `filtee.so.1`. `filtee.so.3` is not a candidate for inclusion in the symbol search, as the `SSE2` capability is not available.

Reducing Filtee Searches

The use of `$HWCAP` within a filter enables one or more filtees to provide implementations of interfaces that are defined within the filter.

All shared objects within the specified $HWCAP directory are inspected to validate their availability, and to sort those found appropriate for the process. Once sorted, all objects are loaded in preparation for use.

A filtee can be built with the link-editor's -z endfiltee option to indicate that it is the last of the available filtees. A filtee identified with this option, terminates the sorted list of filtees for that filter. No objects sorted after this filtee are loaded for the filter. From the previous example, if the filter.so.2 filtee was tagged with -z endfiltee, the filtee search would be:

```
$ LD_DEBUG=symbols prog
.....
debug: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
debug: symbol=foo;  lookup in file=hwcap/filtee.so.2  [ ELF ]
.....
```

Instruction Set Specific Shared Objects

The dynamic token $ISALIST is expanded at runtime to reflect the native instruction sets executable on this platform, as displayed by the utility isalist(1). This token is available for filters, runpath definitions, and dependencies. As this token can expand to multiple objects, its use with dependencies is controlled. Dependencies obtained with dlopen(3C), can use this token with the mode RTLD_FIRST. Explicit dependencies that use this token will load the first appropriate dependency found.

Any string name that incorporates the $ISALIST token is effectively duplicated into multiple strings. Each string is assigned one of the available instruction sets.

The following example shows how the auxiliary filter libfoo.so.1 can be designed to access an instruction set specific filtee libbar.so.1.

```
$ LD_OPTIONS='-f /opt/ISV/lib/$ISALIST/libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
  [1]    SONAME    libfoo.so.1
  [2]    AUXILIARY /opt/ISV/lib/$ISALIST/libbar.so.1
```

Or alternatively the runpath can be used.

```
$ LD_OPTIONS='-f libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R'/opt/ISV/lib/$ISALIST' foo.c
$ dump -Lv libfoo.so.1 | egrep "RUNPATH|AUXILIARY"
  [1]    RUNPATH   /opt/ISV/lib/$ISALIST
  [2]    AUXILIARY libbar.so.1
```

In either case the runtime linker uses the platform available instruction list to construct multiple search paths. For example, the following application is dependent on libfoo.so.1 and executed on a SUNW,Ultra-2:

```
$ ldd -ls prog
.....
  find object=libbar.so.1; required by ./libfoo.so.1
    search path=/opt/ISV/lib/$ISALIST  (RPATH from file ./libfoo.so.1)
      trying path=/opt/ISV/lib/sparcv9+vis/libbar.so.1
      trying path=/opt/ISV/lib/sparcv9/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8plus+vis/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8plus/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8/libbar.so.1
      trying path=/opt/ISV/lib/sparcv8-fsmuld/libbar.so.1
      trying path=/opt/ISV/lib/sparcv7/libbar.so.1
      trying path=/opt/ISV/lib/sparc/libbar.so.1
```

Or an application with similar dependencies is executed on an `MMX` configured `Pentium Pro`:

```
$ ldd -ls prog
.....
  find object=libbar.so.1; required by ./libfoo.so.1
    search path=/opt/ISV/lib/$ISALIST  (RPATH from file ./libfoo.so.1)
      trying path=/opt/ISV/lib/pentium_pro+mmx/libbar.so.1
      trying path=/opt/ISV/lib/pentium_pro/libbar.so.1
      trying path=/opt/ISV/lib/pentium+mmx/libbar.so.1
      trying path=/opt/ISV/lib/pentium/libbar.so.1
      trying path=/opt/ISV/lib/i486/libbar.so.1
      trying path=/opt/ISV/lib/i386/libbar.so.1
      trying path=/opt/ISV/lib/i86/libbar.so.1
```

Reducing Filtee Searches

The use of `$ISALIST` within a filter enables one or more filtees to provide implementations of interfaces defined within the filter.

Any interface defined in a filter can result in an exhaustive search of all potential filtees in an attempt to locate the required interface. If filtees are being employed to provide performance critical functions, this exhaustive filtee searching can be counterproductive.

A filtee can be built with the link-editor's `-z endfiltee` option to indicate that it is the last of the available filtees. This option terminates any further filtee searching for that filter. From the previous SPARC example, if the `sparcv9` filtee existed, and was tagged with `-z endfiltee`, the filtee searches would be:

```
$ ldd -ls prog
.....
  find object=libbar.so.1; required by ./libfoo.so.1
    search path=/opt/ISV/lib/$ISALIST  (RPATH from file ./libfoo.so.1)
      trying path=/opt/ISV/lib/sparcv9+vis/libbar.so.1
      trying path=/opt/ISV/lib/sparcv9/libbar.so.1
```

System Specific Shared Objects

The dynamic tokens `$OSNAME`, `$OSREL` and `$PLATFORM` are expanded at runtime to provide system specific information. These tokens are available for filters, runpath, or dependency definitions.

`$OSNAME` expands to reflect the name of the operating system, as displayed by the utility `uname`(1) with the `-s` option. `$OSREL` expands to reflect the operating system release level, as displayed by `uname -r`. `$PLATFORM` expands to reflect the underlying hardware implementation, as displayed by `uname -i`.

The following example shows how the auxiliary filter `libfoo.so.1` can be designed to access a platform specific filtee `libbar.so.1`.

```
$ LD_OPTIONS='-f /platform/$PLATFORM/lib/libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
  [1]    SONAME    libfoo.so.1
  [2]    AUXILIARY /platform/$PLATFORM/lib/libbar.so.1
```

This mechanism is used in the Solaris operating environment to provide platform specific extensions to the shared object `/lib/libc.so.1`.

Locating Associated Dependencies

Typically, an unbundled product is designed to be installed in a standalone, unique location. This product is composed of binaries, shared object dependencies, and associated configuration files. For example, the unbundled product `ABC` might have the layout shown in the following figure.

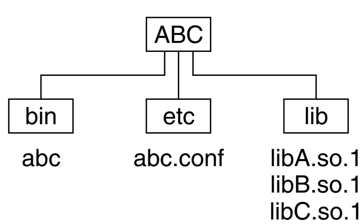


FIGURE C–1 Unbundled Dependencies

Assume that the product is designed for installation under /opt. Normally, you would augment the PATH with /opt/ABC/bin to locate the product's binaries. Each binary locates their dependencies using a hard-coded runpath within the binary. For the application abc, this runpath would be:

```
% cc -o abc abc.c -R/opt/ABC/lib -L/opt/ABC/lib -lA
% dump -Lv abc
  [1]    NEEDED  libA.so.1
  [2]    RUNPATH /opt/ABC/lib
```

and similarly for the dependency libA.so.1 this would be:

```
% cc -o libA.so.1 -G -Kpic A.c -R/opt/ABC/lib -L/opt/ABC/lib -lB
% dump -Lv libA.so.1
  [1]    NEEDED  libB.so.1
  [2]    RUNPATH /opt/ABC/lib
```

This dependency representation works until the product is installed in some directory other than the recommended default.

The dynamic token $ORIGIN expands to the directory in which an object originated. This token is available for filters, runpath, or dependency definitions. Use this technology to redefine the unbundled application to locate its dependencies in terms of $ORIGIN:

```
% cc -o abc abc.c '-R$ORIGIN/../lib' -L/opt/ABC/lib -lA
% dump -Lv abc
  [1]    NEEDED  libA.so.1
  [2]    RUNPATH $ORIGIN/../lib
```

and the dependency libA.so.1 can also be defined in terms of $ORIGIN:

```
% cc -o libA.so.1 -G -Kpic A.c '-R$ORIGIN' -L/opt/ABC/lib -lB
% dump -Lv libA.so.1
  [1]    NEEDED  libB.so.1
  [2]    RUNPATH $ORIGIN
```

If this product is now installed under /usr/local/ABC and the user's PATH is augmented with /usr/local/ABC/bin, invocation of the application abc result in a pathname lookup for its dependencies as follows:

```
% ldd -s abc
.....
  find object=libA.so.1; required by abc
    search path=$ORIGIN/../lib  (RPATH from file abc)
      trying path=/usr/local/ABC/lib/libA.so.1
        libA.so.1 =>     /usr/local/ABC/lib/libA.so.1

  find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
    search path=$ORIGIN  (RPATH from file /usr/local/ABC/lib/libA.so.1)
      trying path=/usr/local/ABC/lib/libB.so.1
        libB.so.1 =>     /usr/local/ABC/lib/libB.so.1
```

Dependencies Between Unbundled Products

Another issue related to dependency location is how to establish a model whereby unbundled products express dependencies between themselves.

For example, the unbundled product XYZ might have dependencies on the product ABC. This dependency can be established by a host package installation script. This script generates a symbolic link to the installation point of the ABC product, as shown in the following figure.

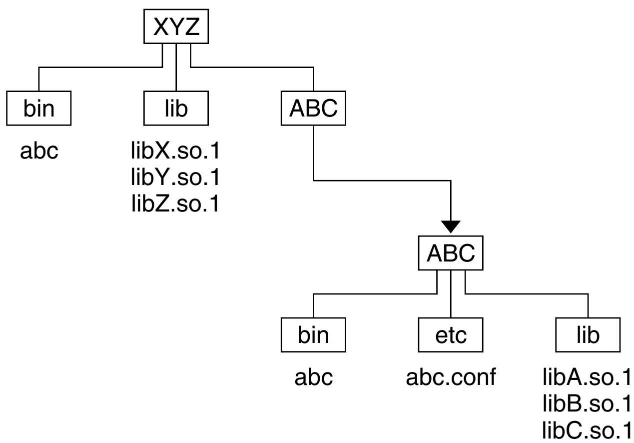


FIGURE C–2 Unbundled Co-Dependencies

The binaries and shared objects of the XYZ product can represent their dependencies on the ABC product using the symbolic link. This link is now a stable reference point. For the application xyz, this runpath would be:

```
% cc -o xyz xyz.c '-R$ORIGIN/../lib:$ORIGIN/../ABC/lib' \
-L/opt/ABC/lib -lX -lA
% dump -Lv xyz
  [1]    NEEDED  libX.so.1
  [2]    NEEDED  libA.so.1
  [3]    RUNPATH $ORIGIN/../lib:$ORIGIN/../ABC/lib
```

and similarly for the dependency libX.so.1 this runpath would be:

```
% cc -o libX.so.1 -G -Kpic X.c '-R$ORIGIN:$ORIGIN/../ABC/lib' \
-L/opt/ABC/lib -lY -lC
% dump -Lv libX.so.1
  [1]    NEEDED  libY.so.1
  [2]    NEEDED  libC.so.1
  [3]    RUNPATH $ORIGIN:$ORIGIN/../ABC/lib
```

If this product is now installed under /usr/local/XYZ, its post-install script would be required to establish a symbolic link of:

```
% ln -s ../ABC /usr/local/XYZ/ABC
```

If the user's PATH is augmented with /usr/local/XYZ/bin, then invocation of the application xyz result in a pathname lookup for its dependencies as follows:

```
% ldd -s xyz
.....
  find object=libX.so.1; required by xyz
    search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib  (RPATH from file xyz)
      trying path=/usr/local/XYZ/lib/libX.so.1
        libX.so.1 =>     /usr/local/XYZ/lib/libX.so.1

  find object=libA.so.1; required by xyz
    search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib  (RPATH from file xyz)
      trying path=/usr/local/XYZ/lib/libA.so.1
      trying path=/usr/local/ABC/lib/libA.so.1
        libA.so.1 =>     /usr/local/ABC/lib/libA.so.1

  find object=libY.so.1; required by /usr/local/XYZ/lib/libX.so.1
     search path=$ORIGIN:$ORIGIN/../ABC/lib \
                (RPATH from file /usr/local/XYZ/lib/libX.so.1)
      trying path=/usr/local/XYZ/lib/libY.so.1
        libY.so.1 =>     /usr/local/XYZ/lib/libY.so.1

  find object=libC.so.1; required by /usr/local/XYZ/lib/libX.so.1
     search path=$ORIGIN:$ORIGIN/../ABC/lib \
                (RPATH from file /usr/local/XYZ/lib/libX.so.1)
      trying path=/usr/local/XYZ/lib/libC.so.1
      trying path=/usr/local/ABC/lib/libC.so.1
        libC.so.1 =>     /usr/local/ABC/lib/libC.so.1

  find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
     search path=$ORIGIN  (RPATH from file /usr/local/ABC/lib/libA.so.1)
       trying path=/usr/local/ABC/lib/libB.so.1
        libB.so.1 =>     /usr/local/ABC/lib/libB.so.1
```

Security

In a secure process, the expansion of the $ORIGIN string is allowed only if it expands to a trusted directory. The occurrence of other relative path names, poses a security risk.

A path like $ORIGIN/../lib apparently points to a fixed location, fixed by the location of the executable. However, the location is not actually fixed. A writable directory in the same file system could exploit a secure program that uses $ORIGIN.

The following example shows this possible security breach if $ORIGIN was arbitrarily expanded within a secure process.

```
% cd /worldwritable/dir/in/same/fs
% mkdir bin lib
% ln $ORIGIN/bin/program bin/program
% cp ~/crooked-libc.so.1 lib/libc.so.1
% bin/program
```

```
..... using crooked-libc.so.1
```

You can use the utility `crle`(1) to specify trusted directories that enable secure applications to use `$ORIGIN`. Administrators who use this technique should ensure that the target directories are suitably protected from malicious intrusion.

APPENDIX D

Linker and Libraries Updates and New Features

This appendix provides an overview of updates and new features that have been added to releases of the Solaris OS.

Solaris 10 Release

- A restructuring of the filesystem has moved many components from under /usr/lib to /lib. Both the link-editor and runtime linkers default search paths have been changed accordingly. See "Directories Searched by the Link-Editor" on page 33, "Directories Searched by the Runtime Linker" on page 66, and "Security" on page 83.
- System archive libraries are no longer provided. Therefore, the creation of a statically linked executable is no longer possible. See "Static Executables" on page 21.
- Greater flexibility for defining alternative dependencies is provided with the -A option of crle(1).
- The link-editors now process environment variables specified without a value. See "Environment Variables" on page 23.
- Pathnames used with dlopen(3C), and as explicit dependency definitions, can now use any reserved tokens. See Appendix C. The evaluation of pathnames that use reserved tokens is provided with the new utility moe(1).
- An optimal means of testing for the existence of an interface is provide with dlsym(3C) and the new handle RTLD_PROBE. See "Providing an Alternative to dlopen()" on page 78.

Solaris 9 7/04 Release

- Greater flexibility in defining the hardware and software requirements of ELF objects is provided with the link-editors. See "Hardware and Software Capabilities Section" on page 207.
- The runtime link auditing interface `la_objfilter()` has been added. See "Audit Interface Functions" on page 160.
- Shared object filtering has been extended to provide filtering on a per-symbol basis. See "Shared Objects as Filters" on page 109.

Solaris 9 4/04 Release

- The new section types `SHT_SUNW_ANNOTATE`, `SHT_SUNW_DEBUGSTR`, `SHT_SUNW_DEBUG`, and `SHT_SPARC_GOTDATA` are supported. See Table 7–12.
- The analysis of runtime interfaces is simplified with the new utility `lari`(1).
- Greater control of direct bindings is provided with the link-editor options `-z direct` and `-z nodirect`, together with the `DIRECT` and `NODIRECT` `mapfile` directives. See "Defining Additional Symbols" on page 46, and "Direct Binding" on page 72.

Solaris 9 12/03 Release

- Performance improvements within `ld`(1) can significantly reduce the link-edit time of very large applications.

Solaris 9 8/03 Release

- `dlsym`(3C) symbol processing can be reduced using a `dlopen`(3C) handle that is created with the `RTLD_FIRST` flag. See "Obtaining New Symbols" on page 93.
- The signal used by the runtime linker to terminate an erroneous process can be managed using the `dlinfo`(3C) flags `RTLD_DI_GETSIGNAL`, and `RTLD_DI_SETSIGNAL`.

Solaris 9 12/02 Release

- The link-editor provides string table compression, that can result in reduced `.dynstr` and `.strtab` sections. This default processing can be disabled using the link-editor's `-z nocompstrtab` option. See "String Table Compression" on page 56.
- The `-z ignore` option has been extended to eliminate unreferenced sections during a link-edit. See "Remove Unused Material" on page 120.
- Unreferenced dependencies can be determined using `ldd`(1). See the `-U` option.
- The link-editors support extended ELF sections. See "ELF Header" on page 182, Table 7–12, "Sections" on page 189, Table 7–16 and "Symbol Table Section" on page 225.
- Greater flexibility in defining a symbols visibility is provided with the `protected` `mapfile` directive. See "Defining Additional Symbols" on page 46.

Solaris 9 Release

- Thread-Local Storage (TLS) support is provided. See Chapter 8.
- The `-z rescan` option provides greater flexibility in specifying archive libraries to a link-edit. See "Position of an Archive on the Command Line" on page 32.
- The `-z ld32` and `-z ld64` options provide greater flexibility in using the link-editor support interfaces. See "32–Bit and 64–Bit Environments" on page 152.
- Additional link-editor support interfaces, `ld_input_done()`, `ld_input_section()`, `ld_input_section64()` and `ld_version()` have been added. See "Support Interface Functions" on page 153.
- Environment variables interpreted by the runtime linker can now be established for multiple processes by specifying these variables within a configuration file. See the `-e` and `-E` options of `crle`(1).
- Support for more than 32,768 procedure linkage table entries within 64–bit SPARC objects has been added. See "SPARC: 64–bit Procedure Linkage Table" on page 267.
- An `mdb`(1) debugger module enables you to inspect runtime linker data structures as part of process debugging. See "Debugger Module" on page 99.
- The `bss` segment declaration directive makes the creation of a `bss` segment easier. See "Segment Declarations" on page 298.

Solaris 8 07/01 Release

- Unused dependencies can be determined using ldd(1). See the -u option.
- Various ELF ABI extensions have been added. See "Initialization and Termination Sections" on page 35, "Initialization and Termination Routines" on page 79, Table 7–4, Table 7–7, Table 7–14, Table 7–15, "Group Section" on page 205, Table 7–16, Table 7–25, Table 7–43, Table 7–44, and "Program Loading (Processor-Specific)" on page 245.
- Greater flexibility in the use of link-editor environment variables has been provided with the addition of _32 and _64 variants. See "Environment Variables" on page 23.

Solaris 8 01/01 Release

- The symbolic information that is made available from dladdr(3C) has been enhanced with the introduction of dladdr1().
- The $ORIGIN of a dynamic object can be obtained from dlinfo(3C).
- The maintenance of runtime configuration files that are created with crle(1) has been simplified. Inspection of a configuration file displays the command-line options used to create the file. An update capability is provided with the -u option.
- The runtime linker and its debugger interface have been extended to detect procedure linkage table entry resolution. This update is identified by a new version number. See rd_init() under "Agent Manipulation Interfaces" on page 169. This update extends the rd_plt_info_t structure. See rd_plt_resolution() under "Procedure Linkage Table Skipping" on page 175.
- An application's stack can be defined non-executable by using the new mapfile segment descriptor STACK. See "Segment Declarations" on page 298.

Solaris 8 10/00 Release

- The environment variable LD_BREADTH is ignored by the runtime linker. See "Initialization and Termination Routines" on page 79.
- The runtime linker and its debugger interface have been extended for better runtime and core file analysis. This update is identified by a new version number. See rd_init() under "Agent Manipulation Interfaces" on page 169. This update extends the rd_loadobj_t structure. See "Scanning Loadable Objects" on page

171.

- You can now validate displacement relocated data in regard to its use, or possible use, with copy relocations. See "Displacement Relocations" on page 60.
- 64–bit filters can be built solely from a `mapfile` by using the link-editor's `-64` option. See "Generating Standard Filters" on page 110.
- The search paths used to locate the dependencies of dynamic objects can be inspected using `dlinfo`(3C).
- `dlsym`(3C) and `dlinfo`(3C) lookup semantics have been expanded with a new handle `RTLD_SELF`.
- The runtime symbol lookup mechanism used to relocate dynamic objects can be significantly reduced by establishing direct binding information within each dynamic object. See "Direct Binding" on page 72.

Solaris 8 Release

- The secure directory from which files can be preloaded is now `/usr/lib/secure` for 32–bit objects, and `/usr/lib/secure/64` for 64–bit objects. See "Security" on page 83.
- Greater flexibility in modifying the runtime linker's search paths can be achieved with the link-editor's `-z nodefaultlib` option, and runtime configuration files created by the new utility `crle`(1). See "Directories Searched by the Runtime Linker" on page 34 and "Configuring the Default Search Paths" on page 69.
- The new `EXTERN` `mapfile` directive enables you to use `-z defs` with externally defined symbols. See "Defining Additional Symbols" on page 46.
- The new `$ISALIST`, `$OSNAME`, and `$OSREL` dynamic string tokens provide greater flexibility in establishing instruction set specific, and system specific dependencies. See "Dynamic String Tokens" on page 69.
- The link-editor options `-p` and `-P` provide additional means of invoking runtime link auditing libraries. See "Recording Local Auditors" on page 159. The runtime link auditing interfaces `la_activity()` and `la_objsearch()` have been added. See "Audit Interface Functions" on page 160.
- A new dynamic section tag, `DT_CHECKSUM`, enables you to coordinate ELF files with core images. See Table 7–43.

Solaris 7 Release

- The 64–bit ELF object format is now supported. See "File Format" on page 179 for details. Link-editor extensions and differences for 64–bit processing include the use of /usr/lib/64 (see "Directories Searched by the Link-Editor" on page 33, "Directories Searched by the Runtime Linker" on page 34, and "Naming Conventions" on page 104), the environment variable LD_LIBRARY_PATH_64 (see "Using an Environment Variable" on page 33, and "Directories Searched by the Runtime Linker" on page 66), and the runtime linker /usr/lib/64/ld.so.1 (see Chapter 3).
- You can build shared objects with optimized relocation sections by using the link-editor's -z combreloc option. See "Combined Relocation Sections" on page 125.
- The new $ORIGIN dynamic string token provides greater flexibility in establishing dependencies within unbundled software. See "Dynamic String Tokens" on page 69.
- The loading of a shared object can now be deferred until the object is actually referenced by the running program. See "Lazy Loading of Dynamic Dependencies" on page 118.
- The new SHT_SUNW_COMDAT section type enables the elimination of multiply-defined symbols. See "COMDAT Section" on page 205.
- The new SHT_SUNW_move section type enables partially initialized symbols. See "Move Section" on page 209.
- The runtime link auditing interfaces la_symbind64(), la_sparcv9_pltenter(), and la_pltexit64(), together with a new link-auditing flag LA_SYMB_ALTVALUE, have been added. See "Audit Interface Functions" on page 160.

Solaris 2.6 Release

- Weak symbol references can trigger archive member extraction by using the link-editor's -z weakextract option. The extraction of all archive members can be achieved using the -z allextract option. See "Archive Processing" on page 28.
- Shared objects specified as part of a link-edit that are not referenced by the object being built, can be ignored by using the link-editor's -z ignore option. See "Shared Object Processing" on page 30.
- The link-editor generates the reserved symbols _START_ and _END_ to provide a means of establishing an object's address range. See "Generating the Output File" on page 56.

- Changes have been made to the runtime ordering of initialization and finalization code to better accommodate dependency requirements. See "Initialization and Termination Routines" on page 79.
- Symbol resolution semantics have been expanded for dlopen(3C). See "Symbol Lookup" on page 87, RTLD_GROUP in "Isolating a Group" on page 92, and RTLD_PARENT in "Object Hierarchies" on page 92.
- Symbol lookup semantics have been expanded with a new dlsym(3C) handle RTLD_DEFAULT. See "Default Symbol Lookup Model" on page 88.
- Extensions have been made to filter processing that allow more than one filtee to be defined, and provide for forcibly loading filtees. See "Shared Objects as Filters" on page 109.
- You can record additional version dependencies by using the mapfile file control directive $ADDVERS. See "Binding to Additional Version Definitions" on page 144.
- A runtime linker audit interface provides support for monitoring and modifying a dynamically linked application from within the process. See "Runtime Linker Auditing Interface" on page 157.
- A runtime linker debugger interface provides support for monitoring and modifying a dynamically linked application from an external process. See "Runtime Linker Debugger Interface" on page 167.
- Additional section information is supported. See Table 7–11 for SHN_BEFORE and SHN_AFTER. See Table 7–14 for SHF_ORDERED and SHF_EXCLUDE.
- A new dynamic section tag, DT_1_FLAGS, is supported. See Table 7–45 for the various flag values.
- A package of demonstration ELF programs is provided. See Chapter 7.
- The link-editors now support internationalized messages. All system errors are reported using strerror(3C).
- The new eliminate mapfile directive, or the -B eliminate option, enable you to elimination local symbol table entries. See "Symbol Elimination" on page 55.

Index

F

G

I

L

M

N

O

P

R

S

T

U

V

W