Oracle® Solaris 11.2 Linkers and Libraries Guide

Part No: E36857
July 2014

ORACLE

Contents

Figures

Tables

Using This Documentation

- **Overview** – In the Oracle Solaris™ operating system (Oracle 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).

  The *Linkers and Libraries Guide* describes the operations of the Oracle 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.

Note - This Oracle Solaris™ release supports systems that use the SPARC® and x86 families of processor architectures. The supported systems appear in the *Oracle Solaris OS: Hardware Compatibility Lists*. This document cites any implementation differences between the platform types.

In this document, these x86 related terms mean the following:

- x86 refers to the larger family of 64-bit and 32-bit x86 compatible products.
- x64 relates specifically to 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

- **Audience** – This guide is intended for a range of programmers who are interested in the Oracle Solaris link-editor, runtime linker, and related tools, 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.
- **Required knowledge** – Readers of this guide should be familiar and be able to use the following technologies.
  - A UNIX® SVR4 system – preferably the current Oracle Solaris™ release.
  - The C programming language, and application development.

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://www.oracle.com/pls/topic/lookup?ctx=E36784.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

PART I

Using the Link-Editor and Runtime Linker

♦ ♦ ♦ CHAPTER 1

Introduction to the Oracle Solaris Link Editors

This manual describes the operations of the Oracle Solaris link-editor and runtime linker, together with the objects on which these utilities operate. The basic operation of the Oracle Solaris link-editor and runtime linker involve 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, dynamic executable, 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 Oracle Solaris link-editor, runtime linker, and related tools, 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

The link-editor, ld(1), takes a variety of input files, typically generated from compilers, assemblers, or previous invocations of the link-editor. The link-editor concatenates and interprets the data within these input files to form an output file. The output file that is produced is one of the following basic types.

- *Dynamic Executable* - A concatenation of input relocatable objects that requires intervention by the runtime linker, ld.so.1(1), to produce a runnable process. Dynamic executables typically have one or more dependencies in the form of shared objects.

  A dynamic executable is created when the `-z type=exec` option is used, or is the default when no other options that control the output file type are provided.
- *Position-independent Executable* - A special case of a shared object, that specifies an interpreter. Position-independent executables should be created from position-independent code. Unlike a dynamic executable, which requires a fixed address space for execution, a position-independent executable can be loaded at an arbitrary address chosen by exec(2).

  A position-independent executable is created when the `-z type=pie` option is used.
- *Relocatable Object* - A concatenation of input relocatable objects that can be used in subsequent link-edit phases.

  A relocatable object is created when the `-z type=reloc` option, or `-r` option are used.
- *Shared Object* - A concatenation of input relocatable objects that provide services that can be bound to a dynamic executable at runtime. Shared objects should be created from position-independent code. Shared objects can have dependencies on other shared objects.

  A shared object is created when the `-z type=shared` option, or `-G` option are used.

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.

Beginning with the Oracle Solaris 10 release, the OS no longer includes 32–bit system archive libraries. Without these libraries, specifically `libc.a`, the creation of a static executable is no

longer achievable without specialized system knowledge. Note, that the link-editors ability 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 Oracle Solaris applications.

Application Binary Interfaces

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

The Oracle Solaris ABI is a technological descendent for work on ABI's 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 Environments and 64–Bit Environments

The link-editor is provided as a 32–bit application and a 64–bit application. Each link-editor can operate on 32–bit objects and 64–bit objects. On systems that are running a 64–bit environment, both versions of the link-editor can be executed. On systems that are running a 32–bit environment, only the 32–bit version of the link-editor can be executed.

The runtime linker is provided as a 32–bit object and a 64–bit object. The 32–bit object is used to execute 32–bit processes, and the 64–bit object is used to execute 64–bit processes.

The operations of the link-editor and runtime linker 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.

Environment Variables

The link-editor and runtime linker 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 Oracle Solaris 10 release, the link-editor and runtime linker 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
```

Beginning with the Oracle 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 Oracle Solaris OS 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 12, “Object File Format”. 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 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 sections that compose the output file. 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 Appendix B, *"System V Release 4 (Version 1) Mapfiles"*.

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.

Note - Starting with Oracle Solaris 11, various compilation components have been moved from `/usr/ccs/bin` and `/usr/ccs/lib`, to `/usr/bin` and `/usr/lib`. However, applications exist that refer to the original `ccs` names. Symbolic links have been used to maintain compatibility.

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, the following command instructs the link-editor to create a dynamic executable that is named `a.out` using only the input file `test.o`.

```
$ ld test.o
```

Typically, a dynamic executable requires specialized startup code and exit processing code. 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.

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

Cross Link-Editing

The link-editor is a cross link-editor, able to link 32–bit objects or 64–bit objects, for SPARC or x86 targets. The mixing of 32–bit objects and 64–bit objects is not permitted. Similarly, only objects of a single machine type are allowed.

Typically, no command line option is required to distinguish the link-edit target. The link-editor uses the ELF machine type 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 native target. To explicitly define the link-edit target use the `-z target` option.

Specifying the Link-Editor Options

Typically, link-edits are completely specified using command line options. However, a variety of environment variables are provided to augment command line processing. These variables

provide for supplying options that might clash with compiler options. These variables also provide for overriding, or unsetting, the command line options that are embedded in scripts and build environments.

Any inconsistencies between command line options result in a fatal error condition. Any inconsistencies that involve an option provided by an environment variable result in a warning, and the first option taking precedence. Any `UNSET` operation is accompanied with a warning notification.

Options are interpreted from the environment and the command line in the following order.

- From the `LD_OPTIONS` environment variable.
- From the command line.
- From the `LD_UNSET` environment variable.

`LD_OPTIONS` can be used to pass arguments to the link-editor that would otherwise be interpreted by the compiler drivers. For example, diagnostics related to the link-edit can be obtained using the `-D` option. This option is normally interpreted by the compiler preprocessor.

```
$ LD_OPTIONS=-Dargs  cc -o main main.c
...
debug: arg[0]  option=-D:  option-argument: args  (LD_OPTIONS)
debug:
debug: arg[0]  /usr/ccs/bin/ld
debug: arg[2]  option=-o:  option-argument: main
debug: arg[3]  option=-Q:  option-argument: y
debug: arg[4]  option=-l:  option-argument: c
```

`LD_OPTIONS` can also be used to override options that have a family of variants. For example, an embedded `-z text` option can be overridden by a `-z textoff` option.

```
$ LD_OPTIONS=-ztextoff  cc -ztext -G null.o
ld: warning: option '-ztextoff' and option '-ztext' are incompatible, \
    first option taken
```

Some options have no alternative variants, and therefore can not be overridden. However, they can be unset. For example, a standard link-edit can create the following sections.

```
$ cc -o main main.c
$ elfdump -c main | egrep "symtab|debug"
Section Header[19]:  sh_name: .symtab
Section Header[22]:  sh_name: .debug_info
Section Header[23]:  sh_name: .debug_line
```

These sections can be removed with the `-z strip-class` option.

```
$ cc -o main -zstrip-class=symbol -zstrip-class=debug main.c
$ elfdump -c main | egrep "symtab|debug"
$
```

Individual strip options can be unset. The follow example unsets the stripping of debug sections.

```
$ LD_UNSET=-zstrip-class=debug  cc -o main -zstrip-class=symbol \
    -zstrip-class=debug main.c
ld: warning: unsetting option '-zstrip-class=debug': LD_UNSET directed
$ elfdump -c main | egrep "symtab|debug"
Section Header[20]:  sh_name: .debug_info
Section Header[21]:  sh_name: .debug_line
```

In addition, options that provide for multiple instances, such as `-z strip-class` can have all family members unset by specifying the option without any qualifying option string. The following example unsets the stripping of debug and symbol table sections.

```
$ LD_UNSET=-zstrip-class  cc -o main -zstrip-class=symbol \
    -zstrip-class=debug main.c
ld: warning: unsetting option '-zstrip-class': LD_UNSET directed
$ elfdump -c main | egrep "symtab|debug"
Section Header[19]:  sh_name: .symtab
Section Header[22]:  sh_name: .debug_info
Section Header[23]:  sh_name: .debug_line
```

The output object type is determined from the `LD_OPTIONS`, command line, and `LD_UNSET` components. This object type is then used to investigate any `LD_{object-type}_UNSET`, and `LD_{object-type}_OPTIONS` environment variables to remove, or add, options specific to the object type being built. The object-type provides the types, in uppercase, defined by the `-z type` option, and is one of `EXEC`, `PIE`, `RELOC` or `SHARED`. For example, the `LD_EXEC_OPTIONS` option is interpreted when the output file type is a dynamic executable. These environment variables are processed in the following order.

- From the `LD_{object-type}_UNSET` environment variable.
- From the `LD_{object-type}_OPTIONS` environment variable.

For example, a build environment, that creates both dynamic executables and shared objects, can use the `LD_EXEC_OPTIONS` environment variable to enable address space layout randomization for all dynamic executables.

```
$ LD_EXEC_OPTIONS=-zaslr  build.sh
```

Any command line options that are inconsistent with this output object type result in a fatal error condition. Any inconsistent option provided by an environment variable results in a warning, and the option being ignored.

See Chapter 5, “Link-Editor Quick Reference” 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. Information sections do not become part of the output file image, as new information sections are generated to take their place. 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 30. 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 under the following conditions.

- 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 archive member contains a symbol definition that matches a reference that requires hidden visibility or protected visibility. See Table 12-23.
- 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 only the relocatable objects needed when an 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 31.

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'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 facility, and `ldd`(1) with the `-u` option, can be used to determine unused dependencies. The link-editor's `-z discard-unused=dependencies` option can be used to

suppress the dependency recording of any unused shared objects. See also “Removing Unused Dependencies” on page 182.

If a shared object has dependencies on other shared objects, these dependencies can also be 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. In the absence of lazy loading, the runtime linker 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, `libfoo.so` is the shared object version of the “*foo*” implementation that is made available to the compilation environment. `libfoo.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 135 and “Coordination of Versioned Filenames” on page 250.

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` options 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`. The link-editor then searches for `libbar.so`, and if that search fails, searches for `libbar.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
```

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.

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

The determination, and maintenance, of repeated archive specifications can be tedious. The `-z rescan-now` option makes this process simpler. The `-z rescan-now` option is processed by the link-editor immediately when the option is encountered on the command line. All archives that have been processed from the command line prior to this option are immediately 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 can be simplified as follows.

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

Alternatively, the `-z rescan-start` and `-z rescan-end` options can be used to group mutually dependent archives together into an archive group. These groups are reprocessed by the link-editor immediately when the closing delimiter is encountered on the command line. Archives found within the group are reprocessed in an attempt to locate additional archive members that resolve symbol references. This archive rescanning continues until a pass over the archive group occurs in which no new members are extracted. Using archive groups, the previous example can be written as follows.

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

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

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 two standard directories in which to look for libraries, `/lib` followed by `/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 /lib, and finally /usr/lib, to find libfoo. The command searches path1 and then path2, followed by /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. Compiler drivers often use the -Y option to provide compiler specific search paths.

Using an Environment Variable

You can also use the environment variable LD_LIBRARY_PATH to add to the link-editor's library search path. Typically, LD_LIBRARY_PATH takes a colon-separated list of directories. In its most general form, LD_LIBRARY_PATH can also take two directory lists separated by a semicolon. These lists are searched before and after the -Y 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:/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:/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 96. 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, the last search paths used are always the default locations.

Note - The default search path can be administrated by using a runtime configuration file. See "Configuring the Default Search Paths" on page 99. However, the creator of a dynamic 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 locations.

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 257.

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. The initialization code of a dynamic object is executed once each time the dynamic object is loaded in a process. The termination code of a dynamic object is executed once each time the dynamic object is unloaded from a process or at process termination. 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 `.pre_initarray`, `.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 that the sections can be called by the runtime linker. A *pre-initialization* array is applicable to dynamic executables only.

Note - Functions that are assigned to these arrays must be provided from the object that is being built.

The sections `.init` and `.fini` provide a runtime initialization and termination code block, respectively. The compiler drivers typically supply `.init` and `.fini` sections with files they add

to the beginning and end of your input file list. These compiler provided files have the effect of encapsulating the `.init` and `.fini` code from your relocatable objects 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 110.

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 `.init_array` element, and the address of `bar` in a `.fini_array` element.

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

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

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

void main()
{
        (void) printf("main()\n");
}
$ 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. Simplified, self contained, initialization and termination code provides predictable runtime behavior. See "Initialization and Termination Order" on page 112 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

Symbols can be categorized as *local* or *global*. See "Symbol Visibility" on page 38.

During input file processing, local symbols are copied from any input relocatable object files to the output object being built, without examination.

The global symbols from all input relocatable objects, and the global symbols from any external dependencies, are analyzed and combined in a process known as *symbol resolution*. The link-editor places each symbol in an internal symbol table in the order that the symbols are encountered. If a symbol with the same name was contributed by an earlier object, and already exists in the symbol table, the symbol resolution process determines which of the two symbols to keep. As a side effect of this process, the link-editor determines how to establish references to external object dependencies.

On successful completion of input file processing, the link-editor applies any symbol visibility adjustment, and determines if any unresolved symbol references remain. If any fatal symbol resolution errors have occurred, or if any unresolved symbol references remain, the link-edit terminates. 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 visibilities, symbol resolution, and undefined symbol processing.

Symbol Visibility

Symbols can be categorized as *local* or *global*. Local symbols can not be referenced from an object other than the object that contains the symbol definition. By default, local symbols are copied from any input relocatable object files to the output object being built. Local symbols can instead be eliminated from the output object. See “Symbol Elimination” on page 53.

Global symbols can be referenced from other objects besides the object that contains the symbol definition. After collection and resolution, global symbols are added to the symbol tables being created in the output object. Although all global symbols are processed and resolved together, their final visibility can be adjusted. Global symbols can define additional visibility attributes. See Table 12-23. In addition, `mapfile` symbol directives can be used to assign symbol visibilities during a link-edit. See Table 8-8. These visibility attributes, and directives, can result in a global symbol having its visibility adjusted when written to the output object.

When creating a relocatable object, all visibility attributes and directives are recorded in the output object. However, the visibility changes implied by these attributes are not applied. Any visibility processing is instead deferred to a subsequent link-edit of a dynamic object that reads these objects as input. In special cases, the `-B reduce` option can be used to force the immediate interpretation of any visibility attributes or directives.

When creating a dynamic executable, or shared object, symbol visibility attributes and directives are applied before the symbols are written to any symbol tables. Visibility attributes can ensure that symbols remain global, and are not affected by any symbol reduction techniques. Visibility attributes and directives can also result in global symbols being demoted to local. This latter technique is most frequently used to explicitly define an objects exported interface. See “Reducing Symbol Scope” on page 49.

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 most common simple resolutions involve binding symbol references from one object to symbol definitions within another object. This binding can occur between two relocatable objects, or between a relocatable object and the first definition found in a shared object dependency. Complex resolutions typically occur between two or more relocatable objects.

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 365. 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;

int d_foo()
{
        return (u_foo(u_bar, t_bar, d_bar));
}
$ cc -o main.o -c main.c
$ elfdump -s main.o

Symbol Table Section:  .symtab
    index  value   size  type bind oth ver shndx          name
    ....
      [7]      0      0  FUNC GLOB  D    0 UNDEF          u_foo
      [8]   0x10   0x40  FUNC GLOB  D    0 .text          d_foo
      [9]    0x4    0x4  OBJT GLOB  D    0 COMMON         t_bar
     [10]      0    0x4  NOTY GLOB  D    0 UNDEF          u_bar
     [11]      0    0x4  OBJT GLOB  D    0 .data          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. This resolution can occur between two relocatable objects, or between a relocatable object and the first definition found in a shared object dependency.

Symbols that undergo resolution can have either a global or weak binding. When processing relocatable objects, weak bindings have lower precedence than global bindings. A weak symbol definition is silently overridden by a 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. Multiply-defined symbols that occur between relocatable objects and shared objects, or between multiple shared objects, are treated identically. A symbols weak binding or global binding is irrelevant. By resolving to the first definition, regardless of the symbols binding, both the link-editor and runtime linker behave consistently.

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 };
$ ld -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
int bar()
{
```

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

int 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
int bar()
{
        return (0);
}
$ ld -r -o temp.o foo.c bar.c
ld: fatal: symbol `bar' is multiply-defined:
    (file foo.o and file bar.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. Undefined symbols can affect the link-edit process according to the type of symbol, together with the type of output file being generated.

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();

int main()
{
        return (foo());
}
$ cc -o prog main.c
Undefined           first referenced
 symbol                 in file
foo                     main.o
ld: fatal: symbol referencing errors
```

Similarly, if a shared object is used to create a dynamic executable and leaves an unresolved symbol definition, an undefined symbol error results.

```
$ cat foo.c
extern int bar;
int 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
```

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
```

`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 "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

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

Historically, weak symbols have been used to circumvent interposition, or test for optional functionality. However, experience has shown that weak symbols are fragile and unreliable in modern programming environments, and their use is discouraged.

Weak symbol aliases were frequently employed within system shared objects. The intent was to provide an alternative interface name, typically the symbol name with a prefixed "_" character. This alias name could be referenced from other system shared objects to avoid interposition issues due to an application exporting their own implementation of the symbol name. In practice, this technique proved to be overly complex and was used inconsistently. Modern versions of Oracle Solaris establish explicit bindings between system objects with direct bindings. See Chapter 6, "Direct Bindings".

Weak symbol references were often employed to test for the existence of an interface at runtime. This technique places restrictions on the build environment, the runtime environment, and can be circumvented by compiler optimizations. The use of `dlsym`(3C) with the `RTLD_DEFAULT`, or `RTLD_PROBE` handles, provides a consistent and robust means of testing for a symbol's existence. See "Testing for Functionality" on page 127.

Tentative Symbol Order Within the Output File

Contributions from input files usually appear in the output file in the order of their contribution. Tentative symbols are an exception to this rule, as these symbols are not fully defined until their resolution is complete. The order of tentative symbols within the output file might not follow the order of their contribution.

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 One_array[0x10];
char Two_array[0x20];
char Three_array[0x30];
$ cc -o libfoo.so -G -Kpic foo.c
$ elfdump -sN.dynsym libfoo.so | grep array | sort -k 2,2
    [11]   0x10614  0x20  OBJT GLOB  D    0 .bss          Two_array
     [3]   0x10634  0x30  OBJT GLOB  D    0 .bss          Three_array
     [4]   0x10664  0x10  OBJT GLOB  D    0 .bss          One_array
```

Sorting the symbols based on their address shows that their output order is different than the order they were defined in the source. In contrast, defining these symbols as initialized data items ensures that 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 libfoo.so -G -Kpic foo.c
$ elfdump -sN.dynsym libfoo.so | grep array | sort -k 2,2
     [4]   0x10614  0x10  OBJT GLOB  D    0 .data         One_array
    [11]   0x10624  0x20  OBJT GLOB  D    0 .data         Two_array
     [3]   0x10644  0x30  OBJT GLOB  D    0 .data         Three_array
```

Defining Additional Symbols

Besides the symbols provided from input files, you can supply additional global symbol references or global symbol 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 global symbol references and a variety of global symbol definitions. Attributes of the symbol such as visibility and type can be specified, See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217 for a complete description of the available options.

Defining Additional Symbols with the `-u` option

The `-u` option provides a mechanism for generating a global 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.

Defining Symbol References

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
#include    <stdio.h>

void foo()
{
        (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
#include    <stdio.h>

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

void main()
{
        foo();
        bar();
}
```

```
$ cc -c foo.c bar.c main.c
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                foo;
                bar;
        global:
                main;
};
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ elfdump -sN.symtab prog | egrep 'main$|foo$|bar$'
    [29]   0x10f30  0x24  FUNC LOCL  H    0 .text          bar
    [30]   0x10ef8  0x24  FUNC LOCL  H    0 .text          foo
    [55]   0x10f68  0x24  FUNC GLOB  D    0 .text          main
```

The significance of reducing symbol scope from global to local is covered in more detail in the section “Reducing Symbol Scope” on page 49.

Defining Absolute Symbols

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
#include    <stdio.h>

extern int foo();
extern int bar;

void main()
{
        (void) printf("&foo = 0x%p\n", &foo);
        (void) printf("&bar = 0x%p\n", &bar);
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION; VALUE=0x400 };
                bar     { TYPE=DATA;     VALUE=0x800 };
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 0x400
&bar = 0x800
$ elfdump -sN.symtab prog | egrep 'foo$|bar$'
    [45]     0x800     0  OBJT GLOB  D    0 ABS            bar
    [69]     0x400     0  FUNC GLOB  D    0 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. A simple `mapfile` definition that is associated with a `size`, but *no* `value` results in the creation of data storage. In this case, the symbol definition is accompanied with a section index. However, a `mapfile` definition that is accompanied with a `value` results in the creation of an absolute symbol. If a symbol is defined in a shared object, an absolute definition should be avoided. See "Augmenting a Symbol Definition" on page 49.

Defining Tentative Symbols

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
#include    <stdio.h>

extern int foo;

int bar[0x10];

void main()
{
        (void) printf("&foo = 0x%p\n", &foo);
        (void) printf("&bar = 0x%p\n", &bar);
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=COMMON; VALUE=0x4;   SIZE=0x200 };
                bar     { TYPE=COMMON; VALUE=0x102; SIZE=0x40 };
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol 'bar' has differing alignments:
    (file mapfile value=0x102; file main.o value=0x4);
    largest value applied
$ prog
&foo = 0x21264
&bar = 0x21224
```

```
$ elfdump -sN.symtab prog | egrep 'foo$|bar$'
    [45]   0x21224   0x40  OBJT GLOB  D    0 .bss          bar
    [69]   0x21264  0x200  OBJT GLOB  D    0 .bss          foo
```

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

Augmenting a Symbol Definition

The creation of an absolute data symbol within a shared object should be avoided. An external reference from a dynamic executable to a data item within a shared object typically requires the creation of a copy relocation. See "Copy Relocations" on page 188. To provide for this relocation, the data item should be associated with data storage. This association can be produced by defining the symbol within a relocatable object file. This association can also be produced by defining the symbol within a `mapfile` together with a `size` declaration and *no* `value` declaration. See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

A data symbol can be filtered. See "Shared Objects as Filters" on page 140. To provide this filtering, an object file definition can be augmented with a `mapfile` definition. The following example creates a filter containing a function and data definition.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION;       FILTER=filtee.so.1 };
                bar     { TYPE=DATA; SIZE=0x4; FILTER=filtee.so.1 };
        local:
                *;
};
$ cc -o filter.so.1 -G -Kpic -h filter.so.1 -M mapfile -R.
$ elfdump -sN.dynsym filter.so.1 | egrep 'foo|bar'
     [1]   0x105f8     0x4  OBJT GLOB  D    1 .data         bar
     [7]         0       0  FUNC GLOB  D    1 ABS           foo
$ elfdump -y filter.so.1 | egrep 'foo|bar'
     [1]  F        [0] filtee.so.1        bar
     [7]  F        [0] filtee.so.1        foo
```

At runtime, a reference from an external object to either of these symbols is resolved to the definition within the *filtee*.

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 which 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. A shared object created with these files, typically results in the creation of three symbols with 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 libfoo.so.1 -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [41]     0x560    0x18  FUNC GLOB  D    0 .text          bar
    [44]     0x520    0x2c  FUNC GLOB  D    0 .text          foo
    [45]   0x106b8     0x4  OBJT GLOB  D    0 .data          str
```

You can now use the functionality offered by `libfoo.so.1` 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 these scenarios 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
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                bar;
                str;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [24]     0x548    0x18  FUNC LOCL  H    0 .text          bar
    [25]   0x106a0     0x4  OBJT LOCL  H    0 .data          str
    [45]     0x508    0x2c  FUNC GLOB  D    0 .text          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 187 for details of symbolic relocation overhead.

As the number of symbols that are processed during a link-edit increases, defining 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
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [26]     0x570    0x18  FUNC LOCL  H    0 .text          bar
    [27]   0x106d8     0x4  OBJT LOCL  H    0 .data          str
    [50]     0x530    0x2c  FUNC GLOB  D    0 .text          foo
```

This example also defines a version name, `ISV_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 9, “Interfaces and Versioning”.

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
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
};
$ cc -o libfoo.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
```

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

```
$ cc -o libfoo.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
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ ld -o libfoo.o -M mapfile -r foo.o bar.o
$ elfdump -s libfoo.o | egrep 'foo$|bar$|str$'
    [29]      0x10    0x2c  FUNC GLOB  D    2 .text          foo
    [30]         0     0x4  OBJT GLOB  H    0 .data          str
```

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 processed from a `mapfile`.

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

```
$ ld -o libfoo.so.1 -G libfoo.o
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [24]     0x508    0x18  FUNC LOCL  H    0 .text          bar
```

```
    [25]   0x10644     0x4  OBJT LOCL  H    0 .data          str
    [42]     0x4c8    0x2c  FUNC GLOB  D    0 .text          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 libfoo.o -M mapfile -B reduce -r foo.o bar.o
$ elfdump -sN.symtab libfoo.o | egrep 'foo$|bar$|str$'
    [20]      0x50    0x18  FUNC LOCL  H    0 .text          bar
    [21]         0     0x4  OBJT LOCL  H    0 .data          str
    [30]      0x10    0x2c  FUNC GLOB  D    2 .text          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 `-z strip-class` option, or after a link-edit by using `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` keyword `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
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                str;
        eliminate:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
    [26]   0x10690     0x4  OBJT LOCL  H    0 .data          str
    [44]     0x4e8    0x2c  FUNC GLOB  D    0 .text          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 Chapter 6, “Direct Bindings”.

String Table Compression

String tables are compressed by the link-editor by removing duplicate entries, together with tail substrings. This compression can significantly reduce the size of any string tables. For example, 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 symbol 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 `mmapobj`(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 be stripped from the image by using the link-editor's -z strip-class 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 all read-only information, typically referred to as the text segment.

_edata
: The first location after initialized data.

_end
: The first location after all data.

_DYNAMIC
: The address of the .dynamic information section.

END
: The same as _end. The symbol has local scope and, together with the _START_ symbol, provides a simple 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 178.

_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 178.

START
: The first location within the text segment. The symbol has local scope and, together with the _END_ symbol, provides a simple 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.

Identifying Capability Requirements

Capabilities identify the attributes of a system that are required to allow code to execute. The following capabilities, in their order of precedence, are available.

- A *platform* capability - identifies a specific platform by name.
- A *machine* capability - identifies a specific machine hardware by name.
- *Hardware* capabilities - identify instruction set extensions and other hardware details with capabilities flags.
- *Software* capabilities - reflect attributes of the software environment with capabilities flags.

Each of these capabilities can be defined individually, or combined to produce a capabilities group.

Code that can only be executed when certain capabilities are available should identify these requirements by means of a capabilities section within the associated ELF object. Recording capability requirements within an object allows the system to validate the object before attempting to execute the associated code. These requirements can also provide a framework where the system can select the most appropriate object from a family of objects. A family consists of variants of the same object, where each variant requires different capabilities.

Dynamic objects, as well as individual functions or initialized data items within an object, can be associated with capability requirements. Ideally, capability requirements are recorded in the relocatable objects that are produced by the compiler, and reflect the options or optimization that was specified 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 “Capabilities Section” on page 342.

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 by using a `mapfile` can augment, or override, the capabilities that are specified within any input relocatable objects. `Mapfiles` are usually used to augment compilers that do not generate the necessary capability information.

System capabilities are the capabilities that describe a running system. The platform name, and machine hardware name can be displayed with `uname`(1) using the `-i` option and `-m` option respectively. The system hardware capabilities can be displayed with `isainfo`(1) using the `-v` option. At runtime, the capability requirements of an object are compared against the system capabilities to determine whether the object can be loaded, or a symbol within the object can be used.

Object capabilities are capabilities that are associated with an object. These capabilities define the requirements of the entire object, and control whether the object can be loaded at runtime. If an object requires capabilities that can not be satisfied by the system, then the object can not be loaded at runtime. Capabilities can be used to provide more than one instance of a given object, each optimized for systems that match the objects requirements. The runtime linker can

transparently select the best instance from such a family of object instances by comparing the objects capability requirements to the capabilities provided by the system.

Symbol capabilities are capabilities that are associated with individual functions, or initialized data items, within an object. These capabilities define the requirements of one or more symbols within an object, and control whether the symbol can be used at runtime. Symbol capabilities allow for the presence of multiple instances of a function within a single object. Each instance of the function can be optimized for a system with different capabilities. Symbol capabilities also allow for the presence of multiple instances of an initialized data item within an object. Each instance of the data can define system specific data. If a symbol instance requires capabilities that can not be satisfied by the system, then that symbol instance can not be used at runtime. Instead, an alternative instance of the same symbol name must be used. Symbol capabilities offer the ability to construct a single object that can be used on systems of varying abilities. A family of functions can provide optimized instances for systems that can support the capabilities, and more generic instances for other, less capable systems. A family of initialized data items can provide system specific data. The runtime linker transparently selects the best instance from such a family of symbol instances by comparing the symbols capability requirements to the capabilities provided by the system.

Object and symbol capabilities provide for selecting the best object, and the best symbol within an object, for the currently running system. Object and symbol capabilities are optional features, both independent of each other. However, an object that defines symbol capabilities may also define object capabilities. In this case, any family of capabilities symbols should be accompanied with one instance of the symbol that satisfies the object capabilities. If no object capabilities exist, any family of capability symbols should be accompanied with one instance of the symbol that requires no capabilities. This symbol instance provides the default implementation, should no capability instance be applicable for a given system.

The following x86 example displays the object capabilities of `foo.o`. These capabilities apply to the entire object. In this example, no symbol capabilities exist.

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1      0x840  [ SSE  MMX ]
```

The following x86 example displays the symbol capabilities of `bar.o`. These capabilities apply to the individual functions `foo` and `bar`. Two instances of each symbol exist, each instance being assigned to a different set of capabilities. In this example, no object capabilities exist.

```
$ elfdump -H bar.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_HW_1      0x40  [ MMX ]
```

```
 Symbols:
    index    value     size  type bind oth ver shndx    name
     [25]        0     0x21  FUNC LOCL  D    0 .text    foo%mmx
     [26]     0x24     0x1e  FUNC LOCL  D    0 .text    bar%mmx

Symbol Capabilities:
    index  tag               value
      [3]  CA_SUNW_HW_1      0x800  [ SSE ]

 Symbols:
    index    value     size  type bind oth ver shndx    name
     [33]     0x44     0x21  FUNC LOCL  D    0 .text    foo%sse
     [34]     0x68     0x1e  FUNC LOCL  D    0 .text    bar%sse
```

Note - In this example, the capability symbols follow a naming convention that appends a capability identifier to the generic symbol name. This convention can be produced by the link-editor when object capabilities are converted to symbol capabilities, and is discussed later in "Converting Object Capabilities to Symbol Capabilities" on page 71.

Capability definitions provide for many combinations that allow you to identify the requirements of an object, or of individual symbols within an object. Hardware capabilities provide the greatest flexibility. Hardware capabilities define hardware requirements without dictating a specific machine hardware name, or platform name. However, sometimes there are attributes of an underlying system that can only be determined from the machine hardware name, or platform name. Identifying a capability name can allow you to code to very specific system capabilities, but the use of the identified object can be restrictive. Should a new machine hardware name or platform name become applicable for the object, the object must be rebuilt to identify the new capability name.

The following sections describe how capabilities can be defined, and used by the link-editor.

Identifying a Platform Capability

A platform capability of an object identifies the platform name of the systems that the object, or specific symbols within the object, can execute upon. Multiple platform capabilities can be defined. This identification is very specific, and takes precedence over any other capability types.

The platform name of a system can be displayed by the utility `uname`(1) with the `-i` option.

A platform capability requirement can be defined using the following `mapfile` syntax.

```
$mapfile_version 2
    CAPABILITY {
            PLATFORM  = platform_name...;
            PLATFORM += platform_name...;
            PLATFORM -= platform_name...;
    };
```

The PLATFORM attribute is qualified with one or more platform names. The "+=" form of assignment augments the platform capabilities specified by the input objects, while the "=" form overrides them. The "-=" form of assignment is used to exclude platform capabilities from the output object. The following SPARC example identifies the object foo.so.1 as being specific to the SUNW,SPARC-Enterprise platform.

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        PLATFORM = 'SUNW,SPARC-Enterprise';
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_PLAT      SUNW,SPARC-Enterprise
```

Relocatable objects can define platform capabilities. These capabilities are gathered together to define the final capability requirements of the object being built.

The platform capability of an object can be controlled explicitly from a mapfile by using the "=" form of assignment to override any platform capabilities that might be provided from any input relocatable objects. An empty PLATFORM attribute used with the "=" form of assignment effectively removes any platform capabilities requirement from the object being built.

A platform capability requirement defined in a dynamic object is validated by the runtime linker against the platform name of the system. The object is only used if one of the platform names recorded in the object match the platform name of the system.

Targeting code to a specific platform can be useful in some instances, however the development of a hardware capabilities family can provide greater flexibility, and is recommended. Hardware capabilities families can provide for optimized code to be exercised on a broader range of systems.

Identifying a Machine Capability

A machine capability of an object identifies the machine hardware name of the systems that the object, or specific symbols within the object, can execute upon. Multiple machine capabilities can be defined. This identification carries less precedence than platform capability definitions, but takes precedence over any other capability types.

The machine hardware name of a system can be displayed by the utility uname(1) with the -m option.

A machine capability requirement can be defined using the following mapfile syntax.

```
$mapfile_version 2
    CAPABILITY {
            MACHINE  = machine_name...;
            MACHINE += machine_name...;
            MACHINE -= machine_name...;
    };
```

The `MACHINE` attribute is qualified with one or more machine hardware names. The "`+=`" form of assignment augments the machine capabilities specified by the input objects, while the "`=`" form overrides them. The "`-=`" form of assignment is used to exclude machine capabilities from the output object. The following SPARC example identifies the object `foo.so.1` as being specific to the `sun4u` machine hardware name.

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        MACHINE = sun4u;
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag              value
       [0]  CA_SUNW_MACH      sun4u
```

Relocatable objects can define machine capabilities. These capabilities are gathered together to define the final capability requirements of the object being built.

The machine capability of an object can be controlled explicitly from a `mapfile` by using the "`=`" form of assignment to override any machine capabilities that might be provided from any input relocatable objects. An empty `MACHINE` attribute used with the "`=`" form of assignment effectively removes any machine capabilities requirement from the object being built.

A machine capability requirement defined in a dynamic object is validated by the runtime linker against the machine hardware name of the system. The object is only used if one of the machine names recorded in the object match the machine name of the system.

Targeting code to a specific machine can be useful in some instances, however the development of a hardware capabilities family can provide greater flexibility, and is recommended. Hardware capabilities families can provide for optimized code to be exercised on a broader range of systems.

Identifying Hardware Capabilities

The hardware capabilities of an object identify the hardware requirements of a system necessary for the object, or specific symbol, 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.

```
$mapfile_version 2
    CAPABILITY {
            HW  = hwcap_flag...;
            HW += hwcap_flag...;
            HW -= hwcap_flag...;
    };
```

The `HW` attribute to the `CAPABILITY` directive is qualified with one or more tokens, which are symbolic representations of hardware capabilities. The "`+=`" form of assignment augments the hardware capabilities specified by the input objects, while the "`=`" form overrides them. The "`-=`" form of assignment is used to exclude hardware capabilities from the output object.

For SPARC systems, hardware capabilities are defined as `AV_` values in `sys/auxv_SPARC.h`. For x86 systems, 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
$mapfile_version 2
CAPABILITY {
        HW += SSE MMX;
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     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 an object can be controlled explicitly from a `mapfile` by using the "`=`" form of assignment to override any hardware capabilities that might be provided from any input relocatable objects. An empty `HW` attribute used with the "`=`" form of assignment effectively removes any hardware capabilities requirement from the object being built.

The following example suppresses any hardware capabilities data defined by the input relocatable object `foo.o` from being included in the output file, `bar.o`.

```
$ elfdump -H foo.o
```

```
Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag              value
       [0]  CA_SUNW_HW_1      0x840  [ SSE  MMX ]
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        HW = ;
};
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

Any hardware capability requirements defined by a dynamic object are validated by the runtime linker against the hardware capabilities that are provided by the system. 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 ]
        ....
```

Multiple variants of a dynamic object that exploit different hardware capabilities can provide a flexible runtime environment using filters. See “Capability Specific Shared Objects” on page 253.

Hardware capabilities can also be used to identify the capabilities of individual functions within a single object. In this case, the runtime linker can select the most appropriate function instance to use based upon the current system capabilities. See “Creating a Family of Symbol Capabilities Functions” on page 65.

Identifying Software Capabilities

The software capabilities of an object identify characteristics of the software that might be important for debugging or monitoring processes. Software capabilities can also influence process execution. Presently, the only software capabilities that are recognized relate to frame pointer usage by the object, and process address space restrictions.

Objects can indicate that their frame pointer use is known. This state is then qualified by declaring the frame pointer as being used or not.

64–bit objects can indicate that at runtime they must be exercised within a 32–bit address space.

Software capabilities flags are defined in sys/elf.h.

```
#define  SF1_SUNW_FPKNWN    0x001
#define  SF1_SUNW_FPUSED    0x002
#define  SF1_SUNW_ADDR32    0x004
```

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

```
        $mapfile_version 2
CAPABILITY {
        SF  = sfcap_flags...;
        SF += sfcap_flags...;
        SF -= sfcap_flags...;
};
```

The `SF` attribute to the `CAPABILITY` directive can be assigned any of the tokens `FPKNWN`, `FPUSED` and `ADDR32`.

Relocatable objects can contain software capabilities values. The link-editor combines the software capabilities values from multiple input relocatable objects. Software capabilities can also be supplied with a `mapfile`. By default, any `mapfile` values are combined with the values supplied by relocatable objects.

The software capability requirements of an object can be controlled explicitly from a `mapfile` by using the "=" form of assignment to override any software capabilities that might be provided from any input relocatable objects. An empty `SF` attribute used with the "=" form of assignment effectively removes any software capabilities requirement from the object being built.

The following example suppresses any software capabilities data defined by the input relocatable object `foo.o` from being included in the output file, `bar.o`.

```
$ elfdump -H foo.o

Object Capabilities:
    index  tag               value
      [0]  CA_SUNW_SF_1      0x3  [ SF1_SUNW_FPKNWN  SF1_SUNW_FPUSED ]
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        SF = ;
};
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

Software Capability Frame Pointer Processing

The computation of a `CA_SUNW_SF_1` value from two frame pointer input values is as follows.

TABLE 2-1 `CA_SUNW_SF_1` Frame Pointer Flag Combination State Table

Input file 1	Input file 2	
SF1_SUNW_FPKNWN SF1_ SUNW_FPUSED	**SF1_SUNW_FPKNWN**	**<unknown>**

Input file 1	Input file 2		
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 frame pointer 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.

Software Capability Address Space Restriction Processing

64–bit objects that are identified with the `SF1_SUNW_ADDR32` software capabilities flag can contain optimized code that requires a 32–bit address space. 64–bit objects that are identified in this manner can interoperate with any other 64–bit objects whether they are identified with the `SF1_SUNW_ADDR32` flag or not. An occurrence of the `SF1_SUNW_ADDR32` flag within a 64–bit input relocatable object is propagated to the `CA_SUNW_SF_1` value that is created for the output file being created by the link-editor.

The existence of the `SF1_SUNW_ADDR32` flag within a 64–bit executable ensures that the associated process is restricted to the lower 32–bit address space. This restricted address space includes the process stack and all process dependencies. Within such a process, all objects, whether they are identified with the `SF1_SUNW_ADDR32` flag or not, are loaded within the restricted 32–bit address space.

64–bit shared objects can contain the `SF1_SUNW_ADDR32` flag. However, the restricted address space requirement can only be established by a 64–bit executable containing the `SF1_SUNW_ADDR32` flag. Therefore, a 64–bit `SF1_SUNW_ADDR32` shared object must be a dependency of a 64–bit `SF1_SUNW_ADDR32` executable.

A 64–bit `SF1_SUNW_ADDR32` shared object that is encountered by the link-editor when building an unrestricted 64–bit executable results in a warning.

```
$ cc -m64 -o main main.c -lfoo
ld: warning: file libfoo.so: section .SUNW_cap: software capability ADDR32: \
    requires executable be built with ADDR32 capability
```

A 64–bit `SF1_SUNW_ADDR32` shared object that is encountered at runtime by a process that is created from an unrestricted 64–bit executable, results in a fatal error.

```
$ ldd main
    libfoo.so =>     ./libfoo.so  - software capability unsupported: 0x4  [ ADDR32 ]
    ....
```

```
$ main
ld.so.1: main: fatal: ./libfoo.so: software capability unsupported: 0x4  [ ADDR32 ]
```

An executable can be seeded with the `SF1_SUNW_ADDR32` using a `mapfile`.

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        SF += ADDR32;
};
$ cc -m64 -o main main.c -Mmapfile -lfoo
$ elfdump -H main

Object Capabilities:
    index  tag               value
      [0]  CA_SUNW_SF_1      0x4  [ SF1_SUNW_ADDR32 ]
```

Creating a Family of Symbol Capabilities Functions

Developers often desire to provide multiple instances of functions, each optimized for a particular set of capabilities, within a single object. It is desirable for the selection and use of these instances to be transparent to any consumers. A generic, front-end function can be created to provide an external interface. This generic instance, together with the optimized instances, can be combined into one object. The generic instance might use `getisax`(2) to determine the systems capabilities and then call the appropriate optimized function instance to handle a task. Although this model works, it suffers from a lack of generality, and incurs a runtime overhead.

Symbol capabilities offer an alternative mechanism to construct such an object. This mechanism is simpler, more efficient, and does not require you to write additional front-end code. Multiple instances of a function can be created and associated with different capabilities. These instances, together with a default instance of the function that is suitable for any system, can be combined into a single dynamic object. The selection of the most appropriate member from this family of symbols is carried out by the runtime linker using the symbol capabilities information.

In the following example, the x86 objects `foobar.mmx.o` and `foobar.sse.o`, contain the same function `foo` and `bar`, that have been compiled to use the `MMX` and `SSE` instructions respectively.

```
$ elfdump -H foobar.mmx.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
    index  tag               value
      [1]  CA_SUNW_ID        mmx
      [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size  type bind oth ver shndx    name
      [10]        0      0x21  FUNC LOCL  D    0 .text    foo%mmx
      [16]     0x24      0x1e  FUNC LOCL  D    0 .text    bar%mmx

$ elfdump -H foobar.sse.o
```

```
Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
    index  tag              value
      [1]  CA_SUNW_ID        sse
      [2]  CA_SUNW_HW_1      0x800  [ SSE ]

  Capabilities symbols:
     index    value      size  type bind oth ver shndx    name
      [16]        0      0x2f  FUNC LOCL  D    0 .text    foo%sse
      [18]     0x48      0x30  FUNC LOCL  D    0 .text    bar%sse
```

Each of these objects contain a local symbol identifying the capabilities function `foo%*` and `bar%*`. In addition, each object also defines a global reference to the function `foo` and `bar`. Any internal references to `foo` or `bar` are relocated through these global references, as are any external references.

These two objects can now be combined with a default instance of `foo` and `bar`. These default instances satisfy the global references, and provide an implementation that is compatible with any object capabilities. These default instances are said to lead each capabilities family. If no object capabilities exist, this default instance should also require no capabilities. Effectively, three instances of `foo` and `bar` exist, the global instance provides the default, and the local instances provide implementations that are used at runtime if the associated capabilities are available.

```
$ cc -o libfoobar.so.1 -G foobar.o foobar.sse.o foobar.mmx.o
$ elfdump -sN.dynsym libfoobar.so.1 | egrep "foo|bar"
     [2]     0x700    0x21  FUNC LOCL  D    0 .text    foo%mmx
     [4]     0x750    0x2f  FUNC LOCL  D    0 .text    foo%sse
     [8]     0x784    0x1e  FUNC LOCL  D    0 .text    bar%mmx
     [9]     0x7b0    0x30  FUNC LOCL  D    0 .text    bar%sse
    [15]     0x7a0    0x14  FUNC GLOB  D    1 .text    foo
    [17]     0x7c0    0x14  FUNC GLOB  D    1 .text    bar
```

The capabilities information for a dynamic object displays the capabilities symbols, and reveals the capabilities families that are available.

```
$ elfdump -H libfoobar.so.1

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag              value
     [1]  CA_SUNW_ID        mmx
     [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
   index    value     size  type bind oth ver shndx    name
     [2]    0x700     0x21  FUNC LOCL  D    0 .text    foo%mmx
     [8]    0x784     0x1e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
   index  tag              value
     [4]  CA_SUNW_ID        sse
```

```
     [5]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
   index    value     size  type bind oth ver shndx    name
     [4]    0x750     0x2f  FUNC LOCL  D    0 .text    foo%sse
     [9]    0x7b0     0x30  FUNC LOCL  D    0 .text    bar%sse

Capabilities Chain Section:  .SUNW_capchain

 Capabilities family: foo
  chainndx  symndx      name
         1  [15]        foo
         2  [2]         foo%mmx
         3  [4]         foo%sse

 Capabilities family: bar
  chainndx  symndx      name
         5  [17]        bar
         6  [8]         bar%mmx
         7  [9]         bar%sse
```

At runtime, all references to `foo` and `bar` are initially bound to the global symbols. However, the runtime linker recognizes that these functions are the lead instance of a capabilities family. The runtime linker inspects each family member to determine if a better capability function is available. There is a one time cost to this operation, which occurs on the first call to the function. Subsequent calls to `foo` and `bar` are bound directly to the function instance selected by the first call. This function selection can be observed by using the runtime linkers debugging capabilities.

In the following example, the underlying system does not provide `MMX` or `SSE` support. The lead instance of `foo` requires no special capabilities support, and thus satisfies any relocation reference.

```
$ LD_DEBUG=symbols  main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo%mmx[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo%mmx[2]:  capability rejected
debug:   symbol=foo%sse[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo%sse[4]:  capability rejected
debug:   symbol=foo[15]:  used
```

In the following example, `MMX` is available, but `SSE` is not. The `MMX` capable instance of `foo` satisfies any relocation reference.

```
$ LD_DEBUG=symbols  main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo%mmx[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo%mmx[2]:  capability candidate
debug:   symbol=foo%sse[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo%sse[4]:  capability rejected
debug:   symbol=foo[2]:  used
```

When more than one capability instance can be exercised on the same system, a set of precedent rules are used to select one instance.

- A capability group that defines a platform name takes precedent over a group that does not define a platform name.
- A capability group that defines a machine hardware name takes precedent over a group that does not define a machine hardware name.
- A larger hardware capabilities value takes precedent over a smaller hardware capabilities value.

A family of capabilities function instances must be accessed from a procedure linkage table entry. See “Procedure Linkage Table (Processor-Specific)” on page 416. This procedure linkage reference requires the runtime linker to resolve the function. During this process, the runtime linker can process the associated symbol capabilities information, and select the best function from the available family of function instances.

When symbol capabilities are not used, there are cases where the link-editor can resolve references to code without the need of a procedure linkage table entry. For example, within a dynamic executable, a reference to a function that exists within the executable can be bound internally at link-edit time. Hidden and protected functions within shared objects can also be bound internally at link-edit time. In these cases, there is normally no need for the runtime linker to be involved in resolving a reference to these functions.

However, when symbol capabilities are used, the function must be resolved from a procedure linkage table entry. This entry is necessary in order for the runtime linker to be involved in selecting the appropriate function, while maintaining a read-only text segment. This mechanism results in an indirection through a procedure linkage table entry for all calls to a capability function. This indirection might not be necessary if symbol capabilities are not used. Therefore, there is a small trade off between the cost of calling the capability function, and any performance improvement gained from using the capability function over its default counterpart.

Note - Although a capability function must be accessed through a procedure linkage table entry, the function can still be defined as hidden or protected. The runtime linker honors these visibility states and restricts any binding to these functions. This behavior results in the same bindings as are produced when symbol capabilities are not associated with the function. A hidden function can not be bound to from an external object. A reference to a protected function from within an object will only be bound to within the same object.

Creating a Family of Symbol Capabilities Data Items

Multiple instances of initialized data, where each instance is specific to a system, can be provided within the same object. However, providing such data through functional interfaces is often simpler, and is recommended. See “Creating a Family of Symbol Capabilities

Functions" on page 65. Special care is required to provide multiple instances of initialized data within an executable.

The following example initializes a data item `foo` within `foo.c`, to point to a machine name string. This file can be compiled for various machines, and each instance is identified with a machine capability. A reference to this data item is made from `bar` from the file `bar.c`. A shared object `foobar.so.1` is then created by combining `bar` with two capabilities instances of `foo`.

```
$ cat foo.c
char *foo = MACHINE;
$ cat bar.c
#include <stdio.h>

extern char *foo = MACHINE;

void bar()
{
        (void) printf("machine: %s\n", foo);
}

$ elfdump -H foobar.so.1

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag               value
     [1]  CA_SUNW_ID        sun4u
     [2]  CA_SUNW_MACH      sun4u

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [1]  0x108d4      0x4  OBJT LOCL  D    0 .data          foo%sun4u

 Symbol Capabilities:
   index  tag               value
     [4]  CA_SUNW_ID        sun4v
     [5]  CA_SUNW_MACH      sun4v

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [2]  0x108d8      0x4  OBJT LOCL  D    0 .data          foo%sun4v
```

An application can reference `bar`, and the runtime linker binds to the instance of `foo` that is associated with the underlying system.

```
$ uname -m
sun4u
$ main
machine: sun4u
```

The proper operation of this code depends on the code having been compiled to be *position-independent*, as is normally the case for code in sharable objects. See "Position-Independent Code" on page 178. Position-independent data references are indirect

references, which allow the runtime linker to locate the required reference and update elements of the data segment. This relocation update of the data segment preserves the text segment as read-only.

However, the code within an executable is typically *position-dependent*. In addition, data references within an executable are bound at link-edit time. Within an executable, a symbol capabilities data reference must remain unresolved through a global data item, so that the runtime linker can select from the symbol capabilities family. If the reference from `bar` in the previous example `bar.c` is compiled as position-dependent code, then the text segment of the executable must be relocated at runtime. By default, this condition results in a fatal link-time error.

```
$ cc -o main main.c bar.c foo.o foo.1.o foo.2.o ...
warning: Text relocation remains               referenced
    against symbol                 offset      in file
foo                                0x0         bar.o
foo                                0x8         bar.o
```

One approach to solve this error condition is to compile `bar.c` as position-independent. Note however, that all references to any symbol capabilities data items from within the executable must be compiled position-independent for this technique to work.

Although data can be accessed using the symbol capabilities mechanism, making data items a part of the public interface to an object can be problematic. An alternative, and more flexible model, is to encapsulate each data item within a symbol capabilities function. This function provides the sole means of access to the data. Hiding data behind a symbol capabilities function has the important benefit of allowing the data to be defined static and kept private. The previous example can be coded to use symbol capabilities functions.

```
$ cat foobar.c
cat bar.c
#include <stdio.h>

static char *foo = MACHINE;

void bar()
{
        (void) printf("machine: %s\n", foo);
}
$ elfdump -H main

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag               value
     [1]  CA_SUNW_ID        sun4u
     [2]  CA_SUNW_MACH      sun4u

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [1]  0x1111c     0x1c  FUNC LOCL  D    0 .text          bar%sun4u

 Symbol Capabilities:
```

```
   index  tag               value
     [4]  CA_SUNW_ID        sun4v
     [5]  CA_SUNW_MACH      sun4v

  Symbols:
   index    value     size  type bind oth ver shndx          name
     [2]  0x11138     0x1c  FUNC LOCL  D    0 .text          bar%sun4v

$ uname -m
sun4u
$ main
machine: sun4u
```

Converting Object Capabilities to Symbol Capabilities

Ideally, the compiler can generate objects that are identified with symbol capabilities. If the compiler can not create symbol capabilities, the link-editor offers a solution.

A relocatable object that defines object capabilities can be transformed into a relocatable object that defines symbol capabilities using the link-editor. Using the link-editor `-z symbolcap` option, any capability data section is converted to define symbol capabilities. All global functions within the object are converted into local functions, and are associated with symbol capabilities. All global initialized data items are converted to local data items, and are associated with symbol capabilities. These transformed symbols are appended with any capability identifier specified as part of the object capabilities group. If a capability identifier is not defined, a default group name is appended.

For each original global function or initialized data item, a global reference is created. This reference is associated to any relocation requirements, and provides for binding to a default, global symbol when this object is finally combined to create a dynamic object.

Note - The `-z symbolcap` option applies to objects that contain an object capabilities section. The option has no affect upon relocatable objects that already contain symbol capabilities, or relocatable objects that contain both object and symbol capabilities. This design allows multiple objects to be combined by the link-editor, with only those objects that contain object capabilities being affected by the option.

In the following example, a x86 relocatable object contains two global functions `foo` and `bar`. This object has been compiled to require the `MMX` and `SSE` hardware capabilities. In these examples, the capabilities group has been named with a capabilities identifier entry. This identifier name is appended to the transformed symbol names. Without this explicit identifier, the link-editor appends a default capabilities group name.

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
```

```
   index  tag              value
     [0]  CA_SUNW_ID       sse,mmx
     [1]  CA_SUNW_HW_1     0x840  [ SSE MMX ]

$ elfdump -s foo.o | egrep "foo|bar"
    [25]         0   0x21  FUNC GLOB  D    0 .text    foo
    [26]      0x24   0x1e  FUNC GLOB  D    0 .text    bar

$ elfdump -r foo.o | fgrep foo
  R_386_PLT32               0x38       .rel.text            foo
```

This relocatable object can now be transformed into a symbols capabilities relocatable object.

```
$ ld -r -o foo.1.o -z symbolcap foo.o
$ elfdump -H foo.1.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
   index  tag              value
     [1]  CA_SUNW_ID       sse,mmx
     [2]  CA_SUNW_HW_1     0x840  [ SSE MMX ]

  Symbols:
   index    value     size  type bind oth ver shndx    name
    [25]        0     0x21  FUNC LOCL  D    0 .text    foo%sse,mmx
    [26]     0x24     0x1e  FUNC LOCL  D    0 .text    bar%sse,mmx

$ elfdump -s foo.1.o | egrep "foo|bar"
    [25]        0     0x21  FUNC LOCL  D    0 .text    foo%sse,mmx
    [26]     0x24     0x1e  FUNC LOCL  D    0 .text    bar%sse,mmx
    [37]        0        0  FUNC GLOB  D    0 UNDEF    foo
    [38]        0        0  FUNC GLOB  D    0 UNDEF    bar

$ elfdump -r foo.1.o | fgrep foo
  R_386_PLT32               0x38       .rel.text            foo
```

This object can now be combined with other objects containing instances of the same functions, associated with different symbol capabilities, to produce an executable or shared object. In addition, a default instance of each function, one that is not associated with any symbol capabilities, should be provided to lead each capabilities family. This default instance provides for all external references, and ensures that an instance of the function is available on any system.

At runtime, any references to `foo` and `bar` are directed to the lead instances. However, the runtime linker selects the best symbol capabilities instance if the system accommodates the appropriate capabilities.

Archive Considerations

Archive libraries usually contain a collection of relocatable objects. The link-editor can extract individual relocatable objects to resolve unresolved symbol references. See “Archive Processing” on page 28.

If a family of capabilities relocatable objects were added to an archive, any reference to the lead capability symbol only extracts the generic relocatable object that defines that symbol. No other capabilities objects are extracted.

If capabilities objects are required to be deployed using an archive library, a single capability family relocatable object should be created. Combine any capabilities objects, and any generic object containing the capabilities lead symbol, into one relocatable object. Add this single object, containing the entire capabilities family collection, to the archive.

```
$ ld -r -o all.foo.o foo.o foo.1.o foo.2.o ....
$ ar -cr libfoo.o all.foo.o
```

Exercising a Capability Family

Objects are normally designed and built so that they can execute on all systems of a given architecture. However, individual systems, with special capabilities, are often targeted for optimization. Optimized code can be identified with the capabilities that the code requires to execute, using the mechanisms described in the previous sections.

To exercise and test optimized instances it is necessary to use a system that provides the required capabilities. For each system, the runtime linker determines the capabilities that are available, and then chooses the most capable instances. To aid testing and experimentation, the runtime linker can be told to use an alternative set of capabilities than those provided by the system. In addition, you can specify that only specific files should be validated against these alternative capabilities.

An alternative set of capabilities is derived from the system capabilities, and can be re-initialized or have capabilities added or removed.

A family of environment variables is available to create and target the use of an alternative set of capabilities.

`LD_PLATCAP={`*name*`}`

Identifies an alternative platform name.

`LD_MACHCAP={`*name*`}`

Identifies an alternative machine hardware name.

`LD_HWCAP=[+-]{`*token* `| [`*index*`]`*number*`},...`

Identifies an alternative hardware capabilities value.

`LD_SFCAP=[+-]{`*token* `| [`*index*`]`*number*`},...`

Identifies an alternative software capabilities value.

`LD_CAP_FILES=`*file*`,...`

Identifies the files that should be validated against the alternative capabilities.

The capabilities environment variables `LD_PLATCAP` and `LD_MACHCAP` accept a string that defines the platform name and machine hardware names respectively. See "Identifying a Platform Capability" on page 58, and "Identifying a Machine Capability" on page 59.

The capabilities environment variables `LD_HWCAP` and `LD_SFCAP` accept a comma separated list of *tokens* as a symbolic representation of capabilities. See "Identifying Hardware Capabilities" on page 60, and "Identifying Software Capabilities" on page 62. A token can also be a numeric value. To provide for setting numeric values for different masks, such as `CA_SUNW_HW_1` and `CA_SUNW_HW_2`, the number can be prefixed with a bracketed index. For example, `LD_HWCAP=[2]0x80` sets `CA_SUNW_HW_2` to the value *0x80*. If no index is specified, 1 is assumed. Invalid indexes are ignored.

A "`+`" prefix results in the capabilities that follow being added to the alternative capabilities. A "`-`" prefix results in the capabilities that follow being removed from the alternative capabilities. The lack of "`+-`" result in the capabilities that follow replacing the alternative capabilities.

The removal of a capability results in a more restricted capabilities environment being emulated. Normally, when a family of capabilities instances is available, a generic, non-capabilities specific instance is also provided. A more restricted capabilities environment can therefore be used to force the use of less capable, or generic code instances.

The addition of a capability results in a more enhanced capabilities environment being emulated. This environment should be created with caution, but can be used to exercise the framework of a capabilities family. For example, a family of functions can be created that define their expected capabilities using *mapfiles*. These functions can use `printf`(3C) to confirm their execution. The creation of the associated objects can then be validated and exercised with various capability combinations. This prototyping of a capabilities family can prove useful before the real capabilities requirements of the functions are coded. However, if the code within a family instance requires a specific capability to execute correctly, and this capability is not provided by the system, but is set as an alternative capability, the code instance will fail to execute correctly.

Establishing a set of alternative capabilities without also using `LD_CAP_FILES` results in all of the capabilities specific objects of a process being validated against the alternative capabilities. This approach should also be exercised with caution, as many system objects require system capabilities to execute correctly. Any alteration of capabilities can cause system objects to fail to execute correctly.

A best environment for capabilities experimentation is to use a system that provides all the capabilities your objects are targeted to use. `LD_CAP_FILES` should also be used to isolate the objects you wish to experiment with. Capabilities can then be disabled, using the "`-`" syntax, so that the various instances of your capabilities family can be exercised. Each instance is fully supported by the true capabilities of the system.

For example, suppose you have two x86 capabilities objects, `libfoo.so` and `libbar.so`. These objects contain capability functions optimized to use `SSE2` instructions, functions optimized to

use MMX instructions, and generic functions that require no capabilities. The underlying system provides both SSE2 and MMX. By default, the fully optimized SSE2 functions are used.

libfoo.so and libbar.so can be restricted to use the functions optimized for MMX instructions by removing the SSE2 capability by using a LD_HWCAP definition. The most flexible means of defining LD_CAP_FILES is to use the base name of the required files.

```
$ LD_HWCAP=-sse2 LD_CAP_FILES=libfoo.so,libbar.so  ./main
```

libfoo.so and libbar.so can be further restricted to use only generic functions by removing the SSE2 and MMX capabilities.

```
$ LD_HWCAP=-sse2,mmx LD_CAP_FILES=libfoo.so,libbar.so  ./main
```

Note - The capabilities available for an application, and any alternative capabilities that have been set, can be observed using the runtime linkers diagnostics.

```
$ LD_DEBUG=basic LD_HWCAP=-sse2,mmx,cx8  ./main
....
02328: hardware capabilities (CA_SUNW_HW_1) - 0x5c6f  \
    [ SSE3 SSE2 SSE FXSR MMX CMOV SEP CX8 TSC FPU ]
02328: alternative hardware capabilities (CA_SUNW_HW_1) - 0x4c2b  \
    [ SSE3 SSE FXSR CMOV SEP TSC FPU ]
....
```

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 178. 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 188.

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 the data, or an external relocation is a displacement into the copied data, the displacement relocation becomes invalidated.

Two areas of validation attempt to catch displacement relocation problems.

- The first occurs when generating a shared object. Any potential copy relocatable data items that can be problematic if the copied data is involved in a displacement relocation are flagged. 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.
- The second occurs when generating an executable. The creation of a copy relocation whose data is known to be involved in a displacement relocation is flagged.

  However, displacement relocations applied to a shared object might be completed during the shared objects creation at link-edit time. These displacement relocations might not have been flagged. The link-edit of an executable that references an unflagged 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 13-10. 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.

```
$ 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 49. Relocation problems of this type can be avoided by accessing data within shared objects by using functional interfaces.

Stub Objects

A stub object is a shared object, built entirely from `mapfiles`, that supplies the same linking interface as the real object, while containing no code or data. Stub objects cannot be used at runtime. However, an application can be built against a stub object, where the stub object provides the real object name to be used at runtime.

When building a stub object, the link-editor ignores any object or library files specified on the command line, and these files need not exist in order to build a stub. Since the compilation step can be omitted, and because the link-editor has relatively little work to do, stub objects can be built very quickly.

Stub objects can be used to solve a variety of build problems.

- Speed

  Modern machines, using a version of the `make` utility with the ability to parallelize operations, are capable of compiling and linking many objects simultaneously, and doing so offers significant speedups. However, it is typical that a given object will depend on other objects, and that there will be a core set of objects that nearly everything else depends on. It is necessary to order the builds so that all objects are built ahead of their use by other objects. This ordering creates bottlenecks that reduce the amount of parallelization that is possible and limits the overall speed at which the code can be built.
- Complexity/Correctness

  In a large body of code, there can be a large number of dependencies between the various objects. The `makefiles` or other build descriptions for these objects can become very complex and difficult to understand or maintain. The dependencies can change as the system evolves. This can cause a given set of `makefiles` to become slightly incorrect over time, leading to race conditions and mysterious rare build failures.
- Dependency Cycles

  It might be desirable to organize code as cooperating shared objects, each of which draw on the resources provided by the other. Such cycles cannot be supported in an environment where objects must be built before the objects that use them, even though the runtime linker is fully capable of loading and using such objects if they could be built.

Stub shared objects offer an alternative method for building code that sidesteps the above issues. Stub objects can be quickly built for all the shared objects produced by the build. Then, all the real shared objects and executables can be built in parallel, in any order, using the stub

objects to stand in for the real objects at link-time. Afterwards, the executables and real shared objects are kept, and the stub shared objects are discarded.

Stub objects are built from one or more `mapfiles`, which must collectively satisfy the following requirements.

- At least one `mapfile` must specify the `STUB_OBJECT` directive. See "STUB_OBJECT Directive" on page 216.
- All function and data symbols that make up the external interface to the object must be explicitly listed in the `mapfile`.
- The `mapfile` must use symbol scope reduction ('*'), to remove any symbols not explicitly listed from the external interface. See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.
- All global data exported from the object must have an `ASSERT` symbol attribute in the `mapfile` to specify the symbol type and size. In the case where there are multiple symbols that reference the same data, the `ASSERT` for one of these symbols must specify the `TYPE` and `SIZE` attributes, while the others must use the `ALIAS` attribute to reference this primary symbol. See "ASSERT Attribute" on page 219.

Given such a `mapfile`, the stub and real versions of the shared object can be built using the same command line for each. The `-z stub` option is added to the link-edit of the stub object, and is omitted from the link-edit of the real object.

To demonstrate these ideas, the following code implements a shared object named `idx5`, which exports data from a 5 element array of integers. Each element is initialized to contain its zero-based array index. This data is made available as a global array, as an alternative alias data symbol with weak binding, and through a functional interface.

```
$ cat idx5.c
int _idx5[5] = { 0, 1, 2, 3, 4 };
#pragma weak idx5 = _idx5

int
idx5_func(int index)
{
        if ((index < 0) || (index > 4))
                return (-1);
        return (_idx5[index]);
}
```

A `mapfile` is required to describe the interface provided by this shared object.

```
$ cat mapfile
$mapfile_version 2
STUB_OBJECT;
SYMBOL_SCOPE {
        _idx5 {
                ASSERT { TYPE=data; SIZE=4[5] };
        };
        idx5 {
```

```
                ASSERT { BINDING=weak; ALIAS=_idx5 };
        };
        idx5_func;
    local:
        *;
};
```

The following main program is used to print all the index values available from the `idx5` shared object.

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

extern int _idx5[5], idx5[5], idx5_func(int);

int
main(int argc, char **argv)
{
        int i;
        for (i = 0; i < 5; i++)
                (void) printf("[%d] %d %d %d\n",
                    i, _idx5[i], idx5[i], idx5_func(i));
        return (0);
}
```

The following commands create a stub version of this shared object in a subdirectory named `stublib`. The `elfdump` command is used to verify that the resulting object is a stub. The command used to build the stub differs from that of the real object only in the addition of the `-z stub` option, and the use of a different output file name. This demonstrates the ease with which stub generation can be added to existing code.

```
$ cc -Kpic -G -M mapfile -h libidx5.so.1 idx5.c -o stublib/libidx5.so.1 -zstub
$ ln -s libidx5.so.1 stublib/libidx5.so
$ elfdump -d stublib/libidx5.so | grep STUB
    [11]  FLAGS_1           0x4000000           [ STUB ]
```

The main program can now be built, using the stub object to stand in for the real shared object, and setting a *runpath* that will find the real object at runtime. However, as the real object has not been built, this program cannot yet be run. Attempts to cause the system to load the stub object are rejected, as the runtime linker knows that stub objects lack the actual code and data found in the real object, and cannot execute.

```
$ cc main.c -L stublib -R '$ORIGIN/lib' -lidx5 -lc
$ ./a.out
ld.so.1: a.out: fatal: libidx5.so.1: open failed: No such file or directory
Killed
$ LD_PRELOAD=stublib/libidx5.so.1  ./a.out
ld.so.1: a.out: fatal: stublib/libidx5.so.1: stub shared object \
    cannot be used at runtime
Killed
```

The real object is built using the same command used to build the stub object. The `-z stub` option is omitted, and the path for the real output file is specified.

```
$ cc -Kpic -G -M mapfile -h libidx5.so.1 idx5.c -o lib/libidx5.so.1
```

Once the real object has been built in the `lib` subdirectory, the program can be run.

```
$ ./a.out
[0] 0 0 0
[1] 1 1 1
[2] 2 2 2
[3] 3 3 3
[4] 4 4 4
```

Using Stub Objects to Hide Obsolete Interfaces

Libraries evolve, and sometimes the original functionality proves to be undesirable. It is common for new abilities to be added, and for older ones to be considered obsolete. When backward compatibility is a concern, it is necessary to maintain such older functionality in the library for the benefit of existing objects. However, you may wish to prevent new use of these features. Stub objects can be used to enforce this policy. The *mapfile* `STUB_ELIMINATE` flag can be used to mark functions or data from an object that should be eliminated from the stub object, while remaining in the real object. This prevents new code, which links to the stub object, from using these obsolete items, and encourages code to be rewritten to use the preferred interfaces. Since the real objects still contain these items, existing objects are able to use them.

The `libidx5` example from the previous section illustrates this. That library demonstrates how to export global data from an object. However, exported global data introduces complexity to dynamic linking, and is best avoided. It is usually a better design to provide a function to access such data, such as the `idx5_func`function provided by `libidx5`. Continuing that example, `STUB_ELIMINATE` can be used to make the global data unavailable to new code that links to the stub, while providing those old interfaces in the real object for the benefit of existing programs.

The `mapfile` is rewritten to apply `STUB_ELIMINATE` to the two global data symbols. A benefit of applying `STUB_ELIMINATE` to global data is that it is no longer necessary to provide an `ASSERT` directive to provide the data size. In this example, the `ASSERT` is commented out. A real `mapfile` might omit it entirely.

```
$ cat better_mapfile
$mapfile_version 2
STUB_OBJECT;
SYMBOL_SCOPE {
        _idx5 {
                FLAGS=STUB_ELIMINATE;
                #ASSERT { TYPE=data; SIZE=4[5] };
        };
        idx5 {
                FLAGS=STUB_ELIMINATE;
                #ASSERT { BINDING=weak; ALIAS=_idx5 };
        };
        idx5_func;
    local:
        *;
```

```
};
```

A new version of the test program only uses the functional interface.

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

extern int idx5_func(int);

int
main(int argc, char **argv)
{
        int i;
        for (i = 0; i < 5; i++)
                (void) printf("[%d] %d\n", i, idx5_func(i));
        return (0);
}
```

The old test program is saved, the stub object is rebuilt using the new `mapfile`, and the test program is rebuilt, linking against the new stub object that employs `STUB_ELIMINATE`:

```
$ cp a.out original_a.out
$ cc -Kpic -G -M better_mapfile -h libidx5.so.1 idx5.c -o stublib/libidx5.so.1 -zstub
$ cc better_main.c -o better_a.out -L stublib -R '$ORIGIN/lib' -lidx5 -lc
$ ./better_a.out
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
```

The original test program can no longer be built, because the stub library lacks the necessary global data symbols. However, the preexisting binary that used them continues to function because the real library still provides the global data symbols.

```
$ cc main.c -L stublib -R '$ORIGIN/lib' -lidx5 -lc
Undefined                       first referenced
 symbol                             in file
idx5                                main.o
_idx5                               main.o
ld: fatal: symbol referencing errors
$ ./original_a.out
[0] 0 0 0
[1] 1 1 1
[2] 2 2 2
[3] 3 3 3
[4] 4 4 4
```

Ancillary Objects

By default, objects contain both *allocable* and *non-allocable* sections. Allocable sections are the sections that contain executable code and the data needed by that code at runtime. Non-

allocable sections contain supplemental information that is not required to execute an object at runtime. These sections support the operation of *debuggers* and other *observability* tools. The non-allocable sections in an object are not loaded into memory at runtime by the operating system, and so, they have no impact on memory use or other aspects of runtime performance no matter their size.

For convenience, both allocable and non-allocable sections are normally maintained in the same file. However, there are situations in which it can be useful to separate these sections.

- To reduce the size of objects in order to improve the speed at which they can be copied across wide area networks.
- To support fine grained debugging of highly optimized code requires considerable debug data. In modern systems, the debugging data can easily be larger than the code it describes. The size of a 32-bit object is limited to 4 Gbytes. In very large 32-bit objects, the debug data can cause this limit to be exceeded and prevent the creation of the object.
- To limit the exposure of internal implementation details.

Traditionally, objects have been stripped of non-allocable sections in order to address these issues. Stripping is effective, but destroys data that might be needed later. The Solaris link-editor can instead write non-allocable sections to an *ancillary object*. This feature is enabled with the `-z ancillary` option.

```
$ cc .... -z ancillary[=outfile] ....
```

By default, the ancillary file is given the same name as the primary output object, with a `.anc` file extension. However, a different name can be provided by providing an *outfile* value to the `-z ancillary` option.

When `-z ancillary` is specified, the link-editor performs the following actions.

- All allocable sections are written to the primary object. In addition, all non-allocable sections containing one or more input sections that have the `SHF_SUNW_PRIMARY` section header flag set are written to the primary object.
- All remaining non-allocable sections are written to the ancillary object.
- The following non-allocable sections are written to both the primary object and ancillary object.

`.shstrtab`	The section name string table.
`.symtab`	The full non-dynamic symbol table.
`.symtab_shndx`	The symbol table extended index section associated with `.symtab`.
`.strtab`	The non-dynamic string table associated with `.symtab`.
`.SUNW_ancillary`	Contains the information required to identify the primary and ancillary objects, and to identify the object being examined.

- The primary object and all ancillary objects contain the same array of sections headers. Each section has the same section index in every file.
- Although the primary and ancillary objects all define the same section headers, the data for most sections will be written to a single file as described above. If the data for a section is not present in a given file, the `SHF_SUNW_ABSENT` section header flag is set, and the `sh_size` field is 0.

This organization makes it possible to acquire a full list of section headers, a complete symbol table, and a complete list of the primary and ancillary objects from either of the primary or ancillary objects.

The following example illustrates the underlying implementation of ancillary objects. An ancillary object is created by adding the `-z ancillary` command line option to an otherwise normal compilation. The `file` utility shows that the result is an executable named `a.out`, and an associated ancillary object named `a.out.anc`.

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

int
main(int argc, char **argv)
{
        (void) printf("hello, world\n");
        return (0);
}
$ cc -g -zancillary hello.c
$ file a.out a.out.anc
a.out: ELF 32-bit LSB executable 80386 Version 1 [FPU], dynamically \
    linked, not stripped, ancillary object a.out.anc
a.out.anc: ELF 32-bit LSB ancillary 80386 Version 1, primary object a.out
$ ./a.out
hello world
```

The resulting primary object is an ordinary executable that can be executed in the usual manner. It is no different at runtime than an executable built without the use of ancillary objects, and then stripped of non-allocable content using the `strip` or `mcs` commands.

As previously described, the primary object and ancillary objects contain the same section headers. To see how this works, it is helpful to use the `elfdump` utility to display these section headers and compare them. The following table shows the section header information for a selection of headers from the previous link-edit example.

Index	Section Name	Type	Primary Flags	Ancillary Flags	Primary Size	Ancillary Size
13	.text	PROGBITS	ALLOC EXECINSTR	ALLOC EXECINSTR SUNW_ABSENT	0x131	0
20	.data	PROGBITS	WRITE ALLOC	WRITE ALLOC SUNW_ABSENT	0x4c	0

Index	Section Name	Type	Primary Flags	Ancillary Flags	Primary Size	Ancillary Size
21	.symtab	SYMTAB	0	0	0x450	0x450
22	.strtab	STRTAB	STRINGS	STRINGS	0x1ad	0x1ad
24	.debug_info	PROGBITS	SUNW_ABSENT	0	0	0x1a7
28	.shstrtab	STRTAB	STRINGS	STRINGS	0x118	0x118
29	.SUNW_ ancillary	SUNW_ancillary	0	0	0x30	0x30

The data for most sections is only present in one of the two files, and absent from the other file. The SHF_SUNW_ABSENT section header flag is set when the data is absent. The data for allocable sections needed at runtime are found in the primary object. The data for non-allocable sections used for debugging but not needed at runtime are placed in the ancillary file. A small set of non-allocable sections are fully present in both files. These are the .SUNW_ancillary section used to relate the primary and ancillary objects together, the section name string table .shstrtab, as well as the symbol table .symtab, and its associated string table .strtab.

It is possible to strip the symbol table from the primary object. A debugger that encounters an object without a symbol table can use the .SUNW_ancillary section to locate the ancillary object, and access the symbol contained within.

The primary object, and all associated ancillary objects, contain a .SUNW_ancillary section that allows all the objects to be identified and related together.

```
$ elfdump -T SUNW_ancillary a.out a.out.anc
a.out:
Ancillary Section:  .SUNW_ancillary
   index  tag                   value
     [0]  ANC_SUNW_CHECKSUM     0x8724
     [1]  ANC_SUNW_MEMBER       0x1         a.out
     [2]  ANC_SUNW_CHECKSUM     0x8724
     [3]  ANC_SUNW_MEMBER       0x1a3       a.out.anc
     [4]  ANC_SUNW_CHECKSUM     0xfbe2
     [5]  ANC_SUNW_NULL         0

a.out.anc:
Ancillary Section:  .SUNW_ancillary
   index  tag                   value
     [0]  ANC_SUNW_CHECKSUM     0xfbe2
     [1]  ANC_SUNW_MEMBER       0x1         a.out
     [2]  ANC_SUNW_CHECKSUM     0x8724
     [3]  ANC_SUNW_MEMBER       0x1a3       a.out.anc
     [4]  ANC_SUNW_CHECKSUM     0xfbe2
     [5]  ANC_SUNW_NULL         0
```

The ancillary sections for both objects contain the same number of elements, and are identical except for the first element. Each object, starting with the primary object, is introduced with a

MEMBER element that gives the file name, followed by a CHECKSUM that identifies the object. In this example, the primary object is a.out, and has a checksum of 0x8724. The ancillary object is a.out.anc, and has a checksum of 0xfbe2. The first element in a .SUNW_ancillary section, preceding the MEMBER element for the primary object, is always a CHECKSUM element, containing the checksum for the file being examined.

- The presence of a .SUNW_ancillary section in an object indicates that the object has associated ancillary objects.
- The names of the primary and all associated ancillary objects can be obtained from the ancillary section from any one of the files.
- It is possible to determine which file is being examined from the larger set of files by comparing the first checksum value to the checksum of each member that follows.

Note - The link editor does not read ancillary objects as input. If a relocatable object is created using the -z ancillary option, and the resulting object is later referenced to build another object, the sections from the ancillary object are not propagated to the final object.

Debugger Access and Use of Ancillary Objects

Debuggers and other *observability* tools must merge the information found in the primary and ancillary object files in order to build a complete view of the object. This is equivalent to processing the information from a single file. This merging is simplified by the primary object and ancillary objects containing the same section headers, and a single symbol table.

The following steps can be used by a debugger to assemble the information contained in these files.

1. Starting with the primary object, or any of the ancillary objects, locate the .SUNW_ancillary section. The presence of this section identifies the object as part of an ancillary group, contains information that can be used to obtain a complete list of the files and determine which of those files is the one currently being examined.
2. Create a section header array in memory, using the section header array from the object being examined as an initial template.
3. Open and read each file identified by the .SUNW_ancillary section in turn. For each file, fill in the in-memory section header array with the information for each section that does not have the SHF_SUNW_ABSENT flag set.

The result will be a complete in-memory copy of the section headers with pointers to the data for all sections. Once this information has been acquired, the debugger can proceed as it would in the single file case, to access and control the running program.

Note - The ELF definition of ancillary objects provides for a single primary object, and an arbitrary number of ancillary objects. At this time, the Oracle Solaris link-editor only produces a single ancillary object containing all non-allocable sections. This may change in the future. *Debuggers* and other *observability* tools should be written to handle the general case of multiple ancillary objects.

Compressed Debug Sections

As discussed in "Ancillary Objects" on page 81, objects contain both *allocable* and *non-allocable* sections. Allocable sections are the sections that contain executable code and the data needed by that code at runtime. Non-allocable sections contain supplemental information that is not required to execute an object at runtime. These sections support the operation of debuggers and other observability tools, and are informally referred to as *debug* sections.

Depending on the level of debug information requested, debug sections can become very large relative to the code they describe. Ancillary objects, which write these sections to a separate file, offer one mechanism for dealing with these large sections. Compressed debug sections offer a second, complimentary, option for reducing debug section size.

Debug sections are compressed with the industry standard ZLIB compression library. Documentation for ZLIB may be found at http://www.zlib.net/.

The link-editor recognizes compressed debug sections within input objects, and automatically decompresses these sections. This operation is transparent to the user of the link-editor, and requires no special action.

By default, the link-editor does not compress debug sections in output objects. Use the `-z compress-debug-sections` option to enable the compression of debug sections in the output file.

```
$ cc .... -z compress-sections[=cmp-type] ....
```

The following values for *cmp-type* are recognized.

`none` — No compression is done. This option is equivalent to not specifying the `-z compress-sections` option.

`zlib` — Compress candidate sections using ZLIB compression. The resulting output sections have the `SHF_COMPRESSED` section flag set to identify the use of compression.

`zlib-gnu` — Compress all candidate sections using ZLIB compression, using the GNU section compression format. This format requires candidate

sections to have a name that begins with `.debug`. The resulting output sections are renamed to start with .zdebug to identify the use of compression.

If *cmp-type* is omitted, the `zlib` style is used.

Compression for any section that would be larger in compressed form than the original non-compressed data is quietly skipped.

To be a candidate for compression, a section must be non-allocable, and belong to one of the following classes.

`annotate` Annotate sections provide information that is used by memory access tools, and coverage related tools. These sections are identified by having a `SHT_SUNW_ANNOTATE` section type.

`debug` Debug sections are identified by having a `.compcom`, `.line`, `.stab*`, `.debug*`, or `.zdebug*` section name. These sections are also identified by having an `SHT_PROGBITS` or `SHT_SUNW_DEBUG*` section type.

The `zlib-gnu` compression type is limited to sections with a name that starts with `.debug`. When `zlib-gnu` is used, sections that would otherwise be candidates for compression are not compressed. The underlying ZLIB compression is identical for the `zlib` and `zlib-gnu` styles, and both formats deliver the same amount of compression for a given input section. The two styles differ in the selection of candidate sections, the format of the compression header, and in how compressed sections are identified. See “Section Compression” on page 330. Unless there is a specific requirement to use the `zlib-gnu` style, the more general default `zlib` style is recommended.

The following program demonstrates the use of compressed debug sections. For the purpose of comparison, the program is built twice, once without compression, and once with compression.

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

int
main(int argc, char **argv)
{
        (void) printf("hello, world\n");
        return (0);
}
% cc -g hello.c -o a.out.uncompressed
% cc -g hello.c -o a.out.compressed -z compress-sections
```

The section headers of the uncompressed, and compressed debug sections can now be compared.

```
$ elfdump -c a.out.uncompressed
```

```
....
Section Header[24]:  sh_name: .debug_info
    sh_addr:      0               sh_flags:   0
    sh_size:      0x17b           sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[25]:  sh_name: .debug_line
    sh_addr:      0               sh_flags:   0
    sh_size:      0x4f            sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[26]:  sh_name: .debug_abbrev
    sh_addr:      0               sh_flags:   0
    sh_size:      0x7c            sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[27]:  sh_name: .debug_pubnames
    sh_addr:      0               sh_flags:   0
    sh_size:      0x1b            sh_type:    [ SHT_PROGBITS ]
    ....

$ elfdump -c a.out.compressed
....
Section Header[24]:  sh_name: .debug_info
    sh_addr:      0               sh_flags:   [ SHF_COMPRESSED ]
    sh_size:      0x14f           sh_type:    [ SHT_PROGBITS ]
    ....
    ch_size:      0x196           ch_type:    [ ELFCOMPRESS_ZLIB ]
    ch_addralign: 0x1

Section Header[25]:  sh_name: .debug_line
    sh_addr:      0               sh_flags:   0
    sh_size:      0x4f            sh_type:    [ SHT_PROGBITS ]
    ....

Section Header[26]:  sh_name: .debug_abbrev
    sh_addr:      0               sh_flags:   [ SHF_COMPRESSED ]
    sh_size:      0x79            sh_type:    [ SHT_PROGBITS ]
    ....
    ch_size:      0x7c            ch_type:    [ ELFCOMPRESS_ZLIB ]
    ch_addralign: 0x1

Section Header[27]:  sh_name: .debug_pubnames
    sh_addr:      0               sh_flags:   0
    sh_size:      0x1b            sh_type:    [ SHT_PROGBITS ]
    ....
```

Each compressed section, `.debug_info`, and `.debug_abbrev`, is identified with a `SHF_COMPRESSED` section flag. In addition, the section header information is accompanied with compression header structure information. The `ch_size` and `ch_addralign` fields provide size and alignment requirements for the uncompressed data. See "Section Compression" on page 330.

The `.debug_line` and `.debug_pubnames` sections would be larger compressed than in their original uncompressed form, and have therefore been left uncompressed.

Compression Costs And Benefits

The primary benefit of compressed debug sections is a size reduction of objects. However, compression imposes additional costs in runtime and memory use at all stages of development

- The compiler must produce each debug section in uncompressed form, allocate additional memory for the compressed version, and perform the compression.
- When reading an input object, the link-editor must read the compressed data into memory, allocate additional memory to hold the decompressed data, and perform the decompression.
- On output, the link-editor must create an uncompressed version of the resulting debug sections. If compression is requested, additional memory and time are used to create the compressed version.
- When a debugger reads an object with compressed debug sections, the debugger must allocate additional memory to hold the decompressed data, and perform the decompression.

Furthermore, compressed debug sections allow for smaller files, but not for larger amounts of information. A common example involves 32-bit objects, which are fundamentally limited to 4 Gbytes due to the use of 32-bit file offsets and sizes within them. It is sometimes assumed that compressing debug data might allow for more debug information to be generated. However, the format of 32-bit debug data also contains 32-bit offsets, and so, is logically constrained to 4 Gbytes in uncompressed form.

For these reasons, compressed debug sections are not recommended for general development, where speed of the compile/link/debug cycle usually outweighs the benefits of smaller debug data. Compressed debug sections may be beneficial in cases where disk space is scarce, or for production objects that are copied widely and debugged rarely.

Parent Objects

Programs that offer extensible functionality often make use of shared objects, loaded at runtime using the `dlopen` function. These shared objects are often referred to as *plugins*, and provide a flexible means to extend the abilities of the core system. The object that loads the *plugins* is referred to as the *parent*.

A parent object loads the *plugin* and accesses functions and data from within the *plugin*. It is also common for the parent object to provide functions and data for use by the *plugin*. This is illustrated by the following parent and *plugin* source files. Here the parent supplies a function named `parent_callback` for the benefit of the *plugin*. The *plugin* supplies a function named `plugin_func` for the parent to call.

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

```
#include    <link.h>

void
parent_callback(void)
{
        (void) printf("plugin_func() has called parent_callback()\n");
}

int
main(int argc, char **argv)
{
        typedef void plugin_func_t(void);

        void            *hdl;
        plugin_func_t   *plugin_func;

        if (argc != 2) {
                (void) fprintf(stderr, "usage: main plugin\n");
                return (1);
        }

        if ((hdl = dlopen(argv[1], RTLD_LAZY)) == NULL) {
                (void) fprintf(stderr, "unable to load plugin: %s\n",
      dlerror());
                return (1);
        }

        plugin_func = (plugin_func_t *) dlsym(hdl, "plugin_func");
        if (plugin_func == NULL) {
                (void) fprintf(stderr, "unable to find plugin_func: %s\n",
                    dlerror());
                return (1);
        }

        (*plugin_func)();

        return (0);
}

$ cat plugin.c
#include    <stdio.h>

extern void parent_callback(void);

void
plugin_func(void)
{
        (void) printf("parent has called plugin_func() from plugin.so\n");
        parent_callback();
}

$ cc -o main main.c -lc
$ cc -Kpic -G -o plugin.so plugin.c -lc
$ ./main ./plugin.so
parent has called plugin_func() from plugin.so
plugin_func() has called parent_callback()
```

When building any shared object, the `-z defs` option is recommended, in order to ensure that the object specifies all of its dependencies. However, the use of `-z defs` prevents the *plugin* object from linking due to the unsatisfied symbol from the parent object.

```
$ cc -zdefs -Kpic -G -o plugin.so plugin.c -lc
Undefined                       first referenced
 symbol                             in file
parent_callback                     plugin.o
ld: fatal: symbol referencing errors
```

A *mapfile* can be used to specify to the link-edit that the `parent_callback` symbol is supplied by the parent object.

```
$ cat plugin.mapfile
$mapfile_version 2

SYMBOL_SCOPE {
        global:
               parent_callback         { FLAGS = PARENT };
};
$ cc -zdefs -Mplugin.mapfile -Kpic -G -o plugin.so plugin.c -lc
```

The preferred solution for building a *plugin* is to use the `-z parent` option to provide the *plugin* with direct access to symbols from the parent. An added benefit of using `-z parent` instead of a *mapfile*, is that the name of the parent object is recorded in the dynamic section of the *plugin*, and is displayed by the `file` utility.

```
$ cc -zdefs -zparent=main -Kpic -G -o plugin.so plugin.c -lc
$ elfdump -d plugin.so | grep PARENT
     [0]  SUNW_PARENT       0xcc               main
$ file plugin.so
plugin.so: ELF 32-bit LSB dynamic lib 80386 Version 1, parent main, \
    dynamically linked, not stripped
```

Debugging Aids

The link-editor provides a debugging facility that allows you to trace the link-editing process in detail. This facility can help you understand and debug the link-edit of your applications and libraries. The type of information that is displayed by using this facility is expected to remain constant. However, 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. This option must be augmented with one or more tokens to indicate the type of debugging that is required.

The tokens that are available with `-D` can be displayed by typing `-D help` at the command line.

```
$ ld -Dhelp
```

By default, all debug output is sent to `stderr`, the standard error output file. Debug output can be directed to a file instead, using the `output` token. For example, the help text can be captured in a file named `ld-debug.txt`.

```
$ ld -Dhelp,output=ld-debug.txt
```

Most compiler drivers assign the `-D` option a different meaning, often to define preprocessing macros. The `LD_OPTIONS` environment variable can be used to bypass the compiler driver, and supply the `-D` option directly to the link-editor.

The following example shows how input files can be traced. This syntax can be useful to determine what libraries are used as part of a link-edit. Objects that are extracted from an archive are also displayed with this syntax.

```
$ 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. Multiple “(again)” diagnostics 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`. This symbol 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 toggled 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 25.

♦ ♦ ♦ 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 Oracle 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 Oracle Solaris OS 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 385. 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 performs the following actions.

- 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 elfdump(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.

```
$ elfdump -d /usr/bin/cat

Dynamic Section:  .dynamic:
   index  tag               value
     [0]  NEEDED            0x211              libc.so.1
     ...
```

In the previous elfdump(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 default search 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 135 and “Recording a Shared Object Name” on page 136.

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.

A *runpath* recording can be displayed using elfdump(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.

```
$ elfdump -d prog | egrep "NEEDED|RUNPATH"
     [1]  NEEDED        0x4ce           libfoo.so.1
```

```
    [3]  NEEDED        0x4f6            libc.so.1
   [21]  RUNPATH      0x210e            /home/me/lib:/home/you/lib
```

Another way to add to the runtime linker's search path is to set one of the `LD_LIBRARY_PATH` family of environment variables. 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 that matches the name being looked for. Or, the other way around. The runtime linker rejects the mismatched 32–bit library and continues its search looking for a valid 64–bit match. If no match is found, an error message is generated. This rejection can be observed in detail by setting the `LD_DEBUG` environment variable to include the `files` token. See "Debugging Facility" on page 129.

```
$ 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.

- `$CAPABILITY` – Indicates a directory in which objects offering differing capabilities can be located. See “Capability Specific Shared Objects” on page 253.
- `$ISALIST` – Expands to the native instruction sets executable on this platform. See “Instruction Set Specific Shared Objects” on page 255.
- `$ORIGIN` – Provides the directory location of the current object. See “Locating Associated Dependencies” on page 257.
- `$OSNAME` – Expands to the name of the operating system. See “System Specific Shared Objects” on page 257.
- `$OSREL` – Expands to the operating system release level. See “System Specific Shared Objects” on page 257.
- `$PLATFORM` – Expands to the processor type of the current machine. See “System Specific Shared Objects” on page 257.

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 "Relocations" on page 353. Two basic types of relocation exist.

- Non-symbolic relocations
- Symbolic relocations

The relocation records for an object can be displayed by using `elfdump`(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.

```
$ elfdump -r libbar.so.1

Relocation Section:  .rel.got:
    type                     offset             section       symbol
  R_SPARC_RELATIVE           0x10438            .rel.got
  R_SPARC_GLOB_DAT           0x1043c            .rel.got      foo
```

The first relocation is a simple relative relocation that can be seen from the relocation type and that no symbol is referenced. This relocation needs to use the base address at which the object is 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 from one of its dependencies.

Relocation Symbol Lookup

The runtime linker is responsible for searching for symbols that are required by objects at runtime. Typically, users become familiar with the default search model that is applied to a dynamic executable and its dependencies, and to the objects obtained through `dlopen`(3C). However, more complex flavors of symbol lookup can result because of the symbol attributes of an object, or through specific binding requirements.

Two attributes of an object affect symbol lookup. The first attribute is the requesting object's symbol *search scope*. The second attribute is 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.

An object can define a *world* search scope, and/or a *group* search scope.

`world`

The object can search for symbols in any other global object within the process.

`group`

The object can search for symbols in any 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.

An object can define that any of the object's exported symbols are *globally* visible or *locally* visible.

`global`

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

`local`

The object's exported symbols can be referenced only from other objects that make up the same *group*.

The runtime symbol search can also be dictated by a symbols visibility. Symbols assigned the `STV_SINGLETON` visibility are not affected by any symbol search scope. All references to a singleton symbol are bound to the first occurrence of a singleton definition within the process. See Table 12-23.

The simplest form of symbol lookup is outlined in the next section "Default Symbol Lookup" on page 101. Typically, symbol attributes are exploited by various forms of `dlopen`(3C). These scenarios are discussed in "Symbol Lookup" on page 120.

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 Chapter 6, "Direct Bindings".

Default Symbol Lookup

A dynamic executable and all the dependencies loaded with the executable are assigned *world* search scope, and *global* symbol visibility. A default 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.

`ldd`(1) lists the dependencies of a dynamic executable in the order in which the dependencies are loaded. For example, suppose the dynamic executable `prog` specifies `libfoo.so.1` and `libbar.so.1` as its dependencies.

```
$ ldd prog
```

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

Should the symbol `bar` be required to perform a relocation, the runtime linker first looks for `bar` in the dynamic executable `prog` If the symbol is not found, the runtime linker then searches 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 Chapter 7, "Building Objects to Optimize System Performance". See Chapter 6, "Direct Bindings" for an alternative lookup model.

The default relocation processing model also provides for a transition 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.

Runtime Interposition

By default, the runtime linker searches for a symbol first in the dynamic executable and then in each dependency. With this model, 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.

An overview of how symbol resolution is affected by interposition is provided in "Simple Resolutions" on page 39. A mechanism for changing symbol visibility, and hence reducing the chance of accidental interposition is provided in "Reducing Symbol Scope" on page 49.

Note - Symbols assigned the `STV_SINGLETON` visibility provide a form of interposition. All references to a singleton symbol are bound to the first occurrence of a singleton definition within the process. See Table 12-23.

Interposition can be enforced, on a per-object basis, if an object is explicitly 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 symbol, any object identified as an interposer is searched after the application, but before any other dependencies.

The use of all of the interfaces offered by an interposer can only be guaranteed if the interposer is loaded before any process relocation has occurred. Interposers provided using

the environment variable LD_PRELOAD, or established as non-lazy loaded dependencies of the application, are loaded before relocation processing starts. Interposers that are brought into a process after relocation has started are demoted to normal dependencies. Interposers can be demoted if the interposer is lazy loaded, or loaded as a consequence of using dlopen(3C). The former category can be detected using ldd(1).

```
$ ldd -Lr prog
        libc.so.1 =>     /lib/libc.so.1
        foo.so.2 =>      ./foo.so.2
        libmapmalloc.so.1 =>     /usr/lib/libmapmalloc.so.1
        loading after relocation has started: interposition request \
            (DF_1_INTERPOSE) ignored: /usr/lib/libmapmalloc.so.1
```

Note - If the link-editor encounters an explicitly defined interposer while processing dependencies for lazy loading, the interposer is recorded as a non-lazy loadable dependency.

Individual symbols within a dynamic executable can be defined as interposers using the INTERPOSE mapfile keyword. This mechanism is more selective that using the -z interpose option, and provides better insulation over adverse interposition that can occur as dependencies evolve. See “Defining Explicit Interposition” on page 168.

When Relocations Are Performed

Relocations can be separated into two types dependent upon when the relocation is performed. This distinction arises due to the type of *reference* being made to the relocated offset.

- 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 an associated relocation.

As part of the first call to a procedure linkage table entry, control is passed to the runtime linker. The runtime linker looks up the required symbol and rewrites the entry information in the

associated object. 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 reference and lazy reference relocations when an object is loaded. These relocations are performed before the application gains, or regains, control. For example, all relocations within the file `prog` together within its dependencies are processed under the following environment variable. These relocations are 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 the object requires complete relocation processing at the time the object is loaded. This relocation requirement is also propagated to any dependencies of the marked object at runtime.

Note - The preceding examples of immediate references and lazy references are typical. However, 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. These relocations are common for position-independent code.

However, a dynamic executable is typically created from position dependent code, which might not indicate that a procedure linkage table entry is required. Because a dynamic executable has a fixed location, the link-editor can create a procedure linkage table entry when a reference is bound to an external function definition. This procedure linkage table entry creation occurs 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 together with the termination of the application. In the following example, the symbol `bar`, which is referenced in the file `libfoo.so.1`, cannot be located.

```
$ 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
```

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. However, a runtime relocation error can occur if a dependency located at runtime is incompatible with the original dependency referenced as part of the link-edit. In the previous example, `prog` was built against a version of the shared object `libbar.so.1` that contained a symbol definition for `bar`.

The use of the `-z nodefs` option during a link-edit suppresses the validation of an objects runtime relocation requirements. This suppression can also lead to runtime relocation errors.

If a relocation error occurs because a symbol used as an immediate reference cannot be found, the error condition occurs immediately during process initialization. With the default mode of lazy binding, if a symbol used as a lazy reference cannot be found, the error condition occurs 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), every dependency is printed and all immediate reference relocations are processed. If a reference cannot be resolved, a diagnostic message is produced. From the previous example, the `-d` option would result in the following error diagnostic.

```
$ 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 reference *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 by using the environment variable `LD_PRELOAD`. This environment variable 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.

In the following example, the dynamic executable `prog` is loaded, followed by the shared object `newstuff.so.1`. The dependencies defined within `prog` are then loaded.

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

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

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

In the following 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).

```
$ ldd -e 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 provide for interposition. 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 original 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 interposition of the new symbol on the original symbol enables the new function to carry out additional processing. The new function can also call through to the original function. This mechanism typically obtains the original symbol's address using `dlsym`(3C) with the special handle `RTLD_NEXT`.

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 the lazy binding of a function call, 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 103. 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 is 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. This model also works well if 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 follows 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 -lelf -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         libelf.so.1
     [3]  NEEDED        0x13d         libc.so.1
     [4]  RUNPATH       0x147         $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 libelf.so.1 has no preceding LAZY flag, this library is loaded at the initial startup of the program.

Note - libc.so.1 has special system requirements, that require the file not be lazy loaded. If -z lazyload is in effect when libc.so.1 is processed, the flag is effectively ignored.

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 117. 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);
                ....
        }
```

```
}
```

Although very flexible, this model of using `dlopen` and `dlsym` is an unnatural coding style, and has some drawbacks.

- The object in which the symbols are expected to exit must be known.
- The calls through function pointers provide no means of verification by either the compiler, or `lint(1)`.

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 that a function reference can trigger 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. This coding is far more natural, and 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. -zdefs -zlazyload -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 dependency name, is desirable. A means of testing for the availability of a dependency that satisfies a function reference is required.

A robust model for testing for the existence of a function can be achieved with explicitly defined *deferred* dependencies, and use of `dlsym`(3C) with the `RTLD_PROBE` handle.

An explicitly defined deferred dependency is an extension to a lazy loadable dependency. A symbol reference that is associated to a deferred dependency is referred to as a deferred symbol. A relocation against this symbol is only processed when the symbol is first referenced. These relocations are not processed as part of `LD_BIND_NOW` processing, or through `dlsym`(3C) with the `RTLD_NOW` flag.

Deferred dependencies are established at link-edit time using the link-editors `-z deferred` option.

```
$ cc -G -o libfoo.so.1 foo.c -L. -zdefs -zdeferred -lbar -R'$ORIGIN'
```

Having established `libbar.so.1` as a deferred dependency, a reference to `bar1` can verify that the dependency is available. This test can be used to control the reference to functions provided

by the dependency in the same manner as dlsym(3C) had been used. This code can then make natural calls to bar1 and bar2. These calls are much more legible and easier to write, and allow the compiler to catch errors in their calling sequences.

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

Deferred dependencies offer an additional level of flexibility. Provided the dependency has not already been loaded, the dependency can be changed at runtime. This mechanism offers a level of flexibility similar to dlopen(3C), where different objects can be loaded and bound to by the caller.

If the original dependency name is known, then the original dependency can be exchanged for a new dependency using dlinfo(3C) with the RTLD_DI_DEFERRED argument. Alternatively, a deferred symbol that is associated with the dependency can be used to identify the deferred dependency using dlinfo(3C) with the RTLD_DI_DEFERRED_SYM argument.

Initialization and Termination Routines

Dynamic objects can supply code that provides for runtime initialization and termination processing. The initialization code of a dynamic object is executed once each time the dynamic object is loaded in a process. The termination code of a dynamic object is executed once each time the dynamic object is unloaded from a process or at process termination.

Before transferring control to an application, the runtime linker processes any initialization sections found in the application and any loaded dependencies. If new dynamic objects are loaded during process execution, their initialization sections are processed as part of loading the object. 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). Termination sections are also called 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 on the creation of initialization and termination sections by the link-editor see “Initialization and Termination Sections” on page 35.

Limitations and Pitfalls of Initialization and Termination Code

ELF initialization and termination sections and routines execute at a sensitive point in the life cycle of the object. During initialization, the object has been loaded into memory, but is not fully initialized. During finalization, the object is still loaded in memory, but is no longer safe to use, and may be partially removed from the process state. In either context, the process state is not fully consistent, and there are significant limits on what code can safely do. Common pitfalls include, but are not limited, to the following.

- Cyclic dependencies resulting in deadlock, where the initialization code for one object triggers the loading of another object, which in turn calls back into the initial object.

- Thread serialization failures when a shared object is used in a multithreaded application. Two threads may attempt to access a lazily loaded library at the same time. The thread that gets there first will cause the runtime linker to load the object and start to run the initialization code. Programmers are often under the mistaken impression that the runtime linker can prevent more than one thread from accessing a given object when ELF initialization and termination code is running, but this is not the case. The runtime linker cannot prevent other threads from attempting to access the library once the initialization code is running. It is therefore possible for a second thread to access the object in an inconsistent state. It is the responsibility of the object to serialize such access, either by providing the necessary locks, or my requiring the caller to do so.

ELF initialization and termination sections and routines allow for the execution of arbitrary code, giving the illusion that they are capable of doing anything that code running in a normal context might do. In this view, such code seems like nothing more than a convenient way to do initialization or cleanup without explicit function calls. This misconception leads to failures that can be difficult to diagnose.

Programmers should be cautious in their use of ELF initialization and termination code, and limit the scope and complexity of operations. The link-editor and runtime linker are not cognizant of the content or purpose of such code, and cannot diagnose or prevent unsafe code. Small self contained operations are safe. Operations involving access to other objects or process state may not be. Rather than attempt complex operations in initialization and termination code, libraries should provide explicit initialization and termination functions for their callers to run, and document the requirement to do so.

The following section considers these issues in detail.

Initialization and Termination Order

To determine the order of executing initialization and termination code within a process at runtime is a complex procedure that involves dependency analysis. This procedure has evolved substantially from the original inception of initialization and termination sections. This procedure attempts to fulfill the expectations of modern languages and current programming techniques. However, scenarios can exist, where user expectations are hard to meet. Flexible, predictable runtime behavior can be achieved by understanding these scenarios together with limiting the content of initialization code and termination code.

The goal of an initialization section is to execute a small piece of code before any other code within the same object is referenced. The goal of a termination section is to execute a small piece of code after an object has finished executing. Self contained initialization sections and termination sections can easily satisfy these requirements.

However, initialization sections are typically more complex and make reference to external interfaces that are provided by other objects. Therefore, a dependency is established where the initialization section of one object must be executed before references are made from

other objects. Applications can establish an extensive dependency hierarchy. In addition, dependencies can creating cycles within their hierarchies. The situation can be further complicated by initialization sections that load additional objects, or change the relocation mode of objects that are already loaded. These issues have resulted in various sorting and execution techniques that attempt to satisfy the original goal of these sections.

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 sections are called in the topological order of the dependencies. The termination sections of any cyclic dependencies are executed in their load order.

A static analysis of the initialization order of an object's dependencies can be obtained by using `ldd`(1) with the `-i` option. For example, the following dynamic executable and its dependencies exhibit a cyclic dependency.

```
$ elfdump -d B.so.1 | grep NEEDED
     [1]     NEEDED      0xa9    C.so.1
$ elfdump -d C.so.1 | grep NEEDED
     [1]     NEEDED      0xc4    B.so.1
$ elfdump -d main | grep NEEDED
     [1]     NEEDED      0xd6    A.so.1
     [2]     NEEDED      0xc8    B.so.1
     [3]     NEEDED      0xe4    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
```

The previous analysis resulted solely from the topological sorting of the explicit dependency relationships. However, objects are frequently created that do not define their required dependencies. For this reason, symbol bindings are also incorporated as part of dependency analysis. The incorporation of symbol bindings with explicit dependencies can help produce a more accurate dependency relationship. A more accurate static analysis of initialization order can be obtained by using `ldd`(1) with the `-i` and `-d` options.

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. These bindings can extend the dependency relationships so far established. A static analysis of the initialization order that incorporates all symbol binding can be obtained by using `ldd`(1) with the `-i` and `-r` options.

In practice, most applications use lazy binding. Therefore, the dependency analysis achieved before computing the initialization order follows the static analysis using `ldd -id`. However, because this dependency analysis can be incomplete, and because cyclic dependencies can exist, the runtime linker provides for dynamic initialization.

Dynamic initialization attempts to execute the initialization section of an object before any functions in the same object are called. During lazy symbol binding, the runtime linker determines whether the initialization section of the object being bound to has been called. If not, the runtime linker executes the initialization section 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 `init`. See "Debugging Facility" on page 129. Extensive runtime initialization information and termination information can be captured by adding the debugging token `detail`. This information includes dependency listings, topological processing, and the identification of cyclic dependencies.

Dynamic initialization is only available when processing lazy references. This dynamic initialization is circumvented by the following.

- Use of the environment variable `LD_BIND_NOW`.
- Objects that have been built with the `-z now` option.
- Objects that are loaded by `dlopen`(3C) with the mode `RTLD_NOW`.

The initialization techniques that have been described so far might still be insufficient to cope with some dynamic activities. Initialization sections can load additional objects, either explicitly using `dlopen`(3C), or implicitly through lazy loading and the use of filters. Initialization sections can also promote the relocations of existing objects. Objects that have been loaded to employ lazy binding have these bindings resolved if the same object is referenced using `dlopen`(3C) with the mode `RTLD_NOW`. This relocation promotion effectively suppresses the dynamic initialization facility that is available when resolving a function call dynamically.

Whenever new objects are loaded, or existing objects have their relocations promoted, a topological sort of these objects is initiated. Effectively, the original initialization execution is suspended while the new initialization requirements are established and the associated initialization sections executed. This model attempts to insure that the newly referenced objects are suitably initialized for the original initialization section to use. However, this parallization can be the cause of unwanted recursion.

While processing objects that employ lazy binding, the runtime linker can detect certain levels of recursion. This recursion can be displayed by setting `LD_DEBUG=init`. For example, the execution of the initialization section of `foo.so.1` might result in calling another object. If this object then references an interface in `foo.so.1` then a cycle is created. The runtime linker can detect this recursion as part of binding the lazy function reference to `foo.so.1`.

```
$ LD_DEBUG=init  prog
00905: ....
00905: warning: calling foo.so.1 whose init has not completed
00905: ....
```

Recursion that occurs through references that have already been relocated can not be detected by the runtime linker.

Recursion can be expensive and problematic. Reduce the number of external references and dynamic loading activities that can be triggered by an initialization section so as to eliminate recursion.

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

The preceding sections describe the various techniques that are 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 make initialization processing and termination processing that is predictable, while 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 43. Do not express dependencies that are not required. See “Shared Object Processing” on page 29. 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 139.

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 that are known to the runtime linker are `/lib/secure/64` and `/usr/lib/secure/64`. The utility `crle`(1) can be used to specify additional trusted directories that are 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 96.

In a secure process, any *runpath* specifications provided by the application or any of its dependencies are used. However, the *runpath* must be a full path name, that is, the path name must start with a '/'.

In a secure process, the expansion of the `$ORIGIN` string is allowed only if the string expands to a trusted directory. See "Security" on page 261. However, should a `$ORIGIN` expansion match a directory that has already provided dependencies, then the directory is implicitly secure. This directory can be used to provide additional dependencies.

In a secure process, `LD_CONFIG` is ignored. However, a configuration file that is recorded in a secure application is used. See the `-c` option of `ld`(1). A recorded configuration file must be a full path name, that is, the path name starts with a '/'. A recorded configuration file that employs the `$ORIGIN` string is restricted to known trusted directories. Developers who record a configuration file within a secure application should ensure that the configuration file directory is suitably protected from malicious intrusion. In the absence of a recorded configuration file, a secure process uses the default configuration file, if the configuration file 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 path names or relative path names are used as is. Therefore, the developer of a secure process should ensure that the target directory referenced as one of these dependencies 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 applies 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 effectively uses the same services of the runtime linker that are used to process the applications standard dependencies.

Delayed object binding has several advantages.

- By processing an object when the object is required rather than during the initialization of an application, startup time can be greatly reduced. If the services provided by an object are not needed during a particular run of the application, the object is not required. This scenario can occur for objects that provide 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 of this shared object 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 that exist within the objects that are being freed are called at this time.
- Any error conditions that occur as a result of using the 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 the Oracle Solaris OS, the dynamic linking interfaces were made available by the shared object `libdl.so.1`. `libdl.so.1` remains available to support any existing dependencies. However, the dynamic linking interfaces offered by `libdl.so.1` are now available from `libc.so.1`. Linking with `-ldl` is no longer necessary.

Loading Additional Objects

Additional objects can be added to a running process's address space by 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 uses a set of rules to generate an appropriate path name. Path names that contain a '/' are 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 96. For example, the file `main.c` contains the following code fragment.

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

int main(int argc, char **argv)
{
        void *handle;
        ....

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

To locate the shared object `foo.so.1`, the runtime linker uses any `LD_LIBRARY_PATH` definition that is present at process initialization. Next, the runtime linker uses 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 that is specified using dlopen(3C) should be referenced by its *versioned* file name. For more information on versioning, see "Coordination of Versioned Filenames" on page 250.

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, from the previous example, `main.c` can contain the following `dlopen` call.

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

The handle returned from this 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

After locating and loading any objects, the runtime linker must process each object and perform any necessary relocations. Any objects that are 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`. However, the mode of a `dlopen` together 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 Chapter 6, “Direct Bindings”.

Note - Symbols assigned the `STV_SINGLETON` visibility are bound using the default symbol search, regardless of any dlopen(3C) attributes. See Table 12-23.

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 120, uses this default model to illustrate typical object group interactions. The sections “Defining a Global Object” on page 123, “Isolating a Group” on page 124, and “Object Hierarchies” on page 124 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 a basic 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.

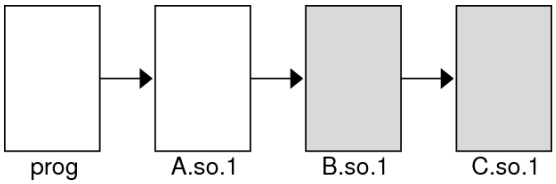
The default symbol lookup model provides for transitioning into a lazy loading environment. If a symbol can not be found in the presently loaded objects, any pending lazy loaded objects are processed in an attempt to locate the symbol. This loading compensates for objects that have not fully defined 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), that is 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 the executable 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 the applications initial dependencies. For example, if `A.so.1` requires a function relocation after the previous `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.

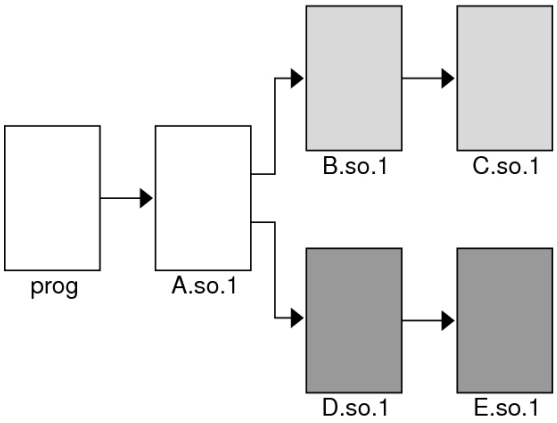
These symbol search and symbol visibility attributes maintain associations between objects. These associations are based on their introduction into the process address space, and on any dependency relationship between the objects. Assigning the objects associated with a given `dlopen`(3C) to a unique 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 that are 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.

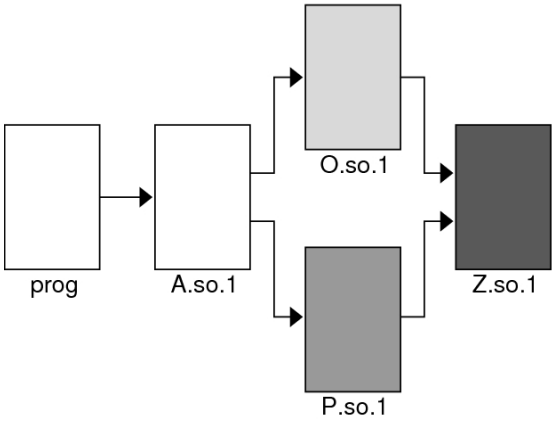
In the following example, the shared objects O.so.1 and P.so.1 have the same common dependency.

```
$ 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 defines local symbol visibility, and is referenced by another group that defines global symbol visibility, then the object's visibility becomes 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 defines a group search requirement, and is referenced by another group that defines a world search requirement, then the object's search requirement becomes 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 is obtained from a dlopen(3C), and uses dlopen to open a secondary object, both objects are assigned to a unique group. This situation can prevent either object from locating symbols from the other.

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 that are provided in these new objects. The file `main.c` contains the following code.

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

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

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

        if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
            ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
                (void) printf("dlsym: %s\n", dlerror());
                return (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 “Relocation Symbol Lookup” on page 100.

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 a symbol.

The RTLD_DEFAULT handle employes the same rules used by the runtime linker to resolve any symbol reference from the calling object. See "Default Symbol Lookup Model" on page 120. Two aspects of this model should be noted.

- A symbol reference that matches the same symbol reference from the dynamic executable is bound to the procedure linkage table entry associated with the reference from the executable. See "Procedure Linkage Table (Processor-Specific)" on page 416. This artifact of dynamic linking ensures that all components within a process see a single address for a function.
- If a symbol definition can not be found to satisfy a non-weak symbol reference within the objects that are presently loaded in the process, a lazy loading fall back is initiated. This fall back iterates through each loaded dynamic object, and loads any pending lazy loadable objects in an attempt to resolve the symbol. This model compensates for objects that have not fully defined their dependencies. However, this compensation can undermine the advantages of lazy loading. Unnecessary objects can be loaded, or an exhaustive loading of all lazy loadable objects can occur should the relocation symbol not be found.

RTLD_PROBE follows a similar model to RTLD_DEFAULT, but differs in the two aspects noted with RTLD_DEFAULT. RTLD_PROBE only binds to explicit symbol definitions, and is not bound to any procedure linkage table entry within the executable. In addition, RTLD_PROBE does not initiate an exhaustive lazy loading fall back. RTLD_PROBE is the most appropriate flag to use to detect the presence of a symbol within an existing process.

RTLD_DEFAULT and RTLD_PROBE can both initiate an explicit lazy load. An object can make reference to a function, and that reference can be established through a lazy loadable dependency. Prior to calling this function, RTLD_DEFAULT or RTLD_PROBE can be used to test for the existence of the function. Because the object makes reference to the function, an attempt is first made to load the associated lazy dependency. The rules for RTLD_DEFAULT and RTLD_PROBE are then followed to bind to the function. In the following example, an RTLD_PROBE call is used both to trigger a lazy load, and to bind to the loaded dependency if the dependency exists.

```
void foo()
{
        if (dlsym(RTLD_PROBE, "foo1")) {
                foo1(arg1);
                foo2(arg2);
                ....
}
```

To provide a robust and flexible model for testing for functionally, the associated lazy dependencies should be explicitly tagged as *deferred*. See "Providing an Alternative to dlopen" on page 108. This tagging also provides a means of changing the deferred dependency at runtime.

The use of RTLD_DEFAULT or RTLD_PROBE provide a more robust 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, the application prog can contain the following code fragment.

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

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

In this case, 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 can be 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 (NULL);
                }
        }

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

malloc.so.1 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, the same interposition can be achieved using the following commands.

```
$ 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) possibly using `malloc`(3C).

The use of `RTLD_NEXT` within a dynamic executable or preloaded object, provides a predictable 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 a debugging `mdb`(1) module are provided with the Oracle Solaris runtime linker. The debugging library enables you to trace the runtime linking process in more detail. The `mdb`(1) module enables interactive process debugging.

Debugging Facility

The runtime linker provides a debugging facility that allows you to trace the runtime linking of applications and their dependencies in detail. The type of information that is displayed by using this facility is expected to remain constant. However, the exact format of the information might change slightly from release to release.

Some of the debugging output might be unfamiliar to users 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. This environment variable must be augmented with one or more tokens to indicate the type of debugging that is required.

The tokens that are available with LD_DEBUG can be displayed by using LD_DEBUG=help.

```
$ LD_DEBUG=help  prog
```

prog can be any dynamic executable. The process is terminated following the display of the help information, before control transfers to prog. The choice of executable is unimportant.

By default, all debug output is sent to stderr, the standard error output file. Debug output can be directed to a file instead, using the output token. For example, the help text can be captured in a file named rtld-debug.txt.

```
$ LD_DEBUG=help,output=rtld-debug.txt  prog
```

Alternatively, debug output can be redirected by setting the environment variable LD_DEBUG_OUTPUT. When LD_DEBUG_OUTPUT is used, the process identifier is added as a suffix to the output filename.

If LD_DEBUG_OUTPUT and the output token are both specified, LD_DEBUG_OUTPUT takes precedence. If LD_DEBUG_OUTPUT and the output token are both specified, LD_DEBUG_OUTPUT takes precedence. Use of the output token with programs that call fork(2) result in each process writing debug output to the same file. The debug output will become jumbled and incomplete. LD_DEBUG_OUTPUT should be used in such cases to direct debug output for each process to a unique file.

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

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

int 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 on the first call to the function. This relocation 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.

By setting `LD_DEBUG=bindings,detail`, additional information regarding the real and relative addresses of the actual binding locations is provided.

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:.  (RUNPATH/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:.  (RUNPATH/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`. A combination of `symbols` and `bindings` produces a complete picture of the symbol relocation process.

```
$ LD_DEBUG=bindings,symbols  prog
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 omission of a data reference lookup is due to an optimization used when processing copy relocations. See "Copy Relocations" on page 188 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 the debugging information requires familiarity with the internals of the runtime linker. These internals 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 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 facilities 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
  ....
> ::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)
  ---------------------------------------------
    lmco        rtmap       ADDR()     NAME()
    ---------------------------------------------
    [0xc]       0xff3f0fdc 0x00010000 main
    [0xc]       0xff3f1394 0xff280000 /lib/libc.so.1
  ---------------------------------------------
  Lm_list: 0xff3f6f88  (LM_ID_LDSO)
  ---------------------------------------------
    [0xc]       0xff3f0c78 0xff3b0000 /lib/ld.so.1
```

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

```
> 0xff3f9040::Rt_map
Rt_map located at: 0xff3f9040
     NAME: main
 PATHNAME: /export/home/user/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, a break point on the function `foo` might be useful. 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
lmco    rtmap       ADDR()     NAME()
----------------------------------------------
[0xc]   0xff3f0fdc 0x00010000 main
[0xc]   0xff3f1394 0xff280000 /lib/libc.so.1
[0xc]   0xff3f9ca4 0xff380000 ./foo.so.1
[0xc]   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 previous examples provide a basic guide to the debugger module facilities, but the exact commands, usage, and output can change from release to release. Refer to the usage and help information from mdb(1) for the exact facilities that are 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 19.

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 304). 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 facilities. See "Shared Object Processing" on page 29.

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 addition, a version number is strongly recommended to provide for the correct binding of the shared object across a series of software releases. Chapter 9, "Interfaces and Versioning" 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 105 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
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED       0x123         libfoo.so.1

$ cc -o prog main.o ../tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED       0x123         ../tmp/libfoo.so
```

```
$ cc -o prog main.o /usr/tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED       0x123        /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 `elfdump`(1) and referring to the entry that has the `SONAME` tag.

```
$ elfdump -d ../tmp/libfoo.so | grep SONAME
     [1]  SONAME       0x123        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
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED       0x123        libfoo.so

$ cc -o prog main.o ../tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED       0x123        libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
     [1]  NEEDED       0x123        libfoo.so
```

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 96 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 are 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
$ elfdump -d main | grep NEEDED
     [1]  NEEDED       0x123        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 this 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
```

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
```

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 96. 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
$ elfdump -d libfoo.so | egrep "NEEDED|RUNPATH"
    [1]  NEEDED        0x123         libbar.so.1
    [2]  RUNPATH       0x456         /home/me/lib
```

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

The `LD_LIBRARY_PATH` family of environment variables have a more global scope. Any path names specified using these variables 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 these environment variables is strongly discouraged in production software. See "Directories Searched by the Runtime Linker" on page 96 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 as follows.

```
$ 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*. This technique involves associating the interfaces that the filter provides with an alternative shared object. At runtime, the alternative shared object supplies one or more of the interfaces provided by the *filter*. This alternative shared object 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. At link-edit time, 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 that are defined within a shared object can be defined as filters by using the `mapfile` keywords `FILTER` or `AUXILIARY`. Alternatively, a shared object can define all of the interfaces the shared object offers as filters by using the link-editor's `-F` or `-f` options. These techniques are typically used individually, but can also be combined within the same shared object.

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` keyword `FILTER`, or by using the link-editor's `-F` option. This `mapfile` keyword or option, is

qualified with the name of one or more *filtees* that must supply the symbol definition at runtime.

A *filtee* that cannot be processed at runtime is skipped. A standard filter symbol that cannot be located within the *filtee*, also causes the *filtee* to be skipped. In both of these cases, the symbol definition provided by the *filter* is *not* used to satisfy 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` keyword `AUXILIARY`, or by using the link-editor's `-f` option. This `mapfile` keyword or option, is qualified with the name of one or more *filtees* that can supply the symbol definition at runtime.

A *filtee* that cannot be processed at runtime is skipped. An auxiliary filter symbol that cannot be located within the *filtee*, also causes the *filtee* to be skipped. In both of these cases, the symbol definition provided by the *filter* is used to satisfy this symbol lookup.

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`, suppling 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, use the link-editor's `-F` option. To declare individual interfaces of a shared object to be filters, use a link-editor `mapfile` and the `FILTER` keyword.

In the following example, the shared object `filter.so.1` is defined to be a filter. `filter.so.1` offers the symbols `foo` and `bar`, and 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 = NULL;

char *foo()
```

```
{
        return (NULL);
}
$ 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
```

The link-editor can reference the standard filter `filter.so.1` as a dependency when creating a dynamic executable or shared object. The link-editor 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();

void 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` keyword, `FUNCTION`, is used to ensure a symbol table entry for `foo` is created.

```
$ cat filter.c
char *bar = "defined in filter";
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=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. The *filtee* 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 convenient mechanism for defining a subset interface of an existing shared object. Standard filters provide for the creation of 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 Oracle Solaris OS.

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. As `libnsl.so.1` continues to export `gethostname`, the interface of this library continues to 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 “SYMBOL_SCOPE / SYMBOL_VERSION Directives” on page 217.

When generating data symbols within a filter, always associate the data with a section. This association can be produced by defining the symbol within a relocatable object file. This

association can also be produced by defining the symbol within a `mapfile` together with a `size` declaration and *no* `value` declaration. See “SYMBOL_SCOPE / SYMBOL_VERSION Directives” on page 217. The resulting data definition ensures that references from a dynamic executable are established correctly.

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 the attributes of the symbols in the *filtee*. Maintaining attribute consistency ensures that the link-editing process analyzes the filter in a manner that is compatible with the symbol definitions used at runtime. See “Symbol Resolution” on page 38.

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`.

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`, supplying 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, use the link-editor's `-f` option. To declare individual interfaces of a shared object to be auxiliary filters, use a link-editor `mapfile` and the `AUXILIARY` keyword.

In the following example, the shared object `filter.so.1` is defined to be an auxiliary filter. `filter.so.1` offers the symbols `foo` and `bar`, and 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
```

The link-editor can reference the auxiliary filter `filter.so.1` as a dependency when creating a dynamic executable or shared object. The link-editor 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();

void 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 the interface `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
$mapfile_version 2
SYMBOL_SCOPE {
        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, the *filtee* is loaded. The *filtee* is then 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. The *filtee* 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 Oracle Solaris OS to provide optimized functionality within hardware capability, and platform specific shared objects. See “Capability Specific Shared Objects” on page 253, “Instruction Set Specific Shared Objects” on page 255, and “System Specific Shared Objects” on page 257 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.

Filtering Combinations

Individual interfaces that define standard filters, together with individual interfaces that define auxiliary filters, can be defined within the same shared object. This combination of filter definitions is achieved by using the `mapfile` keywords `FILTER` and `AUXILIARY` to assign the required *filtees*.

A shared object that defines all of its interfaces to be filters by using the `-F`, or `-f` option, is either a standard or auxiliary filter.

A shared object can define individual interfaces to act as filters, together with defining all the interfaces of the object to act as a filters. In this case, the individual filtering defined for an interface is processed first. When a *filtee* for an individual interface filter can not be established, the *filtee* defined for all the interfaces of the filter provides a fall back if appropriate.

For example, consider the filter `filter.so.1`. This filter defines that all interfaces act as auxiliary filters against the *filtee* `filtee.so.1` using the link-editor's `-f` option. `filter.so.1` also defines the individual interface `foo` to be a standard filter against the *filtee* `foo.so.1` using the `mapfile` keyword `FILTER`. `filter.so.1` also defines the individual interface `bar` to be an auxiliary filter against the *filtee* `bar.so.1` using the `mapfile` keyword `AUXILIARY`.

An external reference to `foo` results in processing the *filtee* `foo.so.1`. If `foo` can not be found from `foo.so.1`, then no further processing of the filter is carried out. In this case, no fall back processing is performed because `foo` is defined to be a standard filter.

An external reference to `bar` results in processing the *filtee* `bar.so.1`. If `bar` can not be found from `bar.so.1`, then processing falls back to the *filtee* `filtee.so.1`. In this case, fall back processing is performed because `bar` is defined to be an auxiliary filter. If `bar` can not be found from `filtee.so.1`, then the definition of `bar` within the filter `filter.so.1` is finally used to resolve the external reference.

Filtee Processing

The runtime linker's processing of a filter defers loading 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 the *filtee* is 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 all *filtees* within a process, can be triggered by setting the `LD_LOADFLTR` environment variable to any value.

PART II

Quick Reference

♦ ♦ ♦ CHAPTER 5

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 18 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 `cc`(1) is used. See “Using a Compiler Driver” on page 25.

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 27.

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 30. This provides additional flexibility in allowing search paths to be specified using the `-L` option. See “Directories Searched by the Link-Editor” on page 32.

Over time, the link-editor has added many features that provide for the creation of high quality objects. These features can enable the object to be used efficiently and reliably in various runtime environments. However, to ensure backward compatibility with existing build environments, many of these features are not enabled by default. For example, features such as direct bindings and lazy loading must be explicitly enabled. The link-editor provides the `-z guidance` option to help simplify the process of selecting which features to apply. When guidance is requested, the link-editor can issue warning guidance messages. These messages suggesting options to use, and other related changes, that can help produce higher quality objects. Guidance messages might change over time, as new features are added to the link-editor, or as better practices are discovered to generate high qualify objects. See `ld`(1).

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 `-r` option.

```
$ ld -r -o temp.o file1.o file2.o file3.o ....
```

Creating a Static Executable

Note - The use of static executables is limited. See "Static Executables" on page 18. Static executables usually contain platform-specific implementation details that restrict the ability of the executable to be run on an alternative platform, or version of the operating system. Many implementations of Oracle Solaris shared objects depend on dynamic linking facilities, such as `dlopen`(3C) and `dlsym`(3C). See "Loading Additional Objects" on page 105. These facilities 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.
- The use of the link-editor `-z guidance` option is recommended. Guidance messages offer suggestions for link-editor options and other actions that can improve the resulting object.
- 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 178. Use the `-z text` option to enforce this requirement.
- Avoid including unused relocatable objects. Or, use the `-z discard-unused=sections` option, which instructs the link-editor to eliminate unreferenced ELF sections. See "Removing Unused Material" on page 181.
- Application registers are a feature of the SPARC architecture which are reserved for use by the end user. SPARC shared objects intended for external use should use the `-xregs=no%appl` option to the C compiler in order to ensure that the shared object does not use any application registers. This makes the application registers available to any external users without compromising the shared object's implementation.
- Establish the shared object's 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 Chapter 9, "Interfaces and Versioning".
- Use a versioned name for the shared object to allow for future upgrades. See "Coordination of Versioned Filenames" on page 250.
- 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 43.
- Avoid unneeded dependencies. Use `ldd` with the `-u` option to detect and remove unneeded dependencies. See "Shared Object Processing" on page 29. Or, use the `-z discard-unused=dependencies` 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 106.
- 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 139.
- If interposing symbols are not used on this object or its dependencies, establish direct binding information with `-B direct`. See Chapter 6, "Direct Bindings".

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 discard-unused=sections -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 136.
- 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 250.

The following example combines the above points.

```
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
    -z discard-unused=sections -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 184: Minimize paging activity, as described in “Minimizing Paging Activity” on page 186: Reduce relocation overhead, especially by minimizing symbolic relocations, as described in “Reducing Symbol Scope” on page 49: Allow access to data through functional interfaces, as described in “Copy Relocations” on page 188.

Creating a Dynamic Executable

- To create a dynamic executable don't use the -G, or -d n options.
- The use of the link-editor -z guidance option is recommended. Guidance messages offer suggestions for link-editor options and other actions that can improve the resulting object.
- 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 106.
- Avoid unneeded dependencies. Use ldd with the -u option to detect and remove unneeded dependencies. See “Shared Object Processing” on page 29. Or, use the -z discard-unused=dependencies 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 Chapter 6, “Direct Bindings”.

The following example combines the above points.

```
$ cc -o prog -R /home/lib -z discard-unused=dependencies -z lazyload -B direct -L. \
    -lfoo file1.o file2.o file3.o ....
```

PART III

Advanced Topics

♦ ♦ ♦ CHAPTER 6

Direct Bindings

As part of constructing a process from a dynamic executable and a number of dependencies, the runtime linker must bind symbol references to symbol definitions. By default, symbol definitions are discovered using a simple search model. Typically, each object is searched, starting with the dynamic executable, and progressing through each dependency in the same order in which the objects are loaded. This model has been in effect since dynamic linking was first introduced. This simple model typically results in all symbol references being bound to one definition. The bound definition is the first definition that is found in the series of dependencies that have been loaded.

Dynamic executables have evolved into far more complex processes than the executables that were developed when dynamic linking was in its infancy. The number of dependencies has grown from tens to hundreds. The number of symbolic interfaces that are referenced between dynamic objects has also grown substantially. The size of symbol names has increased considerably with techniques such as the name mangling used to support languages such as C++. These factors have contributed to an increase in startup time for many applications, as symbol references are bound to symbol definitions.

The increase in the number of symbols within a process has also led to an increase in namespace pollution. Multiple instances of symbols of the same name are becoming more common. Unanticipated, and erroneous bindings that result from multiple instances of the same symbol frequently result in hard to diagnose process failures.

In addition, processes now exist where individual objects of the process need to bind to different instances of multiply defined symbols of the same name.

To address the overhead of the default search model while providing greater symbol binding flexibility, an alternative symbol search model has been created. This model is referred to as *direct binding*.

Direct binding allows for precise binding relationships to be established between the objects of a process. Direct binding relationships can help avoid any accidental namespace clashes, by isolating the associated objects from unintentional bindings. This protection adds to the robustness of the objects within a process, which can help avoid unexpected, hard to diagnose, binding situations.

Direct bindings can affect interposition. Unintentional interposition can be avoided by employing direct bindings. However, intentional interposition can be circumvented by direct bindings.

This chapter describes the direct binding model together with discussing interposition issues that should be considered when converting objects to use this model.

Observing Symbol Bindings

To understand the default symbol search model and compare this model with direct bindings, the following components are used to build a process.

```
$ cat main.c
extern int W(), X();

int main() { return (W() + X()); }
$ cat W.c
extern int b();

int a() { return (1); }
int W() { return (a() - b()); }
$ cat w.c
int b() { return (2); }
$ cat X.c
extern int b();

int a() { return (3); }
int X() { return (a() - b()); }
$ cat x.c
int b() { return (4); }
$ cc -o w.so.1 -G -Kpic w.c
$ cc -o W.so.1 -G -Kpic W.c -R. w.so.1
$ cc -o x.so.1 -G -Kpic x.c
$ cc -o X.so.1 -G -Kpic X.c -R. x.so.1
$ cc -o prog1 -R. main.c W.so.1 X.so.1
```

The components of the application are loaded in the following order.

```
$ ldd prog1
        W.so.1 =>        ./W.so.1
        X.so.1 =>        ./X.so.1
        w.so.1 =>        ./w.so.1
        x.so.1 =>        ./x.so.1
```

Both files `W.so.1` and `X.so.1` define a function that is named `a`. Both files `w.so.1` and `x.so.1` define a function that is named `b`. In addition, both files `W.so.1` and `X.so.1` reference the functions `a` and `b`.

The runtime symbol search, using the default search model, together with the final binding, can be observed by setting the `LD_DEBUG` environment variable. From the runtime linkers diagnostics, the bindings to the functions `a` and `b` can be revealed.

```
$ LD_DEBUG=symbols,bindings prog1
```

```
....
17375: symbol=a;  lookup in file=prog1  [ ELF ]
17375: symbol=a;  lookup in file=./W.so.1  [ ELF ]
17375: binding file=./W.so.1 to file=./W.so.1: symbol 'a'
....
17375: symbol=b;  lookup in file=prog1  [ ELF ]
17375: symbol=b;  lookup in file=./W.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./X.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./w.so.1  [ ELF ]
17375: binding file=./W.so.1 to file=./w.so.1: symbol 'b'
....
17375: symbol=a;  lookup in file=prog1  [ ELF ]
17375: symbol=a;  lookup in file=./W.so.1  [ ELF ]
17375: binding file=./X.so.1 to file=./W.so.1: symbol 'a'
....
17375: symbol=b;  lookup in file=prog1  [ ELF ]
17375: symbol=b;  lookup in file=./W.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./X.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./w.so.1  [ ELF ]
17375: binding file=./X.so.1 to file=./w.so.1: symbol 'b'
```

Each reference to one of the functions `a` or `b`, results in a search for the associated symbol starting with the application `prog1`. Each reference to `a` binds to the first instance of the symbol which is discovered in `W.so.1`. Each reference to `b` binds to the first instance of the symbol which is discovered in `w.so.1`. This example reveals how the function definitions in `W.so.1` and `w.so.1` interpose on the function definitions in `X.so.1` and `x.so.1`. The existence of interposition is an important factor when considering the use of direct bindings. Interposition is covered in detail in the sections that follow.

This example is concise, and the associated diagnostics are easy to follow. However, most applications are far more complex, being constructed from many dynamic components. These components are frequently delivered asynchronously, having been built from separate source bases.

The analysis of the diagnostics from a complex process can be challenging. Another technique for analyzing the interface requirements of dynamic objects is to use the `lari`(1) utility. `lari` analyzes the binding information of a process together with the interface definitions provided by each object. This information allows `lari` to concisely convey interesting information about the symbol dependencies of a process. This information is very useful when analyzing interposition in conjunction with direct bindings.

By default, `lari` conveys information that is considered *interesting*. This information originates from multiple instances of a symbol definition. `lari` reveals the following information for `prog1`.

```
$ lari prog1
[2:2ES]: a(): ./W.so.1
[2:0]: a(): ./X.so.1
[2:2E]: b(): ./w.so.1
[2:0]: b(): ./x.so.1
```

In this example, the process established from `prog1` contains two multiply defined symbols, `a` and `b`. The initial elements of the output diagnostics, those elements that are enclosed in the brackets, describe the associated symbols.

The first decimal value identifies the number of instances of the associated symbol. Two instances of `a` and `b` exist. The second decimal value identifies the number of bindings that have been resolved to this symbol. The symbol definition `a` from `W.so.1` reveals that two bindings have been established to this dependency. Similarly, the symbol definition `b` from `w.so.1` reveals that two bindings have been established to this dependency. The letters that follow the number of bindings, qualify the binding. The letter "`E`" indicates that a binding has been established from an external object. The letter "`S`" indicates that a binding has been established from the same object.

`LD_DEBUG`, `lari`, and the process examples built from these components, are used to further investigate direct binding scenarios in the sections that follow.

Enabling Direct Binding

An object that uses direct bindings maintains the relationship between a symbol reference and the dependency that provided the definition. 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 for a dynamic object is recorded at link-edit time. This information can only be established for the dependencies that are specified with the link-edit of that object. Use the `-z defs` option to ensure that all of the necessary dependencies are provided as part of the link-edit.

Objects that use direct bindings can exist within a process with objects that do not use direct bindings. Those objects that do not use direct bindings use the default symbol search model.

The direct binding of a symbol reference to a symbol definition can be established with one of the following link-editing mechanisms.

- With the `-B direct` option. This option establishes direct bindings between the object being built and all of the objects dependencies. This option also establishes direct bindings between any symbol reference and symbol definition within the object being built.

  The use of the `-B direct` option also enables lazy loading. This enabling is equivalent to adding the `-z lazyload` option to the front of the link-edit command line. This attribute was introduced in "Lazy Loading of Dynamic Dependencies" on page 106.
- With the `-z direct` option. This option establishes direct bindings from the object being built to 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. This option does not establish direct bindings between any symbol reference and symbol definition within the object being built.

- With the `DIRECT` `mapfile` keyword. This keyword provides for directly binding individual symbols. This keyword is described in "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

Note - Direct bindings can be disabled at runtime by setting the environment variable `LD_NODIRECT` to a non-null value. By setting this environment variable, all symbol binding within a process is carried out using the default search model.

The following sections describe the use of each of the direct binding mechanisms.

Using the -B direct Option

The `-B direct` option provides the simplest mechanism of enabling direct binding for any dynamic object. This option establishes direct bindings to any dependencies, and within the object being built.

From the components used in the previous example, a directly bound object, `W.so.2`, can be produced.

```
$ cc -o W.so.2 -G -Kpic W.c -R. -Bdirect w.so.1
$ cc -o prog2 -R. main.c W.so.2 X.so.1
```

The direct binding information is maintained in a symbol information section, `.SUNW_syminfo`, within `W.so.2`. This section can be viewed with `elfdump(1)`.

```
$ elfdump -y W.so.2
     [6]  DB          <self>         a
     [7]  DBL     [1] w.so.1         b
```

The letters "`DB`" indicates a direct binding has been recorded for the associated symbol. The function `a` has been bound to the containing object `W.so.2`. The function `b` has been bound directly to the dependency `w.so.1`. The letter "`L`" indicates that the dependency `w.so.1` should also be lazily loaded.

The direct bindings that are established for `W.so.2` can be observed using the `LD_DEBUG` environment variable. The `detail` token adds additional information to the binding diagnostics. For `W.so.2`, this token indicates the direct nature of the binding. The `detail` token also provides additional information about the binding addresses. For simplification, this address information has been omitted from the output generated from the following examples.

```
$ LD_DEBUG=symbols,bindings,detail  prog2
....
18452: symbol=a;  lookup in file=./W.so.2  [ ELF ]
18452: binding file=./W.so.2 to file=./W.so.2: symbol 'a'  (direct)
```

```
18452: symbol=b;  lookup in file=./w.so.1  [ ELF ]
18452: binding file=./W.so.2 to file=./w.so.1: symbol 'b'  (direct)
```

The lari(1) utility can also reveal the direct binding information.

```
$ lari prog2
[2:2ESD]: a(): ./W.so.2
[2:0]: a(): ./X.so.1
[2:2ED]: b(): ./w.so.1
[2:0]: b(): ./x.so.1
```

The letter "D" indicates that the function a defined by W.so.2 has been bound to directly. Similarly, the function b defined in w.so.1 has been bound to directly.

Note - The direct binding of W.so.2 to W.so.2 for the function a results in a similar effect as would be created had the -B symbolic option been used to build W.so.2. However, the -B symbolic option causes references such as a, that can be resolved internally, to be finalized at link-edit time. This symbol resolution leaves no binding to resolve at runtime.

Unlike -B symbolic bindings, a -B direct binding is left for resolution at runtime. Therefore, this binding can be overridden by explicit interposition, or disabled by setting the environment variable LD_NODIRECT to a non-null value.

Symbolic bindings have often been employed to reduce the runtime relocation overhead incurred when loading complex objects. Direct bindings can be used to establish exactly the same symbol bindings. However, a runtime relocation is still required to create each direct binding. Direct bindings require more overhead than symbolic bindings, but provide for greater flexibility.

Using the -z direct Option

The -z direct option provides a mechanism of establishing direct bindings to any dependencies that follow the option on the link-edit command line. Unlike the -B direct option, no direct bindings are established within the object that is being built.

This option is well suited for building objects that are designed to be interposed upon. For example, shared objects are sometimes designed that contain a number of default, or fall back, interfaces. Applications are free to define their own definitions of these interfaces with the intent that the application definitions are bound to at runtime. To allow an application to interpose on the interfaces of a shared object, build the shared object using the -z direct option rather than the -B direct option.

The -z direct option is also useful if you want to be selective over directly binding to one or more dependencies. The -z nodirect option allows you to toggle the use of direct bindings between the dependencies supplied with a link-edit.

From the components used in the previous example, a directly bound object `X.so.2` can be produced.

```
$ cc -o X.so.2 -G -Kpic X.c -R. -zdirect x.so.1
$ cc -o prog3 -R. main.c W.so.2 X.so.2
```

The direct binding information can be viewed with `elfdump`(1).

```
$ elfdump -y X.so.2
     [6]  D           <self>         a
     [7]  DB      [1] x.so.1         b
```

The function `b` has been bound directly to the dependency `x.so.1`. The function `a`is defined as having a potential direct binding, "`D`", with the object `X.so.2`, but no direct binding is established.

The `LD_DEBUG` environment variable can be used to observe the runtime bindings.

```
$ LD_DEBUG=symbols,bindings,detail  prog3
....
06177: symbol=a;  lookup in file=prog3  [ ELF ]
06177: symbol=a;  lookup in file=./W.so.2  [ ELF ]
06177: binding file=./X.so.2 to file=./W.so.2: symbol 'a'
06177: symbol=b;  lookup in file=./x.so.1  [ ELF ]
06177: binding file=./X.so.2 to file=./x.so.1: symbol 'b'  (direct)
```

The `lari`(1) utility can also reveal the direct binding information.

```
$ lari prog3
[2:2ESD]: a(): ./W.so.2
[2:0]: a(): ./X.so.2
[2:1ED]: b(): ./w.so.1
[2:1ED]: b(): ./x.so.1
```

The function `a` defined by `W.so.2` continues to satisfy the default symbol reference made by `X.so.2`. However, the function `b` defined in `x.so.1` has now been bound to directly from the reference made by `X.so.2`.

Using the `DIRECT` `mapfile` Keyword

The `DIRECT` `mapfile` keyword provides a means of establishing a direct binding for individual symbols. This mechanism is intended for specialized link-editing scenarios.

From the components used in the previous example, the function `main` references the external functions `W` and `X`. The binding of these functions follow the default search model.

```
$ LD_DEBUG=symbols,bindings  prog3
....
18754: symbol=W;  lookup in file=prog3  [ ELF ]
```

```
18754: symbol=W;  lookup in file=./W.so.2  [ ELF ]
18754: binding file=prog3 to file=./W.so.2: symbol 'W'
....
18754: symbol=X;  lookup in file=prog3  [ ELF ]
18754: symbol=X;  lookup in file=./W.so.2  [ ELF ]
18754: symbol=X;  lookup in file=./X.so.2  [ ELF ]
18754: binding file=prog3 to file=./X.so.2: symbol 'X'
```

`prog3` can be rebuilt with `DIRECT` `mapfile` keywords so that direct bindings are established to the functions `W` and `X`.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                W       { FLAGS = EXTERN DIRECT };
                X       { FLAGS = EXTERN DIRECT };
};
$ cc -o prog4 -R. main.c W.so.2 X.so.2 -Mmapfile
```

The `LD_DEBUG` environment variable can be used to observe the runtime bindings.

```
$ LD_DEBUG=symbols,bindings,detail prog4
....
23432: symbol=W;  lookup in file=./W.so.2  [ ELF ]
23432: binding file=prog4 to file=./W.so.2: symbol 'W'  (direct)
23432: symbol=X;  lookup in file=./X.so.2  [ ELF ]
23432: binding file=prog4 to file=./x.so.2: symbol 'X'  (direct)
```

The `lari`(1) utility can also reveal the direct binding information. However in this case, the functions `W` and `X` are not multiply defined. Therefore, by default `lari` does not find these functions interesting. The `-a` option must be used to display all symbol information.

```
$ lari -a prog4
....
[1:1ED]: W(): ./W.so.2
....
[2:1ED]: X(): ./X.so.2
....
```

Note - The same direct binding to `W.so.2` and `X.so.1`, can be produced by building `prog4` with the `-B direct` option or the `-z direct` option. The intent of this example is solely to convey how the `mapfile` keyword can be used.

Direct Bindings and Interposition

Interposition can occur when multiple instances of a symbol, having the same name, exist in different dynamic objects that have been loaded into a process. Under the default search model,

symbol references are bound to the first definition that is found in the series of dependencies that have been loaded. This first symbol is said to interpose on the other symbols of the same name.

Direct bindings can circumvent any implicit interposition. As the directly bound reference is searched for in the dependency associated with the reference, the default symbol search model that enables interposition, is bypassed. In a directly bound environment, bindings can be established to different definitions of a symbol that have the same name.

The ability to bind to different definitions of a symbol that have the same name is a feature of direct binding that can be very useful. However, should an application depend upon an instance of interposition, the use of direct bindings can subvert the applications expected execution. Before deciding to use direct bindings with an existing application, the application should be analyzed to determine whether interposition exists.

To determine whether interposition is possible within an application, use `lari`(1). By default, `lari` conveys *interesting* information. This information originates from multiple instances of a symbol definition, which in turn can lead to interposition.

Interposition only occurs when one instance of the symbol is bound to. Multiple instances of a symbol that are called out by `lari` might not be involved in interposition. Other multiple instance symbols can exist, but might not be referenced. These unreferenced symbols are still candidates for interposition, as future code development might result in references to these symbols. All instances of multiply defined symbols should be analyzed when considering the use of direct bindings.

If multiple instances of a symbol of the same name exist, especially if interposition is observed, one of the following actions should be performed.

- Localize symbol instances to remove namespace collision.
- Remove the multiple instances to leave one symbol definition.
- Define any interposition requirement explicitly.
- Identify symbols that can be interposed upon to prevent the symbol from being directly bound to.

The following sections explore these actions in greater detail.

Localizing Symbol Instances

Multiply defined symbols of the same name that provide different implementations, should be isolated to avoid accidental interposition. The simplest way to remove a symbol from the interfaces that are exported by an object, is to reduce the symbol to local. Demoting a symbol to local can be achieved by defining the symbol “`static`”, or possibly through the use of symbol attributes provided by the compilers.

A symbol can also be reduced to local by using the link-editor and a `mapfile`. The following example shows a `mapfile` that reduces the global function `error` to a local symbol by using the `local` scoping directive.

```
$ cc -o A.so.1 -G -Kpic error.c a.c b.c ....
$ elfdump -sN.symtab A.so.1 | fgrep error
    [36]     0x2d0   0x14  FUNC GLOB  D    0 .text      error
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                error;
};
$ cc -o A.so.2 -G -Kpic -M mapfile error.c a.c b.c ....
$ elfdump -sN.symtab A.so.2 | fgrep error
    [24]     0x2c8   0x14  FUNC LOCL  H    0 .text      error
```

Although individual symbols can be reduced to locals using explicit `mapfile` definitions, defining the entire interface family through symbol versioning is recommended. See Chapter 9, “Interfaces and Versioning”.

Versioning is a useful technique typically employed to identify the interfaces that are exported from shared objects. Similarly, dynamic executables can be versioned to define their exported interfaces. A dynamic executable need only export the interfaces that must be made available for the dependencies of the object to bind to. Frequently, the code that you add to a dynamic executable need export no interfaces.

The removal of exported interfaces from a dynamic executable should take into account any symbol definitions that have been established by the compiler drivers. These definitions originate from auxiliary files that the compiler drivers add to the final link-edit. See “Using a Compiler Driver” on page 25.

The following example `mapfile` exports a common set of symbol definitions that a compiler driver might establish, while demoting all other global definitions to local.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                __Argv;
                __environ_lock;
                _environ;
                _lib_version;
                environ;
        local:
                *;
};
```

You should determine the symbol definitions that your compiler driver establishes. Any of these definitions that are used within the dynamic executable should remain global.

By removing any exported interfaces from a dynamic executable, the executable is protected from future interposition issues than might occur as the objects dependencies evolve.

Removing Multiply Defined Symbols of the Same Name

Multiply defined symbols of the same name can be problematic within a directly bound environment, if the implementation associated with the symbol maintains state. Data symbols are the typical offenders in this regard, however functions that maintain state can also be problematic.

In a directly bound environment, multiple instances of the same symbol can be bound to. Therefore, different binding instances can manipulate different state variables that were originally intended to be a single instance within a process.

For example, suppose that two shared objects contain the same data item `errval`. Suppose also, that two functions `action` and `inspect`, exist in different shared objects. These functions expect to write and read the value `errval` respectively.

With the default search model, one definition of `errval` would interpose on the other definition. Both functions `action` and `inspect` would be bound to the same instance of `errval`. Therefore, if an error code was written to `errval` by `action`, then `inspect` could read, and act upon this error condition.

However, suppose the objects containing `action` and `inspect` were bound to different dependencies that each defined `errval`. Within a directly bound environment, these functions are bound to different definitions of `errval`. An error code can be written to one instance of `errval` by `action` while `inspect` reads the other, uninitialized definition of `errval`. The outcome is that `inspect` detects no error condition to act upon.

Multiple instances of data symbols typically occur when the symbols are declared in headers.

```
int bar;
```

This data declaration results in a data item being produced by each compilation unit that includes the header. The resulting *tentative* data item can result in multiple instances of the symbol being defined in different dynamic objects.

However, by explicitly defining the data item as external, *references* to the data item are produced for each compilation unit that includes the header.

```
extern int bar;
```

These references can then be resolved to one data instance at runtime.

Occasionally, the interface for a symbol implementation that you want to remove, should be preserved. Multiple instances of the same interface can be vectored to one implementation, while preserving any existing interface. This model can be achieved by creating individual symbol filters by using a `FILTER` `mapfile` keyword. This keyword is described in “SYMBOL_SCOPE / SYMBOL_VERSION Directives” on page 217.

Creating individual symbol filters is useful when dependencies expect to find a symbol in an object where the implementation for that symbol has been removed.

For example, suppose the function `error` exists in two shared objects, `A.so.1` and `B.so.1`. To remove the symbol duplication, you want to remove the implementation from `A.so.1`. However, other dependencies are relying on `error` being provided from `A.so.1`. The following example shows the definition of `error` in `A.so.1`. A `mapfile` is then used to allow the removal of the `error` implementation, while leaving a filter for this symbol that is directed to `B.so.1`.

```
$ cc -o A.so.1 -G -Kpic error.c a.c b.c ....
$ elfdump -sN.dynsym A.so.1 | fgrep error
     [3]     0x300    0x14  FUNC GLOB  D    0 .text      error
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                error { TYPE=FUNCTION; FILTER=B.so.1 };
};
$ cc -o A.so.2 -G -Kpic -M mapfile a.c b.c ....
$ elfdump -sN.dynsym A.so.2 | fgrep error
     [3]        0       0  FUNC GLOB  D    0 ABS        error
$ elfdump -y A.so.2 | fgrep error
     [3]  F       [0] B.so.1         error
```

The function `error` is global, and remains an exported interface of `A.so.2`. However, any runtime binding to this symbol is vectored to the *filtee* `B.so.1`. The letter "`F`" indicates the filter nature of this symbol.

This model of preserving existing interfaces, while vectoring to one implementation has been used in several Oracle Solaris libraries. For example, a number of math interfaces that were once defined in `libc.so.1` are now vectored to the preferred implementation of the functions in `libm.so.2`.

Defining Explicit Interposition

The default search model can result in instances of the same named symbol interposing on later instances of the same name. Even without any explicit labelling, interposition still occurs, so that one symbol definition is bound to from all references. This *implicit* interposition occurs as a consequence of the symbol search, not because of any explicit instruction the runtime linker has been given. This implicit interposition can be circumvented by direct bindings.

Although direct bindings work to resolve a symbol reference directly to an associated symbol definition, *explicit* interposition is processed prior to any direct binding search. Therefore, even within a direct binding environment, interposers can be designed, and be expected to interpose on any direct binding associations. Interposers can be explicitly defined using the following techniques.

- With the `LD_PRELOAD` environment variable.

- With the link-editors `-z interpose` option.
- With the `INTERPOSE mapfile` keyword.
- As a consequence of a `singleton` symbol definition.

The interposition facilities of the `LD_PRELOAD` environment variable, and the `-z interpose` option, have been available for some time. See "Runtime Interposition" on page 102. As these objects are explicitly defined to be interposers, the runtime linker inspects these objects before processing any direct binding.

Interposition that is established for a shared object applies to all the interfaces of that dynamic object. This object interposition is established when a object is loaded using the `LD_PRELOAD` environment variable. Object interposition is also established when an object that has been built with the `-z interpose` option, is loaded. This object model is important when techniques such as `dlsym`(3C) with the special handle `RTLD_NEXT` are used. An interposing object should always have a consistent view of the *next* object.

A dynamic executable has additional flexibility, in that the executable can define individual interposing symbols using the `INTERPOSE mapfile` keyword. Because a dynamic executable is the first object loaded in a process, the executables view of the next object is always consistent.

The following example shows an application that explicitly wants to interpose on the `exit` function.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                exit    { FLAGS = INTERPOSE };
};
$ cc -o prog -M mapfile exit.c a.c b.c ....
$ elfdump -y prog | fgrep exit
[6]  DI          <self>         exit
```

The letter "`I`" indicates the interposing nature of this symbol. Presumably, the implementation of this `exit` function directly references the system function `_exit`, or calls through to the system function `exit` using `dlsym` with the `RTLD_NEXT` handle.

At first, you might consider identifying this object using the `-z interpose` option. However, this technique is rather heavy weight, because all of the interfaces exported by the application would act as interposers. A better alternative would be to localize all of the symbols provided by the application except for the interposer, together with using the `-z interpose` option.

However, use of the `INTERPOSE mapfile` keyword provides greater flexibility. The use of this keyword allows an application to export several interfaces while selecting those interfaces that should act as interposers.

Symbols that are assigned the `STV_SINGLETON` visibility effectively provide a form of interposition. See Table 12-23. These symbols can be assigned by the compilation system to

an implementation that might become multiply instantiated in a number of objects within a process. All references to a `singleton` symbol are bound to the first occurrence of a `singleton` symbol within a process.

Preventing a Symbol from being Directly Bound to

Direct bindings can be overridden with explicit interposition. See "Defining Explicit Interposition" on page 168. However, cases can exist where you do not have control over establishing explicit interposition.

For example, you might deliver a family of shared objects that you would like to use direct bindings. Customers are known to be interposing on symbols that are provided by shared objects of this family. If these customers have not explicitly defined their interpositioning requirements, their interpositioning can be compromised by a re-delivery of shared objects that employ direct bindings.

Shared objects can also be designed that provide a number of default interfaces, with an expectation that users provide their own interposing routines.

To prevent disrupting existing applications, shared objects can be delivered that explicitly prevent directly binding to one or more of their interfaces.

Directly binding to a dynamic object can be prevented using one of the following options.

- With the `-B nodirect` option. This option prevents directly binding to any interfaces that are offered by the object being built.
- With the `NODIRECT` `mapfile` keyword. This keyword provides for preventing direct binding to individual symbols. This keyword is described in "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.
- As a consequence of a `singleton` symbol definition.

An interface that is labelled as `nodirect`, can not be directly bound to from an external object. In addition, an interface that is labelled as `nodirect`, can not be directly bound to from within the same object.

The following sections describe the use of each of the direct binding prevention mechanisms.

Using the `-B nodirect` Option

The `-B nodirect` option provides the simplest mechanism of preventing direct binding from any dynamic object. This option prevents direct binding from any other object, and from within the object being built.

The following components are used to build three shared objects, `A.so.1`, `O.so.1` and `X.so.1`. The `-B nodirect` option is used to prevent `A.so.1` from directly binding to `O.so.1`. However, `O.so.1` can continue to establish direct bindings to `X.so.1` using the `-z direct` option.

```
$ cat a.c
extern int o(), p(), x(), y();

int a() { return (o() + p() - x() - y()); }
$ cat o.c
extern int x(), y();

int o() { return (x()); }
int p() { return (y()); }
$ cat x.c
int x() { return (1); }
int y() { return (2); }
$ cc -o X.so.1 -G -Kpic x.c
$ cc -o O.so.1 -G -Kpic o.c -Bnodirect -zdirect -R. X.so.1
$ cc -o A.so.1 -G -Kpic a.c -Bdirect -R. O.so.1 X.so.1
```

The symbol information for `A.so.1` and `O.so.1` can be viewed with `elfdump`(1).

```
$ elfdump -y A.so.1
     [1]  DBL     [3] X.so.1           x
     [5]  DBL     [3] X.so.1           y
     [6]  DL      [1] O.so.1           o
     [9]  DL      [1] O.so.1           p
$ elfdump -y O.so.1
     [3]  DB      [0] X.so.1           x
     [4]  DB      [0] X.so.1           y
     [6]  N                            o
     [7]  N                            p
```

The letter "`N`" indicates that no direct bindings be allowed to the functions `o` and `p`. Even though `A.so.1` has requested direct bindings by using the `-B direct` option, direct bindings have not be established to the functions `o` and `p`. `O.so.1` can still request direct bindings to its dependency `X.so.1` using the `-z direct` option.

The Oracle Solaris library `libproc.so.1` is built with the `-B nodirect` option. Users of this library are expected to provide their own call back interfaces for many of the `libproc` functions. References to the `libproc` functions from any dependencies of `libproc` should bind to any user definitions when such definitions exist.

Using the `NODIRECT` `mapfile` Keyword

The `NODIRECT` `mapfile` keyword provides a means of preventing a direct binding to individual symbols. This keyword allows for more fine grained control over preventing direct binding than the `-B nodirect` option.

From the components used in the previous example, `O.so.2` can be built to prevent direct binding to the function `o`.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                o       { FLAGS = NODIRECT };
};
$ cc -o O.so.2 -G -Kpic o.c -Mmapfile -zdirect -R. X.so.1
$ cc -o A.so.2 -G -Kpic a.c -Bdirect -R. O.so.2 X.so.1
```

The symbol information for `A.so.2` and `O.so.2` can be viewed with `elfdump`(1).

```
$ elfdump -y A.so.2
     [1]  DBL     [3] X.so.1           x
     [5]  DBL     [3] X.so.1           y
     [6]  DL      [1] O.so.1           o
     [9]  DBL     [1] O.so.1           p
$ elfdump -y O.so.1
     [3]  DB      [0] X.so.1           x
     [4]  DB      [0] X.so.1           y
     [6]  N                            o
     [7]  D           <self>           p
```

`O.so.1` only declares that the function `o` can not be directly bound to. Therefore, `A.so.2` is able to directly bind to the function `p` in `O.so.1`.

Several individual interfaces within the Oracle Solaris libraries have been defined to not allow direct binding. One example is the data item `errno`. This data item is defined in `libc.so.1`. This data item can be referenced by including the header file `stdio.h`. However, many applications were commonly taught to defined their own `errno`. These applications would be compromised if a family of system libraries were delivered which directly bound to the `errno` that is defined in `libc.so.1`.

Another family of interfaces that have been defined to prevent direct binding to, are the `malloc`(3C) family. The `malloc` family are another set of interfaces that are frequently implemented within user applications. These user implementations are intended to interpose upon any system definitions.

Note - Various system interposing libraries are provided with the Oracle Solaris OS that provide alternative `malloc` implementations. In addition, each implementation expects to be the only implementation used within a process. All of the `malloc` interposing libraries have been built with the `-z interpose` option. This option is not really necessary as the `malloc` family within `libc.so.1` have been labelled to prevent any direct binding

However, the interposing libraries have been built with `-z interpose` to set a precedent for building interposers. This explicit interposition has no adverse interaction with the direct binding prevention definitions established within `libc.so.1`.

Symbols that are assigned the `STV_SINGLETON` visibility can not be directly bound to. See Table 12-23. These symbols can be assigned by the compilation system to an implementation that might become multiply instantiated in a number of objects within a process. All references to a `singleton` symbol are bound to the first occurrence of a `singleton` symbol within a process.

♦ ♦ ♦ CHAPTER 7

Building Objects to Optimize System Performance

Dynamic executables and shared objects require runtime processing to establish the processes these objects contribute to. Multiple instances of a process can be active at any one time, and shared objects can be used by different processes at the same time. The construction of a dynamic object affects the runtime initialization and potential sharing of the object between processes, and overall system performance.

The following sections investigate the runtime initialization and processing of dynamic objects, examining factors that affect their runtime performance such as text size and purity, and relocation overhead.

Analyzing Files With `elfdump`

Various tools are available to analyze the contents of an ELF file, including the standard Unix utilities `dump`(1), `nm`(1), and `size`(1). Under Oracle Solaris, these tools have been largely superseded by `elfdump`(1).

The use of `eldump` to diagnose the contents of an ELF object can be useful to explore the various performance issues described in the following sections.

The ELF format organizes data into *sections*. Sections are in turn allocated to units known as *segments*. Segments describe how portions of a file are mapped into memory. See `mmapobj`(2). These loadable segments can be displayed by using the `elfdump`(1) command and examining the `PT_LOAD` entries.

```
$ elfdump -p -NPT_LOAD libfoo.so.1
Program Header[0]:
    p_vaddr:      0           p_flags:    [ PF_X PF_R ]
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x53c       p_memsz:    0x53c
    p_offset:     0           p_align:    0x10000

Program Header[1]:
    p_vaddr:      0x1053c     p_flags:    [ PF_X PF_W PF_R ]
```

```
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x114       p_memsz:    0x13c
    p_offset:     0x53c       p_align:    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, `PF_X` and `PF_R`. The data segment is mapped to also allow its contents to be modified, `PF_W`. The memory size, `p_memsz`, of the data segment differs from the file size, `p_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 the `-s` option to `elfdump`.

```
$ elfdump -sN.symtab libfoo.so.1

Symbol Table Section:  .symtab
   index    value     size  type bind oth ver shndx          name
    ....
    [36]  0x10628     0x28  OBJT GLOB  D    0 .data          data
    ....
    [38]  0x10650     0x28  OBJT GLOB  D    0 .bss           bss
    ....
    [40]    0x520      0xc  FUNC GLOB  D    0 .init          _init
    ....
    [44]    0x508     0x14  FUNC GLOB  D    0 .text          foo
    ....
    [46]    0x52c      0xc  FUNC GLOB  D    0 .fini          _fini
```

The symbol table information displayed by `elfdump` includes the section the symbol is associated with. The `elfdump -c` option can be used to display information about these sections.

```
$ elfdump -c libfoo.so.1
....
Section Header[6]:  sh_name: .text
    sh_addr:      0x4f8           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0x28            sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x4f8           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8

Section Header[7]:  sh_name: .init
    sh_addr:      0x520           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0xc             sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x520           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4

Section Header[8]:  sh_name: .fini
    sh_addr:      0x52c           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0xc             sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x52c           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4
....
```

```
Section Header[12]:  sh_name: .data
   sh_addr:      0x10628         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
   sh_size:      0x28            sh_type:    [ SHT_PROGBITS ]
   sh_offset:    0x628           sh_entsize: 0
   sh_link:      0               sh_info:    0
   sh_addralign: 0x4
....
Section Header[14]:  sh_name: .bss
   sh_addr:      0x10650         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
   sh_size:      0x28            sh_type:    [ SHT_NOBITS ]
   sh_offset:    0x650           sh_entsize: 0
   sh_link:      0               sh_info:    0
   sh_addralign: 0x4
....
```

The output from elfdump(1) in the previous 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.

Underlying System

Applications are built from a dynamic executable and one or more shared object dependencies. The entire loadable contents of the dynamic executable and the shared objects are mapped into the virtual address space of that process at runtime. Each process starts by referencing a single copy of the dynamic executable and the shared objects in memory.

Relocations within the dynamic objects 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 objects were 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 objects shares memory among processes at the granularity of a page. Memory pages can be shared between processes as long as the pages are not modified at runtime. If a process writes to a page of an object when writing a data item, or relocating a reference to a shared object, a private copy of that page is generated. This private copy has no effect on other users of the 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 dynamic 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 With elfdump” on page 175 on how to obtain this information from an ELF file.
An overriding goal when developing a dynamic object is to maximize the text segment and

minimize the data segment. This partitioning optimizes the amount of code sharing while reducing the amount of processing needed to initialize and use the dynamic 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 dependencies first reference, by establishing the object as lazy loadable. See “Lazy Loading of Dynamic Dependencies” on page 106.

For small applications, a typical thread of execution can reference all the applications dependencies. The application loads all of its dependencies whether the dependencies 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. Dependencies that are not referenced for a particular thread of execution, are not loaded.

Position-Independent Code

The code within a dynamic executable is typically *position-dependent*, and is tied to a fixed address in memory. Shared objects, on the other hand, can be loaded at different addresses in different processes. Position-*independent* code is not tied to a specific address. This independence allows the code to execute efficiently at a different address in each process that uses the code. Position-independent code is recommended for the creation of shared objects.

The compiler can generate position-independent code under the `-K pic` option.

If a shared object is built from position-dependent code, the text segment can require modification at runtime. This modification allows relocatable references to be assigned to the location that the object has been loaded. The relocation of the text segment requires the segment to be remapped as writable. This modification requires a swap space reservation, and results in a private copy of the text segment for the process. The text segment is no longer sharable between multiple processes. Position-dependent code typically requires more runtime relocations than the corresponding position-independent code. Overall, the overhead of processing text relocations can cause serious performance degradation.

When a shared object is built from position-independent code, relocatable references are generated as indirections through data in the shared object's data segment. The code within the text segment requires no modification. All relocation updates are applied to corresponding entries within the data segment. See “Global Offset Table (Processor-Specific)” on page 415

and “Procedure Linkage Table (Processor-Specific)” on page 416 for more details on the specific indirection techniques.

The runtime linker attempts to handle text relocations should these relocations exist. However, some relocations can not be satisfied at runtime.

The x64 position-dependent code sequence can generate code which can only be loaded into the lower 32–bits of memory. The upper 32–bits of any address must all be zeros. Since shared objects are typically loaded at the top of memory, the upper 32–bits of an address are required. Position-dependent code within an x64 shared object is therefore insufficient to cope with relocation requirements. Use of such code within a shared object can result in runtime relocation errors.

```
$ prog
ld.so.1: prog: fatal: relocation error: R_AMD64_32: file \
    libfoo.so.1: symbol (unknown): value 0xfffffd7fff0cd457 does not fit
```

Position-independent code can be loaded in any region in memory, and hence satisfies the requirements of shared objects for x64.

This situation differs from the default `ABS64` mode that is used for 64–bit SPARCV9 code. This position-dependent code is typically compatible with the full 64–bit address range. Thus, position-dependent code sequences can exist within SPARCV9 shared objects. Use of either the `ABS32` mode, or `ABS44` mode for 64–bit SPARCV9 code, can still result in relocations that can not be resolved at runtime. However, each of these modes require the runtime linker to relocate the text segment.

Regardless of the runtime linkers facilities, or differences in relocation requirements, shared objects should be built using position-independent code.

You can identify a shared object that requires relocations against its text segment. The following example uses `elfdump`(1) to determine whether a `TEXTREL` entry dynamic entry exists.

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ elfdump -d libfoo.so.1 | grep TEXTREL
    [9]  TEXTREL       0
```

Note - The value of the `TEXTREL` entry is irrelevant. The presence of this entry 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 position-dependent code used as input. The following example shows how position-dependent code results in a failure to generate a shared object.

```
$ 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 position-dependent 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`.

Text relocations within a shared object can also occur when hand written assembler code is included and does not include 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.

`-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 415.

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.

Removing Unused Material

The inclusion of functions and data from input relocatable object files, when this material is not used by the object being built, is wasteful. This unneeded material causes the object to be larger than necessary, resulting in added overhead when the object is used at runtime.

References to unused shared object dependencies are also wasteful. Particularly in the absence of lazy loading, these references result in the unnecessary loading and processing of these shared objects at runtime.

Unused sections, unused relocatable object files, and unused shared object dependencies can be diagnosed during a link-edit by using the link-editors debugging option `-D unused`.

Unused files and dependencies are also diagnosed when using the `-z guidance` option.

Unused sections, unused files, and unused dependencies should be removed from the link-edit. This removal reduces the cost of the link-edit, and reduces the runtime cost of using the object being built. However, if removing these items is problematic, unused material can be discarded from the object being built by using the `-z discard-unused` option.

Removing Unused Sections

An ELF section, from an input relocatable object file, is determined to be unused when three conditions are true.

- The section provides no global symbols.
- The section contributes to an allocatable segment.
- The section is not referenced by any other used section, from any object, that contributes to the link-edit.

Unused sections can be discarded from the link-edit by using the `-z discard-unused=sections` option.

You can improve the link-editor's ability to diagnose and discard unused sections by defining the dynamic object's external interfaces. See Chapter 9, “Interfaces and Versioning”. 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 then clearly identified as candidates for discarding.

Individual functions and data variables can be discarded by the link-editor if these items are assigned to their own sections. This section refinement can be achieved by using the `-xF` compiler option.

Removing Unused Files

An input relocatable object file is determined to be unused if all allocatable sections provided by the relocatable object are unused.

Unused files are diagnosed with the `-z guidance` option, and can be discarded from the link-edit by using the `-z discard-unused=files` option.

The `-z discard-unused` option provides independent control over unused sections and unused files in order to compliment `-z guidance` processing. Under `-z guidance`, files that are determined to be unused are identified. Unused files can often easily be removed from a link-edit. However, sections that are determined to be unused are not identified under `-z guidance` processing. Unused sections can involve much more investigation and effort to remove and can be a consequence of compiler actions that are beyond your control.

By using the `-z discard-unused=sections` option together with the `-z guidance` option, unused sections are automatically removed, while unused files are identified for you to remove from the link-edit.

Removing Unused Dependencies

An explicit, shared object dependency is one that is defined on the command line, either using the path name, or more commonly by using the `-l` option. Explicit dependencies include those that might be provided by the compiler drivers, such as `-lc`.

Implicit dependencies are the dependencies of explicit dependencies. Implicit dependencies can be processed as part of a link-edit to complete the closure of all symbol resolution. This symbol closure ensures that the object being built is self-contained, with no unreferenced symbols remaining.

All dynamic objects should define the dependencies they require. This requirement is enforced by default when building a dynamic executable, but not when building a shared object. Use the `-z defs` option to enforce this requirement when building a shared object.

All dynamic objects should refrain from defining dependencies that they do not require. Loading such unused dependencies at runtime is unnecessary and wasteful.

An explicit dependency is determined to be unused if two conditions are true.

- No global symbols that are provided by the dependency are referenced from the object being built.
- The dependency does not compensate for the requirements of any implicit dependencies.

Unused dependencies are diagnosed with the `-z guidance` option. These dependencies should be removed from the link-edit. However, if removing these items is problematic, unused dependencies can be discarded from the object being built by using the `-z discard-unused=dependencies` option.

Unfortunately, shared objects exist that have not defined all the dependencies they require. In these cases, developers often add the missing dependencies to the executable, or other shared objects they are building, rather than rebuild the original dependency correctly. Such dependencies are referred to as *compensating dependencies*.

For example, consider a shared object, `foo.so`, that references the symbol `bar` from the shared object `bar.so`. However, `foo.so` does not express a dependency upon `bar.so`. An inspection of `foo.so` reveals the lack of the required dependency, as the symbol `bar` can not be found.

```
% ldd -r foo.so
     libc.so.1 =>     /lib/libc.so.1
     symbol not found: bar           (foo.so)
```

Now consider an application developer that wishes to create an executable that references the symbol `foo` from the shared object `foo.so`. The required dependency upon `foo.so` is specified, but the link-edit of the executable fails.

```
% cc -Bdirect -o main main.c -L. -lfoo
Undefined                       first referenced
  symbol                             in file
  bar                                ./libfoo.so
ld: fatal: symbol referencing errors
```

The developer forcibly corrects this situation by adding a compensating dependency on `bar.so`.

```
% cc -Bdirect -o main main.c -L. -lfoo -lbar
```

This correction creates an application that loads all the necessary dependencies at runtime, and therefore appears to resolve the issue. However, the result is fragile. If a future delivery of `foo.so` is made that does not require a symbol from `bar.so`, then this application will load `bar.so` for no reason. The better solution is to correct `foo.so` by adding the missing dependency `bar.so`.

The occurrence of a compensating dependency is diagnosed though guidance.

```
% cc -Bdirect -zguidance -o main main.c -L. -lfoo -lbar
ld: guidance: removal of compensating dependency recommended: libbar.so
```

Compensating dependencies are diagnosed through guidance, but they are not removed under `-z discard-unused=dependencies`. Although the dependency might be unused in relation to the object being created, the dependency is used by other components of the link-edit. To remove this dependency could result in creating an object that can not be executed at runtime.

The need for compensating dependencies can be eliminated by the systematic use of the `-z defs` option to build all dynamic objects.

The `-z ignore` and `-z record` options are positional options that can be used in conjunction with the `-z discard-unused=dependencies` option. These positional options turn the discard feature on and off selectively for targeted objects.

Maximizing Shareability

As mentioned in “Underlying System” on page 177, 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 to use the following definition.

```
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 the array as:

```
const char *rdstrs[] = { .... };
```

ensures 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 that are generated by accessing a shared object can 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 improves locality of reference and has the side effect of reducing the object's overall size.

Relocations

In “Relocation Processing” on page 99, the mechanisms by which the runtime linker relocates dynamic executables and shared objects to create a *runable* process was covered. “Relocation Symbol Lookup” on page 100 and “When Relocations are Performed” on page 187 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 49 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 49. 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 Chapter 6, “Direct Bindings”.

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 188. 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

The relocation sections within relocatable objects are typically maintained in a one-to-one relationship with the sections to which the relocations must be applied. However, when the linker editor creates an executable or shared object, all but the procedure linkage table relocations are placed into a single common section named `.SUNW_reloc`.

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 178 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 instructs the runtime linker to copy the data from the shared object to the 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
$ elfdump -sN.dynsym prog | grep ' sys_'
    [24]   0x21240   0x260  OBJT GLOB  D    1 .bss           sys_errlist
    [39]   0x21230     0x4  OBJT GLOB  D    1 .bss           sys_nerr
$ elfdump -c prog
....
Section Header[19]:  sh_name: .bss
    sh_addr:      0x21230         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
    sh_size:      0x270           sh_type:    [ SHT_NOBITS ]
    sh_offset:    0x1230          sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8
....
$ elfdump -r prog

Relocation Section:  .SUNW_reloc
      type                     offset     addend  section        symbol
  ....
  R_SPARC_COPY               0x21240          0  .SUNW_reloc    sys_errlist
  R_SPARC_COPY               0x21230          0  .SUNW_reloc    sys_nerr
  ....
```

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.

```
$ cat foo.c
int _size_gets_smaller[16];
int _size_gets_larger[16];
$ cc -o libfoo.so -G foo.c
$ cc -o prog main.c -L. -R. -lfoo
$ elfdump -sN.symtab prog | grep _size
    [49]   0x211d0   0x40  OBJT GLOB  D    0 .bss           _size_gets_larger
    [59]   0x21190   0x40  OBJT GLOB  D    0 .bss           _size_gets_smaller
$ elfdump -r prog | grep _size
  R_SPARC_COPY               0x211d0          0  .SUNW_reloc    _size_gets_larger
  R_SPARC_COPY               0x21190          0  .SUNW_reloc    _size_gets_smaller
```

A new version of this shared object is supplied that contains different data sizes for these symbols.

```
$ cat foo2.c
int _size_gets_smaller[4];
int _size_gets_larger[32];
$ cc -o libfoo.so -G foo2.c
$ elfdump -sN.symtab libfoo.so | grep _size
    [37]   0x105cc   0x10  OBJT GLOB  D    0 .bss           _size_gets_smaller
    [41]   0x105dc   0x80  OBJT GLOB  D    0 .bss           _size_gets_larger
```

Running ldd(1) against the dynamic executable reveals the following.

```
$ ldd -d prog
    libfoo.so.1 =>   ./libfoo.so.1
    ....
    relocation R_SPARC_COPY sizes differ: _size_gets_larger
        (file prog size=0x40; file ./libfoo.so size=0x80)
        prog size used; possible data truncation
    relocation R_SPARC_COPY sizes differ: _size_gets_smaller
        (file prog size=0x40; file ./libfoo.so size=0x10)
        ./libfoo.so size used; possible insufficient data copied
....
```

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 178.

Using the -B symbolic Option

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 49. 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 188.

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 187 for details of this mechanism.

The `LD_PROFILE` environment variable specifies the name of a shared object to profile. You can analyze a single shared object 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 previous 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 previous 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 `fork`(2) removes this restriction.

♦ ♦ ♦ CHAPTER 8

Mapfiles

Mapfiles provide a large degree of control over the operation of the link-editor, and the resulting output object.

- Create and/or modify output segments.
- Define how input sections are assigned to segments, and the relative order of those sections.
- Specify symbol scope and/or versioning, creating stable backward compatible interfaces for sharable objects.
- Define the versions to use from sharable object dependencies.
- Set header options in the output object.
- Set process stack attributes for a dynamic executable.
- Set or override hardware and software capabilities.

Note - The link-editor used without a `mapfile` will always produce a valid ELF output file. The `mapfile` option provides the user with a great deal of flexibility and control over the output object, some of which has the potential to produce an invalid or unusable object. The user is expected to have knowledge of the rules and conventions that govern the ELF format.

The `-M` command line option is used to specify the `mapfile` to be used. Multiple *mapfiles* can be used in a single link operation. When more than one `mapfile` is specified, the link-editor processes each one in the order given, as if they represented a single logical `mapfile`. This occurs before any input objects are processed.

The system provides sample *mapfiles* for solving common problems in the `/usr/lib/ld` directory.

Mapfile Structure and Syntax

Mapfile directives can span more than one line, and can have any amount of white space, including new lines.

For all syntax discussions, the following notations apply.

- Spaces, or newlines, can appear anywhere except in the middle of a name or value.
- Comments beginning with a hash character (#) and ending at a newline can appear anywhere that a space can appear. Comments are not interpreted by the link-editor, and exist solely for documentation purposes.
- All directives are terminated by a semicolon (;). The final semicolon within a `{ . . . . }` section can be omitted.
- All entries in constant width, all colons (:), semicolons (;), assignment (`=, +=, -=`), and `{ . . . . }` brackets are typed in literally.
- All entries in *italics* are substitutable.
- `[ . . . . ]` brackets are used to delineate optional syntax. The brackets are not literal, and do not appear in the actual directives.
- Names are case sensitive strings. Table 8-2 contains a list of names and other strings commonly found in *mapfiles*. Names can be specified in three different forms.
  - Unquoted

    An unquoted name is a sequence of letters and digits. The first character must be a letter, followed by zero or more letters or digits. The characters percent (%), slash (/), period (.), and underscore (_) count as a letter. The characters dollar ($), and hyphen (-) count as a digit.
  - Single Quotes

    Within single quotes ('), a name can contain any character other than a single quote, or newline. All characters are interpreted as literal characters. This form of quoting is convenient when specifying file paths, or other names that contain normal printable characters that are not allowed in an unquoted name.
  - Double Quotes

    Within double quotes ("), a name can contain any character other than a double quote, or newline. Backslash(\) is an escape character which operates similarly to the way it is used in the C programming language within a string literal. Characters prefixed by a backslash are replaced by the character they represent, as shown in Table 8-1. Any character following a backslash, other than the ones shown in Table 8-1 is an error.
- *value* represents a numeric value, and can be hexadecimal, decimal, or octal, following the rules used by the C language for integer constants. All values are unsigned integer values, and are 32-bit for 32-bit output objects, and 64-bit for 64-bit output objects.
- *segment_flags* specify memory access permissions as a space separated list of one or more of the values given in Table 8-3, which correspond to the PF_ values defined in `<sys/elf.h>`.

TABLE 8-1 Double Quoted Text Escape Sequences

Escape Sequence	Meaning
`\a`	alert (bell)
`\b`	backspace

Escape Sequence	Meaning
`\f`	*formfeed*
`\n`	*newline*
`\r`	return
`\t`	horizontal tab
`\v`	vertical tab
\\	backslash
\'	single quote
\”	double quote
ooo	An octal constant, where *ooo* is one to three octal digits (0....7)

TABLE 8-2 Names And Other Widely Used Strings Found In *Mapfiles*

Name	Purpose
segment_name	Name of ELF segment
section_name	Name of ELF section
symbol_name	Name of ELF symbol
file_path	A Unix file path of slash (/) delimited names used to reference an ELF object, or an archive that contains ELF objects
file_basename	Final component (basename(1)) of a *file_path*
objname	Either a *file_basename* or the name of an object contained within an archive
soname	Sharable object name, as used for the SONAME of a sharable object (e.g. `libc.so.1`)
version_name	Name of a symbol version, as used within an ELF versioning section
inherited_version_name	Name of a symbol version inherited by another symbol version

TABLE 8-3 Segment Flags

Flag Value	Meaning
`READ`	Segment is readable
`WRITE`	Segment is writable
`EXECUTE`	Segment is executable
`0`	All permission flags are cleared

Flag Value	Meaning
`DATA`	The combination of READ, WRITE, and EXECUTE flags appropriate for a data segment on the target platform
`STACK`	The combination of READ, WRITE, and EXECUTE flags appropriate for the target platform, as defined by the platform ABI

Mapfile Version

The first non-comment, non-empty, line in a *mapfile* is expected to be a *mapfile* version declaration. This declaration establishes the version of the *mapfile* language used by the remainder of the file. The *mapfile* language documented in this manual is version 2.

```
$mapfile_version 2
```

A *mapfile* that does not begin with a version declaration is assumed to be written in the original *mapfile* language defined for System V Release 4 Unix (SVR4) by AT&T. The link-editor retains the ability to process such *mapfiles*. Their syntax is documented in Appendix B, "*System V Release 4 (Version 1) Mapfiles*".

Conditional Input

Lines within a *mapfile* can be conditionalized to only apply to a specific ELFCLASS (32 or 64-bit) or machine type.

```
$if expr
....
[$elif expr]
....
[$else]
....
$endif
```

A conditional input expression evaluates to a logical *true* or *false* value. Each of the directives (`$if`, `$elif`, `$else`, and `$endif`) appear alone on a line. The expressions in `$if` and subsequent `$elif` lines are evaluated in order until an expression that evaluates to *true* is found. Text following a line with a *false* value is discarded. The text following a successful directive line is treated normally. *Text* here refers to any material, that is not part of the conditional structure. Once a successful `$if` or `$elif` has been found, and its text processed, succeeding `$elif` and `$else` lines, together with their text, are discarded. If all the expressions are zero, and there is a `$else`, the text following the `$else` is treated normally.

The scope of an `$if` directive cannot extend across multiple *mapfiles*. An `$if` directive must be terminated by a matching `$endif` within the `mapfile` that uses the `$if` directive, or the link-editor issues an error.

The link-editor maintains an internal table of names that can be used in the logical expressions evaluated by `$if` and `$elif`. At startup, this table is initialized with each of the names in the following table that apply to the output object being created.

TABLE 8-4 Predefined Conditional Expression Names

Name	Meaning
`_ELF32`	32–bit object
`_ELF64`	64–bit object
`_ET_DYN`	shared object
`_ET_EXEC`	executable object
`_ET_REL`	relocatable object
`_sparc`	*Sparc* machine (32 or 64–bit)
`_x86`	x86 machine (32 or 64–bit)
`true`	Always defined

The names are case sensitive, and must be used exactly as shown. For example, `true` is defined, but `TRUE` is not. Any of these names can be used by themselves as a logical expression. For example.

```
$if _ELF64
....
$endif
```

This example will evaluate to *true*, and allow the link-editor to process the enclosed text, when the output object is 64-bit. Although numeric values are not allowed in these logical expressions, a special exception is made for the value `1`, which evaluates to *true*, and `0` for *false*.

Any undefined name evaluates to *false*. It is common to use the undefined name `false` to mark lines of input that should be unconditionally skipped.

```
$if false
....
$endif
```

More complex logical expressions can be written, using the operators shown in the following table

TABLE 8-5 Conditional Expression Operators

Operator	Meaning
&&	Logical AND

Operator	Meaning
`||`	Logical OR
`( expr )`	Sub-expression
`!`	Negate boolean value of following expression

Expressions are evaluated from left to right. Sub-expressions are evaluated before enclosing expressions.

For example, the lines in the following construct will be evaluated when building 64-bit objects for x86 platforms.

```
$if _ELF64 && _x86
....
$endif
```

The `$add` directive can be used to add a new name to the link-editor's table of known names. Using the previous example, it might be convenient to define the name `amd64` to stand for 64-bit x86 objects, in order to simplify `$if` directives.

```
$if _ELF64 && _x86
$add amd64
$endif
```

This can be used to simplify the previous example.

```
$if amd64
....
$endif
```

New names can also be added to the link-editor's table of known names by using the link-editor's `-z mapfile-add` option. This option is useful when `mapfile` input needs to be conditionally enabled based on an attribute of the external environment, such as the compiler being used.

The `$clear` directive is the reverse of the `$add` directive. It is used to remove names from the internal table.

```
$clear amd64
```

The effect of the `$add` directive persists beyond the end of the `mapfile` that uses `$add`, and is visible to any subsequent `mapfile` that is processed by the link-editor in the same link operation. If this is not desired, use `$clear` at the end of the `mapfile` containing the `$add` to remove the definition.

Finally, the `$error` directive causes the link-editor to print all remaining text on the line as a fatal error, and halt the link operation. The `$error` directive can be used to ensure that a programmer porting an object to a new machine type will not be able to silently build an incorrect object that is missing a necessary `mapfile` definition.

```
$if _sparc
....
$elif _x86
....
$else
$error unknown machine type
$endif
```

C language programmers will recognize that the syntax used for `mapfile` conditional input resembles that of the C preprocessor macro language. This similarity is intentional. However, `mapfile` conditional input directives are by design considerably less powerful than those provided by the C preprocessor. They provide only the most basic facilities required to support linking operations in a cross platform environment.

Among the significant differences between the two languages.

- The C preprocessor defines a full macro language, and the macros are applied to both the source text, and to the expressions evaluated by the `#if` and `#elif` preprocessor statements. Link-editor *mapfiles* do not implement a macro capability.
- The expressions evaluated by the C preprocessor involve numeric types, and a rich set of operators. *Mapfile* logical expressions involve boolean *true* and *false* values, and a limited set of operators.
- C preprocessor expressions involve arbitrary numeric values, possibly defined as macros, and `defined()` is used to evaluate whether a given macro is defined or not, yielding a *true* (nonzero) or *false* (zero) value. *Mapfile* logical expressions only manipulate boolean values, and names are used directly without a `defined()` operation. The specified names are considered to be *true* if they exist in the link-editor's table of known names, and *false* otherwise.

Those requiring more sophisticated macro processing should consider using an external macro processor, such as `m4`(1).

Directive Syntax

Mapfile directives exist to specify many aspects of the output object. These directives share a common syntax, using name value pairs for attributes, and {....} constructs to represent hierarchy and grouping.

The syntax of *mapfile* directives is based on the following generic forms.

The simplest form is a directive name without a value.

directive;

The next form is a directive name with a value, or a white space separated list of values.

```
directive = value....;
```

In addition to the "=" assignment operator shown, the "+=" and "-=" forms of assignment are allowed. The "=" operator sets the given directive to the given value, or value list. The "+=" operator is used to add the value on the right hand side to the current value, and the "-=" operator is used to remove values.

More complex directives manipulate items that take multiple attributes enclosed within {....} brackets to group the attributes together as a unit.

```
directive [name] {
        attribute [directive = value];
....
} [name];
```

There can be a name before the opening brace ({), which is used to name the result of the given statement. Similarly, one or more optional names can follow the closing brace (}), prior to the terminating semicolon (;). These names are used to express that the defined item has a relationship with other named items.

Note that the format for attributes within a grouping use the same syntax described above for simple directives with a value, with an assignment operator (=, +=, -=) followed by a value, or white space separated list of values, terminated with a semicolon (;).

A directive can have attributes that in turn have sub-attributes. In such cases, the sub-attributes are also grouped within nested {. . . .} brackets to reflect this hierarchy.

```
directive [name] {
        attribute {
                subattribute [= value];
        ....
        };
} [name....];
```

The *mapfile* syntax grammar puts no limit on the depth to which such nesting is allowed. The depth of nesting depends solely on the requirements of the directive.

Mapfile Directives

The following directives are accepted by the link-editor.

TABLE 8-6 *Mapfile* Directives

Directive	Purpose
CAPABILITY	Hardware, software, machine, and platform capabilities

Directive	Purpose
DEPEND_VERSIONS	Specify allowed versions from sharable object dependencies
HDR_NOALLOC	ELF header and program headers are not allocable
LOAD_SEGMENT	Create new loadable segment, or modify an existing load segment
NOTE_SEGMENT	Create note segment, or modify an existing note segment
NULL_SEGMENT	Create null segment, or modify an existing null segment
PHDR_ADD_NULL	Add Null Program Header Entries
SEGMENT_ORDER	Specify the order of segments in the output object and program header array
STACK	Process Stack Attributes
STUB_OBJECT	Specify that object can be built as a stub object
SYMBOL_SCOPE	Set symbol attributes and scope within the unnamed global version
SYMBOL_VERSION	Set symbol attributes and scope within an explicitly named version

The specific syntax for each supported *mapfile* directive is shown in the sections that follow.

CAPABILITY Directive

The hardware, software, machine, and platform capabilities of a relocatable object are typically recorded within an object at compile time. The link-editor combines the capabilities of any input relocatable objects to create a final capabilities section for the output file. Capabilities can be defined within a `mapfile`, to augment, or completely replace, the capabilities that are supplied from input relocatable objects.

```
CAPABILITY [capid] {
        HW  = [hwcap_flag....];
        HW += [hwcap_flag....];
        HW -= [hwcap_flag....];

        HW_1  = [value....];
        HW_1 += [value....];
        HW_1 -= [value....];

        HW_2  = [value....];
        HW_2 += [value....];
        HW_2 -= [value....];

        MACHINE  = [machine_name....];
        MACHINE += [machine_name....];
        MACHINE -= [machine_name....];
```

```
        PLATFORM  = [platform_name....];
        PLATFORM += [platform_name....];
        PLATFORM -= [platform_name....];

        SF  = [sfcap_flag....];
        SF += [sfcap_flag....];
        SF -= [sfcap_flag....];

        SF_1  = [value....];
        SF_1 += [value....];
        SF_1 -= [value....];
};
```

If present, the optional *capid* name provides a symbolic name for the object capabilities, resulting in a `CA_SUNW_ID` capability entry in the output object. If multiple `CAPABILITY` directives are seen, the *capid* provided by the final directive is used.

An empty `CAPABILITY` directive can be used to specify a *capid* for the object capabilities without specifying any capability values.

```
CAPABILITY capid;
```

For each type of capability, the link-editor maintains a current value (*value*), and a set of values to be excluded (*exclude*). For hardware and software capabilities, these values are *bitmasks*. For machine and platform capabilities, they are lists of names. Prior to processing *mapfiles*, the *value* and *exclude* values for all capabilities are cleared. The assignment operators work as follows.

- If the "`+=`" operator is used, the value specified is added to the current *value* for that capability, and removed from the *exclude* values for that capability.
- If the "`-=`" operator is used, the value specified is added to the *exclude* values for that capability, and removed from the current *value* for that capability.
- If the "`=`" operator is used, the value specified replaces the previous *value*, and *exclude* is reset to 0. In addition, the use of "`=`" overrides any capabilities that are collected from input file processing.

Input objects are processed after *mapfiles* have been read. Capability values specified by the input objects are merged with those from the *mapfiles*, unless the "`=`" operator was used, in which case that capability is ignored when encountered in an input object. Hence, the "`=`" operator overrides the input objects, whereas the "`+=`" operator is used to augment them.

Prior to writing the resulting capability value to the output object, the link-editor subtracts any capability values specified with the "`-=`" operator.

To completely eliminate a given capability from the output object, it suffices to use the "`=`" operator and an empty value list. For example, the following suppresses any hardware capabilities contributed by the input objects:

```
$mapfile_version 2
```

```
CAPABILITY {
        HW = ;
};
```

Within an ELF object, hardware and software capabilities are represented as bit assignments within one or more *bitmasks* found in the capabilities section of the object. The `HW` and `SF` `mapfile` attributes provide a more abstract view of this implementation, accepting a space separated list of symbolic capability names that the link-editor translates to the appropriate mask and bit. The numbered attributes (`HW_1`, `HW_2`, `SF_1`) exist in order to allow direct numeric access to the underlying capability *bitmasks*. They can be used to specify capability bits that have not been officially defined. Where possible, use of the `HW` and `SF` attributes is recommended.

HW Attribute

Hardware capabilities are specified as a space separated list of symbolic capability names. 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>`. *Mapfiles* use the same names, without the `AV_` prefix. For example, the x86 `AV_SSE` hardware capability is called `SSE` within a `mapfile`. This list can contain any of the capability names defined for the `CA_SUNW_HW_` capability masks.

HW_1 / HW_2 Attributes

The `HW_1` and `HW_2` attributes allow the `CA_SUNW_HW_1` and `CA_SUNW_HW_2` capability masks to be specified directly as numeric values, or as the symbolic hardware capability names that correspond to that mask.

MACHINE Attribute

The `MACHINE` attribute specifies the machine hardware names for the systems that the object can execute upon. The machine hardware name of a system can be displayed by the utility `uname(1)` with the `-m` option. A `CAPABILITY` directive can specify multiple machine names. Each name results in a `CA_SUNW_MACH` capability entry in the output object.

PLATFORM Attribute

The `PLATFORM` attribute specifies the platform names for the systems that the object can execute upon. The platform name of a system can be displayed by the utility `uname(1)` with the `-i`

option. A `CAPABILITY` directive can specify multiple platform names. Each name results in a `CA_SUNW_PLAT` capability entry in the output object.

SF Attribute

Software capabilities are specified as a space separated list of symbolic capability names. Software capabilities are defined as `SF1_SUNW_` values in `<sys/elf.h>`. *Mapfiles* use the same names, without the `SF1_SUNW_` prefix. For example, the `SF1_SUNW_ADDR32` software capability is called `ADDR32` in a `mapfile`. This list can contain any of the capability names defined for the `CA_SUNW_SF_1`.

SF_1 Attribute

The `SF_1` attribute allows the `CA_SUNW_SF_1` capability mask to be specified directly as a numeric value, or as symbolic software capability names that correspond to that mask.

DEPEND_VERSIONS Directive

When linking against a sharable object, the symbols from all versions exported by the object are normally available for use by the link-editor. The `DEPEND_VERSIONS` directive is used to limit access to specified versions only. Restricting version access can be used to ensure that a given output object does not use newer features that might not be available on an older version of the system.

A `DEPEND_VERSIONS` directive has the following syntax.

```
DEPEND_VERSIONS objname {
        ALLOW = version_name;
        REQUIRE = version_name;
        ....
};
```

objname is the name of the sharable object, as specified on the command line. In the common case where the object is specified using the `-l` command line option, this will be the specified name with a `lib` prefix. For instance, `libc` is commonly referenced as `-lc` on the command line, and is therefore specified as `libc.so` in a `DEPEND_VERSIONS` directive.

ALLOW Attribute

The `ALLOW` attribute specifies that the specified version, and versions inherited by that version, are available to the link-editor for resolving symbols in the output object. The link-editor will

add a requirement for the highest version used in the inheritance chain containing this version to the output object requirements.

REQUIRE Attribute

`REQUIRE` adds the specified version to the output object requirements, whether or not the version is actually required to satisfy the link operation.

HDR_NOALLOC Directive

Every ELF object has an ELF header at offset 0 in the file. Executable and sharable objects also contain program headers, which are accessed through the ELF header. The link-editor normally arranges for these items to be included as part of the first loadable segment. The information contained in these headers is therefore visible within the mapped image, and is typically used by the runtime linker. The `HDR_NOALLOC` directive prevents this.

```
HDR_NOALLOC;
```

When `HDR_NOALLOC` is specified, the ELF header and program header array still appear at the start of the resulting output object file, but are not contained in a loadable segment, and virtual address calculations for the image start at the first section of the first segment rather than at the base of the ELF header.

PHDR_ADD_NULL Directive

The `PHDR_ADD_NULL` directive causes the link-editor to add a specified number of additional program header entries of type `PT_NULL` at the end of the program header array. Extra `PT_NULL` entries can be used by post processing utilities.

```
PHDR_ADD_NULL = value;
```

value must be a positive integer value, and gives the number of extra `PT_NULL` entries to create. All fields of the resulting program header entries will be set to 0.

LOAD_SEGMENT / NOTE_SEGMENT / NULL_SEGMENT Directives

A segment is a contiguous portion of the output object that contains sections. The *mapfile* segment directives allow the specification of three different segment types.

- LOAD_SEGMENT

  A loadable segment contains code or data that is *mapped* into the address space of a process at runtime. The link-editor creates a `PT_LOAD` program header entry for each allocable segment, which is used by the runtime linker to locate and map the segment.
- NOTE_SEGMENT

  A note segment contains note sections. The link-editor creates a `PT_NOTE` program header entry that references the segment. Note segments are not allocable.
- NULL_SEGMENT

  A null segment holds sections that are included in the output object, but which are not available to the object at runtime. Common examples of such sections are the `.symtab` symbol table, and the various sections produced for the benefit of debuggers. No program header is created for a null segment.

Segment directives are used to create new segments in the output file, or to change the attribute values of an existing segment. An existing segment is one that was previous defined, or one of the built-in segments discussed in "Predefined Segments" on page 223. Each new segment is added to the object after the last such segment of the same type. Loadable segments are added first, then note segments, and finally null segments. Any program headers associated with these segments are placed in the program header array in the same relative order as the segments themselves. This default placement can be altered by setting an explicit address in the case of a loadable segment, or using the `SEGMENT_ORDER` directive.

If *segment_name* is a preexisting segment, then the attributes specified modify the existing segment. Otherwise, a new segment is created and the specified attributes are applied to the new segment. The link-editor fills in default values for attributes not explicitly supplied.

Note - When selecting a segment name, bear in mind that a future version of the link-editor might add new predefined segments. If the name used in your segment directive matches this new name, the new predefined segment will alter the meaning of your `mapfile`, from creating a new segment to modifying an existing one. The best way to prevent this situation is to avoid generic names for segments, and give all of your segment names a unique prefix, such as a company/project identifier, or even the name of the program. For example, a program named `hello_world` might use the segment name `hello_world_data_segment`.

All three segment directives share a common set of core attributes. Substituting one of `LOAD_SEGMENT`, `NOTE_SEGMENT`, `NULL_SEGMENT` for *directive*, a segment declaration is as follows.

```
directive segment_name {
        ASSIGN_SECTION [assign_name];
        ASSIGN_SECTION [assign_name] {
                FILE_BASENAME = file_basename;
                FILE_OBJNAME = objname;
                FILE_PATH = file_path;
                FLAGS = section_flags;
                IS_NAME = section_name;
```

```
                TYPE = section_type;
        };

        DISABLE;

        IS_ORDER  = assign_name....;
        IS_ORDER += assign_name....;

        OS_ORDER  = section_name....;
        OS_ORDER += section_name....;
};
```

The LOAD_SEGMENT directive accepts an additional set of attributes specific to loadable segments. The syntax of these additional attributes is as follows.

```
LOAD_SEGMENT segment_name {
       ALIGN = value;

       FLAGS  = segment_flags;
       FLAGS += segment_flags;
       FLAGS -= segment_flags;

       MAX_SIZE = value;

       NOHDR;

       PADDR = value;
       ROUND = value;

       SIZE_SYMBOL  = symbol_name....;
       SIZE_SYMBOL += symbol_name....;

       VADDR = value;
};
```

Any of the segment directives can be specified as an empty directive. When an empty segment directive creates a new segment, default values are established for all segment attributes. Empty segments are declared as follows.

```
LOAD_SEGMENT segment_name;

NOTE_SEGMENT segment_name;

NULL_SEGMENT segment_name;
```

All of the attributes accepted by one or more of the segment directives are described below.

ALIGN Attribute (LOAD_SEGMENT only)

The ALIGN attribute is used to specify the alignment for a loadable segment. The value specified is set in the p_align field of the program header corresponding to the segment. Segment alignment is used in calculating the virtual address of the beginning of the segment.

The alignment specified must be 0 or a power of 2. By default, the link-editor sets the alignment of a segment to the built-in default. This default differs from one CPU to another and might even be different between software revisions.

The `ALIGN` attribute is related to the `PADDR` and `VADDR` attributes, and must be compatible with them.

ASSIGN_SECTION Attribute

`ASSIGN_SECTION` specifies a combination of section attributes, such as section name, type, and flags, that collectively qualify a section for assignment to a given segment. Each such set of attributes is called an *entrance criterion*. A section matches when the section attributes match those of an entrance criterion exactly. An `ASSIGN_SECTION` that does not specify any attributes matches any section that criterion is compared to.

Multiple `ASSIGN_SECTION` attributes are allowed for a given segment. Each `ASSIGN_SECTION` attribute is independent of the others. A section will be assigned to a segment if the section matches any one of the `ASSIGN_SECTION` definitions associated with that segment. The link-editor will not assign sections to a segment unless the segment has at least one `ASSIGN_SECTION` attribute.

The link-editor uses an internal list of entrance criteria to assign sections to segments. Each `ASSIGN_SECTION` declaration encountered in the `mapfile` is placed on this list, in the order encountered. The entrance criteria for the built-in segments discussed in "Predefined Segments" on page 223 are placed on this list immediately following the final `mapfile` defined entry.

The entrance criterion can be given an optional name (*assign_name*). This name can be used in conjunction with the `IS_ORDER` attribute to specify the order in which input sections are placed in the output section.

To place an input section, the link-editor starts at the head of the entrance criteria list, and compares the attributes of the section to each entrance criterion in turn. The section is assigned to the segment associated with the first entrance criterion that matches the section attributes exactly. If there is no match, the section is placed at the end of the file, as is generally the case for all non-allocable sections.

ASSIGN_SECTION accepts the following.

`FILE_BASENAME`, `FILE_OBJNAME`, `FILE_PATH`

These attributes allow the selection of sections based on the path (`FILE_PATH`), basename (`FILE_BASENAME`), or object name (`FILE_OBJNAME`) of the file they come from.

File paths are specified using the standard Unix slash delimited convention. The final path segment is the *basename* of the path, also known simply as the *filename*. In the case of an

archive, the basename can be augmented with the name of the archive member, using the form `archive_name(component_name)`. For example, `/lib/libfoo.a(bar.o)`specifies the object `bar.o`, found in an archive named `/lib/libfoo.a`.

`FILE_BASENAME` and `FILE_OBJNAME` are equivalent when applied to a non-archive, and compare the given name to the basename of the file. When applied to an archive, `FILE_BASENAME` examines the basename of the archive name, while `FILE_OBJNAME` examines the name of the object contained within the archive.

Each `ASSIGN_SECTION` maintains a list of all `FILE_BASENAME`, `FILE_PATH`, and `FILE_OBJNAME` values. A file match occurs if any one of these definitions match an input file.

`IS_NAME`

Input section name.

`TYPE`

Specifies an ELF *section_type*, which can be any of the `SHT_` constants defined in `<sys/elf.h>`, with the `SHT_` prefix removed. For example, `PROGBITS`, `SYMTAB`, or `NOBITS`.

`FLAGS`

The `FLAGS` attribute uses *section_flags* to specify section attributes as a space separated list of one or more of the values given in Table 8-7, which correspond to the `SHF_` values defined in `<sys/elf.h>`. If an individual flag is preceded by an exclamation mark (`!`), that attribute must explicitly not be present. In the following example, a section is defined allocable and not writable.

```
ALLOC !WRITE
```

Flags not explicitly in a *section_flags* list are ignored. In the above example, only the value of `ALLOC` and `WRITE` are examined when matching a section against the specified flags. The other section flags can have any value.

TABLE 8-7 Section FLAGS Values

Flag Value	Meaning
`ALLOC`	Section is allocable
`WRITE`	Section is writable
`EXECUTE`	Section is executable
`AMD64_LARGE`	Section can be larger than 2 Gbytes

DISABLE Attribute

The `DISABLE` attribute causes the link-editor to ignore the segment. No sections will be assigned to a disabled segment. The segment is automatically re-enabled when referenced by a following segment directive. Hence, an empty reference suffices to re-enable a disabled section.

```
segment segment_name;
```

FLAGS Attribute (LOAD_SEGMENT only)

The `FLAGS` attribute specifies segment permissions as a space separated list of the permissions in Table 8-3. By default, user defined segments receive `READ`, `WRITE`, and `EXECUTE` permissions. The default flags for the predefined segments described in "Predefined Segments" on page 223 are supplied by the link-editor, and in some cases can be platform-dependent.

There are three forms allowed.

```
FLAGS  = segment_flags....;
FLAGS += segment_flags....;
FLAGS -= segment_flags....;
```

The simple "`=`" assignment operator replaces the current flags with the new set, the "`+=`" form adds the new flags to the existing set, and the "`-=`" form removes the specified flags from the existing set.

IS_ORDER Attribute

The link-editor normally places output sections into the segment in the order they are encountered. Similarly, the input sections that make up the output section are placed in the order they are encountered. The `IS_ORDER` attribute can be used to alter this default placement of input sections. `IS_ORDER` specifies a space separated list of entrance criterion names (*assign_name*). Sections matched by one of these entrance criteria are placed at the head of the output section, sorted in the order given by `IS_ORDER`. Sections matched by entrance criteria not found in the `IS_ORDER` list are placed following the sorted sections, in the order they are encountered.

When the "`=`" form of assignment is used, the previous value of `IS_ORDER` for the given segment is discarded, and replaced with the new list. The "`+=`" form of `IS_ORDER` concatenates the new list to the end of the existing list.

The `IS_ORDER` attribute is of particular interest when used 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`. When the link-editor places these sections into the output, the `%` and anything following the `%` are removed. Hence, all three of these functions will be placed in the `.text` output section. The `IS_ORDER` attribute can be used to force them to be placed in a specific order within the `.text` output section relative to each other.

Consider the following user-defined `mapfile`.

```
$mapfile_version 2
LOAD_SEGMENT text {
        ASSIGN_SECTION text_bar  { IS_NAME = .text%bar };
        ASSIGN_SECTION text_main { IS_NAME = .text%main };
        ASSIGN_SECTION text_foo  { IS_NAME = .text%foo };
        IS_ORDER = text_foo text_bar text_main;
};
```

No matter the order in which these three functions are found in the source code, or encountered by the link-editor, their order in the output object text segment will be `foo`, `bar`, and `main`.

MAX_SIZE Attribute (LOAD_SEGMENT only)

By default, the link-editor will allow a segment to grow to the size required by the contents of the segment. The `MAX_SIZE` attribute can be used to specify a maximum size for the segment. If `MAX_SIZE` is set, the link-editor will generate an error if the segment grows beyond the specified size.

NOHDR Attribute (LOAD_SEGMENT only)

If a segment with the `NOHDR` attribute set becomes the first loadable segment in the output object, the ELF and program headers will not be included within the segment.

The `NOHDR` attribute differs from the top level `HDR_NOALLOC` directive in that `HDR_NOALLOC` is a per-segment value, and only has an effect if the segment becomes the first loadable segment. This feature exists primarily to provide feature parity with the older *mapfiles*. See Appendix B, *"System V Release 4 (Version 1) Mapfiles"* for more details.

The `HDR_NOALLOC` directive is recommended in preference to the segment `NOHDR` attribute.

OS_ORDER Attribute

The link-editor normally places output sections into the segment in the order they are encountered. The `OS_ORDER` attribute can be used to alter this default placement of output sections. `OS_ORDER` specifies a space separated list of output section names (*section_name*). The

listed sections are placed at the head of the segment, sorted in the order given by `OS_ORDER`. Sections not listed in `OS_ORDER` are placed following the sorted sections, in the order they are encountered.

When the "`=`" form of assignment is used, the previous value of `OS_ORDER` for the given segment is discarded, and replaced with the new list. The "`+=`" form of `OS_ORDER` concatenates the new list to the end of the existing list.

PADDR Attribute (LOAD_SEGMENT only)

The `PADDR` attribute is used to specify an explicit physical address for the segment. The value specified must be 0 or a power of 2. The value specified is set in the `p_addr` field of the program header corresponding to the segment. By default, the link-editor sets the physical address of segments to 0, as this field has no meaning for user mode objects, and is primarily of interest *non-userland* objects such as operating system kernels.

ROUND Attribute (LOAD_SEGMENT only)

The `ROUND` attribute is used to specify that the size of the segment should be rounded up to the given value. The rounding value specified must be 0 or a power of 2. By default, the link-editor sets the rounding factor of a segment to 1, meaning that the segment size is not rounded up.

SIZE_SYMBOL Attribute (LOAD_SEGMENT only)

The `SIZE_SYMBOL` attribute defines a space separated list of section size symbol names to be created by the link-editor. A size symbol is a global-absolute symbol that represents the size, in bytes, of the segment. These symbols can be referenced in your object files. In order to access the symbol within your code, you should ensure that *symbol_name* is a legal identifier in that language. The symbol naming rules for the C programming language are recommended, as such symbols are likely to be accessible from any other language.

The "`=`" form of assignment can be used to establish an initial value, and can only be used once per link-editor session. The "`+=`" form of `SIZE_SYMBOL` concatenates the new list to the end of the existing list, and can be used as many times as desired.

VADDR (LOAD_SEGMENT only)

The `VADDR` attribute is used to specify an explicit virtual address for the segment. The value specified is set in the `p_vaddr` field of the program header corresponding to the segment. By default, the link-editor assigns virtual addresses to segments as the output file is created.

SEGMENT_ORDER Directive

The SEGMENT_ORDER directive is used to specify a non-default ordering for segments in the output object.

SEGMENT_ORDER accepts a space separated list of segment names.

```
SEGMENT_ORDER  = segment_name....;
SEGMENT_ORDER += segment_name....;
```

When the "=" form of assignment is used, the previous segment order list is discarded, and replaced with the new list. The "+=" form of assignment concatenates the new list to the end of the existing list.

By default, the link-editor orders segments as follows.

1. Loadable segments with explicit addresses set with the VADDR attribute of the LOAD_SEGMENT directive, sorted by address.
2. Segments ordered using the SEGMENT_ORDER directive, in the order specified.
3. Loadable segments without explicit addresses, not found in the SEGMENT_ORDER list.
4. Note segments without explicit addresses, not found in the SEGMENT_ORDER list.
5. Null segments without explicit addresses, not found in the SEGMENT_ORDER list.

Note - ELF has some implicit conventions that must be followed by a well formed object.

- The first loadable segment is expected to be read-only, allocable, and executable, and receives the ELF header and program header array. This is usually the predefined text segment.
- The final loadable segment in an executable is expected to be writable, and the head of the dynamic heap is usually located immediately following within the same virtual memory mapping.

Mapfiles can be used to create objects that violate these requirements. This should be avoided, as the result of running such an object is undefined.

Unless the HDR_NOALLOC directive is specified, the link-editor enforces the requirement that the first segment must be a loadable segment, and not a note or null segment. HDR_NOALLOC cannot be used for *userland* objects, and is therefore of little practical use. This feature is used when building operating system kernels.

STACK Directive

The STACK directive specifies attributes of the process stack.

```
STACK {
        FLAGS  = segment_flags....;
        FLAGS += segment_flags....;
        FLAGS -= segment_flags....;
};
```

The FLAGS attribute specifies a white space separated list of segment permissions consisting of any of the values described in Table 8-3.

There are three forms allowed. The simple "=" assignment operator replaces the current flags with the new set, the "+=" form adds the new flags to the existing set, and the "-=" form removes the specified flags from the existing set.

The default stack permissions are defined by the platform ABI, and vary between platforms. The value for the target platform is specified using the segment flag name STACK.

On some platforms, the ABI mandated default permissions include EXECUTE. EXECUTE is rarely if ever needed and is generally considered to be a potential security risk. Removing EXECUTE permission from the stack is a recommended practice.

```
STACK {
        FLAGS -= EXECUTE;
};
```

The STACK directive is reflected in the output ELF object as a PT_SUNWSTACK program header entry.

STUB_OBJECT Directive

The STUB_OBJECT directive informs the link-editor that the object described by the mapfile can be built as a stub object.

```
STUB_OBJECT;
```

A stub shared object is built entirely from the information in the mapfiles supplied on the command line. When the -z stub option is specified to build a stub object, the presence of the STUB_OBJECT directive in a mapfile is required, and the link-editor uses the information in symbol ASSERT attributes to create global symbols that match those of the real object.

SYMBOL_SCOPE / SYMBOL_VERSION Directives

The SYMBOL_SCOPE and SYMBOL_VERSION directives are used to specify the scope and attributes of global symbols. SYMBOL_SCOPE operates within the context of the unnamed base symbol version, while SYMBOL_VERSION is used to gather symbols into explicitly named global versions. The SYMBOL_VERSION directive allows the creation of stable interfaces that support object evolution in a backward compatible manner.

SYMBOL_VERSION has the following syntax.

```
SYMBOL_VERSION version_name {
        symbol_scope:
                *;

        symbol_name;
        symbol_name {
                ASSERT = {
                        ALIAS = symbol_name;
                        BINDING = symbol_binding;
                        TYPE = symbol_type;

                        SIZE = size_value;
                        SIZE = size_value[count];
          VALUE = value;
                };
                AUXILIARY = soname;
                FILTER = soname;
                FLAGS = symbol_flags....;

                SIZE = size_value;
                SIZE = size_value[count];

                TYPE = symbol_type;
                VALUE = value;
        };
} [inherited_version_name....];
```

SYMBOL_SCOPE does not accept version names, but is otherwise identical.

```
SYMBOL_SCOPE {
        ....
};
```

In a SYMBOL_VERSION directive, *version_name* provides a label for this set of symbol definitions. This label identifies a *version definition* within the output object. One or more inherited versions (*inherited_version_name*) can be specified, separated by white space, in which case the newly defined version inherits from the versions named. See Chapter 9, “Interfaces and Versioning”.

symbol_scope defines the scope of symbols in a SYMBOL_SCOPE or SYMBOL_VERSION directive. By default, symbols are assumed to have global scope. This can be modified by specifying a

symbol_scope followed by a colon (:). These lines determine the symbol scope for all symbols that follow, until changed by a subsequent scope declaration. The possible scope values and their meanings are given in the following table.

TABLE 8-8 Symbol Scope Types

Scope	Meaning
`default / global`	Global symbols of this scope are visible to all external objects. References to such symbols from within the object are bound at runtime, thus allowing interposition to take place. This visibility scope provides a default, that can be demoted, or eliminated by other symbol visibility techniques. This scope definition has the same affect as a symbol with `STV_DEFAULT` visibility. See Table 12-23.
`hidden / local`	Global 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 12-23.
`protected / symbolic`	Global symbols of this scope are visible to all external objects. References to these symbols from within the object are bound at link-edit, thus preventing runtime interposition. This visibility scope can be demoted, or eliminated by other symbol visibility techniques. This scope definition has the same affect as a symbol with `STV_PROTECTED` visibility. See Table 12-23.
`exported`	Global symbols of this scope are visible to all external objects. References to such symbols from within the object are bound at runtime, thus allowing interposition to take place. This symbol visibility can not be demoted, or eliminated by any other symbol visibility technique. This scope definition has the same affect as a symbol with `STV_EXPORTED` visibility. See Table 12-23.
`singleton`	Global symbols of this scope are visible to all external objects. References to such symbols from within the object are bound at runtime, and ensure that only one instance of the symbol is bound to from all references within a process. This symbol visibility can not be demoted, or eliminated by any other symbol visibility technique. This scope definition has the same affect as a symbol with `STV_SINGLETON` visibility. See Table 12-23.
`eliminate`	Global symbols of this scope are hidden. Their symbol table entries are eliminated. This scope definition has the same affect as a symbol with `STV_ELIMINATE` visibility. See Table 12-23.

A *symbol_name* is the name of a symbol. This name can result in a symbol definition, or a symbol reference, depending on any qualifying attributes. In the simplest form, without any qualifying attributes, a symbol reference is created. This reference is exactly the same as would be generated using the `-u` option discussed in “Defining Additional Symbols with the `-u` option” on page 45. Typically, if the symbol name is followed by any qualifying attributes, then a symbol definition is generated using the associated attributes.

When a local scope is defined, the symbol name can be defined as the special “*” auto-reduction directive. Symbols that have no explicitly defined visibility are demoted to a local binding within the dynamic object being generated. Explicit visibility definitions originate from `mapfile` definitions, or visibility definitions that are encapsulated within relocatable objects.

Similarly, when an eliminate scope is defined, the symbol name can be defined as the special "*" auto-elimination directive. Symbols that have no explicitly defined visibility are eliminated from the dynamic object being generated.

If a `SYMBOL_VERSION` directive is specified, or if auto-reduction is specified with either `SYMBOL_VERSION` or `SYMBOL_SCOPE`, 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 9, "Interfaces and Versioning".

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

A *symbol_name* can be listed by itself in order to simply assign the symbol to a version and/or specify its scope. Optional symbol attributes can be specified within {} brackets. Valid attributes are described below.

ASSERT Attribute

The `ASSERT` attribute is used to specify the expected characteristics of the symbol. The link-editor compares the symbol characteristics that result from the link-edit to those given by `ASSERT` attributes. If the real and asserted attributes do not agree, a fatal error is issued and the output object is not created.

The interpretation of the `ASSERT` attribute is dependent on whether the `STUB_OBJECT` directive or `-z stub` command line option are used. The three possible cases are as follows.

1. `ASSERT` attributes are not required when the `STUB_OBJECT` directive is not used. However, if `ASSERT` attributes exist, their attributes are verified against the real values collected with the link-edit. Should any `ASSERT` attributes not match their associated real values, the link-edit terminates unsuccessfully.
2. When the `STUB_OBJECT` directive is used, and the `-z stub` command line option is specified, the link-editor uses the `ASSERT` directives to define the attributes of the global symbols provided by the object. See "Stub Objects" on page 77.

3. When the `STUB_OBJECT` directive is used, and `-z stub` command line option is not specified, the link-editor requires that all global data in the resulting object have an associated `ASSERT` directive that declares it as data and supplies a size. In this mode, if the `TYPE ASSERT` attribute is not specified, `GLOBAL` is assumed. Similarly, if `SH_ATTR` is not specified, a default value of `BITS` is assumed. These defaults ensure that the data attributes of the stub and real objects are compatible. The resulting `ASSERT` statements are evaluated in the same manner as in the first case above. See "STUB_OBJECT Directive" on page 216.

ASSERT accepts the following attributes.

`ALIAS`

Defines an alias for a previously defined symbol. An alias symbol has the same type, value, and size as the main symbol. The `ALIAS` attribute cannot be used with the `TYPE`, `SIZE`, and `SH_ATTR` attributes. When `ALIAS` is specified, the type, size, and section attributes are obtained from the alias symbol.

`BIND`

Specifies an ELF *symbol_binding*, which can be any of the `STB_` values defined in `<sys/elf.h>`, with the `STB_` prefix removed. For example, `GLOBAL`, or `WEAK`.

`TYPE`

Specifies an ELF *symbol_type*, which can be any of the `STT_` constants defined in `<sys/elf.h>`, with the `STT_` prefix removed. For example, `OBJECT`, `COMMON`, or `FUNC`. In addition, for compatibility with other `mapfile` usage, `FUNCTION` and `DATA` can be specified for `STT_FUNC` and `STT_OBJECT`, respectively. `TYPE` cannot be used with `ALIAS`.

`SH_ATTR`

Specifies attributes of the section associated with the symbol. The *section_attributes* that can be specified are given in Table 8-9. `SH_ATTR` cannot be used with `ALIAS`.

`SIZE`

Specifies the expected symbol size. `SIZE` cannot be used with `ALIAS`. The syntax for the *size_value* argument is as described in the discussion of the `SIZE` attribute. See "SIZE Attribute" on page 222.

`VALUE`

Specifies the expected symbol value.

TABLE 8-9 SH_ATTR Values

Section Attribute	Meaning
`BITS`	Section is not of type `SHT_NOBITS`
`NOBITS`	Section is of type `SHT_NOBITS`

AUXILIARY Attribute

Indicates that this symbol is an auxiliary filter on the shared object name (*soname*). See "Generating Auxiliary Filters" on page 144.

FILTER Attribute

Indicates that this symbol is a filter on the shared object *name*. See "Generating Standard Filters" on page 141. 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.

FLAGS Attribute

symbol_flags specify symbol attributes as a space separated list of one or more of the following values.

TABLE 8-10 Symbol FLAG Values

Flag	Meaning
`DIRECT`	Indicates that this symbol should be directly bound to. When used with a symbol definition, this keyword results in any reference from within the object being built to be directly bound to the definition. When used with a symbol reference, this flag results in a direct binding to the dependency that provides the definition. See Chapter 6, "Direct Bindings". This flag can also be used with the `PARENT` flag to establish a direct binding to any parent at runtime.
`DYNSORT`	Indicates that this symbol should be included in a sort section. See "Symbol Sort Sections" on page 374. The symbol type must be `STT_FUNC`, `STT_OBJECT`, `STT_COMMON`, or `STT_TLS`.
`EXTERN`	Indicates the symbol is defined externally to the object being created. This keyword is typically defined to label callback routines. Undefined symbols that would be flagged with the `-z defs` option are suppressed with this flag. This flag is only meaningful when generating a symbol reference. Should a definition for this symbol occur within the objects combined at link-edit, then the keyword is silently ignored.
`INTERPOSE`	Indicates that this symbol acts an interposer. This flag can only be used when generating a dynamic executable. This flag provides for finer control of defining interposing symbols than is possible by using the `-z interpose` option.
`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. See Chapter 6, "Direct Bindings". This flag can also be used with the `PARENT` flag to prevent a direct binding to any parent at runtime.
`NODYNSORT`	Indicates that this symbol should not be included in a sort section. See "Symbol Sort Sections" on page 374.

Flag	Meaning
PARENT	Indicates the symbol is defined in the parent of the object being created. A parent is an object that references this object at runtime as an explicit dependency. A parent can also reference this object at runtime using dlopen(3C). This flag is typically defined to label callback routines. This flag can be used with the DIRECT or NODIRECT flags to establish individual direct, or no-direct references to the parent. Undefined symbols that would be flagged with the -z defs option are suppressed with this flag. This flag is only meaningful when generating a symbol reference. Should a definition for this symbol occur within the objects combined at link-edit, then the keyword is silently ignored.
STUB_ELIMINATE	Indicates that this symbol should be omitted from stub objects. See "Using Stub Objects to Hide Obsolete Interfaces" on page 80.

SIZE Attribute

Sets the size attribute. This attribute results in the creation of a symbol definition.

The *size_value* argument can be a numeric value, or it can be the symbolic name addrsize. addrsize represents the size of a machine word capable of holding a memory address. The link-editor substitutes the value 4 for addrsize when building 32-bit objects, and the value 8 when building 64-bit objects. addrsize is useful for representing the size of pointer variables and C variables of type long, as it automatically adjusts for 32 and 64-bit objects without requiring the use of conditional input.

The *size_value* argument can be optionally suffixed with a *count* value, enclosed in square brackets. If *count* is present, *size_value* and *count* are multiplied together to obtain the final size value.

TYPE Attribute

The symbol type attribute. This attribute can be either COMMON, DATA, or FUNCTION. COMMON results in a tentative symbol definition. DATA and FUNCTION result in a section symbol definition or an absolute symbol definition. See "Symbol Table Section" on page 365.

A data attribute results in the creation of an OBJT symbol. A data attribute that is accompanied with a size, but no value creates a section symbol by associating the symbol with an ELF section. This section is filled with zeros. A function attribute results in the creation of an FUNC symbol.

A function attribute that is accompanied with a size, but no value creates a section symbol by associating the symbol with an ELF section. This section is assigned a void function, generated by the link-editor, with the following signature.

```
void (*)(void)
```

A data or function attribute that is accompanied with a value results in the appropriate symbol type together with an absolute, `ABS`, section index.

The creation of a section data symbol is useful for the creation of filters. External references to a section data symbol of a filter from an executable result in the appropriate copy relocation being generated. See "Copy Relocations" on page 188.

VALUE Attribute

Indicates the value attribute. This attribute results in the creation of a symbol definition.

Predefined Segments

The link-editor provides a predefined set of output segment descriptors and entrance criteria. These definitions satisfy the needs of most linking scenarios, and comply with the ELF layout rules and conventions expected by the system.

The text, data, and extra segments are of primary interest, while the others serve more specialized purposes, as described below.

- `text`

  The `text` segment defines a read-only executable loadable segment that accepts allocable, non-writable sections. This includes executable code, read-only data needed by the program, and read-only data produced by the link-editor for use by the runtime linker such as the dynamic symbol table.

  The `text` segment is the first segment in the process, and is therefore assigned the ELF header, and the program header array by the link-editor. This can be prevented using the `HDR_NOALLOC` *mapfile* directive.

- `data`

  The `data` segment defines a writable loadable segment. The `data` segment is used for writable data needed by the program, and for writable data used by the runtime linker, such as the Global Offset Table (GOT), and the Procedure Linkage Table (PLT), on architectures such as SPARC that require the PLT sections to be writable.

- `extra`

  The *extra* segment captures all sections not assigned elsewhere, directed there by the final entrance criterion record. Common examples are the full symbol table (`.symtab`), and the various sections produced for the benefit of debuggers. This is a null segment, and has no corresponding program header table entry.

- `note`

The `note` segment captures all sections of type `SHT_NOTE`. The link-editor provides a `PT_NOTE` program header entry to reference the `note` segment.

- `lrodata` / `ldata`

  The x86–64 ABI defines small, medium, and large compilation models. The ABI requires sections for the medium and large models to set the `SHF_AMD64_LARGE` section flag. An input section lacking the `SHF_AMD64_LARGE` must be placed in an output segment that does not exceed 2 Gbytes in size. The `lrodata` and `ldata` predefined segments are present for x86–64 output objects only, and are used to handle sections with the `SHF_AMD64_LARGE` flag set. `lrodata` receives read-only sections, and `ldata` receives the others.

- `bss`

  ELF allows for any segment to contain `NOBITS` sections. The link-editor places such sections at the end of the segment they are assigned to. This is implemented using the program header entry `p_filesz` and `p_memsz` fields, which must follow the following rule.

  ```
  p_memsz >= p_filesz
  ```

  If `p_memsz` is greater than `p_filesz`, the extra bytes are `NOBITS`. The first `p_filesz` bytes come from the object file, and any remaining bytes up to `p_memsz` are zeroed by the system prior to use.

  The default assignment rules assign read-only `NOBITS` sections to the text segment, and writable `NOBITS` sections to the data segment. The link-editor defines the `bss` segment as an alternative segment that can accept writable `NOBITS` sections. This segment is disabled by default, and must be explicitly enabled to be used.

  Since writable `NOBITS` sections are easily handled as part of the data segment, the benefit of having a separate `bss` segment may not be immediately obvious. By convention, the process dynamic memory heap starts at the end of the final segment, which must be writable. This is usually the data segment, but if `bss` is enabled, `bss` becomes the final segment. When building a dynamic executable, enabling the `bss` segment with an appropriate alignment can be used to enable large page assignment of the heap. For example, the following enables the `bss` segment and sets an alignment of 4 Mbytes.

  ```
  LOAD_SEGMENT bss {
          ALIGN=0x400000;
  };
  ```

Note - Users are cautioned that an alignment specification can be machine-specific, and may not have the same benefit on different hardware platforms. A more flexible means of requesting the most optimal underlying page size may evolve in future releases.

Mapping Examples

The following are examples of user-defined *mapfiles*. 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: Section to Segment Assignment

This example demonstrates how to define segments and assign input sections to them.

EXAMPLE 8-1 Basic Section to Segment Assignment

```
1   $mapfile_version 2
2   LOAD_SEGMENT elephant {
3           ASSIGN_SECTION {
4                   IS_NAME=.data;
5                   FILE_PATH=peanuts.o;
6           };
7           ASSIGN_SECTION {
8                   IS_NAME=.data;
9                   FILE_OBJNAME=popcorn.o;
10          };
11  };
12
13  LOAD_SEGMENT monkey {
14          VADDR=0x80000000;
15          MAX_SIZE=0x4000;
16          ASSIGN_SECTION {
17                  TYPE=progbits;
18                  FLAGS=ALLOC EXECUTE;
19          };
20          ASSIGN_SECTION {
21                  IS_NAME=.data
22          };
23  };
24
25  LOAD_SEGMENT donkey {
26          FLAGS=READ EXECUTE;
27          ALIGN=0x1000;
28          ASSIGN_SECTION {
29                  IS_NAME=.data;
30          };
31  };
32
33  LOAD_SEGMENT text {
34          VADDR=0x80008000
35  };
```

Four separate segments are manipulated in this example. Every `mapfile` starts with a `$mapfile_version` declaration as shown on line 1. Segment elephant (lines 2-11) receives all

of the data sections from the files `peanuts.o` or `popcorn.o`. The object `popcorn.o` can come from an archive, in which case the archive file can have any name. Alternatively, `popcorn.o` can come from any file with a basename of `popcorn.o`. In contrast, `peanuts.o` can only come from a file with exactly that name. For example, the file `/var/tmp/peanuts.o` supplied to a link-edit does not match `peanuts.o`.

Segment monkey (lines 13-23) has a virtual address of *0x80000000*, and a maximum length of *0x4000*. This segment receives all sections that are both `PROGBITS` and allocable-executable, as well as all sections not already in the segment elephant with the name .data. The .data sections entering the monkey segment need not be `PROGBITS` or allocable-executable, because they match the entrance criterion on line 20 rather than the one on line 16. This illustrates that and *and* relationship exists between the sub-attributes within a `ASSIGN_SECTION` attribute, while an *or* relationship exists between the different `ASSIGN_SECTION` attributes for a single segment.

The donkey segment (lines 25-31) is given non-default permission flags and alignment, and will accept all sections named .data. However, this segment will never be assigned any sections, and as a result, segment donkey will never appear in the output object. The reason for this is that the link-editor examines entrance criteria in the order they appear in the `mapfile`. In this `mapfile`, segment elephant accepts some .data sections, and segment monkey takes any that are left, leaving none for donkey.

Lines 33-35 set the virtual address of the text segment to *0x80008000*. The text segment is one of the standard predefined segments, as described in “Predefined Segments” on page 223, so this statement modifies the existing segment rather than creating a new one.

Example: Predefined Section Modification

The following `mapfile` example manipulates the predefined text and data segments, header options and section within segment ordering.

EXAMPLE 8-2 Predefined Section Manipulation and Section to Segment Assignment

```
1    $mapfile_version 2
2    HDR_NOALLOC;
3
4    LOAD_SEGMENT text {
5            VADDR=0xf0004000;
6            FLAGS=READ EXECUTE;
7            OS_ORDER=.text .rodata;
9            ASSIGN_SECTION {
10                   TYPE=PROGBITS;
11                   FLAGS=ALLOC !WRITE;
12           };
13   };
14
15   LOAD_SEGMENT data {
16           FLAGS=READ WRITE EXECUTE;
```

```
17          ALIGN=0x1000;
18          ROUND=0x1000;
19   };
```

As always, the first line declares the *mapfile* language version to be used. The `HDR_NOALLOC` directive (line 2) specifies that the resulting object should not include the ELF header or program header array within the first allocable segment in the object, which is the predefined text segment.

The segment directive on lines 4-13 set a virtual address and permission flags for the text segment. This directive also specifies that sections named `.text` sections should be placed at the head of the segment, followed by any sections named `.rodata`, and that all other sections will follow these. Finally, allocable, non-writable `PROGBITS` sections are assigned to the segment.

The segment directive on lines 15-19 specifies that the data segment must be aligned on a boundary of *0x1000*. This has the effect of aligning the first section within the segment at the same alignment. The length of the segment is to be rounded up to a multiple of the same value as the alignment. The segment permissions are set to read, write, and execute.

Link-Editor Internals: Section and Segment Processing

The internal process used by the link-editor to assign sections to output segments is described here. This information is not necessary in order to use *mapfiles*. This information is primarily of interest to those interested in link-editor internals, and for those who want a deep understanding of how segment *mapfile* directives are interpreted and executed by the link-editor.

Section To Segment Assignment

The process of assigning input sections to output segments involves the following data structures.

- Input Sections

  Input sections are read from relocatable objects input to the link editor. Some are examined and processed by the link-editor, while others are simply passed to the output without examination of their contents (e.g. `PROGBITS`).
- Output Sections

  Output sections are sections that are written to the output object. Some are formed from the concatenation of sections passed through from the input objects. Others, such as symbol tables and relocation sections are generated by the link-editor itself, often incorporating information read from the input objects.

  When the link-editor passes an input section through to become an output section, the section usually retains the input section name. However, the link-editor can modify the

name in certain circumstances. For instance, the link-editor translates input section names of the form `name%XXX`, dropping the `%` character and any characters following from the output section name.

- Segment Descriptors

  The link-editor maintains a list of known segments. This list initially contains the predefined segments, described in "Predefined Segments" on page 223. When a `LOAD_SEGMENT`, `NOTE_SEGMENT`, or `NULL_SEGMENT` *mapfile* directive is used to create a new segment, an additional segment descriptor for the new segment is added to this list. The new segment goes at the end of the list following other segments of the same type, unless explicitly ordered by setting a virtual address (`LOAD_SEGMENT`), or by using the `SEGMENT_ORDER` directive.

  When creating the output object, the link-editor only creates program headers for the segments that receive a section. Empty segments are quietly ignored. Hence, user specified segment definitions have the power to completely replace the use of the predefined segments definitions, despite the fact that there is no explicit facility for removing a segment definition from the link-editor list.

- Entrance Criteria

  A set of section attributes required in order to place that section in a given segment is called an *entrance criterion* for the segment. A given segment can have an arbitrary number of entrance criteria.

  The link-editor maintains an internal list of all defined entrance criteria. This list is used to place sections into segments, as described below. Each `mapfile` inserts the entrance criterion created by the `ASSIGN_SECTION` attribute to the `LOAD_SEGMENT`, `NOTE_SEGMENT`, or `NULL_SEGMENT` *mapfile* directive at the top of this list, in the order they are encountered in the `mapfile`. The entrance criteria for the built-in segments discussed in "Predefined Segments" on page 223 are placed at the end of this list. Therefore, `mapfile` defined entrance criteria take precedence over the built in rules, and *mapfiles* at the end of the command line take precedence over those found at the beginning.

For each section written to the output object, the link-editor performs the following steps to place the section in an output segment.

1. The attributes of the section are compared to each record in the internal entrance criteria list, starting at the head of the list and considering each entrance criterion in turn. A match occurs when every attribute in the entrance criterion matches exactly, and the segment associated with the entrance criterion is not disabled. The search stops with the first entrance criterion that matches, and the section is directed to the associated segment.

   If no Entrance Criterion 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. Most non-allocable sections (e.g. debug sections) end up in this area.
2. When the section falls into a segment, the link-editor checks the list of existing output sections in that segment as follows.

If the section attribute values match those of an existing output section exactly, the section is placed at the end of the list of sections associated with that output section.

If no matching output section is found, a new output section is created with the attributes of the section being placed, and the input section is placed within the new output section. This new output section is positioned within the segment following any other output sections with the same section type, or at the end of the segment if there are none.

Note - If the input section has a user-defined section type value between `SHT_LOUSER` and `SHT_HIUSER`, the section 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 criterion.

Mapfile Directives for Predefined Segments and Entrance Criteria

The link-editor provides a predefined set of output segment descriptors and entrance criteria, as described in "Predefined Segments" on page 223. The link-editor already knows about these sections, so *mapfile* directives are not required to create them. The *mapfile* directives that could be used to produce them are shown for illustrative purposes, and as an example of a relatively complex `mapfile` specification. *Mapfile* segment directives can be used to modify or augment these built in definitions.

Normally, section to segment assignments are done within a single segment directive. However, the predefined sections have more complex requirements, requiring their entrance criteria to be processed in a different order than the segments are laid out in memory. Two passes are used to achieve this, the first to define all the segments in the desired order, and the second to establish entrance criteria in an order that will achieve the desired results. It is rare for a user `mapfile` to require this strategy.

```
# Predefined segments and entrance criteria for the Oracle Solaris
# link-editor
$mapfile_version 2

# The lrodata and ldata segments only apply to x86-64 objects.
# Establish amd64 as a convenient token for conditional input
$if _ELF64 && _x86
$add amd64
$endif

# Pass 1: Define the segments and their attributes, but
# defer the entrance criteria details to the 2nd pass.
LOAD_SEGMENT text {
        FLAGS = READ EXECUTE;
};
LOAD_SEGMENT data {
```

```
        FLAGS = READ WRITE EXECUTE;
};
LOAD_SEGMENT bss {
        DISABLE;
        FLAGS=DATA;
};
$if amd64
LOAD_SEGMENT lrodata {
        FLAGS = READ
};
LOAD_SEGMENT ldata {
        FLAGS = READ WRITE;
};
$endif
NOTE_SEGMENT note;
NULL_SEGMENT extra;

# Pass 2: Define ASSIGN_SECTION attributes for the segments defined
# above, in the order the link-editor should evaluate them.

# All SHT_NOTE sections go to the note segment
NOTE_SEGMENT note {
        ASSIGN_SECTION {
                TYPE = NOTE;
        };
};
$if amd64
# Medium/large model x86-64 readonly sections to lrodata
LOAD_SEGMENT lrodata {
        ASSIGN_SECTION {
                FLAGS = ALLOC AMD64_LARGE;
        };
};
$endif

# text receives all readonly allocable sections
LOAD_SEGMENT text {
        ASSIGN_SECTION {
                FLAGS = ALLOC !WRITE;
        };
};

# If bss is enabled, it takes the writable NOBITS sections
# that would otherwise end up in ldata or data.
LOAD_SEGMENT bss {
        DISABLE;
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE;
                TYPE = NOBITS;
        };
};

$if amd64
# Medium/large model x86-64 writable sections to ldata
LOAD_SEGMENT ldata {
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE AMD64_LARGE;
        };
```

```
        ASSIGN_SECTION {
                TYPE = NOBITS;
                FLAGS = AMD64_LARGE
        };
};
$endif

# Any writable allocable sections not taken above go to data
LOAD_SEGMENT data {
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE;
        };
};

# Any section that makes it to this point ends up at the
# end of the object file in the extra segment. This accounts
# for the bulk of non-allocable sections.
NULL_SEGMENT extra {
        ASSIGN_SECTION;
};
```

♦ ♦ ♦ CHAPTER 9

Interfaces and Versioning

ELF objects processed by the link-editor and runtime linker 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 controlled evolution of the object, while maintaining backward compatibility.

This chapter describes how to define an object's ABI. Also covered, are how changes to this ABI interface can affect backward compatibility. These concepts are explored with models that convey how interface, together with implementation changes, can be incorporated into a new release of an 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.

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 that are made available by any dynamic object represent the object's public interface. Frequently, the number of global symbols that remain in an object after a link-edit are more than you would like to make public. These global symbols stem from the symbol state that is required between the relocatable objects used to create the object. These symbols represent private interfaces within the object.

To define an object'sABI, you should first determine those global symbols that you want to make publicly available from the object. 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 49. 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 these interfaces 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. All previously available interfaces remain intact.
- *Incompatible* updates. These updates change the existing interface. Existing users of the interface can fail, or behave incorrectly.

The following table categorizes some common object changes.

TABLE 9-1 Examples of Interface Compatibility

Object Change	Update Type
The addition of a symbol	Compatible
The removal of a symbol	Incompatible
The addition of an argument to a *non-variadic* 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

Note - Because of interposition, the addition of a symbol can constitute an incompatible update. The new symbol might conflict with an applications use of that symbol. However, this form of incompatibility does seem rare in practice as source-level namespace management is commonly used.

Compatible updates can be accommodated by maintaining version definitions that are 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. These techniques 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 the object. 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.

Version definitions within a shared object provide two facilities.

- 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 49.

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
#include    <stdio.h>

extern const char *_foo1;

void foo1()
{
        (void) printf(_foo1);
}
$ cat data.c
const char *_foo1 = "string used by foo1()\n";
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                 # Release X
        global:
                foo1;
```

```
        local:
                *;
};
$ cc -c -Kpic foo.c data.c
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ elfdump -sN.symtab libfoo.so.1 | grep 'foo.$'
    [32]   0x1074c    0x4  OBJT LOCL  H    0 .data      _foo1
    [53]     0x560   0x38  FUNC GLOB  D    0 .text      foo1
```

The symbol `foo1` is the only global symbol that is 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. The auto-reduction directive is described in "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217. The associated version name, `SUNW_1.1`, causes the generation of a version definition. Thus, the shared object's public interface consists of the global symbol `foo1` associated to the internal version definition `SUNW_1.1`.

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 object being built. This base version is used to associate any reserved symbols generated by the link-editor. See "Generating the Output File" on page 54 for a list of reserved symbols.

The version definitions that are 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.

From this initial version definition, the object can evolve by adding new interfaces together with 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
#include    <stdio.h>

extern const char *_foo1, *_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 the interface describes the evolution of the object. These interfaces enable users to verify and select the interfaces to bind with. These concepts are covered in more detail in "Binding to a Version Definition" on page 240 and in "Specifying a Version Binding" on page 244.

The following example shows the `mapfile` directives that create these two interfaces.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                  # Release X+1
        global:
                foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ elfdump -sN.symtab libfoo.so.1 | grep 'foo.$'
    [28]   0x107a4    0x4  OBJT LOCL  H    0 .data      _foo1
    [29]   0x107a8    0x4  OBJT LOCL  H    0 .data      _foo2
    [48]     0x5e8   0x20  FUNC GLOB  D    0 .text      foo1
    [51]     0x618   0x20  FUNC GLOB  D    0 .text      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 version `SUNW_1.1`. `foo2` is assigned to version `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 version definition is a useful technique. This inheritance reduces the version information that is eventually 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 240.

A version definition symbol is created and associated with a version definition. 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. The version definition has no global interface symbols associated with the definition.

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
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

SYMBOL_VERSION 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 detail. An application can bind to the version definition that is associated with an implementation detail that the application requires. The section "Binding to a Version Definition" on page 240 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 interface expresses a dependency on the original `SUNW_1.2` interface.

The following `mapfile` definition creates the required association.

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};
```

```
SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

SYMBOL_VERSION SUNW_1.2.1 { } SUNW_1.2;     # Release X+2

SYMBOL_VERSION SUNW_1.3a {                  # Release X+3
        global:
                bar1;
} SUNW_1.2;

SYMBOL_VERSION SUNW_1.3b {                  # Release X+3
        global:
                bar2;
} SUNW_1.2;
```

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};
```

Version definitions can be used to verify runtime binding requirements. Version definitions can also be used to control the binding of an object during the objects creation. The following sections explore these version definition usages in more detail.

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 29 and "Recording a Shared Object Name" on page 136 for more details. If a dependency also contain version definitions, then an associated version dependency is recorded in the object being built.

The following example uses the data files from the previous section to generate a shared object, `libfoo.so.1`, which is suitable for a compile time environment.

```
$ 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
```

Six public interfaces are offered by the shared object `libfoo.so.1`. 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 object. One interface, `libfoo.so.1`, defines several reserved labels. Dynamic objects created with `libfoo.so.1` 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 that is 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 only need to record the one dependency. This inheritance provides for the normalization of version definition dependencies. This normalization reduces the amount of version information that is maintained within an object. This normalization also reduces the version verification processing that is 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. A weak version definition results in a separate dependency recording.

Had there been multiple weak version definitions that inherited from each other, then these definitions are 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.

The runtime linker validates the existence of any recorded version definitions from 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 dependency 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)
```

If prog had not recorded any version definition dependencies, the nonexistence of the symbol foo2 could result in a fatal relocation error a runtime. This relocation error might occur at process initialization, or during process execution. An error condition might not occur at all if the execution path of the application did not call the function foo2. See “Relocation Errors” on page 104.

A version definition dependency provides an alternative and immediate indication of the availability of the interfaces required by the application.

For example, `prog` might run in an environment in which `libfoo.so.1` only contains the version definitions `SUNW_1.1` and `SUNW_1.2`. In this event, all non-weak version definition requirements are satisfied. The absence of the weak version definition `SUNW_1.2.1` is deemed nonfatal. In this case, no runtime error condition is generated.

```
$ 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`(1) can be used to display all version definitions that cannot be found.

```
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          (version not found)
        ....
```

At runtime, if an implementation of a dependency contains no version definition information, then any version verification of the dependency is silently ignored. This policy provides a level of backward compatibility as a transition from non-versioned to versioned shared objects occurs. `ldd`(1) can always be used to display any version requirement discrepancies.

Note - 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). An object that is added to the process's address space by using `dlopen`(3C) receives no automatic version dependency verification. Thus, the caller of `dlopen`(3C) 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). The availability of the interface `SUNW_1.2` is then verified.

```
#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());
                return (1);
        }

        if (dlsym(handle, vers) == NULL) {
                (void) printf("fatal: %s: version '%s' not found\n", file, vers);
                return (1);
        }
        ....
```

Note - The use of the `dlopen`(3C) flag `RTLD_FIRST` ensures that the `dlsym`(3C) search is restricted to `libfoo.so.1`.

Specifying a Version Binding

When creating a dynamic object that is linked 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 `DEPEND_VERSIONS` *mapfile* directive. This directive is supplied using the link-editor's `-M` option and an associated `mapfile`. The `DEPEND_VERSIONS` directive uses the following syntax.

```
$mapfile_version 2
DEPEND_VERSIONS objname {
        ALLOW   = version_name;
        REQUIRE = version_name;
        ....
};
```

- *objname* 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 30.
- The `ALLOW` attribute is used to specify version definition names within the shared object that should be made available for binding. Multiple `ALLOW` attributes can be specified.
- The `REQUIRE` attribute allows additional version definitions to be recorded. Multiple `REQUIRE` attributes can be specified.

The control of version binding 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 the object 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 that were 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
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = 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 must only use the interfaces available in `Release X`. 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
```

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 `REQUIRE` attribute to the `DEPEND_VERSIONS` *mapfiile* directive. The following sections describe scenarios where this additional binding can be useful.

Redefining an Interface

One scenario is the consumption of an ISV specific interface into a public standard interface.

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
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
        REQUIRE = 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.

Binding to a Weak Version

"Creating a Weak Version Definition" on page 238 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 `libfoo.so.1`, 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 implementation of `libfoo.so.1` that is used at runtime.

The `REQUIRE` attribute to the `DEPEND_VERSIONS` *mapfile* directive can be used to generate an explicit dependency on a version definition. If this definition is weak, then this explicit reference also 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
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
        REQUIRE = 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 an explicit dependency on the interface `STAND_A`. Because the version definition `SUNW_1.2.1` is promoted to a strong version, the version `SUNW_1.2.1` is 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

Various models have been described that provide for binding to a version definition within an object. These models allow for the runtime validation of interface requirements. This verification only remains valid if the individual version definitions remain constant over the life time of the object.

A version definition for an object can be created for other objects to bind with. This version definition must continue to exist in subsequent releases of the object. Both the version name and the symbols associated with the version must remain constant. To help enforce these requirements, wildcard expansion of the symbol names defined within a version definition is not supported. The number of symbols that can match a wildcard might differ over the course of an objects evolution. This difference can lead to accidental interface instability.

Relocatable Objects

The previous sections have described how version information can be recorded within dynamic objects. Relocatable objects can maintain versioning information in a similar manner. However, subtle differences exist regarding how this information is used.

Any version definitions supplied to the link-edit of a relocatable object are recorded in the object. These definitions follow the same format as version definitions recorded in dynamic objects. However, by default, symbol reduction is not carried out on the relocatable object being created. Symbol reductions that are defined by the versioning information are applied to the relocatable object when the object is used to create a dynamic object.

In addition, any version definition found in a relocatable object is propagated to the dynamic object. For an example of version processing in relocatable objects, see “Reducing Symbol Scope” on page 49.

Note - Symbol reduction that is implied by a version definition can be applied to a relocatable object by using the link-editors `-B reduce` option.

External Versioning

Runtime references to a shared object should always refer to the versioned file name. A versioned file name is usually expressed as a file name with a version number suffix.

Should a shared object's interface changes in an incompatible manner, such a change can break old applications. In this instance, 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 the application 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

A link-edit commonly references shared object dependencies using the link-editors `-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 as a versioned file name. Instead of maintaining two distinct shared objects that follow two naming conventions, create file system links between the two file names.

For example, the shared object `libfoo.so.1` can be made available to the compilation environment by using a symbolic link. The compilation file name is a symbolic link to the runtime file name.

```
$ 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. The symbolic link `libfoo.so`, has also enabled this file's use in a compilation environment.

```
$ 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 `libfoo.so` can be distributed with changed interfaces. The compilation environment can be constructed to use the interface that is applicable by changing the symbolic link.

```
$ 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
```

In this example, three major versions of the shared object are available. Two versions, `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.

The use of this symbolic link mechanism solely is insufficient to coordinate the compilation shared object with a runtime versioned file name. As the example presently stands, the link-editor records in the dynamic executable `prog` the file name of the shared object the link-editor processes. In this case, that file name seen by the link-editor is the compilation environment file.

```
$ elfdump -d prog | grep libfoo
     [0]  NEEDED       0x1b7           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 ensure the correct runtime name is 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 the shared object. This definition can be provided using the `-h` option during the creation of the shared object.

```
$ 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
$ elfdump -d prog | grep libfoog
     [0]  NEEDED       0x1b7           libfoo.so.1
```

This symbolic link and the *soname* mechanism establish 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.

Multiple External Versioned Files in the Same Process

The creation of a new externally versioned shared object is a major change. Be sure you understand the complete dependencies of any processes that use a member of a family of externally versioned shared objects.

For example, an application might have a dependency on `libfoo.so.1` and an externally delivered object `libISV.so.1`. This latter object might also have a dependency on

`libfoo.so.1`. The application might be redesigned to use the new interfaces in `libfoo.so.2`. However, the application might not change the use of the external object `libISV.so.1`. Depending on the scope of visibility of the implementations of `libfoo.so` that get loaded at runtime, both major versions of the file can be brought into the running process. The only reason to change the version of `libfoo.so` is to mark an incompatible change. Therefore, having both versions of the object within a process can lead to incorrect symbol binding and hence undesirable interactions.

The creation of an incompatible interface change should be avoided. Only if you have full control over the interface definition, and all of the objects that reference this definition, should an incompatible change be considered.

♦ ♦ ♦ CHAPTER 10

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 following reserved dynamic string tokens.

- `$CAPABILITY` (`$HWCAP`)
- `$ISALIST`
- `$OSNAME`, `$OSREL`, `$PLATFORM` and `$MACHINE`
- `$ORIGIN`

The following sections provide examples of how each of these tokens can be employed.

Capability Specific Shared Objects

The dynamic token `$CAPABILITY` can be used to specify a directory in which 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.

Note - The original capabilities implementation was based solely on hardware capabilities. The token `$HWCAP` was used to select this capability processing. Capabilities have since been extended beyond hardware capabilities, and the `$HWCAP` token has been replaced by the `$CAPABILITY` token. For compatibility, the `$HWCAP` token is interpreted as an alias for the `$CAPABILITY` token.

The path name specification must consist of a full path name terminated with the `$CAPABILITY` token. Shared objects that exist in the directory that is specified with the `$CAPABILITY` token are inspected at runtime. These objects should indicate their capability requirements. See

"Identifying Capability Requirements" on page 56. Each object is validated against the capabilities that are available to the process. Those objects that are applicable for use with the process, are sorted in descending order of their capability values. These sorted *filtees* are used to resolve symbols that are defined within the filter.

Filtees within the 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/cap/$CAPABILITY' \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
     [2]  SONAME            0x1               libfoo.so.1
     [3]  AUXILIARY         0x96              /opt/ISV/lib/cap/$CAPABILITY
$ elfdump -H /opt/ISV/lib/cap/*

/opt/ISV/lib/cap/filtee.so.3:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/cap/filtee.so.1:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/cap/filtee.so.2:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
   index  tag               value
     [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

If the *filter* `libfoo.so.1` is processed on a system where the `MMX` and `SSE` hardware capabilities are available, the following *filtee* search order occurs.

```
$ cc -o prog prog.c -R. -lfoo
$ LD_DEBUG=symbols prog
....
01233: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01233: symbol=foo;  lookup in file=cap/filtee.so.2  [ ELF ]
01233: symbol=foo;  lookup in file=cap/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 `$CAPABILITY` 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 `$CAPABILITY` 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 as follows.

```
$ LD_DEBUG=symbols prog
....
01424: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01424: symbol=foo;  lookup in file=cap/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.

Note - This token is obsolete, and might be removed in a future release of Oracle Solaris. See “Capability Specific Shared Objects” on page 253 for the recommended technique for handling instruction set extensions.

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
```

```
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
     [2]  SONAME            0x1               libfoo.so.1
     [3]  AUXILIARY         0x96              /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
$ elfdump -d libfoo.so.1 | egrep 'RUNPATH|AUXILIARY'
     [3]  AUXILIARY         0x96              libbar.so.1
     [4]  RUNPATH           0xa2              /opt/ISV/lib/$ISALIST
```

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 as follows.

```
$ 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`, `$PLATFORM` and `$MACHINE` 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`. `$PLATFORM` expands to reflect the underlying platform name, as displayed by `uname -i`. `$MACHINE` expands to reflect the underlying machine hardware name, as displayed by `uname -m`.

The following example shows how the auxiliary filter `libfoo.so.1` can be designed to access a platform specific *filtee* `libbar.so.1`.

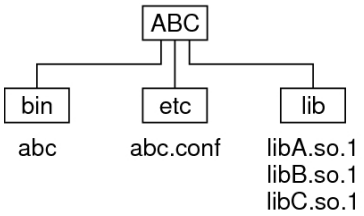
```
$ LD_OPTIONS='-f /platform/$PLATFORM/lib/libbar.so.1' \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
     [2]  SONAME            0x1               libfoo.so.1
     [3]  AUXILIARY        0x96               /platform/$PLATFORM/lib/libbar.so.1
```

This mechanism is used in the Oracle Solaris OS 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 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 10-1 Unbundled Dependencies



Assume that the product is designed for installation under `/opt`. Normally, you would augment your `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 as follows.

```
$ cc -o abc abc.c -R/opt/ABC/lib -L/opt/ABC/lib -lA
$ elfdump -d abc | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x1b5               libA.so.1
     ....
     [4]  RUNPATH          0x1bf               /opt/ABC/lib
```

Similarly, for the dependency `libA.so.1` the *runpath* would be as follows.

```
$ cc -o libA.so.1 -G -Kpic A.c -R/opt/ABC/lib -L/opt/ABC/lib -lB
$ elfdump -d libA.so.1 | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x96                libB.so.1
     [4]  RUNPATH          0xa0                /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
$ elfdump -d abc | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x1b5               libA.so.1
     ....
     [4]  RUNPATH          0x1bf               $ORIGIN/../lib
```

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
$ elfdump -d lib/libA.so.1 | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED           0x96                libB.so.1
     [4]  RUNPATH          0xa0                $ORIGIN
```

If this product is now installed under `/usr/local/ABC` and your `PATH` is augmented with `/usr/local/ABC/bin`, invocation of the application `abc` result in a path name lookup for its dependencies as follows.

```
$ ldd -s abc
....
  find object=libA.so.1; required by abc
    search path=$ORIGIN/../lib  (RUNPATH/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  (RUNPATH/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
```

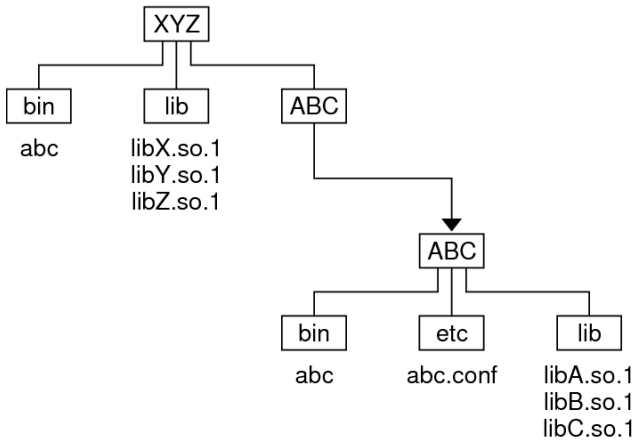
Note - Objects that contain a `$ORIGIN` token can be referenced using a symbolic link. In this case, the symbolic link is fully resolved in order to determine the true origin of the object.

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 10-2 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 as follows.

```
$ cc -o xyz xyz.c '-R$ORIGIN/../lib:$ORIGIN/../ABC/lib' \
    -L/opt/ABC/lib -lX -lA
$ elfdump -d xyz | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED          0x1b5              libX.so.1
     [1]  NEEDED          0x1bf              libA.so.1
     ....
     [2]  NEEDED          0x18f              libc.so.1
     [5]  RUNPATH         0x1c9              $ORIGIN/../lib:$ORIGIN/../ABC/lib
```

and similarly for the dependency `libX.so.1` this *runpath* would be as follows.

```
$ cc -o libX.so.1 -G -Kpic X.c '-R$ORIGIN:$ORIGIN/../ABC/lib' \
    -L/opt/ABC/lib -lY -lC
$ elfdump -d libX.so.1 | egrep 'NEEDED|RUNPATH'
     [0]  NEEDED          0x96               libY.so.1
     [1]  NEEDED          0xa0               libC.so.1
     [5]  RUNPATH         0xaa               $ORIGIN:$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.

```
$ ln -s ../ABC /usr/local/XYZ/ABC
```

If your `PATH` is augmented with `/usr/local/XYZ/bin`, then invocation of the application `xyz` results in a path name lookup for its dependencies as follows.

```
$ ldd -s xyz
....
  find object=libX.so.1; required by xyz
    search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib  (RUNPATH/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  (RUNPATH/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 \
               (RUNPATH/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 \
               (RUNPATH/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  (RUNPATH/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
```

Note - An objects origin can be obtained at runtime using `dlinfo`(3C) together with the `RTLD_DI_ORIGIN` flag. This origin path can be used to access additional files from the associated product hierarchy.

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.

♦ ♦ ♦ CHAPTER 11

Extensibility Mechanisms

The link-editor and runtime linker provide interfaces that enable the monitoring, and 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 263
- *rtld-audit* – "Runtime Linker Auditing Interface" on page 271
- *rtld-debugger* – "Runtime Linker Debugger Interface" on page 285

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. This interface provides 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 and the `make`(1S) utility. The link editor uses the interface to process debugging information within relocatable objects. The `make` utility uses the interface 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. Any support routines that are found in the library 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:support.so.2  cc ....
```

The -S option specifies a single support library. Multiple -S options can be specified.

```
$ LD_OPTIONS="-Ssupport.so.1 -Ssupport.so.2"  cc ....
```

A support library is a shared object. The link-editor opens each support library, in the order the libraries 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 Environments and 64–Bit Environments" on page 264 for more details.

Note - By default, the Solaris OS 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. The default processing of libldstab.so.1 can be required in addition to your support library services. In this case, add libldstab.so.1 explicitly to the list of support libraries that are supplied to the link-editor.

32–Bit Environments and 64–Bit Environments

As described in "32–Bit Environments and 64–Bit Environments" on page 20, the 64–bit link-editor, ld(1), is capable of generating 32–bit objects. In addition, 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 the interface 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 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 that the link-editor is capable of supporting. The support library can verify this version is sufficient for its use. The support library can then return the version that the support library 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 the version `LD_SUP_VNONE`, the link-editor silently unloads the support library, and proceeds without using it. If the returned version is greater than the *ld-support* interface the link-editor supports, a fatal error is issued, and the link-editor terminates execution. Otherwise, execution continues, using the support library at the specified *ld-support* interface version.

`ld_start`

This function is called after initial validation of the link-editor command line. This function 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/bin/ld`, or `/usr/ccs/bin/ld`.

`ld_open`

This function is called for each file input to the link-edit. This function, which was added in version `LD_SUP_VERSION3`, provides greater flexibility than the `ld_file` function. This function allows the support library to replace the file descriptor, ELF descriptor, together with the associated file names. This function provides the following possible usage scenarios.

- The addition of new sections to an existing ELF file. In this case, the original ELF descriptor should be replaced with a descriptor that allows the ELF file to be updated.

  See the `ELF_C_RDWR` argument of `elf_begin`(3ELF).
- The entire input file can be replaced with an alternative. In this case, the original file descriptor and ELF descriptor should be replaced with descriptors that are associated with the new file.

In both scenarios the path name and file name can be replaced with alternative names that indicate the input file has been modified.

```
void ld_open(const char **pname, const char **fname, int *fd,
    int flags, Elf **elf, Elf *ref, size_t off, Elf_Kind kind);

void ld_open64(const char **pname, const char **fname, int *fd,
    int flags, Elf **elf, Elf *ref, size_t off, Elf_Kind kind);
```

`pname` is the path name of the input file about to be processed. `fname` is the file name of the input file about to be processed. `fname` is typically the base name of the `pname`. Both `pname` and `fname` can be modified by the support library.

fd is the file descriptor of the input file. This descriptor can be closed by the support library, and a new file descriptor can be returned to the link-editor. A file descriptor with the value `-1` can be returned to indicate that the file should be ignored.

Note - The *fd* passed to `ld_open` is set to the value `-1` if the link-editor is unable to allow `ld_open` to close the file descriptor. The most common reason where this can occur is in the case of processing an archive member. If a value of `-1` is passed to `ld_open`, the descriptor can not be closed, nor should a replacement descriptor be returned by the support library.

The `flags` field indicates how the link-editor obtained the file. This field can be one or more of the following definitions.

- `LD_SUP_DERIVED` – The file name was not explicitly named on the command line. The file was derived from a `-l` expansion. Or, the file 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 the ELF descriptor of the input file. This descriptor can be closed by the support library, and a new ELF descriptor can be returned to the link-editor. An ELF descriptor with the value `0` can be returned to indicate that the file should be ignored. When the *elf* descriptor is associated with a member of an archive library, the *ref* descriptor is the ELF descriptor of the underlying archive file. The *off* represents the offset of the archive member within the archive file.

`kind` indicates the input file type, which is either `ELF_K_AR`, or `ELF_K_ELF`, as defined in `libelf.h`.

`ld_file`

This function is called for each file input to the link-edit. This function is called 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`. The `flags` field indicates how the link-editor obtained the file. This field can contain the same definitions as the `flags` field for `ld_open`.

- `LD_SUP_DERIVED` – The file name was not explicitly named on the command line. The file was derived from a `-l` expansion. Or, the file 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 the ELF descriptor of the input file.

`section`

This function is called for each section of the input file. This function, which was added in version `LD_SUP_VERSION2`, 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.

This function is called before compressed sections are decompressed, complicating the task of examining the data or replacing the data. For this reason, it is recommended that support libraries defer the examination and possible replacement of section data to `ld_section`.

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 is propagated to the output file. This function is called before any processing of the section data is carried out. However, sections containing compressed data are decompressed before this function is called.

```
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 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.

Note - Sections that are removed from the output file are not reported to `ld_section`. Sections are stripped using the link-editor's `-z strip-class` option. Sections are discarded due to `SHT_SUNW_COMDAT` processing or `SHF_EXCLUDE` identification. See *"COMDAT Section" on page 340*, and Table 12-8.

`ld_input_done`

This function, which was added in version `LD_SUP_VERSION2`, is called when input file processing is complete.

At this point, all input sections have been assigned to the output file image. In addition, the information required to create and update this image, has been collected in preparation to apply to the initial image. Great care should be exercised with any attempt by `ld_input_done` to alter any data recorded from previous support routines. Any change to the identification or relationship of sections will be lost, or may compromise the creation

of the output file image. Minor updates, such as the addition of section data that does not require relocation, or affect existing relocations, can be applied.

```
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 the following.

```
$ 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 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 that are 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 you to access information pertaining to the runtime linking of a process. 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 following information.

- 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.
- The lazy binding mechanism that is provided by procedure linkage table entries, allow the auditing of function calls and their return values. See "Procedure Linkage Table (Processor-Specific)" on page 416. The arguments to a function and return value of a function can be modified by the audit library.

Some of this information can be obtained by preloading specialized shared objects. However, a preloaded object exists within the same namespace as the objects of a application. This preloading often restricts, or complicates the implementation of the preloaded shared object. The *rtld-audit interface* offers you a unique namespace in which to execute audit libraries. This namespace ensures that the audit library does not intrude upon the normal bindings that occur within the application.

An example of using the *rtld-audit interface* is the runtime profiling of shared objects that is described in "Profiling Shared Objects" on page 192.

Establishing a Namespace

When the runtime linker binds a dynamic executable with its dependencies, a linked list of *link-maps* is generated to describe the application. The link-map structure describes each object within the application. The link-map structure is defined in `/usr/include/sys/link.h`. The symbol search mechanism that is required to bind together the objects of an application traverse this list of link-maps. This link-map list is said to provide the *namespace* for the applications symbol resolution.

The runtime linker is also described by a link-map. This link-map is maintained on a different list from the list of application objects. The runtime linker therefore resides in its own unique namespace, which prevents the application from seeing, or being able to directly access, any services within the runtime linker. An application can therefore only access the runtime linker through the filters provided by `libc.so.1`, or `libdl.so.1`.

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 */
```

In addition to these two standard link-map lists, the runtime linker allows the creation of an arbitrary number of additional link-map lists. Each of these additional link-map lists provides a unique namespace. The *rtld-audit interface* employs its own link-map list on which the audit libraries are maintained. The audit libraries are therefore isolated from the symbol binding requirements of the application. Every *rtld-audit* support library is assigned a unique new link-map identifier.

An audit library can inspect the application link-map list using `dlmopen`(3C). When `dlmopen` is used with the `RTLD_NOLOAD` flag, the audit library can query the existence of an object without causing the object to be loaded.

Creating an Audit Library

An audit library is built like any other shared object. However, the audit libraries 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 an audit library references external interfaces, then the audit library must define the dependency that provides the interface definition. For example, 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 43. Because the audit library has a unique namespace, symbol references cannot be satisfied by the `libc` that is 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 the interfaces are the only instance of their implementation within a process. Examples of such implementations are 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.

- *Local* auditing is enabled by defining one or more auditors at the time the object is built. See "Recording Local Auditors" on page 273. The audit libraries that are made available at runtime by this method are provided with information regarding the dynamic objects that have requested local auditing.
- *Global* auditing is enabled by defining one or more auditors using the environment variable `LD_AUDIT`. Global auditing can also be enabled for an application by combining a local auditing definition with the `-z globalaudit` option. See "Recording Global Auditors" on page 274. The audit libraries that are made available at runtime by these methods are provided with information regarding all dynamic objects used by the application.

Both methods of defining auditors employ a string consisting of 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.

Secure applications can only obtain audit libraries from trusted directories. By default, the only trusted directories that are 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`.

Note - Auditing can be disabled at runtime by setting the environment variable `LD_NOAUDIT` to a non-null value.

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
$ elfdump -d libfoo.so.1 | grep AUDIT
     [2]  AUDIT            0x96               audit.so.1
```

At runtime, the existence of this audit identifier results in the audit library being loaded. Information is then passed to the audit library 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 with a dependency on `libfoo.so.1`, then the application is identified to indicate its dependencies require auditing.

```
$ cc -o main main.c libfoo.so.1
$ elfdump -d main | grep AUDIT
     [4]  DEPAUDIT          0x1be              audit.so.1
```

The auditing enabled through 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
$ elfdump -d main | grep AUDIT
     [3]  DEPAUDIT          0x1b2              audit.so.1
```

Recording Global Auditors

Global auditing requirements can be established by setting the environment variable `LD_AUDIT`. For example, this environment variable can be used to audit the application `main` together with all the dependencies of the application, with the audit library `audit.so.1`.

```
$ LD_AUDIT=audit.so.1 main
```

Global auditing can also be achieved by recording a local auditor in the application, together with the `-z globalaudit` option. For example, the application `main` can be built to enable global auditing by using the link-editor's `-P` option and `-z globalaudit` option.

```
$ cc -o main main.c -Wl,-Paudit.so.1 -z globalaudit
$ elfdump -d main | grep AUDIT
     [3]  DEPAUDIT          0x1b2              audit.so.1
    [26]  FLAGS_1       0x1000000             [ GLOBAL-AUDITING ]
```

The auditing enabled through either of these mechanisms results in the audit library being passed information regarding *all* of the dynamic objects of the application.

Audit Interface Interactions

Audit routines are provided one or more *cookies*. A cookie is a data item that describes an individual dynamic object. An initial cookie is provided to the `la_objopen` routine when a dynamic object is initially loaded. This cookie is a pointer to the associated `Link_map` of the loaded dynamic object. However, the `la_objopen` routine is free to allocate, and return to the runtime linker, an alternative cookie. This mechanism provides the auditor a means of maintaining their own data with each dynamic object, and receiving this data with all subsequent audit routine calls.

The *rtld-audit interface* enables multiple audit libraries to be supplied. In this case, the return information from one auditor is passed to the same audit routine of the next auditor. Similarly, a cookie that is established by one auditor is passed to the next auditor. Care should be taken when designing an audit library that expects to coexist with other audit libraries. A safe approach should not alter the bindings, or cookies, that would normally be returned by the runtime linker. Alteration of these data can produce unexpected results from audit libraries that follow. Otherwise, all auditors should be designed to cooperate in safely changing any binding or cookie information.

Audit Interface Functions

The following routines are provided by the *rtld-audit interface*. The routines 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 routine 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* the runtime linker is capable of supporting. The audit library can verify this version is sufficient for its use, and return the version the audit library 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 that is greater than the *rtld-audit interface* the runtime linker supports, the audit library is discarded.

The remaining audit routines are provided one or more *cookies*. See “Audit Interface Interactions” on page 274.

Following the `la_version` call, two calls are made to the `la_objopen` routine. The first call provides link-map information for the dynamic executable, and the second call provides link-map information for the runtime linker.

`la_objopen`

This routine 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 reassigned by the audit library to better identify the object to other *rtld-audit interface* routines.

The `la_objopen` routine 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 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`.

With the auditing version `LAV_VERSION5`, an `la_objopen` call that represents the dynamic executable is provided to a local auditor. In this case, the auditor should not return a symbol binding flag, as the auditor may have been loaded too late to monitor any symbol bindings associated with the dynamic executable. Any flags that are returned by the auditor are ignored. The `la_objopen` call provides the local auditor an initial cookie which is required for any subsequent `la_preinit` or `la_activity` calls.

`la_activity`

This routine 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.

An `LA_ACT_ADD` activity is called on process start up, following the `la_objopen` calls for the dynamic executable and runtime linker, to indicate that new dependencies are being added. This activity is also called for lazy loading and `dlopen`(3C) events. An `LA_ACT_DELETE` activity is also called when objects are deleted with `dlclose`(3C).

Both the `LA_ACT_ADD` and `LA_ACT_DELETE` activities are a *hint* of the events that are expected to follow. There are a number of scenarios where the events that unfold might be

different. For example, the addition of new objects can result in some of the new objects being deleted should the objects fail to relocate fully. The deletion of objects can also result in new objects being added should `.fini` executions result in lazy loading new objects. An `LA_ACT_CONSISTENT` activity follows any object additions or object deletions, and can be relied upon to indicate that the application link-map list is consistent. Auditors should be careful to verify actual results rather than blindly trusting the `LA_ACT_ADD` and `LA_ACT_DELETE` hints.

For auditing versions `LAV_VERSION1` through `LAV_VERSION4`, `la_activity` was only called for global auditors. With the auditing version `LAV_VERSION5`, activity events can be obtained by local auditors. An activity event provides a cookie that represents the application link-map. To prepare for this activity, and allow the auditor to control the content of this cookie, an `la_objopen` call is first made to the local auditor. The `la_objopen` call provides an initial cookie representing the application link-map. See "Audit Interface Interactions" on page 274.

`la_objsearch`

This routine 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` – The initial search name. Typically, this name indicates the file name that is recorded as a `DT_NEEDED` entry, or the argument supplied to `dlopen`(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_objfilter`

This routine is called when a filter loads a new *filtee*. See "Shared Objects as Filters" on page 140.

```
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 calls to `la_objopen` for both the filter and *filtee* have been made.

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 routine is called once after all immediate dependencies have been loaded for the application.

```
void la_preinit(uintptr_t *cookie);
```

`cookie` identifies the primary object that started the process, normally the dynamic executable.

When `la_preinit` is called, the process still requires threads initialization, including the creation of any initial thread local storage. See "Program Startup" on page 430. In addition, the initialization sections of all loaded objects still require collecting and sorting prior to their execution. See "Initialization and Termination Routines" on page 110. This function provides a convenient control point to add additional objects to the initial process. These objects can contribute to the initial thread local storage, and initialization of the process.

For auditing versions `LAV_VERSION1` through `LAV_VERSION4`, `la_preinit` was only called for global auditors. With the auditing version `LAV_VERSION5`, a preinit event can be obtained by local auditors. A preinit event provides a cookie that represents the application link-map. To prepare for this preinit, and allow the auditor to control the content of this cookie, an `la_objopen` call is first made to the local auditor. The `la_objopen` call provides an initial cookie representing the application link-map. See "Audit Interface Interactions" on page 274.

`la_callinit`

This routine is called after threads initialization has completed, and all initial thread local storage has been established. In addition, all initialization routines have been collected and sorted ready for execution.

```
void la_callinit(uintptr_t *cookie);
```

`cookie`, and the calling from global or local auditors, is as described for `la_preinit`.

This interface, added with auditing version `LAV_VERSION6`, marks the transition to executing application code.

`la_callentry`

This routine is called after all initialization routines have been executed.

```
void la_callentry(uintptr_t *cookie);
```

`cookie`, and the calling from global or local auditors, is as described for `la_preinit`.

This interface, added with auditing version LAV_VERSION6, marks the transition to the applications entry point.

la_symbind

This routine 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 routine 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 this procedure linkage table entry. 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 routines exist, these routines are called after la_symbind for procedure linkage table entries. These routines are called each time that the symbol is referenced. See also “Audit Interface Limitations” on page 284.

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 routine for this symbol.
- LA_SYMB_NOPLTEXIT – Do *not* call the la_pltexit routine 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 routines are system specific. These routines 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);

uintptr_t la_amd64_pltenter(Elf64_Sym *sym, uint_t ndx,
    uintptr_t *refcook, uintptr_t *defcook,
    La_amd64_regs *regs, uint_t *flags, const char *sym_name);
```

`sym`, `ndx`, `refcook`, `defcook` and `sym_name` provide the same information as passed to `la_symbind`.

For `la_sparcv8_pltenter` and `la_sparcv9_pltenter`, `regs` points to the out registers. For `la_i86_pltenter`, `regs` points to the stack and frame registers. For `la_amd64_pltenter`, `regs` points to the stack and frame registers, and the registers used in passing integer arguments. `regs` are defined in `/usr/include/link.h`.

`flags` points to a data item that can convey information regarding the binding. This data item can be used to modify the continued 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 routine is called when a procedure linkage table entry, between two objects that have been tagged for binding notification, returns. This routine 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 284.

`la_objclose`

This routine 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 Control Flow

The following sections describe the auditing interface routines and actions an audit library can perform with each interface. The emphasis is on process initialization. These routines are presented in the order they are called in the common case of a global auditor that is provided at process startup.

Auditing interfaces fall into one of two categories, informational, and control.

Informational interfaces provide the audit library information about the executing process, such as object searching, object loading, and symbol bindings. In addition, these interfaces allow the auditor to modify the objects loaded, and to ask for notification of future symbol binding events.

Control interfaces are called to allow the audit library to track the start or end of a phase of activity within the process execution. These interfaces allow the auditor to safely inspect a consistent set of objects, and can even allow new objects to be loaded.

When an auditing library is first loaded, an immediate call is made to the library's `la_version` interface. This handshake verifies that the audit library is supportable, and allows the audit library to define the interface version that the library requires from the runtime linker.

An audit library can be established at process startup, either from using `LD_AUDIT`, or from a local auditing definition within the executable object that starts the process. See "Invoking the Auditing Interface" on page 273. In this scenario, an `la_objopen` call, for both the executable object, and the runtime linker, are provided to the audit library.

At this point the process is still in the early stages of construction. The auditor should refrain from performing any actions that might disturb this construction, such as adding additional objects to the process, or exhaustive symbol searches of the process. These actions can result in prematurely loading and relocating objects in an attempt to satisfy a symbol look up.

Dependencies that are immediately loaded at process initialization are each reported to the auditor library's `la_objopen` interface. For processes that employ lazy loading, only a few dependencies may be loaded at process initialization. See "Lazy Loading of Dynamic Dependencies" on page 106. Each loaded object is relocated, which results in symbol bindings being established between symbol references and symbol definitions. These bindings are reported to the audit library's `la_symbind` interface.

Once all immediate dependencies have been loaded, and relocated, the audit library's `la_preinit` interface is called. At this point, the process is still under construction. Threads initialization and initialization routine collection are still pending. However, this interface provides a convenient control point to add additional objects to the initial process.

Once threads initialization is completed, the audit library's `la_callinit` interface is called. At this point, all loaded objects are ready to execute, and their initialization routines have been collected and sorted in preparation for execution. See "Initialization and Termination Routines" on page 110. The `la_callinit` control point marks the transition to executing application code.

The execution of application code results in function call bindings being established between symbol references and symbol definitions. These bindings are reported to the audit library's `la_symbind` and/or `la_pltenter` interfaces. With lazy loading, additional objects can be loaded to satisfy symbol references, which are reported to the audit library's `la_objopen`.

Once all initialization code has been executed, the audit library's `la_callentry` interface is called. The `la_callentry` control point marks the end of processes initialization, and the transition to the applications entry point, typically `start` or `main`.

As the process continues to execute, more symbol bindings can occur, resulting in `la_symbind` and/or `la_pltenter` calls. Addition dependencies can be loaded, resulting in `la_objopen` calls. New dependencies can also be unloaded, resulting in `la_objclose` calls. Any loading or unloading of objects is bound by a pair of `la_activity` calls. The first `la_activity` hints at the targeted behavior, an object addition or deletion. The second `la_activity` indicates that the dependency structure within the process is consistent. Auditors should restrict their inspection of the process to follow a consistent notification.

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 2012
```

Audit Interface Demonstrations

A number of demonstration applications that use the *rtld-audit* interface are provided in the `pkg:/solaris/source/demo/system` 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 pkg:/developer/linker package. perfcnt and symbindrep are example programs. These applications are not intended for use in a production environment.

Audit Interface Limitations

Limitations exist within the *rtld-audit* implementation. Take care to understand these limitation when designing an auditing library.

Exercising Application Code

An audit library receives information as objects are added to a process. At the time the audit library receives such information, the object being monitored might not be ready to execute. For example, an auditor can receive an la_objopen call for a loaded object. However, the object must load its own dependencies and be relocated before any code within the object can be exercised. An audit library might want to inspect the loaded object by obtaining a handle using dlopen(3C). This handle can then be used to search for interfaces using dlsym(3C). However, interfaces obtained in this manner should not be called unless it is known that the initialization of the destination object has completed.

Use of la_pltexit

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 that is 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 from the application the debugger is 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 the interface to inspect the dynamic aspects of the target processes. A 64–bit controlling process can debug both 64–bit targets 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 perform the following.

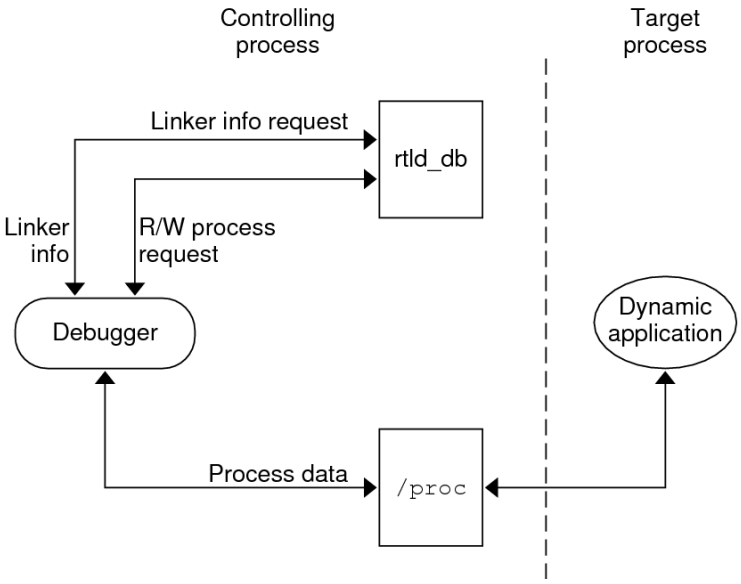
- Look up symbols in a target process.
- Read and write memory in the target process.

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 295.

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 11-1 *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 `pkg:/solaris/source/demo/system` 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 289.

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 293.

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;
        unsigned long   rl_tlsmodid;
} 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 271).

`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 295).

`rl_padend`

The base address of the padding after the dynamic object (refer to "Dynamic Object Padding" on page 295).

`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 13-8).

`rl_tlsmodid`

This field, added with `RD_VERSION4`, provides the module identifier for thread local storage, `TLS`, references. The module identifier is a small integer unique to the object. This identifier can be passed to the `libc_db` function `td_thr_tlsbase` in order to obtain the base address of a thread's `TLS` block for the object in question. See `td_thr_tlsbase`(3C_DB).

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 by means of 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 from a `mmapobj`(2) request. Effectively, an area of the virtual address space of the target process, adjacent to any loaded objects, is reserved. These areas can later be used by the controlling process.

A `padsize` of 0 disables any object padding for later objects.

Note - Reservations obtained using `mmapobj`(2) 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.

PART IV

ELF Application Binary Interface

♦ ♦ ♦ CHAPTER 12

Object File Format

This chapter describes the executable and linking format (ELF) of the object files produced by the assembler and link-editor. Three significant types of object file exist.

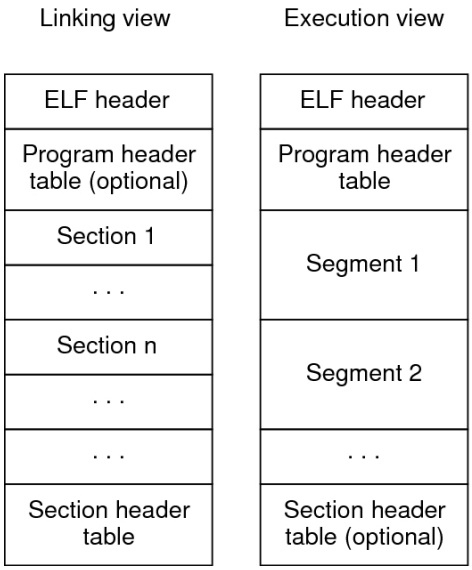
- A *relocatable object* file holds sections containing code and data. This file is suitable to be linked with other relocatable object files to create dynamic executable files, shared object files, or another relocatable object.
- A *dynamic executable* file holds a program that is ready to execute. The file specifies how `exec`(2) creates a program's process image. This file is typically bound to shared object files at runtime to create a process image.
- A *shared object* file holds code and data that is suitable for additional linking. The link-editor can process this file with other relocatable object files and shared object files to create other object files. The runtime linker combines this file with a dynamic executable file and other shared object files to create a process image.

Programs can manipulate object files with the functions that are provided by 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 `pkg:/solaris/source/demo/system` 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 12-1 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 the Oracle Solaris OS.

Sections represent the smallest indivisible units that can be processed within an ELF file. *Segments* are a collection of sections. Segments 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. This data includes instructions, data, symbol table, and relocation information. 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, executable files and shared objects, must have a program header table. Relocatable object files do not need a program header 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 and section size. Files that are used in link-editing must have a section header table.

Data Representation

The object file format supports various processors with 8-bit bytes, 32–bit architectures and 64–bit architectures. Nevertheless, the data representation is intended to be extensible to larger, or smaller, architectures. Table 12-1 and Table 12-2 list the 32–bit data types and 64–bit data types.

Object files represent some control data with a machine-independent format. This format provides for the common identification and interpretation of object files. 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 12-1 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 12-2 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

Name	Size	Alignment	Purpose
`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. Data structures can 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 is aligned on a 4-byte boundary within the file. Similarly, a structure containing an `Elf64_Addr` member is aligned on an 8–byte boundary.

Note - For portability, ELF uses no bit-fields.

ELF Header

Some control structures within object files can grow because the ELF header contains their actual sizes. If the object file format does change, 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 is specified if and when extensions are defined.

The ELF header has the following structure. See `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;
```

`e_ident`

The initial bytes mark the file as an object file. These bytes provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear in "ELF Identification" on page 308.

`e_type`

Identifies the object file type, as listed in the following table.

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_LOSUNW`	`0xfefe`	Start operating system specific range
`ET_SUNW_ANCILLARY`	`0xfefe`	Ancillary object file
`ET_HISUNW`	`0xfefd`	End operating system specific range
`ET_LOPROC`	`0xff00`	Start processor-specific range
`ET_HIPROC`	`0xffff`	End processor-specific range

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 for future use.

`e_machine`

Specifies the required architecture for an individual file. Relevant architectures are listed in the following table.

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
EM_AMD64	62	AMD 64

Other values are reserved for future use. Processor-specific ELF names are distinguished by using the machine name. For example, the flags defined for `e_flags` use the prefix `EF_`. A flag that is 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.

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.

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.

If the number of program headers is greater than or equal to `PN_XNUM` (`0xffff`), this member has the value `PN_XNUM` (`0xffff`). The actual number of program header table entries is contained in the `sh_info` field of the section header at index `0`. Otherwise, the `sh_info` member of the initial section header entry contains the value zero. See Table 12-6 and Table 12-7.

`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`), `e_shnum` has the value zero. 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 section header entry contains the value zero. See Table 12-6 and Table 12-7.

`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 the `sh_link` field of the section header at index `0`. Otherwise, the `sh_link` member of the initial section header entry contains the value zero. See Table 12-6 and Table 12-7.

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 12-3 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 following values.

`EI_MAG0 - EI_MAG3`

A 4–byte *magic number*, identifying the file as an ELF object file, as listed in the following table.

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.

Name	Value	Meaning
`ELFCLASSNONE`	`0`	Invalid class
`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. The data that is 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. This class uses the basic types that are defined in Table 12-1.

Class `ELFCLASS64` is reserved for 64–bit architectures such as 64–bit SPARC and x64. This class uses the basic types that are defined in Table 12-2.

`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.

Name	Value	Meaning
`ELFDATANONE`	`0`	Invalid data encoding
`ELFDATA2LSB`	`1`	See Figure 12-2.

Name	Value	Meaning
ELFDATA2MSB	2	See Figure 12-3.

More information on these encodings appears in the section "Data Encoding" on page 310. Other values are reserved for future use.

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 together with the 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. ABI values relevant to Oracle Solaris are listed in the following table

Name	Value	Meaning
ELFOSABI_NONE / ELFOSABI_SYSV	0	No extensions or unspecified
ELFOSABI_SOLARIS	6	Solaris

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 zero is used to indicate unspecified.

EI_PAD

This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and are set to zero. Programs that read object files should ignore these values.

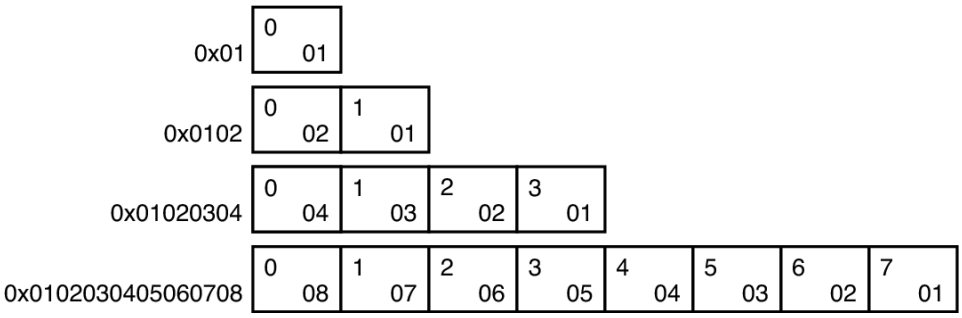
Data Encoding

A file's data encoding specifies how to interpret the integer types in a file. Class ELFCLASS32 files and class ELFCLASS64 files use integers that occupy 1, 2, 4, and 8 bytes to represent offsets,

addresses and other information. Under the defined encodings, objects are represented as described by the figures that follow. Byte numbers appear in the upper left corners.

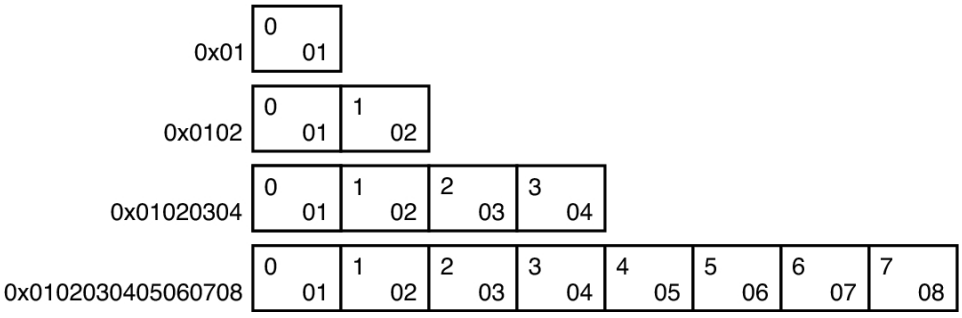
`ELFDATA2LSB` encoding specifies 2's complement values, with the least significant byte occupying the lowest address. This encoding if often referred to informally as *little endian*.

FIGURE 12-2 Data Encoding ELFDATA2LSB



`ELFDATA2MSB` encoding specifies 2's complement values, with the most significant byte occupying the lowest address. This encoding if often referred to informally as *big endian*.

FIGURE 12-3 Data Encoding ELFDATA2MSB



Sections

An object file's section header table allows you to locate all of the sections of the file. The section header table is an array of `Elf32_Shdr` or `Elf64_Shdr` structures. A section header table

index is a subscript into this array. The ELF header's `e_shoff` member indicates the byte offset from the beginning of the file to the section header table. The `e_shnum` member indicates how many entries that the section header table contains. The `e_shentsize` member indicates 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`). 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 the value zero.

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 12-4 ELF Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_AMD64_LCOMMON	0xff02
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 indicates a file has `6` entries in the section header table, the sections 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_LOSUNW` - `SHN_HISUNW`

Values in this inclusive range are reserved for Sun-specific semantics.

`SHN_SUNW_IGNORE`

This section index provides a temporary symbol definition within relocatable objects. Reserved for internal use by `dtrace`(1M).

`SHN_BEFORE, SHN_AFTER`

Provide for initial and final section ordering in conjunction with the `SHF_LINK_ORDER` and `SHF_ORDERED` section flags. See Table 12-8.

`SHN_AMD64_LCOMMON`

x64 specific common block label. This label is similar to `SHN_COMMON`, but provides for identifying a large common block.

`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 the section index 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 the section. 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. See `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;
```

`sh_name`

The name of the section. This members 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 12-12.

`sh_type`

Categorizes the section's contents and semantics. Section types and their descriptions are listed in Table 12-5.

`sh_flags`

Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions are listed in Table 12-8.

`sh_addr`

If the section appears 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 the value zero.

`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 the section occupies no space in the file.

`sh_link`

A section header table index link, whose interpretation depends on the section type. Table 12-9 describes the values.

`sh_info`

Extra information, whose interpretation depends on the section type. Table 12-9 describes the values. If the `sh_flags` field for this section header includes the attribute `SHF_INFO_LINK`, then this member represents a section header table index.

`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. In this case, 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 the value zero 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 12-5 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
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	0x6fffffee
SHT_SUNW_ancillary	0x6fffffee
SHT_SUNW_capchain	0x6fffffef
SHT_SUNW_capinfo	0x6ffffff0
SHT_SUNW_symsort	0x6ffffff1

Name	Value
SHT_SUNW_tlssort	0x6ffffff2
SHT_SUNW_LDYNSYM	0x6ffffff3
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_cap	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_AMD64_UNWIND	0x70000001
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
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`, `SHT_SUNW_LDYNSYM`

Identifies a symbol table. Typically, a `SHT_SYMTAB` section provides symbols for link-editing. As a complete symbol table, the table can contain many symbols that are 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.

`SHT_DYNSYM` can also be augmented with a `SHT_SUNW_LDYNSYM` section. This additional section provides local function symbols to the runtime environment, but is not required for dynamic linking. This section allows debuggers to produce accurate stack traces in runtime contexts when the non-allocable `SHT_SYMTAB` is not available, or has been stripped from the file. This section also provides the runtime environment with additional symbolic information for use with `dladdr`(3C).

When both a `SHT_SUNW_LDYNSYM` section and a `SHT_DYNSYM` section exist, the link-editor places their data regions immediately adjacent to each other. The `SHT_SUNW_LDYNSYM` section precedes the `SHT_DYNSYM` section. This placement allows the two tables to be viewed as a single larger contiguous symbol table, containing a reduced set of symbols from `SHT_SYMTAB`.

See "Symbol Table Section" on page 365 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 364 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 351 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 345 for details.

`SHT_DYNAMIC`

Identifies information for dynamic linking. Currently, an object file can have only one dynamic section. See "Dynamic Section" on page 398 for details.

`SHT_NOTE`

Identifies information that marks the file in some way. See "Note Section" on page 349 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 351 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 340 for details.

`SHT_SYMTAB_SHNDX`

Identifies a section containing extended section indexes, that are 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. This array contains one entry 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 Oracle Solaris OS semantics.

`SHT_SUNW_ancillary`

Indicates that the object is part of a group of ancillary objects. Contains information required to identify all the files that make up the group. See "Ancillary Section" on page 338 for details.

`SHT_SUNW_capchain`

An array of indices that collect capability family members. The first element of the array is the chain version number. Following this element are a chain of `0` terminated capability symbol indices. Each `0` terminated group of indices represents a capabilities family. The first element of each family is the capabilities lead symbol. The following elements point to family members. See "Capabilities Section" on page 342 for details.

`SHT_SUNW_capinfo`

An array of indices that associate symbol table entries to capabilities requirements, and their lead capabilities symbol. An object that defines symbol capabilities contains a `SHT_SUNW_cap` section. The `SHT_SUNW_cap` section header information points to the associated `SHT_SUNW_capinfo` section. The `SHT_SUNW_capinfo` section header information points to the associated symbol table section. See "Capabilities Section" on page 342 for details.

`SHT_SUNW_symsort`

An array of indices into the dynamic symbol table that is formed by the adjacent `SHT_SUNW_LDYNSYM` section and `SHT_DYNSYM` section. These indices are relative to the start of the `SHT_SUNW_LDYNSYM` section. The indices reference those symbols that contain memory addresses. The indices are sorted such that the indices reference the symbols by increasing address.

`SHT_SUNW_tlssort`

An array of indices into the dynamic symbol table that is formed by the adjacent `SHT_SUNW_LDYNSYM` section and `SHT_DYNSYM` section. These indices are relative to the start of the `SHT_SUNW_LDYNSYM` section. The indices reference thread-local storage symbols. See Chapter 14, "Thread-Local Storage". The indices are sorted such that the indices reference the symbols by increasing offset.

`SHT_SUNW_LDYNSYM`

Dynamic symbol table for non-global symbols. See previous `SHT_SYMTAB`, `SHT_DYNSYM`, `SHT_SUNW_LDYNSYM` description.

`SHT_SUNW_dof`

Reserved for internal use by `dtrace`(1M).

`SHT_SUNW_cap`

Specifies capability requirements. See "Capabilities Section" on page 342 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 unsatisfied relocations against this section.

`SHT_SUNW_DEBUGSTR`, `SHT_SUNW_DEBUG`

Identifies debugging information. Sections of this type are stripped from the object using the link-editor's `-z strip-class` option, or after the link-edit using `strip`(1).

`SHT_SUNW_move`

Identifies data to handle partially initialized symbols. See "Move Section" on page 347 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 340 for details.

`SHT_SUNW_syminfo`

Identifies additional symbol information. See "*Syminfo* Table Section" on page 377 for details.

`SHT_SUNW_verdef`

Identifies fine-grained versions defined by this file. See "Version Definition Section" on page 379 for details.

`SHT_SUNW_verneed`

Identifies fine-grained dependencies required by this file. See "Version Dependency Section" on page 381 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 383 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_AMD64_UNWIND`

Identifies x64 specific data, containing unwind function table entries for stack unwinding.

`SHT_LOUSER`

Specifies the lower boundary of the range of indexes that are reserved for application programs.

`SHT_HIUSER`

Specifies the upper boundary of the range of indexes that are 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 12-6 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

Name	Value	Note
sh_addralign	0	No alignment
sh_entsize	0	No entries

Should the number of sections or program headers exceed the ELF header data sizes, elements of section header 0 are used to define extended ELF header attributes. The following table shows the values.

TABLE 12-7 ELF Extended 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	e_shnum	The number of entries in the section header table
sh_link	e_shstrndx	The section header index of the entry that is associated with the section name string table
sh_info	e_phnum	The number of entries in the program header table
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 12-8 ELF Section Attribute Flags

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MERGE	0x10
SHF_STRINGS	0x20

Name	Value
SHF_INFO_LINK	0x40
SHF_LINK_ORDER	0x80
SHF_OS_NONCONFORMING	0x100
SHF_GROUP	0x200
SHF_TLS	0x400
SHF_COMPRESSED	0x800
SHF_MASKOS	0x0ff00000
SHF_SUNW_NODISCARD	0x00100000
SHF_SUNW_ABSENT	0x00200000
SHF_SUNW_PRIMARY	0x00400000
SHF_MASKPROC	0xf0000000
SHF_AMD64_LARGE	0x10000000
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000

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 are 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 header's 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 linked-to section must be unordered, and cannot in turn specify SHF_LINK_ORDER or SHF_ORDERED.

The special sh_link values SHN_BEFORE and SHN_AFTER (see Table 12-4) 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 are contiguous. These section have the same relative ordering as the sections 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 object file containing this section is rejected with an error.

SHF_GROUP

This section is a member, perhaps the only member, 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 that are contained in relocatable objects. See “Group Section” on page 340 for details.

SHF_TLS

This section holds thread-local storage. Each thread within a process has a distinct instance of this data. See Chapter 14, “Thread-Local Storage” for details.

`SHF_COMPRESSED`

Identifies a section containing compressed data. `SHF_COMPRESSED` applies only to non-allocable sections, and cannot be used in conjunction with `SHF_ALLOC`. In addition, `SHF_COMPRESSED` cannot be applied to sections of type `SHT_NOBITS`. See "Section Compression" on page 330 for details.

`SHF_MASKOS`

All bits that are included in this mask are reserved for operating system-specific semantics.

`SHF_SUNW_NODISCARD`

This section cannot be discarded by the link-editor, and is always copied to the output object. The link-editor provides the ability to discard unused input sections from a link-edit. The `SHF_SUNW_NODISCARD` section flag excludes the section from such optimizations.

`SHF_SUNW_ABSENT`

Indicates that the data for this section is not present in this file. When ancillary objects are created, the primary object and any ancillary objects, all have the same section header array. This organization facilitates the merging of the information contained in these objects, and allows the use of a single symbol table. Each file contains a subset of the section data. The data for allocable sections is written to the primary object while the data for non-allocable sections is written to an ancillary file. The `SHF_SUNW_ABSENT` flag indicates that the data for the section is not present in the object being examined. When the `SHF_SUNW_ABSENT` flag is set, the `sh_size` field of the section header must be 0. An application encountering an `SHF_SUNW_ABSENT` section can choose to ignore the section, or to search for the section data within one of the related ancillary files. See "Debugger Access and Use of Ancillary Objects" on page 85.

`SHF_SUNW_PRIMARY`

The default behavior when ancillary objects are created is to write all allocable sections to the primary object and all non-allocable sections to the ancillary objects. The `SHF_SUNW_PRIMARY` flag overrides this behavior. Any output section containing one more input section with the `SHF_SUNW_PRIMARY` flag set is written to the primary object.

`SHF_MASKPROC`

All bits that are included in this mask are reserved for processor-specific semantics.

`SHF_AMD64_LARGE`

The default compilation model for x64 only provides for 32–bit displacements. This displacement limits the size of sections, and eventually segments, to 2 Gbytes. This attribute flag identifies a section that can hold more than 2 Gbyte. This flag allows the linking of object files that use different code models.

An x64 object file section that does not contain the `SHF_AMD64_LARGE` attribute flag can be freely referenced by objects using small code models. A section that contains this flag can only be referenced by objects that use larger code models. For example, an x64 medium

code model object can refer to data in sections that contain the attribute flag and sections that do not contain the attribute flag. However, an x64 small code model object can only refer to data in a section that does not contain this flag.

`SHF_ORDERED`

`SHF_ORDERED` is an older version of the functionality provided by `SHF_LINK_ORDER`, and has been superseded by `SHF_LINK_ORDER`. `SHF_ORDERED` offers two distinct and separate abilities. First, an output section can be specified, and second, special ordering requirements are required from the link-editor.

The `sh_link` field of an `SHF_ORDERED` section forms a linked list of sections. This list is terminated by a final section with a `sh_link` that points at itself. All sections in this list are assigned to the output section with the name of the final section in the list.

If the `sh_info` entry of the ordered section is a valid section within the same input file, the ordered section is sorted based on the relative ordering within the output file of the section pointed to by the `sh_info` entry. The section pointed at by the `sh_info` entry must be unordered, and cannot in turn specify `SHF_LINK_ORDER` or `SHF_ORDERED`.

The special `sh_info` values `SHN_BEFORE` and `SHN_AFTER` (see Table 12-4) 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 are contiguous. These sections have the same relative ordering as the sections appear 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.

Two members in the section header, `sh_link` and `sh_info`, hold special information, depending on section type.

TABLE 12-9 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, otherwise `0`. See also Table 12-12 and “Relocation Sections” on page 351.

sh_type	sh_link	sh_info
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, 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_ancillary	The section header index of the associated string table.	0
SHT_SUNW_cap	If symbol capabilities exist, the section header index of the associated SHT_SUNW_capinfo table, otherwise 0.	If any capabilities refer to named strings, the section header index of the associated string table, otherwise 0.
SHT_SUNW_capinfo	The section header index of the associated symbol table.	For a dynamic object, the section header index of the associated SHT_SUNW_capchain table, otherwise 0.
SHT_SUNW_symsort	The section header index of the associated symbol table.	0
SHT_SUNW_tlssort	The section header index of the associated symbol table.	0
SHT_SUNW_LDYNSYM	The section header index of the associated string table. This index is the same string table used by the SHT_DYNSYM section.	One greater than the symbol table index of the last local symbol, STB_LOCAL. Since SHT_SUNW_LDYNSYM only contains local symbols, sh_info is equivalent to the number of symbols in the table.
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

Section Merging

The SHF_MERGE section flag can be used to mark SHT_PROGBITS sections within relocatable objects. See Table 12-8. This flag indicates that the section can be merged with compatible sections from other objects. Such merging has the potential to reduce the size of any executable or shared object that is built from these relocatable objects. This size reduction can also have a positive effect on the runtime performance of the resulting object.

A SHF_MERGE flagged section indicates that the section adheres to the following characteristics.

- The section is read-only. It must not be possible for a program containing this section to alter the section data at runtime.
- Every item in the section is accessed from an individual relocation record. The program code must not make any assumptions about the relative position of items in the section when generating the code that accesses the items.
- If the section also has the SHF_STRINGS flag set, then the section can only contain null terminated strings. Null characters are only allowed as string terminators, and null characters must not appear within the middle of any string.

SHF_MERGE is an optional flag indicating a possible optimization. The link-editor is allowed to perform the optimization, or to ignore the optimization. The link-editor creates a valid output object in either case. The link-editor presently implements section merging only for sections containing string data marked with the SHF_STRINGS flag.

When the SHF_STRINGS section flag is set in conjunction with the SHF_MERGE flag, the strings in the section are available to be merged with strings from other compatible sections. The link-editor merges such sections using the same string compression algorithm as used to compress the SHT_STRTAB string tables, .strtab and .dynstr.

- Duplicate strings are reduced to a single copy.
- Tail strings are eliminated. For example, if input sections contain the strings “*bigdog*” and “dog”, then the smaller “dog” string is eliminated, and the tail of the larger string is used to represent the smaller string.

The link-editor currently implements string merging only for strings that consist of byte sized characters that do not have special alignment constraints. Specifically, the following section characteristics are required.

- sh_entsize must be 0, or 1. Sections containing wide characters are not supported.
- Only sections with byte alignment, where sh_addralign is 0, or 1, are merged.

Note - Any string table compression can be suppressed with the link-editors -z nocompstrtab option.

Section Compression

The SHF_COMPRESSED section flag identifies a section containing compressed data. SHF_COMPRESSED applies only to non-allocable sections, and cannot be used in conjunction with SHF_ALLOC. In addition, SHF_COMPRESSED cannot be applied to sections of type SHT_NOBITS.

Any relocations that must be applied to a compressed section specify offsets to the uncompressed section data. It is therefore necessary to uncompress section data before relocations can be applied. Each compressed section specifies the algorithm independently. Different sections in a given ELF object can employ different compression algorithms.

Compressed sections start with a compression header structure that identifies the compression algorithm.

```
typedef struct {
        Elf32_Word      ch_type;
        Elf32_Word      ch_size;
        Elf32_Word      ch_addralign;
} Elf32_Chdr;

typedef struct {
        Elf64_Word      ch_type;
        Elf64_Word      ch_reserved;
        Elf64_Xword     ch_size;
        Elf64_Xword     ch_addralign;
} Elf64_Chdr;
```

ch_type

Specifies the compression algorithm. Supported algorithms and their descriptions are listed in Table 12-10.

ch_size

The size in bytes of the uncompressed data. See sh_size.

ch_addralign

Required alignment for the uncompressed data. See sh_addralign.

The sh_size and sh_addralign fields of the section header for a compressed section reflect the requirements of the compressed section. The ch_size and ch_addralign fields of the compression header provide the corresponding values for the uncompressed data, thereby supplying the values that sh_size and sh_addralign would have if the section had not been compressed.

The layout and interpretation of the data that follows the compression header is specific to each algorithm. This layout may contain algorithm specific parameters and alignment padding in addition to compressed data bytes.

A compression header's ch_type member specifies the compression algorithm employed, as shown in the following table.

TABLE 12-10 ELF Compression Types, `ch_type`

Name	Value
`ELFCOMPRESS_ZLIB`	`1`
`ELFCOMPRESS_LOOS`	`0x60000000`
`ELFCOMPRESS_HIOS`	`0x6fffffff`
`ELFCOMPRESS_LOPROC`	`0x70000000`
`ELFCOMPRESS_HIPROC`	`0x7fffffff`

`ELFCOMPRESS_ZLIB`

The section data is compressed with the ZLIB compression algorithm. The compressed ZLIB data bytes begin with the byte immediately following the compression header, and extend to the end of the section. Documentation for ZLIB may be found at http://www.zlib.net/.

`ELFCOMPRESS_LOOS` - `ELFCOMPRESS_HIOS`

Values in this inclusive range are reserved for operating system-specific semantics.

`ELFCOMPRESS_LOPROC` - `ELFCOMPRESS_HIPROC`

Values in this inclusive range are reserved for processor-specific semantics.

GNU-Style Section Compression

In addition to the compression format discussed previously, the Oracle Solaris link-editor understands an alternative format used by the GNU tool chain. This format does not employ a section flag to indicate compression. Instead, a section name that starts with the `.zdebug` prefix identifies a section containing compressed data. GNU-style compressed sections start with the following compression header structure.

```
typedef struct {
        uchar_t         gch_magic[4];
        uchar_t         gch_size[8];
} Chdr_GNU;
```

`gch_magic`

A 4-byte magic number identifying the compression algorithm. At this time, only ZLIB compression is supported. The values of `gch_magic` for ZLIB compression is as listed in Table 12-11.

`gch_size`

The size in bytes of the uncompressed data, encoded as a 64-bit `ELFDATA2MSB` big endian integer value.

TABLE 12-11 GNU ZLIB Compression, `gch_magic`

Name	Value
`gch_magic[0]`	`'Z'`
`gch_magic[1]`	`'L'`
`gch_magic[2]`	`'I'`
`gch_magic[3]`	`'B'`

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 12-12 ELF Special Sections

Name	Type	Attribute
`.bss`	`SHT_NOBITS`	`SHF_ALLOC + SHF_WRITE`
`.comment`	`SHT_PROGBITS`	*None*
`.data`, `.data1`	`SHT_PROGBITS`	`SHF_ALLOC + SHF_WRITE`
`.dynamic`	`SHT_DYNAMIC`	`SHF_ALLOC + SHF_WRITE`
`.dynstr`	`SHT_STRTAB`	`SHF_ALLOC`
`.dynsym`	`SHT_DYNSYM`	`SHF_ALLOC`
`.eh_frame_hdr`	`SHT_AMD64_UNWIND`	`SHF_ALLOC`
`.eh_frame`	`SHT_AMD64_UNWIND`	`SHF_ALLOC + SHF_WRITE`
`.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 415
`.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 397
`.note`	`SHT_NOTE`	*None*

Name	Type	Attribute
.lbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.ldata, .ldata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.lrodata, .lrodata1	SHT_PROGBITS	SHF_ALLOC + SHF_AMD64_LARGE
.plt	SHT_PROGBITS	See "Procedure Linkage Table (Processor-Specific)" on page 416
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	*None*
.rel*name*	SHT_REL	See "Relocation Sections" on page 351
.rela*name*	SHT_RELA	See "Relocation Sections" on page 351
.rodata, .rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	*None*
.strtab	SHT_STRTAB	Refer to the explanation following this table.
.symtab	SHT_SYMTAB	See "Symbol Table Section" on page 365
.symtab_shndx	SHT_SYMTAB_SHNDX	See "Symbol Table Section" on page 365
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata, .tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_ancillary	SHT_SUNW_ancillary	*None*
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_cap	SHT_SUNW_cap	SHF_ALLOC
.SUNW_capchain	SHT_SUNW_capchain	SHF_ALLOC
.SUNW_capinfo	SHT_SUNW_capinfo	SHF_ALLOC
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_ldynsym	SHT_SUNW_LDYNSYM	SHF_ALLOC
.SUNW_dynsymsort	SHT_SUNW_symsort	SHF_ALLOC
.SUNW_dymtlssort	SHT_SUNW_tlssort	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_REL	SHF_ALLOC

Name	Type	Attribute
	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 398 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 365 for details.

`.eh_frame_hdr`, `.eh_frame`

Call frame information used to unwind the stack.

`.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 110 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 110 for details.

`.got`

The global offset table. See “Global Offset Table (Processor-Specific)” on page 415 for details.

`.hash`

Symbol hash table. See “Hash Table Section” on page 345 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 110 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 110 for details.

`.interp`

The path name of a program interpreter. See “Program Interpreter” on page 397 for details.

`.lbss`

x64 specific uninitialized data. This data is similar to `.bss`, but provides for a section that is larger than 2 Gbytes.

`.ldata`, `.ldata1`

x64 specific initialized data. This data is similar to `.data`, but provides for a section that is larger than 2 Gbytes.

`.lrodata`, `.lrodata1`

x64 specific read-only data. This data is similar to `.rodata`, but provides for a section that is larger than 2 Gbytes.

`.note`

Information in the format described in “Note Section” on page 349.

`.plt`

The procedure linkage table. See “Procedure Linkage Table (Processor-Specific)” on page 416 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 110 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 376 for details.

`.rel`*name*, `.rela`*name*

Relocation information, as "Relocation Sections" on page 351 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 385 for details.

`.shstrtab`

Section names.

`.strtab`

Strings, most commonly the strings that represent the names that are 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 365 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 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 14, "Thread-Local Storage" 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 14, "Thread-Local Storage" for details.

`.text`

The *text* or executable instructions of a program.

`.SUNW_ancillary`

Ancillary group information. See “Ancillary Section” on page 338 for details.

`.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_cap`

Capability requirements. See “Capabilities Section” on page 342 for details.

`.SUNW_capchain`

Capability chain table. See “Capabilities Section” on page 342 for details.

`.SUNW_capinfo`

Capability symbol information. See “Capabilities Section” on page 342 for details.

`.SUNW_heap`

The *heap* of a dynamic executable created from `dldump`(3C).

`.SUNW_dynsymsort`

An array of indices to symbols in the combined `.SUNW_ldynsym` – `.dynsym` symbol table. The indices are sorted to reference symbols in order of increasing address. Symbols that do not represent variables or do not represent functions are not included. In the case of redundant global symbols and weak symbols, only the weak symbol is kept. See “Symbol Sort Sections” on page 374 for details.

`.SUNW_dyntlssort`

An array of indices to thread-local storage symbols in the combined `.SUNW_ldynsym` – `.dynsym` symbol table. The indices are sorted to reference symbols in order of increasing offset. Symbols that do not represent TLS variables are not included. In the case of redundant global symbols and weak symbols, only the weak symbol is kept. See “Symbol Sort Sections” on page 374 for details.

`.SUNW_ldynsym`

Augments the `.dynsym` section. This section contains local function symbols, for use in contexts where the full `.symtab` section is not available. The link-editor always places the data for a `.SUNW_ldynsym` section immediately before, and adjacent to, the `.dynsym` section. Both sections always use the same `.dynstr` string table section. This placement and organization, allows both symbol tables to be treated as a single larger symbol table. See “Symbol Table Section” on page 365.

`.SUNW_move`

Additional information for partially initialized data. See “Move Section” on page 347 for details.

`.SUNW_reloc`

Relocation information, as “Relocation Sections” on page 351 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 is meaningful, thus the section `sh_info` value is zero.

`.SUNW_syminfo`

Additional symbol table information. See “*Syminfo* Table Section” on page 377 for details.

`.SUNW_version`

Versioning information. See “Versioning Sections” on page 379 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 that are not reserved. An object file can have more than one section with the same name.

Section names that are 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 that are used for `e_machine`. For example, `.Foo.psect` is the `psect` section defined by the `FOO` architecture.

Existing extensions use their historical names.

Ancillary Section

In addition to the primary output object, the Solaris link-editor can produce one or more ancillary objects. Ancillary objects contain non-allocable sections that are normally written to the primary object. When ancillary objects are produced, the primary object and all of the associated ancillary objects contain a `SHT_SUNW_ancillary` section, containing information that identifies these related objects. The ancillary section from any of these objects provides the information needed to identify and interpret the other members of the group.

This section contains an array of the following structures. See `sys/elf.h`.

```
typedef struct {
        Elf32_Word      a_tag;
        union {
```

```
                Elf32_Word      a_val;
                Elf32_Addr      a_ptr;
        } a_un;
} Elf32_Ancillary;

typedef struct {
        Elf64_Xword     a_tag;
        union {
                Elf64_Xword     a_val;
                Elf64_Addr      a_ptr;
        } a_un;
} Elf64_Ancillary;
```

For each object with this type, `a_tag` controls the interpretation of `a_un`.

`a_val`

These objects represent integer values with various interpretations.

`a_ptr`

These objects represent program virtual addresses.

The following ancillary tags exist.

TABLE 12-13 ELF Ancillary Array Tags

Name	Value	`c_un`
`ANC_SUNW_NULL`	`0`	Ignored
`ANC_SUNW_CHECKSUM`	`1`	`a_val`
`ANC_SUNW_MEMBER`	`2`	`a_ptr`

`ANC_SUNW_NULL`

Marks the end of a group of the ancillary section.

`ANC_SUNW_CHECKSUM`

Provides the checksum for a file in the `c_val` element. When `ANC_SUNW_CHECKSUM` precedes the first instance of `ANC_SUNW_MEMBER`, it provides the checksum for the object from which the ancillary section is being read. When it follows an `ANC_SUNW_MEMBER` tag, it provides the checksum for that member.

`ANC_SUNW_MEMBER`

Specifies an object name. The `a_ptr` element contains the string table offset of a null-terminated string, that provides the file name.

An ancillary section must always contain an `ANC_SUNW_CHECKSUM` before the first instance of `ANC_SUNW_MEMBER`, identifying the current object. Following that, there should be

an ANC_SUNW_MEMBER for each object that makes up the complete set of objects. Each ANC_SUNW_MEMBER should be followed by an ANC_SUNW_CHECKSUM for that object. A typical ancillary section is therefore be structured as follows.

Tag	Meaning
ANC_SUNW_CHECKSUM	Checksum of this object
ANC_SUNW_MEMBER	Name of object #1
ANC_SUNW_CHECKSUM	Checksum for object #1
…	
ANC_SUNW_MEMBER	Name of object N
ANC_SUNW_CHECKSUM	Checksum for object N
ANC_SUNW_NULL	

An object can therefore identify itself by comparing the initial ANC_SUNW_CHECKSUM to each of the ones that follow, until it finds a match.

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 that are 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 that is used for section reordering when the compiler is invoked with the -xF option. If a function is placed in a SHT_SUNW_COMDAT section that is named *.sectname%funcname*, the final SHT_SUNW_COMDAT sections that are retained are coalesced into the section that is named *.sectname*. This method can be used to place SHT_SUNW_COMDAT sections 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.

There can be internal references among group sections. However, 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 the value zero. 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 12-14 ELF Group Section Flag

Name	Value
GRP_COMDAT	0x1

GRP_COMDAT

GRP_COMDAT is a COMDAT group. This group 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. A definition of the same symbol in the object containing the reference must have a separate symbol table entry from the reference. Sections outside of the group can not reference symbols with STB_LOCAL binding for addresses that are 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.

Capabilities Section

A `SHT_SUNW_cap` section identifies the capability requirements of an object. These capabilities are referred to as *object* capabilities. This section can also identify the capability requirements of functions, or initialized data items, within an object. These capabilities are referred to as *symbol* capabilities. This section contains an array of the following structures. See `sys/elf.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 12-15 ELF Capability Array Tags

Name	Value	c_un
CA_SUNW_NULL	0	Ignored
CA_SUNW_HW_1	1	c_val

Name	Value	c_un
CA_SUNW_SF_1	2	c_val
CA_SUNW_HW_2	3	c_val
CA_SUNW_PLAT	4	c_ptr
CA_SUNW_MACH	5	c_ptr
CA_SUNW_ID	6	c_ptr

CA_SUNW_NULL

Marks the end of a group of capabilities.

CA_SUNW_HW_1, CA_SUNW_HW_2

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.

CA_SUNW_PLAT

Specifies a platform name. The c_ptr element contains the string table offset of a null-terminated string, that defines a platform name.

CA_SUNW_MACH

Specifies a machine name. The c_ptr element contains the string table offset of a null-terminated string, that defines a machine hardware name.

CA_SUNW_ID

Specifies a capability identifier name. The c_ptr element contains the string table offset of a null-terminated string, that defines an identifier name. This element does not define a capability, but assigns a unique symbolic name to the capability group by which the group can be referenced. This identifier name is appended to any global symbol names that are transformed to local symbols as part of the link-editors -z symbolcap processing. See “Converting Object Capabilities to Symbol Capabilities” on page 71.

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 Capability Requirements” on page 56.

Multiple CA_SUNW_NULL terminated groups of capabilities can exist within an object. The first group, starting at index 0, identifies the object capabilities. A dynamic object that defines object capabilities, has a PT_SUNWCAP program header associated to the section. This program header allows the runtime linker to validate the object against the system capabilities that are available to the process. Dynamic objects that use different object capabilities can provide a flexible runtime environment using filters. See “Capability Specific Shared Objects” on page 253.

Additional groups of capabilities identify symbol capabilities. Symbol capabilities allow multiple instances of the same symbol to exist within an object. Each instance is associated to a set of capabilities that must be available for the instance to be used. When symbol capabilities are present, the sh_link element of the SHT_SUNW_cap section points to the associated SHT_SUNW_capinfo table. Dynamic objects that use symbol capabilities can provide a flexible means of enabling optimized functions for specific systems. See “Creating a Family of Symbol Capabilities Functions” on page 65.

The SHT_SUNW_capinfo table parallels the associated symbol table. The sh_link element of the SHT_SUNW_capinfo section points to the associated symbol table. Functions that are associated with capabilities, have indexes within the SHT_SUNW_capinfo table that identify the capabilities group within the SHT_SUNW_cap section.

Within a dynamic object, the sh_info element of the SHT_SUNW_capinfo section points to a capabilities chain table, SHT_SUNW_capchain. This table is used by the runtime linker to locate members of a capabilities family.

A SHT_SUNW_capinfo table entry has the following format. See sys/elf.h.

```
typedef Elf32_Word    Elf32_Capinfo;
typedef Elf64_Xword   Elf64_Capinfo;
```

Elements within this table are interpreted using the following macros. See sys/elf.h.

```
#define ELF32_C_SYM(info)       ((info)>>8)
#define ELF32_C_GROUP(info)     ((unsigned char)(info))
#define ELF32_C_INFO(sym, grp)  (((sym)<<8)+(unsigned char)(grp))

#define ELF64_C_SYM(info)       ((info)>>32)
#define ELF64_C_GROUP(info)     ((Elf64_Word)(info))
#define ELF64_C_INFO(sym, grp)  (((Elf64_Xword)(sym)<<32)+(Elf64_Xword)(grp))
```

A SHT_SUNW_capinfo entry group element contains the index of the SHT_SUNW_cap table that this symbol is associated with. This element thus associates symbols to a capability group. A reserved group index, CAPINFO_SUNW_GLOB, identifies a lead symbol of a family of capabilities instances, that provides a default instance.

Name	Value	Meaning
CAPINFO_SUNW_GLOB	0xff	Identifies a default symbol. This symbol is not associated with any specific capabilities, but leads a symbol capabilities family.

A SHT_SUNW_capinfo entry symbol element contains the index of the lead symbol associated with this symbol. The group and symbol information allow the link-editor to process families of capabilities symbols from relocatable objects, and construct the necessary capabilities information in any output object. Within a dynamic object, the symbol element of a lead symbol, one tagged with the group CAPINFO_SUNW_GLOB, is an index into the SHT_SUNW_capchain table. This index allows the runtime linker to traverse the capabilities chain table, starting at this index, and inspects each following entry until a 0 entry is found. The chain entries contain symbol indices for each capabilities family member.

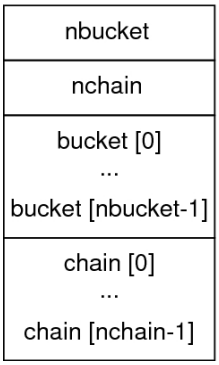
A dynamic object that defines symbol capabilities, has a DT_SUNW_CAP dynamic entry, and a DT_SUNW_CAPINFO dynamic entry. These entries identify the SHT_SUNW_cap section, and SHT_SUNW_capinfo section respectively. The object also contains DT_SUNW_CAPCHAIN, DT_SUNW_CAPCHAINENT and DT_SUNW_CAPCHAINSZ entries that identify the SHT_SUNW_capchain section, the sections entry size and total size. These entries allow the runtime linker to establish the best symbol to use, from a family of symbol capability instances.

An object can define only object capabilities, or can define only symbol capabilities, or can define both types of capabilities. An object capabilities group starts at index 0. Symbol capabilities groups start at any index other than 0. If an object defines symbol capabilities, but no object capabilities, then a single CA_SUNW_NULL entry must exist at index 0 to indicate the start of symbol capabilities.

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 are used in the following figure to help explain the hash table organization, but these labels are not part of the specification.

FIGURE 12-4 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 index 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 int 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`). These items occupy no space in the object file, but contribute 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 zero indicates that the initialization be performed contiguously for 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'}
};
```

A `SHT_SUNW_move` section can be used to describe this data. The data item is defined within the `.bss` section. The non-zero elements of the data item are initialized with the appropriate move entries.

```
$ elfdump -s data | fgrep move
    [17]    0x20868   0x8000  OBJT GLOB 0   .bss       move
$ elfdump -m data

Move Section:  .SUNW_move
  symndx     offset   size repeat stride            value  with respect to
    [17]       0x44      4      1      1       0x45000000  move
    [17]       0x40      4      1      1              0xe  move
    [17]       0x34      4      1      1       0x45000000  move
    [17]       0x30      4      1      1              0xe  move
    [17]       0x1c      4      2      1       0x46000000  move
    [17]       0x18      4      2      1              0xf  move
    [17]        0xc      4      1      1       0x31000000  move
    [17]        0x8      4      1      1              0x1  move
```

Move sections that are 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. This processing expands the move entry 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

A vendor or system engineer might need to mark an object file with special information that other programs can check for conformance or compatibility. 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–bit objects 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 12-6 to help explain note information organization, but are not part of the specification.

FIGURE 12-5 Note Information

`namesz` and `name`

The first `namesz` bytes in name contain a null-terminated character representation of the entry's owner or originator. No formal mechanism exists 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 the value zero. Padding is present, if

necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in `namesz`.

`descsz` and `desc`

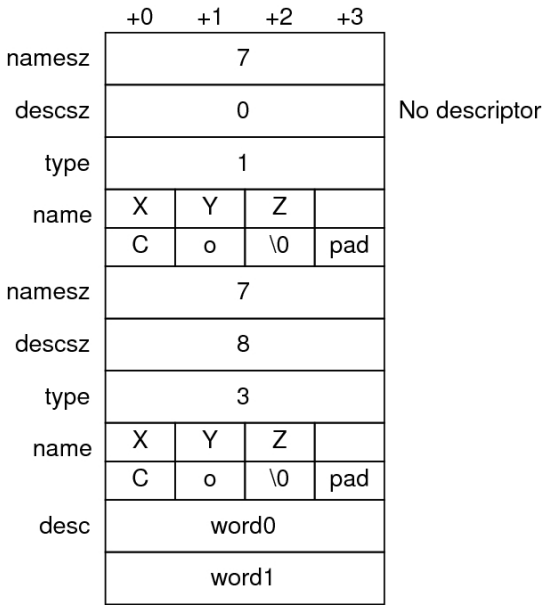
The first `descsz` bytes in `desc` hold the note descriptor. If no descriptor is present, `descsz` contains the value zero. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in `descsz`.

`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 that is shown in the following figure holds two entries.

FIGURE 12-6 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. This information allows 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. See `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;
```

`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 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 zero 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 64–bit SPARC `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. For the existing relocation types, the data field is zero. New relocation types, however, might make use of the data bits.

```
#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. In all cases, the addend and the computed result use the same byte order. The relocation entry type and interpretation of the addend value are defined by the platform specific ABI.

SPARC — 32–bit SPARC uses `Elf32_Rela` relocation entries. 64–bit SPARC uses `Elf64_Rela` relocation entries. The prior value of the field to be relocated, added to the `r_addend` member, serves as the relocation addend.

32–bit x86 — 32–bit x86 uses `Elf32_Rel` relocation entries. The field to be relocated holds the addend.

64–bit x86 — 64–bit x86 uses `Elf64_Rela` relocation entries. The `r_addend` member serves as the relocation addend. The prior value of the field to be relocated is ignored.

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 311 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.

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.

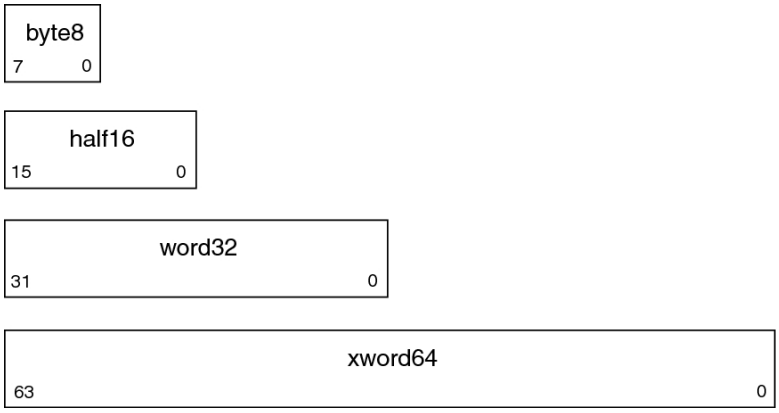
Relocation Calculations

The following notation is used to describe relocation computations.

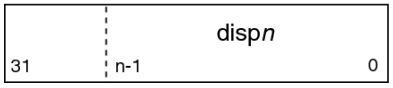
`A`	The addend used to compute the value of the relocatable field.
`B`	The base address at which a shared object is loaded into memory during execution. Generally, a shared object file is built with a base virtual address of `0`. However, the execution address of the shared object is different. See "Program Header" on page 385.
`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 415.
`GOT`	The address of the global offset table. See "Global Offset Table (Processor-Specific)" on page 415.
`L`	The section offset or address of the procedure linkage table entry for a symbol. See "Procedure Linkage Table (Processor-Specific)" on page 416.
`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.
`Z`	The size of the symbol whose index resides in the relocation entry.

SPARC: Relocations

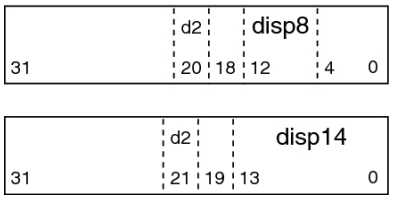
On the SPARC platform, relocation entries apply to bytes (`byte8`), half-words (`half16`), words (`word32`), and extended-words (`xword64`).



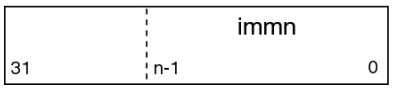
The `disp`*n* family of relocation fields (`disp19`, `disp22`, `disp30`) are word-aligned, sign-extended, PC-relative displacements. All encode a value with its least significant bit in position 0 of the word, and differ only in the number of bits allocated to the value.



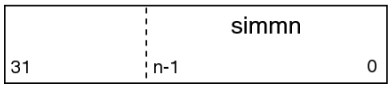
The `d2/disp8` and `d2/disp14` variants encode 16 and 10-bit displacement values using two non-contiguous bit fields, `d2`, and `disp`*n*.



The `imm`*n* `family` of relocation fields (`imm5`, `imm6`, `imm7`, `imm10`, `imm13`, `imm22`) represent unsigned integer constants. All encode a value with its least significant bit in position 0 of the word, and differ only in the number of bits allocated to the value.



The simm*n* family of relocation fields (simm10, simm11, simm13, simm22) represent signed integer constants. All encode a value with its least significant bit in position 0 of the word, and differ only in the number of bits allocated to the value.



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 12-16 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

Name	Value	Field	Calculation
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*	Refer to the explanation following this table.
R_SPARC_GLOB_DAT	20	V-word32	S + A
R_SPARC_JMP_SLOT	21	*None*	Refer to the explanation following this table.
R_SPARC_RELATIVE	22	V-word32	B + A
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

Name	Value	Field	Calculation
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
R_SPARC_GOTDATA_HIX22	80	V-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 31)
R_SPARC_GOTDATA_LOX10	81	T-imm13	((S + A - GOT) & 0x3ff) | (((S + A - GOT) >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP_HIX22	82	T-imm22	(G >> 10) ^ (G >> 31)
R_SPARC_GOTDATA_OP_LOX10	83	T-imm13	(G & 0x3ff) | ((G >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP	84	Word32	Refer to the explanation following this table.
R_SPARC_SIZE32	86	V-word32	Z + A
R_SPARC_WDISP10	88	V-d2/disp8	(S + A - P) >> 2

Note - Additional relocations are available for thread-local storage references. These relocations are covered in Chapter 14, “Thread-Local Storage”.

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 `GOT` 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 `GOT` 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 `GOT` 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 188.

R_SPARC_GLOB_DAT

Resembles R_SPARC_32, except that the relocation sets a GOT entry to the address of the specified symbol. The special relocation type enables you to determine the correspondence between symbols and GOT 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 a value of zero 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_LOX10, 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 that are listed in the following table extend, or alter, the relocations defined for 32–bit SPARC. See “Relocation Types” on page 355.

TABLE 12-17 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

Name	Value	Field	Calculation
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
R_SPARC_SIZE64	87	V-xword64	Z + A

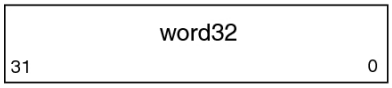
The following relocation type has semantics beyond simple calculation.

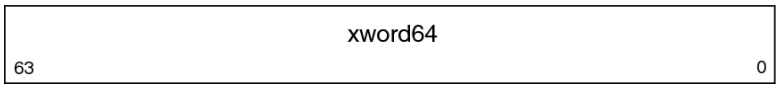
R_SPARC_OLO10

Resembles R_SPARC_LO10, except that an extra offset is added to make full use of the 13-bit signed immediate field.

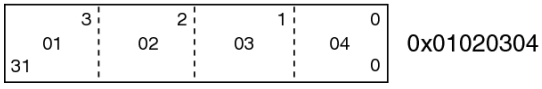
x86: Relocations

On x86, relocation entries apply to words (word32), and extended-words (xword64).





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.



32-bit x86: Relocation Types

The relocations that are listed in the following table are defined for 32–bit x86.

TABLE 12-18 32-bit 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*	Refer to the explanation following this table.
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
R_386_16	20	word16	S + A
R_386_PC16	21	word16	S + A - P
R_386_8	22	word8	S + A
R_386_PC8	23	word8	S + A - P
R_386_SIZE32	38	word32	Z + A

Note - Additional relocations are available for thread-local storage references. These relocations are covered in Chapter 14, “Thread-Local Storage”.

Some relocation types have semantics beyond simple calculation.

R_386_GOT32

Computes the distance from the base of the GOT to the symbol's GOT 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 188.

R_386_GLOB_DAT

Used to set a GOT entry to the address of the specified symbol. The special relocation type enable you to determine the correspondence between symbols and GOT 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 a value of zero for the symbol table index.

R_386_GOTOFF

Computes the difference between a symbol's value and the address of the GOT. 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 GOT 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.

x64: Relocation Types

The relocations that are listed in the following table are defined for x64.

TABLE 12-19 x64: ELF Relocation Types

Name	Value	Field	Calculation
R_AMD64_NONE	0	*None*	*None*
R_AMD64_64	1	word64	S + A

Name	Value	Field	Calculation
R_AMD64_PC32	2	word32	S + A - P
R_AMD64_GOT32	3	word32	G + A
R_AMD64_PLT32	4	word32	L + A - P
R_AMD64_COPY	5	*None*	Refer to the explanation following this table.
R_AMD64_GLOB_DAT	6	word64	S
R_AMD64_JUMP_SLOT	7	word64	S
R_AMD64_RELATIVE	8	word64	B + A
R_AMD64_GOTPCREL	9	word32	G + GOT + A - P
R_AMD64_32	10	word32	S + A
R_AMD64_32S	11	word32	S + A
R_AMD64_16	12	word16	S + A
R_AMD64_PC16	13	word16	S + A - P
R_AMD64_8	14	word8	S + A
R_AMD64_PC8	15	word8	S + A - P
R_AMD64_PC64	24	word64	S + A - P
R_AMD64_GOTOFF64	25	word64	S + A - GOT
R_AMD64_GOTPC32	26	word32	GOT + A + P
R_AMD64_SIZE32	32	word32	Z + A
R_AMD64_SIZE64	33	word64	Z + A

Note - Additional relocations are available for thread-local storage references. These relocations are covered in Chapter 14, “Thread-Local Storage”.

The special semantics for most of these relocation types are identical to those used for x86. Some relocation types have semantics beyond simple calculation.

`R_AMD64_GOTPCREL`

This relocations has different semantics from the `R_AMD64_GOT32` or equivalent `R_386_GOTPC` relocation. The x64 architecture provides an addressing mode that is relative to the instruction pointer. Therefore, an address can be loaded from the `GOT` using a single instruction.

The calculation for the R_AMD64_GOTPCREL relocation provides the difference between the location in the GOT where the symbol's address is given, and the location where the relocation is applied.

R_AMD64_32

The computed value is truncated to 32–bits. The link-editor verifies that the generated value for the relocation zero-extends to the original 64–bit value.

R_AMD64_32S

The computed value is truncated to 32–bits. The link-editor verifies that the generated value for the relocation sign-extends to the original 64–bit value.

R_AMD64_8, R_AMD64_16, R_AMD64_PC16, and R_AMD64_PC8

These relocations are not conformant to the x64 ABI, but are added here for documentation purposes. The R_AMD64_8 relocation truncates the computed value to 8-bits. The R_AMD64_16 relocation truncates the computed value to 16-bits.

String Table Section

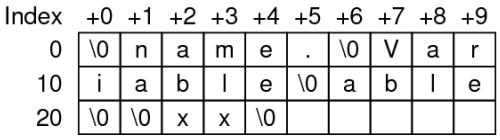
String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. You reference a string as an index into the string table section.

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. The section header string table is 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.

FIGURE 12-7 ELF String Table



The following table shows the strings of the string table that are shown in the preceding figure.

TABLE 12-20 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 symbolic 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 12-24.

A symbol table entry has the following format. See `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;
```

`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, the value 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. The value can be an absolute value or an address, depending on the context. See "Symbol Values" on page 372.

`st_size`

Many symbols have associated sizes. For example, a data object's size is the number of bytes that are contained in the object. This member holds the value zero 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 12-21. The following code shows how to manipulate the values. See `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 12-23. The following code shows how to manipulate the values for both 32–bit objects and 64–bit objects. Other bits are set to zero, 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 12-4.

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 12-21 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, multiple definitions of STB_GLOBAL symbols with the same name are not allowed. However, 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 definitions.

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 definition. A common symbol has the `st_shndx` field holding `SHN_COMMON`. See "Symbol Resolution" on page 38.

- When the link-editor searches archive libraries, archive members that contain definitions of undefined or tentative global symbols are extracted. 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. This options 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 symbols and global symbols. As "Sections" on page 311 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, as determined from its `st_info` field, provides a general classification for the associated entity.

TABLE 12-22 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`

Name	Value
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 14, “Thread-Local Storage” 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 12-23 ELF Symbol Visibility

Name	Value
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3
STV_EXPORTED	4
STV_SINGLETON	5
STV_ELIMINATE	6

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 interpreted the same as `STV_HIDDEN`.

`STV_EXPORTED`

This visibility attribute ensures that a symbol remains global. This visibility can not be demoted, or eliminated by any other symbol visibility technique. A symbol with `STB_LOCAL` binding will not have `STV_EXPORTED` visibility.

`STV_SINGLETON`

This visibility attribute ensures that a symbol remains global, and that a single instance of the symbol definition is bound to by all references within a process. This visibility can not be demoted, or eliminated by any other symbol visibility technique. A symbol with `STB_LOCAL` binding will not have `STV_SINGLETON` visibility. A `STV_SINGLETON` can not be directly bound to.

`STV_ELIMINATE`

This visibility attribute extends `STV_HIDDEN`. A symbol that is defined in the current component as eliminate is not visible to other components. The symbol is not written to any symbol table of a dynamic executable or shared object from which the component is used.

The `STV_SINGLETON` visibility attribute can affect the resolution of symbols within an executable or shared object during link-editing. Only one instance of a singleton can be bound to from any reference within a process.

A `STV_SINGLETON` can be combined with a `STV_DEFAULT` visibility attribute, with the `STV_SINGLETON` taking precedence. A `STV_EXPORT` can be combined with a `STV_DEFAULT` visibility attribute, with the `STV_EXPORT` taking precedence. A `STV_SINGLETON` or `STV_EXPORT` visibility can not be combined with any other visibility attribute. Such an event is deemed fatal to the link-edit.

Other visibility attributes do not affect 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.

- 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 object being linked. If this type of symbol reference has no definition within the object being linked, then the reference must have `STB_WEAK` binding. In this case, the reference is resolved to zero.
- If any reference to a name, or definition of a name is a symbol with a non-default visibility attribute, the visibility attribute is propagated to the resolving symbol in the object being linked. If different visibility attributes are specified for distinct instances of a symbol, the most constraining visibility attribute is propagated to the resolving symbol in the object

being linked. 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, the symbols's 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, and SHN_AMD64_LCOMMON

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 indicates that the symbol is undefined. When the link-editor combines this object file with another object that defines the indicated symbol, this file's references to the symbol is bound to the definition.

As mentioned previously, the symbol table entry for index 0 (STN_UNDEF) is reserved. This entry holds the values listed in the following table.

TABLE 12-24 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.

Symbol Table Layout and Conventions

The symbols in a symbol table are written in the following order.

- Index `0` in any symbol table is used to represent undefined symbols. This first entry in a symbol table is always completely zeroed. The symbol type is therefore `STT_NOTYPE`.
- If the symbol table contains any local symbols, the second entry of the symbol table is an `STT_FILE` symbol giving the name of the file.
- Section symbols of type `STT_SECTION`.
- Register symbols of type `STT_REGISTER`.
- Global symbols that have been reduced to local scope.
- For each input file that supplies local symbols, a `STT_FILE` symbol giving the name of the input file, followed by the symbols in question.
- The global symbols immediately follow the local symbols in the symbol table. The first global symbol is identified by the symbol table `sh_info` value. Local and global symbols are always kept separate in this manner, and cannot be mixed together.

Three symbol tables are of special interest in the Oracle Solaris OS.

`.symtab` (`SHT_SYMTAB`)

This symbol table contains every symbol that describes the associated ELF file. This symbol table is typically non-allocable, and is therefore not available in the memory image of the process.

Global symbols can be eliminated from the `.symtab` by using a `mapfile` together with the `ELIMINATE` keyword. See “Symbol Elimination” on page 53, and “SYMBOL_SCOPE / SYMBOL_VERSION Directives” on page 217.

`.dynsym` (`SHT_DYNSYM`)

This table contains a subset of the symbols from the `.symtab` table that are needed to support dynamic linking. This symbol table is allocable, and is therefore available in the memory image of the process.

The `.dynsym` table begins with the standard NULL symbol, followed by the files global symbols. `STT_FILE` symbols are typically not present in this symbol table. `STT_SECTION` symbols might be present if required by relocation entries.

`.SUNW_ldynsym` (`SHT_SUNW_LDYNSYM`)

An optional symbol table that augments the information that is found in the `.dynsym` table. The `.SUNW_ldynsym` table contains local function symbols. This symbol table is allocable, and is therefore available in the memory image of the process. This section allows debuggers to produce accurate stack traces in runtime contexts when the non-allocable `.symtab` is not available, or has been stripped from the file. This section also provides the runtime environment with additional symbolic information for use with `dladdr`(3C).

A `.SUNW_ldynsym` table only exists when a `.dynsym` table is present. When both a `.SUNW_ldynsym` section and a `.dynsym` section exist, the link-editor places their data regions directly adjacent to each other, with the `.SUNW_ldynsym` first. This placement allows the two tables to be viewed as a single larger contiguous symbol table. This symbol table follows the standard layout rules that were enumerated previously.

The `.SUNW_ldynsym` table can be eliminated by using the link-editor `-z noldynsym` option.

Symbol Sort Sections

The dynamic symbol table formed by the adjacent `.SUNW_ldynsym` section and `.dynsym` section can be used to map memory addresses to their corresponding symbol. This mapping can be used to determine which function or variable that a given address represents. However, analyzing the symbol tables to determine a mapping is complicated by the order in which symbols are written to symbol tables. See “Symbol Table Layout and Conventions” on page 373. This layout complicates associating an address to a symbol name in the follows ways.

- Symbols are not sorted by address, which forces an expensive linear search of the entire table.
- More than one symbol can refer to a given address. Although these symbols are valid and correct, the choice of which of these equivalent names to use by a debugging tool might not be obvious. Different tools might use different alternative names. These issues are likely to lead to user confusion.
- Many symbols provide non-address information. These symbols should not be considered as part of such a search.

Symbol sort sections are used to solve these problems. A symbol sort section is an array of `Elf32_Word` or `Elf64_Word` objects. Each element of this array is an index into the combined `.SUNW_ldynsym` – `.dynsym` symbol table. The elements of the array are sorted so that the symbols that are reference are provided in sorted order. Only symbols representing functions or variables are included. The symbols that are associated with a sort array can be displayed using `elfdump`(1) with the `-S` option.

Regular symbols and thread-local storage symbols can not be sorted together. The value of a regular symbol is the address of the function or the address of the variable the symbol references. The value of a thread-local storage symbol is the variable's thread offset. Therefore, regular symbols and thread-local storage symbols use two different sort sections.

`.SUNW_dynsymsort`

A section of type `SHT_SUNW_SYMSORT`, containing indexes to regular symbols in the combined `.SUNW_ldynsym` – `.dynsym` symbol table, sorted by address. Symbols that do not represent variables or functions are not included.

`.SUNW_dyntlssort`

A section of type `SHT_SUNW_TLSSORT`, containing indexes to TLS symbols in the combined `.SUNW_ldynsym` – `.dynsym` symbol table, sorted by offset. This section is only produced if the object file contains TLS symbols.

The link-editor uses the following rules, in the order that is shown, to select which symbols are referenced by the sort sections.

- The symbol must have a function or variable type: `STT_FUNC`, `STT_OBJECT`, `STT_COMMON`, or `STT_TLS`.
- The following symbols are always included, if present: `_DYNAMIC`, `_end`, `_fini`, `_GLOBAL_OFFSET_TABLE_`, `_init`, `_PROCEDURE_LINKAGE_TABLE_`, and `_start`.
- If a global symbol and a weak symbol are found to reference the same item, the weak symbol is included and the global symbol is excluded.
- The symbol must not be undefined.
- The symbol must have a non-zero size.

These rules filter out automatically generated compiler and link-editor generated symbols. The symbols that are selected are of interest to the user. However, two cases exist where manual intervention might be necessary to improve the selection process.

- The rules did not select a needed special symbol. For example, some special symbols have a zero size.
- Unwanted extra symbols are selected. For example, shared objects can define multiple symbols that reference the same address and have the same size. These alias symbols effectively reference the same item. You might prefer to include only one of a multiple symbol family, within the sort section.

The mapfile keywords DYNSORT and NODYNSORT provide for additional control over symbol selection. See “SYMBOL_SCOPE / SYMBOL_VERSION Directives” on page 217.

DYNSORT

Identifies a symbol that should be included in a sort section. The symbol type must be STT_FUNC, STT_OBJECT, STT_COMMON, or STT_TLS.

NODYNSORT

Identifies a symbol that should not be included in a sort section.

For example, an object might provide the following symbol table definitions.

```
$ elfdump -sN.symtab foo.so.1 | egrep "foo$|bar$"
    [37]    0x4b0   0x1c  FUNC GLOB  D   0 .text      bar
    [38]    0x4b0   0x1c  FUNC WEAK  D   0 .text      foo
```

The symbols foo and bar represent an aliases pair. By default, when creating a sorted array, only the symbol foo is represented.

```
$ cc -o foo.so.1 -G foo.c
$ elfdump -S foo.so.1 | egrep "foo$|bar$"
    [13]    0x4b0   0x1c  FUNC WEAK  D   0 .text      foo
```

In the case where a global and a weak symbol are found by the link-editor to reference the same item, the weak symbol is normally kept. The symbol bar is omitted from the sorted array because of the association to the weak symbol foo.

The following mapfile results in the symbol bar being represented in the sorted array. The symbol foo is omitted.

```
$ cat mapfile
{
        global:
                bar = DYNSORT;
                foo = NODYNSORT;
};
$ cc -M mapfile -o foo.so.2 -Kpic -G foo.c
$ elfdump -S foo.so.2 | egrep "foo$|bar$"
    [13]    0x4b0   0x1c  FUNC GLOB  D   0 .text      bar
```

The .SUNW_dynsymsort section and .SUNW_dyntlssort section, require that a .SUNW_ldynsym section be present. Therefore, use of the -z noldynsym option also prevents the creation of any sort section.

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 that are listed in the following table.

TABLE 12-25 SPARC: ELF Symbol Table Entry: Register Symbol

Field	Meaning
`st_name`	Index into the string table for the name of the symbol, or the value `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 that are defined for SPARC are listed in the following table.

TABLE 12-26 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`. The `.SUNW_syminfo` section contains one entry 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` entry 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 usage does not pose any conflicts.

An `Syminfo` entry has the following format. See `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;
```

`si_boundto`

An index to an entry in the `.dynamic` section, identified by the `sh_info` field, which 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`.

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.
`SYMINFO_BT_EXTERN`	0`xfffc`	Symbol definition is external.

`si_flags`

This bit-field can have flags set, as shown in the following table.

Name	Value	Meaning
`SYMINFO_FLG_DIRECT`	`0x01`	Symbol reference has a direct association to the object containing the definition.
`SYMINFO_FLG_FILTER`	`0x02`	Symbol definition acts as a standard filter.
`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 an external reference to directly bind to this symbol definition.
`SYMINFO_FLG_AUXILIARY`	`0x40`	Symbol definition acts as an auxiliary filter.

Name	Value	Meaning
SYMINFO_FLG_INTERPOSE	0x80	Symbol definition acts as an interposer. This attribute is only applicable for dynamic executables.
SYMINFO_FLG_CAP	0x100	Symbol is associated with capabilities.
SYMINFO_FLG_DEFERRED	0x200	Symbol should not be included in BIND_NOW relocations.

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_verneedSHT_SUNW_versym.

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. These two structures provide an association of symbols to version definitions within the file. See "Creating a Version Definition" on page 235. 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;
```

`vd_version`

This member identifies the version of the structure, as listed in the following table.

Name	Value	Meaning
`VER_DEF_NONE`	`0`	Invalid version.
`VER_DEF_CURRENT`	`>=1`	Current version.

The value 1 signifies the original section format. Extensions require 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.

Name	Value	Meaning
`VER_FLG_BASE`	`0x1`	Version definition of the file.
`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 the version. See “Creating a Weak Version Definition” on page 238.

`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 that is described in "Hash Table Section" on page 345.

`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. This element 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 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;
```

`vn_version`

This member identifies the version of the structure, as listed in the following table.

Name	Value	Meaning
`VER_NEED_NONE`	`0`	Invalid version.
`VER_NEED_CURRENT`	`>=1`	Current version.

The value 1 signifies the original section format. Extensions require 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, providing the file name of a version dependency. This name matches one of the `.dynamic` dependencies found in the file. See "Dynamic Section" on page 398.

`vn_aux`

The byte offset, from the start of this `Elf32_Verneed` entry, to the `Elf32_Vernaux` array of version definitions that are required from the associated file dependency. At least one version dependency must exist. 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 that is described in "Hash Table Section" on page 345.

`vna_flags`

Version dependency specific information, as listed in the following table.

Name	Value	Meaning
`VER_FLG_WEAK`	`0x2`	Weak version identifier.
`VER_FLG_INFO`	`0x4`	`SHT_SUNW_versym` reference exists for informational purposes, and need not be validated at runtime.

A weak version dependency indicates an original binding to a weak version definition.

`vna_other`

If non-zero, the version index assigned to this dependency version. This index is used within the `SHT_SUNW_versym` to assign global symbol references to this version.

Versions of Solaris up to and including the Oracle Solaris 10 release, did not assign version symbol indexes to dependency versions. In these objects, the value of `vna_other` is `0`.

`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.

Version Symbol Section

The version symbol section is defined by the type `SHT_SUNW_versym`. This section consists of an array of elements of 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 that are 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 12-27 ELF Version Dependency Indexes

Name	Value	Meaning
VER_NDX_LOCAL	0	Symbol has local scope.
VER_NDX_GLOBAL	1	Symbol has global scope and is assigned to the base version definition.
	>1	Symbol has global scope and is assigned to a user-defined version definition, SHT_SUNW_verdef, or a version dependency, SHT_SUNW_verneed.

A symbol may be assigned the special reserved index 0. This index can be assigned for any of the following reasons.

- A non-global symbol is always assigned VER_NDX_LOCAL. However, this is rare in practice. Versioning sections are usually created only in conjunction with the dynamic symbol table, .dynsym, which only contains global symbols.
- A global symbol defined within an object that does not have a SHT_SUNW_verdef version definition section.
- An undefined global symbol defined within an object that does not have a SHT_SUNW_verneed version dependency section. Or, an undefined global symbol defined within an object in which the version dependency section does not assign version indexes.
- The first entry of a symbol table is always NULL. This entry always receives VER_NDX_LOCAL, however the value has no particular meaning.

Versions defined by an object are assigned version indexes starting at 1 and incremented by 1 for each version. Index 1 is reserved for the first global version. If the object does not have a SHT_SUNW_verdef version definition section, then all the global symbols defined by the object receive index 1. If the object does have a version definition section, then VER_NDX_GLOBAL simply refers to the first such version.

Versions required by the object from other SHT_SUNW_verneed dependencies, are assigned version indexes that start 1 past the final version definition index. These indexes are also incremented by 1 for each version. Since index 1 is always reserved for VER_NDX_GLOBAL, the first possible index for a dependency version is 2.

Versions of Solaris up to and including the Oracle Solaris 10 release, did not assign a version index to a SHT_SUNW_verneed dependency version. In such an object, any symbol reference had a version index of 0 indicating that no versioning information is available for that symbol.

♦ ♦ ♦ CHAPTER 13

Program Loading and Dynamic Linking

This chapter 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 following major sections are provided.

- "Program Header" on page 385 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 that is needed to create the memory image of the program.
- "Program Loading (Processor-Specific)" on page 391 describes the information used to load a program into memory.
- "Runtime Linker" on page 398 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 structure describes 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 391.

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. See `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;
```

`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 13-1.

`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 that are relevant to the segment. Type values and their meanings are specified in Table 13-2.

`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 391.

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 13-1 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_LOOS	0x60000000
PT_SUNW_UNWIND	0x6464e550
PT_SUNW_EH_FRAME	0x6474e550
PT_LOSUNW	0x6ffffffa
PT_SUNWBSS	0x6ffffffa
PT_SUNWSTACK	0x6ffffffb
PT_SUNWDTRACE	0x6ffffffc
PT_SUNWCAP	0x6ffffffd
PT_HISUNW	0x6fffffff
PT_HIOS	0x6fffffff
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL

Unused. Member 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`. These bytes follow the initialized area of the segment. The file size can not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, and are sorted on the `p_vaddr` member.

`PT_DYNAMIC`

Specifies dynamic linking information. See "Dynamic Section" on page 398.

`PT_INTERP`

Specifies the location and size of a null-terminated path name to invoke as an interpreter. This type is mandatory for dynamic executable files. This type can occur in shared objects. This type cannot occur more than once in a file. This type, if present, must precede any loadable segment entries. See "Program Interpreter" on page 397 for details.

`PT_NOTE`

Specifies the location and size of auxiliary information. See "Note Section" on page 349 for details.

`PT_SHLIB`

Reserved but has unspecified semantics.

`PT_PHDR`

Specifies the location and size of the program header table, both in the file and in the memory image of the program. This segment type cannot occur more than once in a file. Moreover, this segment 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 397 for details.

`PT_TLS`

Specifies a thread-local storage template. See "Thread-Local Storage Section" on page 428 for details.

`PT_LOOS - PT_HIOS`

Values in this inclusive range are reserved for OS-specific semantics.

`PT_SUNW_UNWIND`

This segment contains the stack unwind tables.

`PT_SUNW_EH_FRAME`

This segment contains the stack unwind table. `PT_SUNW_EH_FRAME` is equivalent to `PT_SUNW_EH_UNWIND`.

`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. Only one `PT_SUNWSTACK` element can exist. Only access permissions, as defined in the `p_flags` field, are meaningful.

`PT_SUNWDTRACE`

Reserved for internal use by `dtrace`(1M).

`PT_SUNWCAP`

Specifies capability requirements. See "Capabilities Section" on page 342 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 that are 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 391.

To compute the base address, you determine the memory address that are 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 restriction is not required by the file format. When the system creates loadable segment memory images, the system gives access permissions, as specified in the `p_flags` member. All bits that are included in the `PF_MASKPROC` mask are reserved for processor-specific semantics.

TABLE 13-2 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 between systems. 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 this permission is specified explicitly. The following table lists both the exact flag interpretation and the allowable flag interpretation.

TABLE 13-3 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 section or many sections, is also immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The following diagrams 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 12-12 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 415 and “Procedure Linkage Table (Processor-Specific)” on page 416 for details.

Sections of type `SHT_NOBITS` occupy no space in the file, but contribute 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, the system 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 the process references the logical page during execution. Processes commonly leave many pages unreferenced. Therefore, delaying physical reads can improve system performance. To obtain this efficiency in practice, executable files and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

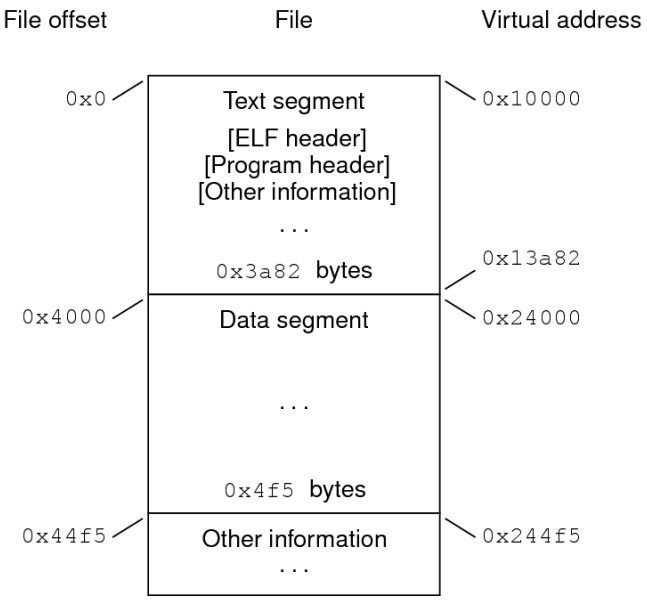
Virtual addresses and file offsets for 32–bit segments are congruent modulo 64K (`0x10000`). Virtual addresses and file offsets for 64–bit segments are congruent modulo 1 megabyte (`0x100000`). By aligning segments to the maximum page size, the files are suitable for paging regardless of physical page size.

By default, 64–bit SPARC programs are linked with a starting address of `0x100000000`. The whole program is located 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 the program truncates any of its pointers. While 64–bit programs are linked above 4 gigabytes, you can still link programs

below 4 gigabytes by using a `mapfile` and the `-M` option to the link-editor. See `/usr/lib/ld/sparcv9/map.below4G`.

The following figure presents the SPARC version of the executable file.

FIGURE 13-1 SPARC: Executable File (64K alignment)



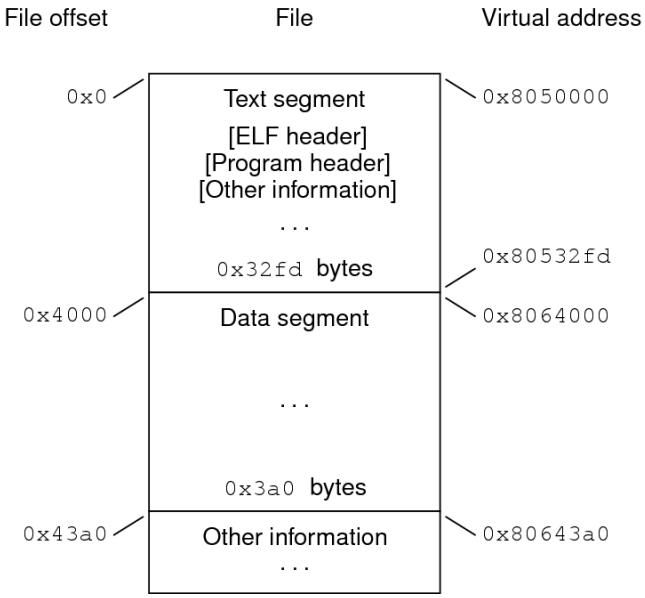
The following table defines the loadable segment elements for the previous figure.

TABLE 13-4 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 13-2 32-bit x86: Executable File (64K alignment)



The following table defines the loadable segment elements for the previous figure.

TABLE 13-5 32-bit 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 previous examples, 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 previous examples reflect typical Oracle Solaris OS 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 13-3 32-bit SPARC: Process Image Segments

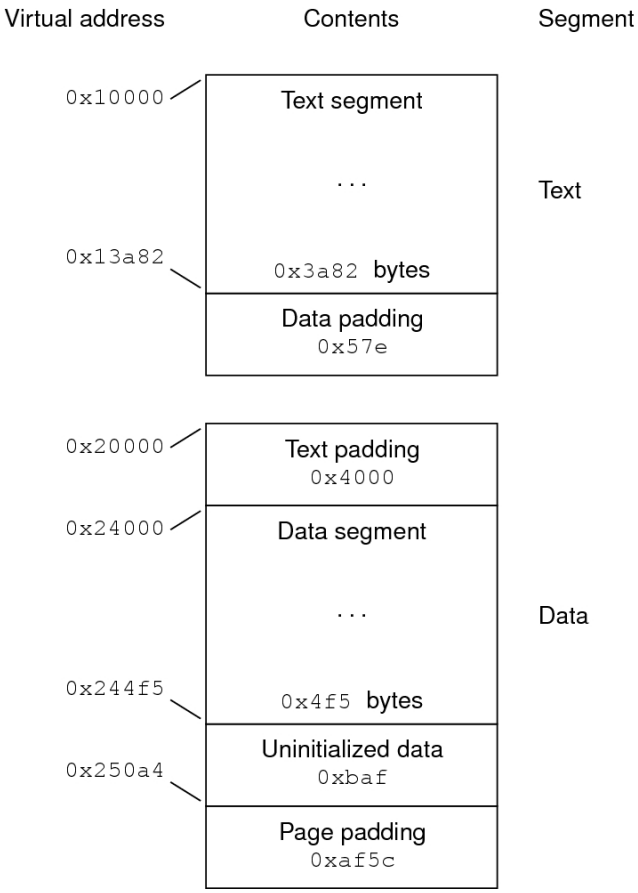
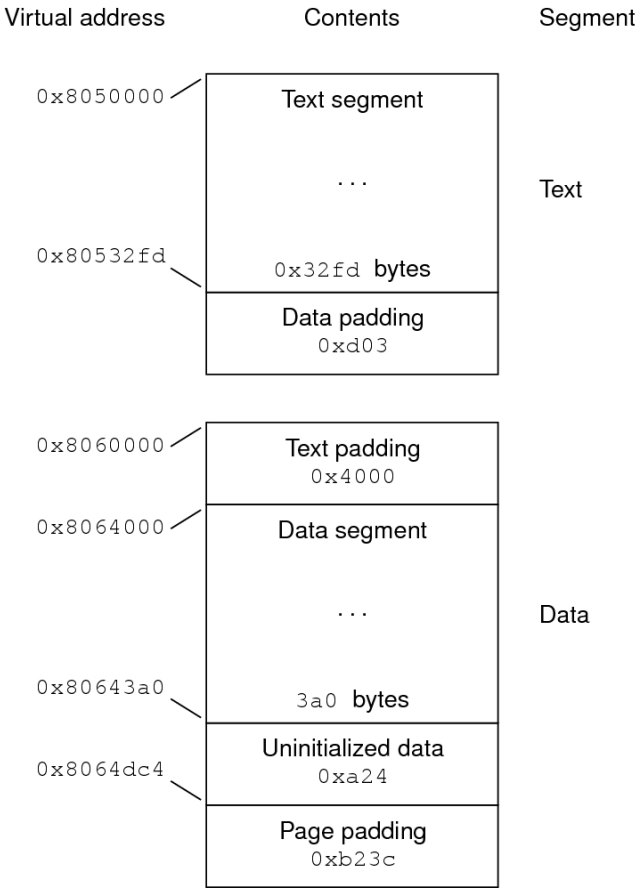


FIGURE 13-4 x86: Process Image Segments



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 between different processes, 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 13-6 32-bit 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 13-7 32-bit 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 Oracle 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, thus making the data 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 that follow, 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 that are 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. See `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.

In general, the value of each dynamic tag determines the interpretation of the `d_un` union. This convention provides for simpler interpretation of dynamic tags by third party tools. 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 the tag uses neither `d_ptr` nor `d_val`. Tags with values in the following special compatibility ranges do not follow these rules. Third party tools must handle these exception ranges explicitly on an item by item basis.

- Tags whose values are less than the special value `DT_ENCODING`.
- Tags with values that fall between `DT_LOOS` and `DT_SUNW_ENCODING`.
- Tags with values that fall between `DT_HIOS` and `DT_LOPROC`.

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 13-8 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

Name	Value	d_un	Executable	Shared Object
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
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_MAXPOSTAGS	34	Unspecified	Unspecified	Unspecified
DT_LOOS	0x6000000d	Unspecified	Unspecified	Unspecified
DT_SUNW_AUXILIARY	0x6000000d	d_ptr	Unspecified	Optional
DT_SUNW_RTLDINF	0x6000000e	d_ptr	Optional	Optional
DT_SUNW_FILTER	0x6000000e	d_ptr	Unspecified	Optional
DT_SUNW_CAP	0x60000010	d_ptr	Optional	Optional

Name	Value	d_un	Executable	Shared Object
DT_SUNW_SYMTAB	0x60000011	d_ptr	Optional	Optional
DT_SUNW_SYMSZ	0x60000012	d_val	Optional	Optional
DT_SUNW_ENCODING	0x60000013	Unspecified	Unspecified	Unspecified
DT_SUNW_SORTENT	0x60000013	d_val	Optional	Optional
DT_SUNW_SYMSORT	0x60000014	d_ptr	Optional	Optional
DT_SUNW_SYMSORTSZ	0x60000015	d_val	Optional	Optional
DT_SUNW_TLSSORT	0x60000016	d_ptr	Optional	Optional
DT_SUNW_TLSSORTSZ	0x60000017	d_val	Optional	Optional
DT_SUNW_CAPINFO	0x60000018	d_ptr	Optional	Optional
DT_SUNW_STRPAD	0x60000019	d_val	Optional	Optional
DT_SUNW_CAPCHAIN	0x6000001a	d_ptr	Optional	Optional
DT_SUNW_LDMACH	0x6000001b	d_val	Optional	Optional
DT_SUNW_CAPCHAINENT	0x6000001d	d_val	Optional	Optional
DT_SUNW_CAPCHAINSZ	0x6000001f	d_val	Optional	Optional
DT_SUNW_PARENT	0x60000021	d_val	Optional	Optional
DT_SUNW_ASLR	0x60000023	d_val	Optional	Ignored
DT_SUNW_RELAX	0x60000025	d_val	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_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

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 96.

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 416.

DT_PLTGOT

An address associated with the procedure linkage table or the global offset table. See "Procedure Linkage Table (Processor-Specific)" on page 416 and "Global Offset Table (Processor-Specific)" on page 415.

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 345.

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 364.

DT_SYMTAB

The address of the symbol table. See "Symbol Table Section" on page 365.

DT_RELA

The address of a relocation table. See "Relocation Sections" on page 351.

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, the runtime linker 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 136.

`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 96.

`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 the `-B symbolic` Option” on page 191.

`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 416. 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 178.

DT_JMPREL

The address of relocation entries that are associated solely with the procedure linkage table. See “Procedure Linkage Table (Processor-Specific)” on page 416. The separation of these relocation entries enables the runtime linker to ignore these entries when the object is loaded with lazy binding 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 13-11.

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 by means of dlopen(3C). This element's use has been superseded by the DF_BIND_NOW flag. See “When Relocations are Performed” on page 187.

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 96.

DT_FLAGS

Flag values specific to this object. See Table 13-9.

DT_ENCODING

Dynamic tag values that are greater than or equal to DT_ENCODING, and less than or equal to DT_LOOS, 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. This array 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_MAXPOSTAGS

The number of positive dynamic array tag values.

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_AUXILIARY

The DT_STRTAB string table offset of a null-terminated string that names one or more per-symbol, auxiliary *filtees*. See "Generating Auxiliary Filters" on page 144.

DT_SUNW_RTLDINF

Reserved for internal use by the runtime-linker.

DT_SUNW_FILTER

The DT_STRTAB string table offset of a null-terminated string that names one or more per-symbol, standard *filtees*. See "Generating Standard Filters" on page 141.

DT_SUNW_CAP

The address of the capabilities section. See "Capabilities Section" on page 342.

DT_SUNW_SYMTAB

The address of the symbol table containing local function symbols that augment the symbols provided by DT_SYMTAB. These symbols are always adjacent to, and immediately precede the symbols provided by DT_SYMTAB. See "Symbol Table Section" on page 365.

DT_SUNW_SYMSZ

The combined size of the symbol tables given by DT_SUNW_SYMTAB and DT_SYMTAB.

DT_SUNW_ENCODING

Dynamic tag values that are greater than or equal to DT_SUNW_ENCODING, and less than or equal to DT_HIOS, follow the rules for the interpretation of the d_un union.

DT_SUNW_SORTENT

The size, in bytes, of the DT_SUNW_SYMSORT and DT_SUNW_TLSSORT symbol sort entries.

DT_SUNW_SYMSORT

The address of the array of symbol table indices that provide sorted access to function and variable symbols in the symbol table referenced by DT_SUNW_SYMTAB. See “Symbol Sort Sections” on page 374.

DT_SUNW_SYMSORTSZ

The total size, in bytes, of the DT_SUNW_SYMSORT array.

DT_SUNW_TLSSORT

The address of the array of symbol table indices that provide sorted access to thread local symbols in the symbol table referenced by DT_SUNW_SYMTAB. See “Symbol Sort Sections” on page 374.

DT_SUNW_TLSSORTSZ

The total size, in bytes, of the DT_SUNW_TLSSORT array.

DT_SUNW_CAPINFO

The address of the array of symbol table indices that provide the association of symbols to their capability requirements. See “Capabilities Section” on page 342.

DT_SUNW_STRPAD

The total size, in bytes, of the unused reserved space at the end of the dynamic string table. If DT_SUNW_STRPAD is not present in an object, no reserved space is available.

DT_SUNW_CAPCHAIN

The address of the array of capability family indices. Each family of indices is terminated with a 0 entry.

DT_SUNW_LDMACH

The machine architecture of the link-editor that produced the object. DT_SUNW_LDMACH uses the same EM_ integer values used for the e_machine field of the ELF header. See “ELF Header” on page 304. DT_SUNW_LDMACH is used to identify the class, 32–bit or 64–bit, and the platform of the link-editor that built the object. This information is not used by the runtime linker, but exists purely for documentation.

DT_SUNW_CAPCHAINENT

The size, in bytes, of the DT_SUNW_CAPCHAIN entries.

DT_SUNW_CAPCHAINSZ

The total size, in bytes, or the DT_SUNW_CAPCHAIN chain.

`DT_SUNW_PARENT`

The `DT_STRTAB` string table offset of a null terminated parent object name. The name provided is a *basename*, containing only a file name without any path component. See "Parent Objects" on page 89.

`DT_SUNW_ASLR`

The Address Space Layout Randomization (ASLR) flag values specific to this object. See Table 13-12.

`DT_SUNW_RELAX`

The validity checking relaxation options, that were specified with the link-editor's `-z relax` option, when the object was built. See Table 13-13.

`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 377.

`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 379.

`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 381.

`DT_VERNEEDNUM`

The number of entries in the `DT_VERNEEDNUM` table.

`DT_RELACOUNT`

Indicates the `RELATIVE` relocation count, which is produced from the concatenation of all `Elf32_Rela`, or `Elf64_Rela` relocations. See "Combined Relocation Sections" on page 188.

DT_RELCOUNT

Indicates the RELATIVE relocation count, which is produced from the concatenation of all Elf32_Rel relocations. See "Combined Relocation Sections" on page 188.

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 144.

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 141.

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 347.

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 99.

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 271.

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 271.

DT_FLAGS_1

Flag values specific to this object. See Table 13-10.

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 the object 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. One dynamic entry exists for every STT_SPARC_REGISTER symbol in the symbol table. See "Register Symbols" on page 376.

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 13-9 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 257.

DF_SYMBOLIC

Indicates that the object contains symbolic bindings that were applied during its link-edit. See "Using the -B symbolic Option" on page 191.

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 178.

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 by means of dlopen(3C). See "When Relocations are Performed" on page 187.

DF_STATIC_TLS

Indicates that the object contains code using a static thread-local storage scheme. Static thread-local storage should not be used in objects that are dynamically loaded, either using dlopen(3C), or using lazy loading.

TABLE 13-10 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 the 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 has been carried out.
DF_1_DISPRELPND	0x10000	Displacement relocation pending.
DF_1_NODIRECT	0x20000	Object contains non-direct bindings.
DF_1_IGNMULDEF	0x40000	Internal use.
DF_1_NOKSYMS	0x80000	Internal use.
DF_1_NOHDR	0x100000	Internal use.
DF_1_EDITED	0x200000	Object has been modified since originally built.
DF_1_NORELOC	0x400000	Internal use.

Name	Value	Meaning
DF_1_SYMINTPOSE	0x800000	Individual symbol interposers exist.
DF_1_GLOBAUDIT	0x1000000	Establish global auditing.
DF_1_SINGLETON	0x2000000	Singleton symbols exist.
DF_1_STUB	0x4000000	Object is a stub.
DF_1_PIE	0x8000000	Object is a position-independent executable.

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 by means of dlopen(3C). See "When Relocations are Performed" on page 187.

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 124.

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), the object cannot be unloaded with dlclose(3C). This flag is recorded in the object using the link-editor -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 147.

DF_1_INITFIRST

Indicates that this object's initialization section be run before any other objects loaded. 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 257.

`DF_1_DIRECT`

Indicates that the object should use direct binding information. See Chapter 6, "Direct Bindings".

`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 "Runtime Interposition" on page 102.

`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 256.

`DF_1_DISPRELDNE`

Indicates that this object has displacement relocations applied. The displacement relocation records no longer exist within the object as the records were discarded once the relocation was applied. See "Displacement Relocations" on page 75.

`DF_1_DISPRELPND`

Indicates that this object has displacement relocations pending. The displacement relocations exits within the object so the relocation can be completed at runtime. See "Displacement Relocations" on page 75.

`DF_1_NODIRECT`

Indicates that this object contains symbols that can not be directly bound to. See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

`DF_1_IGNMULDEF`

Reserved for internal use by the kernel runtime-linker.

`DF_1_NOKSYMS`

Reserved for internal use by the kernel runtime-linker.

`DF_1_NOHDR`

Reserved for internal use by the kernel runtime-linker.

`DF_1_EDITED`

Indicates that this object has been edited or has been modified since the objects original construction by the link-editor. This flag serves as a warning to debuggers that an object might have had an arbitrary change made since the object was originally built.

`DF_1_NORELOC`

Reserved for internal use by the kernel runtime-linker.

`DF_1_SYMINTPOSE`

Indicates that the object contains individual symbols that should interpose before all symbols except the primary load object, which is typically the executable. This flag is recorded when the object is built using a `mapfile` and the `INTERPOSE` keyword. See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

`DF_1_GLOBAUDIT`

Indicates that the dynamic executable requires global auditing. See "Recording Global Auditors" on page 274.

`DF_1_SINGLETON`

Indicates that the object defines, or makes reference to `singleton` symbols. See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

`DF_1_STUB`

Indicates that the object is a stub. See "Stub Objects" on page 77.

`DF_1_PIE`

Indicates that the object is a position-independent executable, which is a special case of a shared object, that specifies an interpreter. See the link-editor's `-z type` option.

TABLE 13-11 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 106.

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 124.

TABLE 13-12 ELF ASLR Values, DT_SUNW_ASLR

Name	Value	Meaning
DV_SUNW_ASLR_DEFAULT	0	Follow system default
DV_SUNW_ASLR_DISABLE	1	Disable ASLR
DV_SUNW_ASLR_ENABLE	2	Enable ASLR

DV_SUNW_ASLR_DISABLE and DV_SUNW_ASLR_ENABLE are recorded in the object using the link-editor's -z aslr option.

TABLE 13-13 ELF Dynamic Relaxation Flags, DT_SUNW_RELAX

DF_SUNW_RELAX_COMDAT	0x1	Relocation symbols substituted for discarded COMDAT
DF_SUNW_RELAX_SECADJ	0x2	Section adjacency verification disabled
DF_SUNW_RELAX_SYMBOUND	0x4	Symbol/section boundary verification disabled
DF_SUNW_RELAX_COMMON	0x8	Tentative (common) data with different size or different alignment enabled

DF_SUNW_RELAX_ flags are recorded in the object as a consequence of using the link-editor's -z relax option.

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 GOT using position-independent addressing and extracts absolute values. This technique redirects position-independent references to absolute locations.

Initially, the GOT 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 relocations can be of type R_xxxx_GLOB_DAT, referring to the GOT.

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 GOT entry. Because the executable file and shared objects have a separate GOT, a symbol's address can appear in several tables. The runtime linker processes all the GOT 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. The system 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 GOT format and interpretation are processor-specific. 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 between

different dynamic objects. 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.

32-bit SPARC: 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 that is shown in Table 13-14. 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 13-14 shows four entries. Two of the four are initial reserved entries. The third entry is a call to `name101`. The fourth entry is a call to `name102`. The example assumes that the entry for `name102` is the table's last entry. A `nop` instruction follows this last entry. The left column shows the instructions from the object file before dynamic linking. The right column illustrates a possible instruction sequence that the runtime linker might use to fix the procedure linkage table entries.

TABLE 13-14 32-bit SPARC: Procedure Linkage Table Example

Object File	*Memory Segment*

`.PLT0:`
`  unimp`
`  unimp`
`  unimp`
`.PLT1:`
`  unimp`
`  unimp`
`  unimp`
 	
`.PLT101:`
`  sethi  (.-.PLT0), %g1`
`  ba,a  .PLT0`
`  nop`
`.PLT102:`
	`.PLT0:`
	`  save  %sp, -64, %sp`
	`  call  runtime_linker`
	`  nop`
	`.PLT1:`
	`  .word  identification`
	`  unimp`
	`  unimp`
	
	`.PLT101:`
	`  nop`
	`  ba,a  name101`
	`  nop`
	`.PLT102:`

Object File	*Memory Segment*
`sethi (.-.PLT0), %g1`	`sethi (.-.PLT0), %g1`
`ba,a .PLT0`	`sethi %hi(name102), %g1`
`nop`	`jmpl %g1+%lo(name102), %g0`
`nop`	`nop`

The following steps describe how the runtime linker and program jointly resolve the symbolic references through the procedure linkage table. The steps that are described are for explanation only. The precise execution-time behavior of the runtime linker is not specified.

1. When the memory image of the program is initially created, the runtime linker changes the initial procedure linkage table entries. These entries are modified so that control can be transferred 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, this word is examined to identify the caller.
2. All other procedure linkage table entries initially transfer to the first entry. Thus, the runtime linker gains control at the first execution of a 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`. This relocation 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 the runtime linkers 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 `jmp1` 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 `jmp1`. The required, trailing `nop` fills this delay slot.

Note - The different instruction sequences that are 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 before transferring control to the program.

64-bit SPARC: 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 destination 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 that is shown in Table 13-15. 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 13-15 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 instruction sequence that the runtime linker might use to fix the procedure linkage table entries.

TABLE 13-15 64-bit SPARC: Procedure Linkage Table Example

Object File	*Memory Segment*

`.PLT0:`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`.PLT1:`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`  unimp`
`.PLT2:`
`  unimp`
 	
`.PLT101:`
`  sethi  (.-.PLT0), %g1`
`  ba,a  %xcc, .PLT1`
`  nop`
`  nop`
`  nop;  nop`
`  nop;  nop`
`.PLT102:`
`  sethi  (.-.PLT0), %g1`
`  ba,a  %xcc, .PLT1`
`  nop`
`  nop`
`  nop;  nop`
`  nop;  nop`
`.PLT103:`
`  sethi  (.-.PLT0),  %g1`
	`.PLT0:`
	`  save  %sp, -176, %sp`
	`  sethi  %hh(runtime_linker_0), %l0`
	`  sethi  %lm(runtime_linker_0), %l1`
	`  or  %l0, %hm(runtime_linker_0), %l0`
	`  sllx  %l0, 32, %l0`
	`  or  %l0, %l1, %l0`
	`  jmpl  %l0+%lo(runtime_linker_0), %o1`
	`  mov  %g1, %o0`
	`.PLT1:`
	`  save  %sp, -176, %sp`
	`  sethi  %hh(runtime_linker_1), %l0`
	`  sethi  %lm(runtime_linker_1), %l1`
	`  or  %l0, %hm(runtime_linker_1), %l0`
	`  sllx  %l0, 32, %l0`
	`  or  %l0, %l1, %l0`
	`  jmpl  %l0+%lo(runtime_linker_0), %o1`
	`  mov  %g1, %o0`
	`.PLT2:`
	`  .xword  identification`
	
	`.PLT101:`
	`  nop`
	`  mov  %o7,  %g1`
	`  call  name101`
	`  mov  %g1, %o7`
	`  nop;  nop`
	`  nop;  nop`
	`.PLT102:`
	`  nop`
	`  sethi  %hi(name102), %g1`
	`  jmpl  %g1+%lo(name102), %g0`
	`  nop`
	`  nop;  nop`
	`  nop;  nop`
	`.PLT103:`
	`  nop`

Object File	*Memory Segment*
`    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`
`.PLT32768:`	`.PLT32768:`
`    mov     %o7, %g5`	`    <unchanged>`
`    call    .+8`	`    <unchanged>`
`    nop`	`    <unchanged>`
`    ldx     [%o7+.PLTP32768 -` `      (.PLT32768+4)], %g1`	`    <unchanged>`
`    jmpl    %o7+%g1, %g1`	`    <unchanged>`
`    mov     %g5, %o7`	`    <unchanged>`
`    ....`	`    ....`
`.PLT32927:`	`.PLT32927:`
`    mov     %o7, %g5`	`    <unchanged>`
`    call    .+8`	`    <unchanged>`
`    nop`	`    <unchanged>`
`    ldx     [%o7+.PLTP32927 -` `      (.PLT32927+4)], %g1`	`    <unchanged>`
`    jmpl    %o7+%g1, %g1`	`    <unchanged>`
`    mov     %g5, %o7`	`    <unchanged>`
`.PLTP32768`	`.PLTP32768`
`    .xword  .PLT0 -` `      (.PLT32768+4)`	`    .xword  name32768 -` `              (.PLT32768+4)`
`    ....`	`    ....`
`.PLTP32927`	`    .PLTP32927`
`    .xword  .PLT0 -` `      (.PLT32927+4)`	`    .xword  name32927 -` `              (.PLT32927+4)`

The following steps describe how the runtime linker and program jointly resolve the symbolic references through the procedure linkage table. The steps that are described are for explanation only. The precise execution-time behavior of the runtime linker is not specified.

1. When the memory image of the program is initially created, the runtime linker changes the initial procedure linkage table entries. These entries are modified so that control is transfer to 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, this word is examined to identify the caller.
2. All other procedure linkage table entries initially transfer to the first or second entry. These 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. If the runtime linker 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, the instruction 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 that are 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 before transferring control to the program.

32-bit x86: 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 13-16 32-bit 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 13-17 32-bit 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.

The following steps describe how the runtime linker and program cooperate to resolve the symbolic references through the procedure linkage table and the global offset table.

1. When the memory image of the program is initially created, the runtime linker sets the second and third entries in the global offset table to special values. The following steps 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 calling 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 before transferring control to the program.

x64: Procedure Linkage Table

For x64 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 13-18 x64: Procedure Linkage Table Example

```
.PLT0:
    pushq   GOT+8(%rip)                     # GOT[1]
    jmp     *GOT+16(%rip)                   # GOT[2]
    nop;    nop
    nop;    nop
.PLT1:
    jmp     *name1@GOTPCREL(%rip)           # 16 bytes from .PLT0
    pushq   $index1
    jmp     .PLT0
.PLT2:
    jmp     *name2@GOTPCREL(%rip)           # 16 bytes from .PLT1
    pushl   $index2
```

```
    jmp     .PLT0
```

The following steps describe how the runtime linker and program cooperate to resolve the symbolic references through the procedure linkage table and the global offset table.

1. When the memory image of the program is initially created, the runtime linker sets the second and third entries in the global offset table to special values. The following steps explain these values.
2. 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.
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 `pushq` instruction, not the real address of `name1`.
5. The program pushes a relocation index (`index1`) on the stack. The relocation offset is a 32–bit, nonnegative index into the relocation table. The relocation table is identified by the `DT_JUMPREL` dynamic section entry. The designated relocation entry has the type `R_AMD64_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 index, the program jumps to `.PLT0`, the first entry in the procedure linkage table. The `pushq` instruction pushes the value of the second global offset table entry (`GOT+8`) 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+16`), 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 `pushq` instruction.

The `LD_BIND_NOW` environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes `R_AMD64_JMP_SLOT` relocation entries before transferring control to the program.

♦ ♦ ♦ CHAPTER 14

Thread-Local Storage

The compilation environment supports the declaration of thread-local data. This data is sometimes 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.

- A foundation is provided upon which the POSIX interfaces for allocating thread specific data are built.
- A convenient, and efficient mechanism for direct use of thread local variables by applications and libraries is provided.
- Compilers can 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 externally and referenced externally. Thread-local variables are subject to the same interposition rules as normal symbols.

Dynamic loading restrictions

Various TLS access models are available. See "Thread-Local Storage Access Models" on page 433. Shared object developers should be aware of the restrictions imposed by some of these access models in relation to object loading. A shared object can be dynamically loaded during process startup, or after process startup by means of lazy loading, filters, or `dlopen`(3C). At the completion of process startup, the thread pointer for the main thread is established. All static TLS storage requirements are calculated before the thread pointer is established.

Shared objects that reference thread-local variables, should insure that every translation unit containing the reference is compiled with a dynamic TLS model. This model of access provides the greatest flexibility for loading shared objects. However, static TLS models can generate faster code. Shared objects that use a static TLS model can be loaded as part of process initialization. However, after process initialization, shared objects that use a static TLS model can only be loaded if sufficient backup TLS storage is available. See "Program Startup" on page 430.

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 that is 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. The relocated values are 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 TLS offset for defined symbols. This field contains zero for undefined symbols.

Several relocations are defined to support access to TLS. See “Thread-Local Storage Relocation Types” on page 440, “Thread-Local Storage Relocation Types” on page 446 and “Thread-Local Storage Relocation Types” on page 451. TLS relocations typically reference symbols of type `STT_TLS`. TLS relocations can also reference local section symbols in association with a `GOT` entry. In this case, the assigned TLS offset is stored in the associated `GOT` entry.

For relocations against static TLS items, the relocation address is encoded as a negative offset from the end of the static TLS template. This offset is calculated by first rounding the template size to the nearest 8-byte boundary in a 32-bit object, and to the nearest 16-byte boundary in a 64-bit object. This rounding ensures that the static TLS template is suitably aligned for any use.

In dynamic executables and shared objects, a `PT_TLS` program entry describes a TLS template. This template has the following members.

TABLE 14-1 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`	`0`
`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

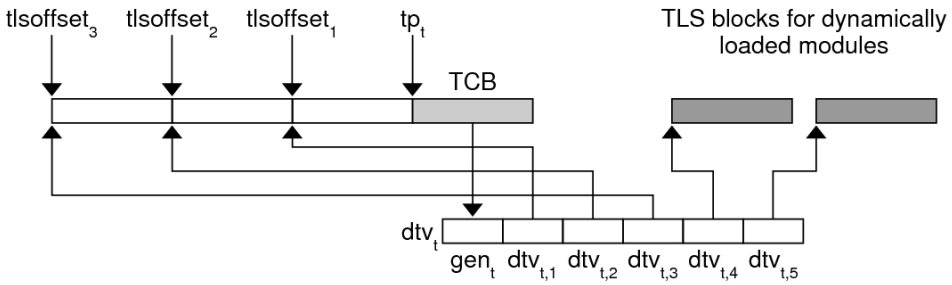
Runtime Allocation of Thread-Local Storage

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 object is loaded following program startup.

Thread-local data storage is laid out at runtime as illustrated in Figure 14-1.

FIGURE 14-1 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.

- $\texttt{tlsoffset}_1 = \texttt{round}(\texttt{tlssize}_1, \texttt{align}_1)$
- $\texttt{tlsoffset}_{m+1} = \texttt{round}(\texttt{tlsoffset}_m + \texttt{tlssize}_{m+1}, \texttt{align}_{m+1})$

$\texttt{tlssize}_{m+1}$ and $\texttt{align}_{m+1}$ are the size and alignment, respectively, for the allocation template for dynamic object `m`. Where `1 <= m <= M`, and `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*.

Next, the runtime linker computes the allocation size that is required for the startup TLS, $\texttt{tlssize}_S$. This size is equal to $\texttt{tlsoffset}_M$, plus an additional `512` bytes. This addition

provides a backup reservation for static TLS references. Shared objects that make static TLS references, and are loaded after process initialization, are assigned to this backup reservation. However, this reservation is a fixed, limited size. In addition, this reservation is only capable of providing storage for uninitialized TLS data items. For maximum flexibility, shared objects should reference thread-local variables using a dynamic TLS model.

The static TLS arena associated with the calculated TLS size $\texttt{tlssize}_S$, is placed immediately preceding the thread pointer $\texttt{tp}_t$. Accesses to this TLS data is based off of subtractions from $\texttt{tp}_t$.

The static TLS arena is associated with a linked list of initialization records. Each record in this list describes the TLS initialization image for one loaded dynamic object. Each record contains the following fields.

- A pointer to the TLS initialization image.
- The size of the TLS initialization image.
- The $\texttt{tlsoffset}_m$ 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 $\texttt{tp}_t$, which points to the thread control block, `TCB`. The thread pointer, `tp`, always contains the value of $\texttt{tp}_t$ for the current running thread.

The thread library then creates a vector of pointers, $\texttt{dtv}_t$, for the current thread `t`. The first element of each vector contains a generation number $\texttt{gen}_t$, which is used to determine when the vector needs to be extended. See “Deferred Allocation of Thread-Local Storage Blocks” on page 432.

Each element remaining in the vector $\texttt{dtv}_{t,m}$, is a pointer to the block that is reserved for the TLS belonging to the dynamic object `m`.

For dynamically loaded, post-startup objects, the thread library defers the allocation of TLS blocks. Allocation occurs when the first reference is made to a TLS variable within the loaded object. For blocks whose allocation has been deferred, the pointer $\texttt{dtv}_{t,m}$ is set to an implementation-defined special value.

Note - The runtime linker can group TLS templates for all startup objects so as to share a single element in the vector, $dtv_{t,1}$. This grouping does not affect the offset calculations described previously 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

A shared object containing only dynamic TLS can be loaded following process startup without limitations. The runtime linker extends the list of initialization records to include the initialization template of the new object. The new object is given an index of `m = M + 1`. The counter `M` is incremented by `1`. However, the allocation of new TLS blocks is deferred until the blocks are actually referenced.

When a shared object that contains only dynamic TLS is unloaded, the TLS blocks used by that shared object are freed.

A shared object containing static TLS can be loaded following process startup with limitations. Static TLS references can only be satisfied from any remaining backup TLS reservation. See "Program Startup" on page 430. This reservation is limited in size. In addition, this reservation can only provide storage for uninitialized TLS data items.

A shared object that contains static TLS is never unloaded. The shared object is tagged as non-deletable as a consequence of processing the static TLS.

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 $\mathbf{dtv}_t$ 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 and x64)
extern void *___tls_get_addr(TLS_index *ti);    (32–bit x86)
```

Note - The SPARC and 64–bit x86 definitions of this function have the same function signature. However, the 32–bit x86 version does not use the default calling convention of passing arguments on the stack. Instead, the 32–bit x86 version passes its arguments by means of the `%eax` register which is more efficient. To denote that this alternate calling method is used, the 32–bit 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 the vector 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 the vector has not been allocated, the routine allocates and initializes the block. The routine uses 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 the block is first referenced from a specific thread.

Local Dynamic (LD) - dynamic TLS of local symbols

This model is a optimization of the *GD* model. The compiler might determine that a variable is bound locally, or protected, within the object being built. In this case, the compiler instructs the link-editor to statically bind the dynamic `tlsoffset` and use this model. This model provides a performance benefit over the *GD* model. Only one call to `tls_get_addr` is required per function, to determine the address of $dtv_{0,m}$. The dynamic TLS offset, bound at link-edit time, is added to the $dtv_{0,m}$ 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 that are available at process startup, plus a small backup reservation. See “Program Startup” on page 430. In this model, the thread pointer-relative offset for a given variable *x* is stored in the `GOT` entry for *x*.

This model can reference a limited number of TLS variables from shared libraries loaded after initial process startup, such as by means of lazy loading, filters, or `dlopen`(3C).

This access is satisfied from a fixed backup reservation. This reservation can only provide storage for uninitialized TLS data items. For maximum flexibility, shared objects should reference thread-local variables using a dynamic TLS model.

Note - Filters can be employed to dynamically select the use of static TLS. A shared object can be built to use dynamic TLS, and act as an auxiliary filter upon a counterpart built to use static TLS. If resourses allow the static TLS object to be loaded, the object is used. Otherwise, a fall back to the dynamic TLS object insures that the functionality provided by the shared object is always available. For more information on filters see "Shared Objects as Filters" on page 140.

Local Executable (LE) - static TLS

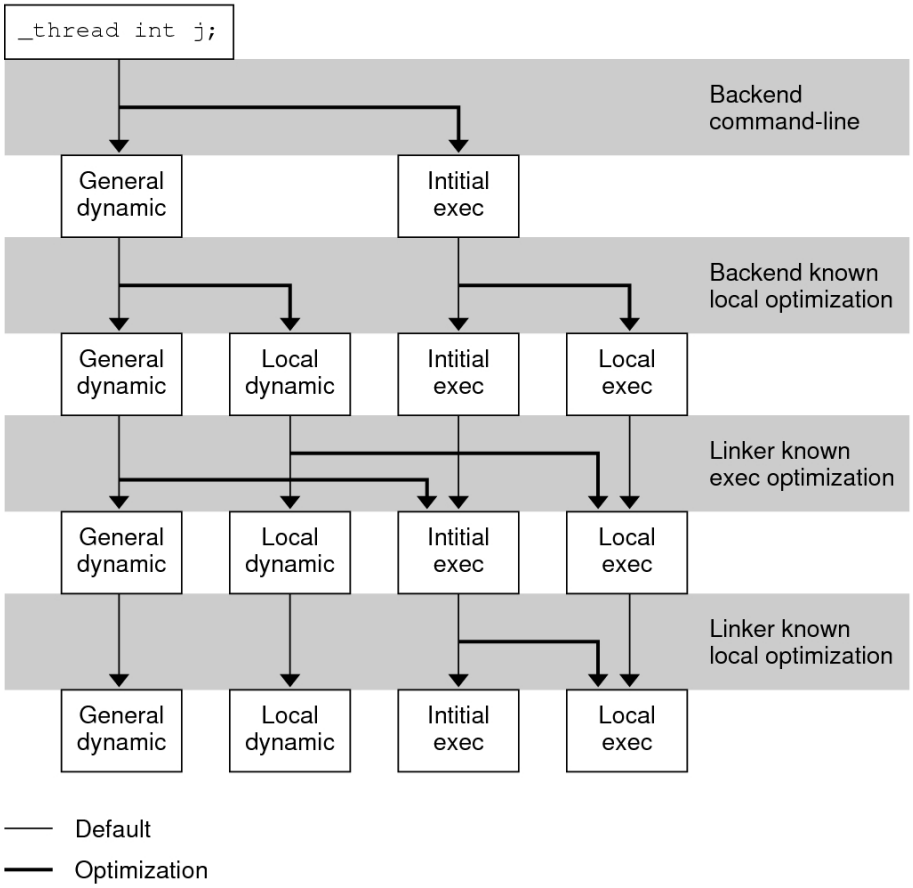
This model can only reference TLS variables which are part of the TLS block of the dynamic executable. The link-editor calculates the thread pointer-relative offsets statically, without the need for dynamic relocations, or the extra reference to the `GOT`. This model can not be used to reference variables outside of the dynamic executable.

The link-editor can transition code from the more general access models to the more optimized models, if the transition is determined appropriate. 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.

Knowledge of the TLS access model, together with the type of object being created, allows the link-editor to perform translations. An example is if a relocatable object using the *GD* access model is being linked into a dynamic executable. In this case, the link-editor can transition the references using the *IE* or *LE* access models, as appropriate. The relocations that are required for the model are then performed.

The following diagram illustrates the different access models, together with the transition of one model to another model.

FIGURE 14-2 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: General Dynamic (GD)

This code sequence implements the `GD` model described in “Thread-Local Storage Access Models” on page 433.

TABLE 14-2 SPARC: General Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`# %l7 - initialized to GOT pointer`		
		x
`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	
`# %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 `GOT` 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.

The load object index and TLS block index for `x` are not known until runtime. Therefore, 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 uses the special syntax, `x@TLSPLT`. This call references the TLS variable and 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. The `add` instruction can not be placed into the delay slot for the call. This requirement is necessary 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 requirement permits the link-editor to identify the `GOT`-pointer register during a code transformation.

SPARC: Local Dynamic (LD)

This code sequence implements the LD model described in “Thread-Local Storage Access Models” on page 433.

TABLE 14-3 SPARC: 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`		
	R_SPARC_TLS_LDO_HIX22	x1
`0x10 sethi %hi(@dtpoff(x1)), %l1`	R_SPARC_TLS_LDO_LOX10	x1
`0x14 xor %l1, %lo(@dtpoff(x1)), %l1`	R_SPARC_TLS_LDO_ADD	x1
`0x18 add %o0, %l1, %l1`		
`# %l1 - contains address of local TLS variable x1`		
	R_SPARC_TLS_LDO_HIX22	x2
`0x20 sethi %hi(@dtpoff(x2)), %l2`	R_SPARC_TLS_LDO_LOX10	x2
`0x24 xor %l2, %lo(@dtpoff(x2)), %l2`	R_SPARC_TLS_LDO_ADD	x2
`0x28 add %o0, %l2, %l2`		
`# %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` instruction and `add` instruction 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 `GOT` 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.

The load object index is not known until runtime. Therefore, 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` instruction and `xor` instruction 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 requirement permits the link-editor to identify the register during a code transformation.

32-bit SPARC: Initial Executable (IE)

This code sequence implements the `IE` model described in "Thread-Local Storage Access Models" on page 433.

TABLE 14-4 32-bit SPARC: Initial Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
`# %l7 - initialized to GOT pointer, %g7 - thread pointer`		
	`R_SPARC_TLS_IE_HI22`	`x`
`0x00 sethi %hi(@tpoff(x)), %o0`	`R_SPARC_TLS_IE_LO10`	`x`
`0x04 or %o0, %lo(@tpoff(x)), %o0`	`R_SPARC_TLS_IE_LD`	`x`
`0x08 ld [%l7 + %o0], %o0`	`R_SPARC_TLS_IE_ADD`	`x`
`0x0c add %g7, %o0, %o0`		
`# %o0 - contains address of TLS variable`		
	Outstanding Relocations	**Symbol**
`GOT[n]`	`R_SPARC_TLS_TPOFF32`	`x`

The `sethi` instruction and `or` instruction 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 `GOT` to store the static TLS offset for symbol `x`. An `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 requirement permits the link-editor to identify the GOT-pointer register during a code transformation.

64-bit SPARC: Initial Executable (IE)

This code sequence implements the IE model described in "Thread-Local Storage Access Models" on page 433.

TABLE 14-5 64-bit SPARC: Initial Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
# %l7 - initialized to GOT pointer, %g7 - thread pointer		
	R_SPARC_TLS_IE_HI22	x
0x00 sethi %hi(@tpoff(x)), %o0	R_SPARC_TLS_IE_LO10	x
0x04 or %o0, %lo(@tpoff(x)), %o0	R_SPARC_TLS_IE_LD	x
0x08 ldx [%l7 + %o0], %o0	R_SPARC_TLS_IE_ADD	x
0x0c add %g7, %o0, %o0		
# %o0 - contains address of TLS variable		
	Outstanding Relocations	**Symbol**
GOT[n]	R_SPARC_TLS_TPOFF64	x

SPARC: Local Executable (LE)

This code sequence implements the LE model described in "Thread-Local Storage Access Models" on page 433.

TABLE 14-6 SPARC: 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 that are 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 GOT 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 GOT 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 a TLS symbol.

TABLE 14-7 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	Refer to the explanation following this table.
R_SPARC_TLS_GD_CALL	59	V-disp30	Refer to the explanation following this table.
R_SPARC_TLS_LDM_HI22	60	T-simm22	@tmndx(S + A) >> 10
R_SPARC_TLS_LDM_LO10	61	T-simm13	@tmndx(S + A) & 0x3ff

Name	Value	Field	Calculation
R_SPARC_TLS_LDM_ADD	62	None	Refer to the explanation following this table.
R_SPARC_TLS_LDM_CALL	63	V-disp30	Refer to the explanation following this table.
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	Refer to the explanation following this table.
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	Refer to the explanation following this table.
R_SPARC_TLS_IE_LDX	70	None	Refer to the explanation following this table.
R_SPARC_TLS_IE_ADD	71	None	Refer to the explanation following this table.
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 relocation 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 relocation 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 that is computed in the initial part of the code sequence is the first register in this instruction. This relocation 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 relocation 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 relocation 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.

32-bit x86: Thread-Local Variable Access

On x86, the following code sequence models are available for accessing TLS.

32-bit x86: General Dynamic (GD)

This code sequence implements the `GD` model described in “Thread-Local Storage Access Models” on page 433.

TABLE 14-8 32-bit x86: General Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol

`0x00 leal  x@tlsgd(,%ebx,1), %eax`
`0x07 call  x@tlsgdplt`
 	
`# %eax - contains address of TLS variable`
	`R_386_TLS_GD`
	`R_386_TLS_GD_PLT`
		`x`
		`x`

	Outstanding Relocations	Symbol
GOT[n] GOT[n + 1]	R_386_TLS_DTPMOD32 R_386_TLS_DTPOFF32	x

The leal instruction generates a R_386_TLS_GD relocation which instructs the link-editor to allocate space in the GOT 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 requirement is necessary to permit the code transformations.

x86: Local Dynamic (LD)

This code sequence implements the LD model described in “Thread-Local Storage Access Models” on page 433.

TABLE 14-9 32-bit x86: Local Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 leal x1@tlsldm(%ebx), %eax 0x06 call x1@tlsldmplt # %eax - contains address of TLS block of current object 0x10 leal x1@dtpoff(%eax), %edx # %edx - contains address of local TLS variable x1 0x20 leal x2@dtpoff(%eax), %edx # %edx - contains address of local TLS variable x2	R_386_TLS_LDM R_386_TLS_LDM_PLT R_386_TLS_LDO_32 R_386_TLS_LDO_32	x1 x1 x1 x2
	Outstanding Relocations	**Symbol**
GOT[n] GOT[n + 1]	R_386_TLS_DTPMOD32 <none>	x

The first `leal` instruction generates a `R_386_TLS_LDM` relocation. This relocation instructs the link-editor to allocate space in the `GOT` 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.

The load object index is not known until runtime. Therefore, a `R_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.

32-bit x86: Initial Executable (IE)

This code sequence implements the `IE` model described in "Thread-Local Storage Access Models" on page 433.

Two code-sequences for the `IE` model exist. 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.

TABLE 14-10 32-bit x86: Initial Executable, Position Independent, Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol

`0x00 movl  %gs:0, %eax`
`0x06 addl  x@gotntpoff(%ebx), %eax`
 	
`# %eax - contains address of TLS variable`
	`<none>`
	`R_386_TLS_GOTIE`
		
		`x`

	Outstanding Relocations	**Symbol**
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

The `addl` instruction generates a `R_386_TLS_GOTIE` relocation. This relocation instructs the link-editor to create space in the `GOT` 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 14-11 32-bit x86: Initial Executable, Position Dependent, Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol

0x00 movl %gs:0, %eax 0x06 addl x@indntpoff, %eax # %eax - contains address of TLS variable	<none> R_386_TLS_IE	 x
	Outstanding Relocations	**Symbol**
GOT[n]	R_386_TLS_TPOFF	x

The addl instruction generates a R_386_TLS_IE relocation. This relocation instructs the link-editor to create space in the GOT 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 using 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 than the address, can be loaded by embedding the offset directly into the memory reference as shown in the next two sequences.

TABLE 14-12 32-bit x86: Initial Executable, Position Independent, Dynamic Thread-Local Variable Access Codes

Code Sequence	**Initial Relocations**	**Symbol**
0x00 movl x@gotntpoff(%ebx), %eax 0x06 movl %gs:(%eax), %eax # %eax - contains address of TLS variable	R_386_TLS_GOTIE <none>	x
	Outstanding Relocations	**Symbol**
GOT[n]	R_386_TLS_TPOFF	x

TABLE 14-13 32-bit x86: Initial Executable, Position Independent, Thread-Local Variable Access Codes

Code Sequence	**Initial Relocations**	**Symbol**
0x00 movl x@indntpoff, %ecx 0x06 movl %gs:(%ecx), %eax # %eax - contains address of TLS variable	R_386_TLS_IE <none>	x
	Outstanding Relocations	**Symbol**
GOT[n]	R_386_TLS_TPOFF	x

In the last sequence, if the %eax register is used instead of the %ecx register, the first instruction can be either 5 or 6 bytes long.

32-bit x86: Local Executable (LE)

This code sequence implements the LE model described in “Thread-Local Storage Access Models” on page 433.

TABLE 14-14 32-bit x86: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 movl %gs:0, %eax 0x06 leal x@ntpoff(%eax), %eax # %eax - contains address of TLS variable	<none> R_386_TLS_LE	x

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 14-15 32-bit x86: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 movl %gs:0, %eax 0x06 movl x@ntpoff(%eax), %eax # %eax - contains address of TLS variable	<none> R_386_TLS_LE	x

Rather than computing the address of the variable, a load from the variable or store to the variable can be accomplished using the following sequence. Note, the x@ntpoff expression is not used as an immediate value, but as an absolute address.

TABLE 14-16 32-bit x86: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 movl %gs:x@ntpoff, %eax # %eax - contains address of TLS variable	R_386_TLS_LE	x

32-bit x86: Thread-Local Storage Relocation Types

The TLS relocations that are 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`. The `ti_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 the entry with the negative `tlsoffset` relative to the static TLS block. This sequence is performed at runtime using the `R_386_TLS_TPOFF` relocation.

`@indntpoff(x)`

This expression is similar to `@gotntpoff`, but is 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 a TLS symbol.

TABLE 14-17 32-bit 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)

Name	Value	Field	Calculation
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)

x64: Thread-Local Variable Access

On x64, the following code sequence models are available for accessing TLS

x64: General Dynamic (GD)

This code sequence implements the GD model described in "Thread-Local Storage Access Models" on page 433.

TABLE 14-18 x64: General Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 .byte 0x66 0x01 leaq x@tlsgd(%rip), %rdi 0x08 .word 0x6666 0x0a rex64 0x0b call __tls_get_addr@plt # %rax - contains address of TLS variable	<none> R_AMD64_TLSGD <none> <none> R_AMD64_PLT32	 x __tls_get_addr
	Outstanding Relocations	**Symbol**
GOT[n] GOT[n + 1]	R_AMD64_DTPMOD64 R_AMD64_DTPOFF64	x x

The __tls_get_addr function takes a single parameter, the address of the tls_index structure. The R_AMD64_TLSGD relocation that is associated with the x@tlsgd(%rip) expression, instructs the link-editor to allocate a tls_index structure within the GOT. The two elements required for the tls_index structure are maintained in consecutive GOT entries, GOT[n] and GOT[n+1]. These GOT entries are associated to the R_AMD64_DTPMOD64 and R_AMD64_DTPOFF64 relocations.

The instruction at address 0x00 computes the address of the first GOT entry. This computation adds the PC relative address of the beginning of the GOT, which is known at link-edit time, to the current instruction pointer. The result is passed using the %rdi register to the __tls_get_addr function.

Note - The leaq instruction computes the address of the first GOT entry. This computation is carried out by adding the PC-relative address of the GOT, which was determined at link-edit time, to the current instruction pointer. The .byte, .word, and .rex64 prefixes insure that the whole instruction sequence occupies 16 bytes. Prefixes are employed, as prefixes have no negative inpact on the code.

---

x64: Local Dynamic (LD)

This code sequence implements the LD model described in "Thread-Local Storage Access Models" on page 433.

TABLE 14-19 x64: Local Dynamic Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 leaq x1@tlsld(%rip), %rdi 0x07 call __tls_get_addr@plt	R_AMD64_TLSLD R_AMD64_PLT32	x1 __tls_get_addr
# %rax - contains address of TLS block		
0x10 leaq x1@dtpoff(%rax), %rcx	R_AMD64_DTOFF32	x1
# %rcx - contains address of TLS variable x1		
0x20 leaq x2@dtpoff(%rax), %r9	R_AMD64_DTOFF32	x2
# %r9 - contains address of TLS variable x2		
	Outstanding Relocations	**Symbol**
GOT[n]	R_AMD64_DTMOD64	x1

The first two instructions are equivalent to the code sequence used for the general dynamic model, although without any padding. The two instructions must be consecutive. The x1@tlsld(%rip) sequence generates a the tls_index entry for symbol x1. This index refers to the current module that contains x1 with an offset of zero. The link-editor creates one relocation for the object, R_AMD64_DTMOD64.

The R_AMD64_DTOFF32 relocation is unnecessary, because offsets are loaded separately. The x1@dtpoff expression is used to access the offset of the symbol x1. Using the instruction as address 0x10, the complete offset is loaded and added to the result of the __tls_get_addr call

in %rax to produce the result in %rcx. The x1@dtpoff expression creates the R_AMD64_DTPOFF32 relocation.

Instead of computing the address of the variable, the value of the variable can be loaded using the following instruction. This instruction creates the same relocation as the original leaq instruction.

```
movq  x1@dtpoff(%rax), %r11
```

Provided the base address of a TLS block is maintained within a register, loading, storing or computing the address of a protected thread-local variable requires one instruction.

Benefits exist in using the local dynamic model over the general dynamic model. Every additional thread-local variable access only requires three new instructions. In addition, no additional GOT entries, or runtime relocations are required.

x64: Initial Executable (IE)

This code sequence implements the IE model described in “Thread-Local Storage Access Models” on page 433.

TABLE 14-20 x64: Initial Executable, Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 movq %fs:0, %rax 0x09 addq x@gottpoff(%rip), %rax # %rax - contains address of TLS variable	<none> R_AMD64_GOTTPOFF	 x
	Outstanding Relocations	**Symbol**
GOT[n]	R_AMD64_TPOFF64	x

The R_AMD64_GOTTPOFF relocation for the symbol x requests the link-editor to generate a GOT entry and an associated R_AMD64_TPOFF64 relocation. The offset of the GOT entry relative to the end of the x@gottpoff(%rip) instruction, is then used by the instruction. The R_AMD64_TPOFF64 relocation uses the value of the symbol x that is determined from the presently loaded modules. The offset is written in the GOT entry and is later loaded by the addq instruction.

To load the contents of x, rather than the address of x, the following sequence is available.

TABLE 14-21 x64: Initial Executable, Thread-Local Variable Access Codes II

Code Sequence	Initial Relocations	Symbol
0x00 movq x@gottpoff(%rip), %rax 0x07 movq %fs:(%rax), %rax	R_AMD64_GOTTPOFF <none>	x

# %rax - contains contents of TLS variable		
	Outstanding Relocations	**Symbol**
GOT[n]	R_AMD64_TPOFF64	x

x64: Local Executable (LE)

This code sequence implements the LE model described in "Thread-Local Storage Access Models" on page 433.

TABLE 14-22 x64: Local Executable Thread-Local Variable Access Codes

Code Sequence	Initial Relocations	Symbol
0x00 movq %fs:0, %rax 0x09 leaq x@tpoff(%rax), %rax # %rax - contains address of TLS variable	<none> R_AMD64_TPOFF32	x

To load the contents of a TLS variable instead of the address of a TLS variable, the following sequence can be used.

TABLE 14-23 x64: Local Executable Thread-Local Variable Access Codes II

Code Sequence	Initial Relocations	Symbol
0x00 movq %fs:0, %rax 0x09 movq x@tpoff(%rax), %rax # %rax - contains contents of TLS variable	<none> R_AMD64_TPOFF32	x

The following sequence is even shorter.

TABLE 14-24 x64: Local Executable Thread-Local Variable Access Codes III

Code Sequence	Initial Relocations	Symbol
0x00 movq %fs:x@tpoff, %rax # %rax - contains contents of TLS variable	R_AMD64_TPOFF32	x

x64: Thread-Local Storage Relocation Types

The TLS relocations that are listed in the following table are defined for x64. Descriptions in the table use the following notation.

`@tlsgd(%rip)`

Allocates two contiguous entries in the `GOT` to hold a `TLS_index` structure. This structure is passed to `__tls_get_addr`. This instruction can only be used in the exact general dynamic code sequence.

`@tlsld(%rip)`

Allocates two contiguous entries in the `GOT` to hold a `TLS_index` structure. This structure is passed to `__tls_get_addr`. At runtime, the `ti_offset` offset field of the object is set to zero, and the `ti_module` offset is initialized. A call to the `__tls_get_addr` function returns the starting offset if the dynamic TLS block. This instruction can be used in the exact code sequence.

`@dtpoff`

Calculates the offset of the variable relative to the start of the TLS block which contains the variable. The computed 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 a TLS symbol.

`@gottpoff(%rip)`

Allocates a entry in the `GOT`, to hold a variable offset in the initial TLS block. This offset is relative to the TLS blocks end, `%fs:0`. The operator can only be used with a `movq` or `addq` instruction.

`@tpoff(x)`

Calculates the offset of a variable relative to the TLS block end, `%fs:0`. No `GOT` entry is created.

TABLE 14-25 x64: Thread-Local Storage Relocation Types

Name	Value	Field	Calculation
R_AMD64_DPTMOD64	16	Word64	@dtpmod(s)
R_AMD64_DTPOFF64	17	Word64	@dtpoff(s)
R_AMD64_TPOFF64	18	Word64	@tpoff(s)
R_AMD64_TLSGD	19	Word32	@tlsgd(s)
R_AMD64_TLSLD	20	Word32	@tlsld(s)
R_AMD64_DTPOFF32	21	Word32	@dtpoff(s)
R_AMD64_GOTTPOFF	22	Word32	@gottpoff(s)
R_AMD64_TPOFF32	23	Word32	@gottpoff(s)

PART V

Appendices

♦ ♦ ♦ APPENDIX A

Linker and Libraries Updates and New Features

This appendix provides an overview of the updates and new features that have been added to releases of the Oracle Solaris OS.

Oracle Solaris 11.2 Release

- The link-editor can decompress and compress debug sections. See "Compressed Debug Sections" on page 86 and "Section Compression" on page 330.
- Improved synchronization between runtime auditors and process initialization is provided with the `la_callinit` and `la_callentry` functions. See "Audit Interface Functions" on page 275 and "Audit Interface Control Flow" on page 281.
- The `-z relax` option can be used to relax items of the link-editor's default validity checking. This option allows the creation of an output object that would otherwise be rejected. The `-z relax` option supersedes the `-t` and `-z relaxreloc` options. See `ld`(1)).
- Greater flexibility in option processing is provided with the link-editor `-z type` option, and the additional `LD_UNSET`, `LD_{object-type}_OPTIONS` and `LD_{object-type}_UNSET` environment variables. See "Specifying the Link-Editor Options" on page 25.
- The new `-F` option to `elfdump`(1) provides output format options.
- Stub objects can omit symbols found in the associated real object. This technique prevents the use of those symbols in new code development that link against the stub object, while maintaining the symbols in the real object for backward compatibility. See "Using Stub Objects to Hide Obsolete Interfaces" on page 80.

Oracle Solaris 11.1 Release

- Ancillary objects allow debug sections that are not required at runtime to be written to a separate object file. See "Ancillary Objects" on page 81.
- Parent Objects simplify the construction of *plugin* objects, by allowing a *plugin* to link directly against its parent. See "Parent Objects" on page 89.

- `ld`(1) provides the `-z aslr` option to provide per-object control of Address Space Layout and Randomization. `elfedit`(1) has been modified to allow simplified editing of the associated `DT_SUNW_ASLR` dynamic section entry. See Table 13-12.

Oracle Solaris 11

- Archive libraries and their members can be examined more fully with the new utility `elffile`(1).
- 64-bit processes can be restricted to the lower 32-bit address space by encoding a software capabilities attribute. See "Software Capability Address Space Restriction Processing" on page 64.

Oracle Solaris 10 1/13 Release

- Greater flexibility in discarding unused material from a link-edit is provided with the link-editor `-z discard-unused` option. See "Removing Unused Material" on page 181.
- Greater flexibility in stripping nonessential sections from an object is provided with the link-editor `-z strip-class` option. The `-z strip-class` option supersedes the older `-s` option, and provides finer grained control over the sections to be stripped.

Oracle Solaris 10 8/11 Release

- The link-editor can create stub objects. Stub objects are shared objects, built entirely from `mapfiles`, that supply the same linking interface as the real object while containing no code or data. Stub objects can be built very quickly by the link-editor, and can be used to increase build *parallelism* and to reduce build complexity. See "Stub Objects" on page 77.
- The link-editor can provide guidance in creating high quality objects using the `-z guidance` option. See `ld`(1).
- Archive processing now allows the creation of archives greater than 4 Gbytes in size.
- Local auditors can now receive `la_preinit` and `la_activity` events. See "Runtime Linker Auditing Interface" on page 271.
- A more robust model for testing for the existence of functionality is provided with deferred dependencies. See "Testing for Functionality" on page 127 and "Providing an Alternative to `dlopen`" on page 108.
- A new *mapfile* syntax is provided. See Chapter 8, "*Mapfiles*". This syntax provides a more human readable, and extensible language than the original System V Release 4 language.

Full support for processing original *mapfiles* is maintained within the link-editor. See Appendix B, *"System V Release 4 (Version 1) Mapfiles"* for the original *mapfile* syntax and use.

- Individual symbols can be associated with capability requirements. See "Identifying Capability Requirements" on page 56. This functionality provides for the creation of a family of optimized functions within a dynamic object. See "Creating a Family of Symbol Capabilities Functions" on page 65, and "Capabilities Section" on page 342.
- Objects that are created with the link-editor, and contain Oracle Solaris specific ELF data, are tagged with `ELFOSABI_SOLARIS` in the `e_ident[EI_OSABI]` ELF header. Historically, `ELFOSABI_NONE` has been used for all objects. This change is primarily of informational value, as the runtime linker continues to consider `ELFOSABI_NONE` and `ELFOSABI_SOLARIS` to be equivalent. However, `elfdump`(1), and similar diagnostic tools, can use this ABI information to produce more accurate information for a given object.
- `elfdump`(1) has been extended to use the value of `e_ident[EI_OSABI]` ELF header, or the new `-O` option, to identify ELF data types and values that are specific to a given ABI, and to use this information to provide a more accurate display of the object contents. The ability to display ABI-specific information in objects from the `Linux` operating system has been greatly expanded.
- The segment mapping information for an object that is loaded with a process can be obtained using the `dlinfo`(3C) flags `RTLD_DI_MMAPCNT` and `RTLD_DI_MMAPS`.
- The link-editor recognizes a number of GNU link-editor options. See `ld`(1).
- The link-editor provides cross linking for SPARC and x86 targets. See "Cross Link-Editing" on page 25.
- The link-editor now provides for merging `SHF_MERGE | SHF_STRING` string sections. See "Section Merging" on page 329.
- The merging of relocation sections when creating executables and shared objects is now the default behavior. See "Combined Relocation Sections" on page 188. This behavior used to require the link-editor's `-z combreloc` option. The `-z nocombreloc` is provided to disable this default behavior, and preserve the one-to-one relationship with the sections to which the relocations must be applied.
- ELF objects can be edited with the new utility `elfedit`(1).
- Arbitrary data files can be encapsulated within ELF relocatable objects using the new utility `elfwrap`(1).
- Additional symbol visibility attributes are provided. See the exported, singleton and eliminate attribute descriptions under "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217 and Table 12-23.
- The link-editor, and associated ELF utilities have been moved from `/usr/ccs/bin` to `/usr/bin`. See "Invoking the Link-Editor" on page 24.
- Symbol sort sections have been added, that allow for simplified correlation of memory addresses to symbolic names. See "Symbol Sort Sections" on page 374.

- The symbol table information that is available with dynamic objects has been extended with the addition of a new `.SUNW_ldynsym` section. See "Symbol Table Section" on page 365 and Table 12-5.
- The format of configuration files that are managed with `crle`(1) has been enhanced for better file identification. The improved format ensures that the runtime linker does not use a configuration file generated on an incompatible platform.
- New relocation types have been added that use the size of the associated symbol in the relocation calculation. See "Relocations" on page 353.
- The `-z rescan-now`, `-z recan-start`, and `-z rescan-end` options provide additional flexibility in specifying archive libraries to a link-edit. See "Position of an Archive on the Command Line" on page 31.

Obsolete Feature

The following items have been made obsolete. These items provided internal, or seldom used features. Any existing use of the associated ELF definitions is ignored, however the definitions can still be displayed by tools such as `elfdump`(1).

`DT_FEATURE_1`

This dynamic section tag identified runtime feature requirements. See "Dynamic Section" on page 398. This tag provided the feature flags `DTF_1_PARINIT` and `DTF_1_CONVEXP`. The `DT_FEATURE_1` tag and the associated flags are no longer created by the link-editor, or processed by the runtime linker.

Solaris 10 5/08 Release

- Global auditing can now be enabled by recording an auditor within an application together with the link-editor `-z globalaudit` option. See "Recording Global Auditors" on page 274.
- Additional link-editor support interfaces, `ld_open` and `ld_open64` have been added. See "Support Interface Functions" on page 265.

Solaris 10 8/07 Release

- Greater flexibility in executing an alternative link-editor is provided with the link-editor `-z altexec64` option, and the `LD_ALTEXEC` environment variable.
- Symbol definitions that are generated using *mapfiles* can now be associated to ELF sections. See "SYMBOL_SCOPE / SYMBOL_VERSION Directives" on page 217.

- The link-editor and runtime linker provide for the creation of static TLS within shared objects. In addition, a backup TLS reservation is established to provide for limited use of static TLS within post-startup shared objects. See "Program Startup" on page 430.

Solaris 10 1/06 Release

- Support for the x64 medium code model is provided. See Table 12-4, Table 12-8, and Table 12-12.
- The command line arguments, environment variables, and auxiliary vector array of the process, can be obtained using the `dlinfo`(3C) flag `RTLD_DI_ARGSINFO`.
- Greater flexibility in prohibiting direct binding from external references is provided with the link-editor `-B nodirect` option. See Chapter 6, "Direct Bindings".

Solaris 10 Release

- x64 is now supported. See Table 12-5, "Special Sections" on page 332, "Relocation Types" on page 362, "Thread-Local Variable Access" on page 448, and "Thread-Local Storage Relocation Types" on page 451.
- 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 32, "Directories Searched by the Runtime Linker" on page 96, and "Security" on page 115.
- System archive libraries are no longer provided. Therefore, the creation of a statically linked executable is no longer possible. See "Static Executables" on page 18.
- Greater flexibility for defining alternative dependencies is provided with the `-A` option of `crle`(1).
- The link-editor and runtime linker process environment variables specified without a value. See "Environment Variables" on page 21.
- Path names used with `dlopen`(3C), and as explicit dependency definitions, can now use any reserved tokens. See Chapter 10, "Establishing Dependencies with Dynamic String Tokens". The evaluation of path names 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 108.

♦ ♦ ♦ APPENDIX B

System V Release 4 (Version 1) Mapfiles

Note - This appendix describes the original System V Release 4 *mapfile* language (version 1). Although this *mapfile* syntax remains supported, the version 2 *mapfile* language described in Chapter 8, "*Mapfiles*" is recommended for new applications.

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 the following 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 B-1 *Mapfile* Segment Attributes

Attribute	Value
`segment_type`	`LOAD | NOTE | NULL | STACK`

Attribute	Value
`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*

Four built-in segments exist with the following default attribute values.

- `text` – `LOAD`, `?RX`, no `virtual_address`, `physical_address`, or `length` specified. `alignment` values are set to defaults per CPU type.
- `data` – `LOAD`, `?RWX`, no `virtual_address`, `physical_address`, or `length` specified. `alignment` values are set to defaults per CPU type.
- `bss` – disabled, `LOAD`, `?RWX`, no `virtual_address`, `physical_address`, or `length` specified. `alignment` values are 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 12-5 for a full description of `SHT_NOBITS` sections. The simplest `bss` declaration is sufficient to enable the creation of a `bss` segment.

```
bss =;
```

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 470.

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`, `NULL` or `STACK`. If unspecified, the segment type 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 the alignment is specified. Other segments still have the default alignment unless their `alignment` values 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 the output file is created.
- If an `alignment` value is not specified for a segment, the alignment 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 is rounded to the next address that conforms to the value that is given. This value only effects the segments that the value 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 the segment. This segment can be a `LOAD` segment or a `NULL` segment. Empty `LOAD` segments can only be specified for executables. These segments must have a specified size

and alignment. These segments result in the creation of memory reservations at process startup. Empty `NULL` segments provide for adding program header entries that can be used by post-processing utilities. These segments should have no additional attributes specified. Multiple definitions for `LOAD` segments and `NULL` segments are permitted.

The `?N` flag enables you to 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 as follows.

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

The syntax allows for the following entry.

```
text: .text%bar: a.o b.o;
```

However, 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 B-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 previously. The `section_types` listed previously are built-in types. For more information on `section_types`, see "Sections" on page 311.
- 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 determines the `basename`(1) of the file from the command line. This base name is used 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 468.
- 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 as follows.

```
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 B-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.

```
1.  text = LOAD ?RXN V0xf0004000;
2.  text | .text;
3.  text | .rodata;
4.  text : $PROGBITS ?A!W;
5.  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.

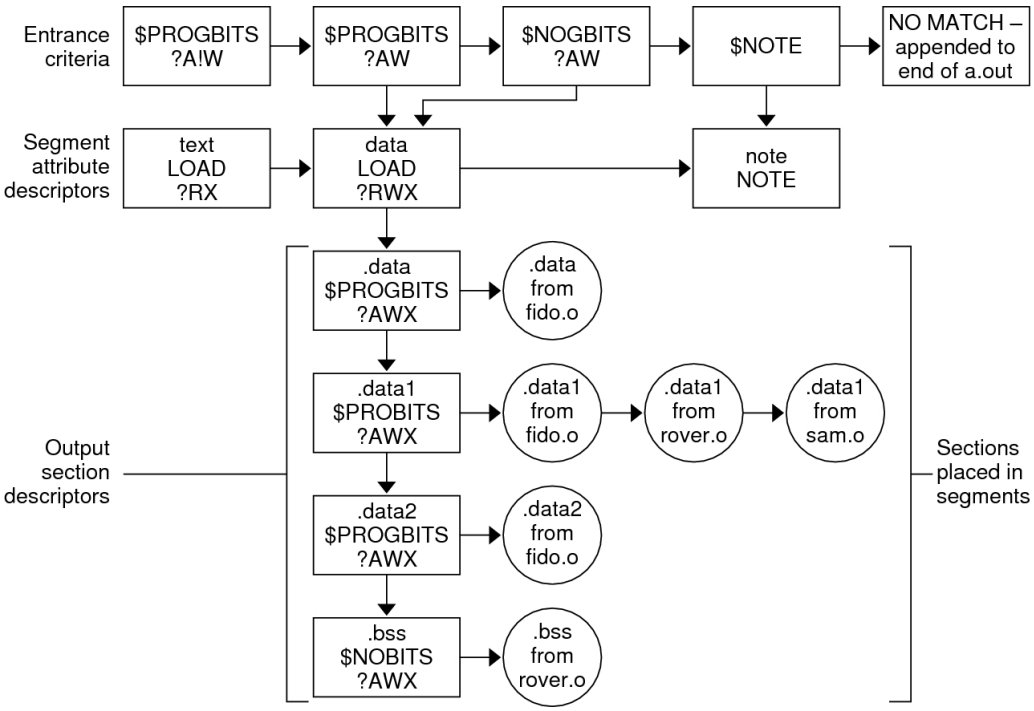
As each mapping directive in a `mapfile` is read in, the directive is added after any other mapping directives that you already specified for the same segment but before the default mapping directives for that segment.

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 B-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 B-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 B-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 `$PROGBITS`, and a `section_flags` value of `?AWX` falls into the second Entrance Criteria box in Figure B-1, placing it in the `data` segment. The section matches the second Output Section Descriptor box exactly (`.data1`, `$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 B-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.

Index

D

E

M

N

O

P

R

S

T

U

V

W