Linker and Libraries

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Contents

Figures

Tables

Preface

The SunOS™ operating system provides an environment in which application developers can build applications and libraries using the link-editor `ld(1)`, and execute these utilities with the aid of the runtime linker `ld.so.1`. For many application developers, the fact that the link-editor is called via the compilation system, and that the runtime linker may play a part in the execution of their application, is mildly interesting. This manual is for those who wish to understand more fully the concepts involved.

About This Manual

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

Intended Audience

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

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

Not many programmers should find it necessary to read this manual from cover to cover.

Organization

Chapter 1, “Introduction”, gives an overview of the linking processes under the SunOS operating system. This chapter is intended for all programmers.

Chapter 2, “Link-Editor”, describes the functions of the link-editor, its two modes of linking (*static* and *dynamic*), scope and forms of input, and forms of output. This chapter is intended for all programmers.

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

Chapter 4, “Shared Objects”, gives definitions of shared objects, describes their mechanisms, and explains how to build and use them. This chapter is intended for all programmers.

Chapter 5, “Object Files”, is a reference chapter on ELF files. This chapter is intended for advanced programmers.

Chapter 6, “Mapfile Option”, describes the mapfile directives to the linker, which specify the layout of the output file. This chapter is intended for advanced programmers.

Appendix A, “Link-Editor Quick Reference”, gives an overview of the most commonly used link-editor options, and is intended for all programmers.

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

Introduction 1

This manual describes the operations of the SunOS operating system link-editor and runtime linker, together with the objects on which they operate. The basic operation of the SunOS linkers involves the combination of objects and the connection of symbolic references from one object to the symbolic definitions within another. This operation is often referred to as *binding*.

The main areas this manual expands upon are:

- The *Link-Editor*

  The link-editor, `ld(1)`, is responsible for concatenating one or more input files (either relocatable objects, shared objects, or archive libraries) to produce one output file (either a relocatable object, an executable application, or a shared object). The link-editor is most commonly invoked as part of the compilation environment (see `cc(1)`).

- The *Runtime Linker*

  The runtime linker, `ld.so.1`[1], is responsible for processing dynamic executables and shared objects at runtime, and binding them to create a runable process.

1. ld.so.1 is a special case of a shared object and thus allows itself to be versioned. Here we use a version number of 1, however later releases of the SunOS operating system may provide higher version numbers.

- *Shared Objects* (sometimes also referred to as *Shared Libraries*)

  Shared objects are one form of output from the link-edit phase. However, their importance in creating a powerful, flexible runtime environment warrants a section of its own.

- *Object Files*

  The SunOS linkers work with files that conform to the *executable and linking format* (ELF).

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

Link-Editing

Link-editing takes a variety of input modules, from `cc(1)`, `as(1)` or `ld(1)`, and performs concatenation and interpretation of the data within these input modules to form a single output module. Although the link-editor provides numerous options, the output module produced is one of four basic kinds shown in Figure 1-1 on page 3.

- *Relocatable object* – a concatenation of input relocatable objects, which may be used in subsequent link-edit phases.
- *Static executable* –a concatenation of input relocatable objects that has all symbolic references bound to the executable, and thus represents a ready to run process.
- *Dynamic executable* – a concatenation of input relocatable objects that requires intervention by the run-time linker to produce a runable process. Its symbolic references may still need to be bound at run-time, and it may have one or more dependencies in the form of shared objects.
- *Shared object* – a concatenation of input relocatable objects that provides services that may be bound to a dynamic executable at runtime. The shared object may also have dependencies on other shared objects.

Of the above four types of link-editor output, the last two, *dynamic executables* and *shared objects,* are the main focus of this document.

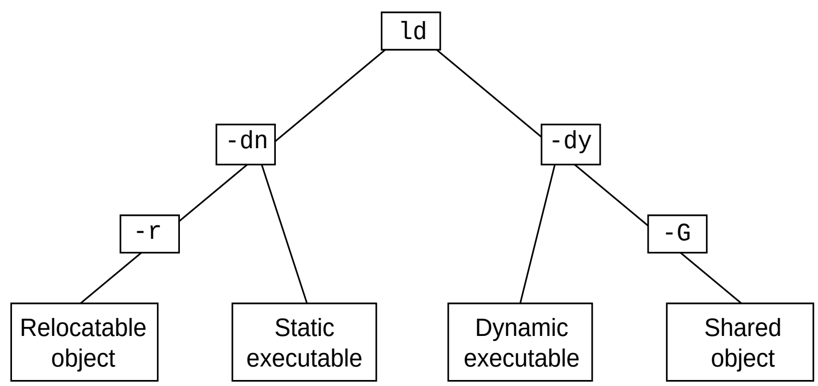

Figure 1-1 Static or Dynamic Link-editing

Runtime Linking

Runtime linking involves the binding of objects, normally generated from one or more previous link-edits, to generate a runable process. During the generation of these objects by the link-editor, the binding requirements are verified and appropriate bookkeeping information is added to each object to allow the runtime linker to map, relocate, and complete the binding process. During the execution of the process, the facilities of the runtime linker are also made available and may be used to extend the process' address space by adding additional shared libraries on demand. The two most common components involved in runtime linking are *dynamic executables* and *shared objects*.

Dynamic Executables

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

Shared Objects

Shared objects provide the key building block to a dynamically linked system. Basically, a shared object is similar to a dynamic executable, however shared objects normally have no entry point and they have not yet been assigned a virtual address. Dynamic executables normally have dependencies on one or more shared objects. That is, the shared object(s) must be bound to the dynamic executable to produce a runable process. Because shared objects may be used by many applications, many of the aspects of their construction directly affect shareability, versioning and performance.

It is useful to distinguish the processing of shared objects by either the link-editor or the runtime linker by referring to the *environments* in which the shared objects are being used:

- The *compilation* environment. Here, 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.
- The *runtime* environment. Here, shared objects are processed by the runtime linker, together with a dynamic executable, to produce a runable process.

Related Topics

Dynamic Linking

Dynamic linking is a term often used to embrace those portions of the link-editing process that generate dynamic executables and shared objects, together with the runtime linking of these objects to generate a runable process. Dynamic linking allows multiple applications to use the code provided by a shared object by enabling the application to bind 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, and is a crucial factor in providing an *application binary interface* (ABI). Dynamic linking is the preferred compilation method for SunOS applications.

Application Binary Interfaces

To enable the asynchronous evolution of system and application components, binary interfaces between these facilities are defined. The SunOS linkers operate upon these interfaces to assemble applications for execution. Although all components handled by the SunOS linkers have binary interfaces, one family of such interfaces of particular interest to applications writers is the *System V Application Binary Interface*.

The *System V Application Binary Interface*, or *ABI*, defines a system interface for compiled application programs. Its purpose is to document a standard binary interface for application programs on systems that implement the *System V Interface Definition, Third Edition*. The SunOS operating system provides for the generation and execution of ABI-conformant applications. On SPARC systems, the ABI is contained as a subset of the *SPARC® Compliance Definition* (SCD).

Many of the topics covered in the following chapters are influenced by the ABI. For more detailed information refer to the appropriate ABI manuals.

Support Tools

Together with the objects mentioned in the previous sections come a number of support tools and libraries. These tools provide for the analysis and inspection of these objects and the linking processes. Among these tools are: `nm(1)`, `dump(1)`, `ldd(1)`, `elf(3E)`, and a linker debugging support library. Throughout this document we augment any discussions with examples of the use of these tools.

Link-Editor 2

Overview

The link-editing process consists of building an output file from one or more input files. The building of the output file is directed by the options supplied to the link-editor together with 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 refer to Chapter 5, “Object Files”, however, for this introduction it is first necessary to introduce two ELF structures, *sections* and *segments*. Sections represent the smallest indivisible units that may be processed within an ELF file. Segments are a collection of sections that represent the smallest individual units that may be mapped to a memory image by `exec(2)` or by the runtime linker.

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

- Sections that contain *program data*, whose interpretation is only meaningful to the application itself (examples of these include the program instructions *.text*, and the associated data *.data* and *.bss*).
- Sections that contain *link-editing information* (examples of these include 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 and may result in modifications to other sections, or the generation of new output information sections for use in later processing of the output file.

The following is a simple breakdown of the link-editors functionality, and introduces the topics covered in this chapter:

- It verifies and checks for consistency all the options passed to it.
- It concatenates sections of the same characteristics (for example, type, attributes and name) from the input relocatable objects to form new sections within the output file. These concatenated sections may in turn be associated to output segments.
- It reads symbol table information from both relocatable objects and shared objects to verify and unite references with definitions, and normally generates a new symbol table, or tables, within the output file.
- It reads relocation information from the input relocatable objects and applies this information to the output file by updating other input sections. In addition, output relocation sections may be generated for use by the runtime linker.
- It generates *program headers* that describe any segments created.
- It generates a dynamic linking information section if necessary, which provides information such as shared library dependencies to the runtime linker.

The process of concatenating like *sections*, together with the association of *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 normally sufficient for most users, however, the defaults may be manipulated using the `-M` option with an associated `mapfile` (refer to Chapter 6, “Mapfile Option” for more details).

Invoking the Link-Editor

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

Direct Invocation

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

```
$ ld test.o
```

the link-editor tries to build a dynamic executable named `a.out` using *only* the input file `test.o`. For the `a.out` to be a useful executable, it should include start-up and exit processing code. This code may be language or operating system specific, and is normally provided through files supplied by the compiler drivers. Additionally, you may also supply your own initialization and termination code. This code must be encapsulated and labelled correctly for it to be correctly recognized and made available to the runtime linker. This encapsulation and labelling is also provided through files supplied by the compiler drivers.

In practice, there is little reason to invoke the link-editor directly.

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)`, `f77(1)`, etc., with the input files that make up your application, and the compiler driver will add additional files and default libraries to complete the link-edit. These additional files may be seen by expanding the compilation invocation, for example:

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

Note – This is an example; the actual files included by your compiler driver and the mechanism used to display the link-editor invocation may vary.

Specifying the Link-Editor Options

Most options to the link-editor can be passed via the compiler driver command-line. For the most part there is no conflict between the compiler and the link-editor options. In cases where a conflict arises, the compiler drivers normally provide a command-line syntax that allows specific options to be passed to the link-editor. However, an alternative mechanism to provide options to the link-editor is to set the `LD_OPTIONS` environment variable. For example:

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

Here the `-R` and `-L` options will be interpreted by the link-editor and *prepended* to any command-line options received from the compiler driver.

The link-editor parses the entire option list looking for any invalid options or any options with invalid associated arguments. If either of these cases are found, a suitable error message is generated, and if the error is deemed fatal the link-edit terminates. For example:

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

Here the illegal option `-X` is identified, and the illegal argument to the `-z` option is caught by the link-editor's checking. If an option requiring an associated argument is mistakenly specified twice the link-editor will provide a suitable warning but will continue with the link-edit. For example:

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

The link-editor also checks the option list for any fatal inconsistences. For example:

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

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

Refer to Appendix A, "Link-Editor Quick Reference" for the most commonly used link-editor options, and to the `ld(1)` manual page for a complete description of all link-editor options.

Input File Processing

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

Under *static* linking the link-editor will only accept relocatable objects or archive libraries as input files. Under *dynamic* linking the link-editor will also accept 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, but will not become part of the output file image, as new sections will be generated to take their place. Symbols are gathered into a special internal symbol table that allows for their verification and resolution, and eventually the creation of one or more symbol tables in the output image.

Although any input file can be specified directly on the link-edit command-line, archive libraries and shared objects are commonly specified using the `-l` option (refer to the section "Linking with Additional Libraries" on page 14 for coverage of the use of this mechanism and how it relates to the two different linking modes). However, even though shared objects are often referred to as shared *libraries*, and both of these objects may be specified using the same option, the interpretation of shared objects and archive libraries is quite different. The next two sections expand upon these differences.

Archive Processing

Archives are built using `ar(1)`, and normally 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. When the link-editor reads an archive, it uses information within the internal symbol table it is creating to select *only* the objects from the archive it requires to complete the binding process. To be more precise, the link-editor will extract a relocatable object from an archive if:

- It contains a symbol definition that satisfies a symbol reference (sometimes referred to as an *undefined* symbol) presently held in the link-editor's internal symbol table, or
- It contains a *data* symbol definition that satisfies a *tentative* symbol definition presently held in the link-editor's internal symbol table. An example of this would be that a FORTRAN COMMON block definition would result in the extraction of a relocatable object that defines the same DATA symbol.

Note – A *weak* symbol reference will not cause the extraction of an object from an archive. Weak symbols are expanded upon in section "Simple Resolutions" on page 22.

The link-editor will make multiple passes through an archive extracting relocatable objects as needed to satisfy the symbol information being accumulated in the link-editors internal symbol table. Once the link-editor has made a complete pass through the archive without extracting any relocatable objects, it will move on to process the next input file. This mechanism of only extracting from the archive the relocatable objects needed *at the time* the archive was encountered means that the position of the archive within the input file list may be significant (refer to section "Position of an Archive on the Command-Line" on page 16 for more details).

Note – Although the link-editor will make multiple passes through an archive to resolve symbols, this mechanism may be quite costly for large archives containing random organizations of relocatable objects. In these cases it is recommended that tools like `lorder(1)` and `tsort(1)` be used to order the relocatable objects within the archive and thus reduce the number of passes the link-editor must carry out.

Shared Object Processing

Shared objects are indivisible, whole units that have been generated via 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. The shared object is not copied physically during the link-edit as its actual inclusion is deferred until process execution. 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, as these will be interpreted by the runtime linker when the shared object is bound to generate a runable process. However, the occurrence of a shared object will be remembered, and information will be stored in the output file image to indicate that this object is a dependency and must be made available at runtime.

If a shared object has dependencies on other shared objects, these too will be processed. This processing will occur *after* all command-line input files have been processed. These shared objects will be used to complete the symbol resolution process, however their names *will not* be recorded as dependencies in the output file image being generated.

Although the position of a shared object on the link-edit command-line has less significance than it does for archive processing, it may 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 (refer to the section "Symbol Resolution" on page 21 for more details). The *order* of shared objects processed by the link-editor is maintained in the dependency information stored in the output file image. As the runtime linker reads this information it will load the specified shared objects in the same order. Therefore, the link-editor and the runtime linker will select the first occurrence of a symbol of a multiply defined series of symbols.

Note – Multiple symbol definitions, and thus the information to describe the interposing of one definition of a symbol for another, are reported in the load map output generated using the `-m` option.

Linking with Additional Libraries

Although the compiler drivers will often insure that appropriate libraries are specified to the link-editor, it is frequently necessary for developers to supply their 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 normally designated by the prefix `lib` and the suffix `.so`, and archives are designated by the prefix `lib` and the suffix `.a`. For example, `libc.so` is the shared object version of the standard C library made available to the compilation environment, and `libc.a` is its archive version. These conventions are recognized by the `-l` option of the link-editor. Developers commonly use this option to supply additional libraries to their link-edit, for example

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

directs the link-editor to search for `libfoo.so`, and if it does not find it, to search for `libfoo.a`.

Note – There is a naming convention 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. Refer to section "Naming Conventions" on page 68, and "Versioning" on page 73 for more details.

When link-editing in dynamic mode, you may choose to link with a mix of shared objects and archives. When link-editing in static mode, only archive libraries are acceptable for input. When in dynamic mode and using the `-l` option to enable a library search, the link-editor will first search in a given directory for a shared object that matches the specified name. If no match is found the link-editor will then look for an archive library in the same directory. When in static mode and using the `-l` option, only archive libraries will be sought.

Linking with a Mix of Shared Objects and Archives

Although the library search mechanism, in dynamic mode, searches a given directory for a shared object, and then an archive library, finer control of the type of search required can be achieved using the `-B` option. By specifying the `-Bdynamic` and `-Bstatic` options on the command-line, as many times as required, the library search can be toggled 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.o -Bstatic -lfoo -Bdynamic -lbar
```

The `-Bstatic` and `-Bdynamic` keywords are not exactly symmetrical. When you specify `-Bstatic`, the link-editor does *not* accept shared objects as input until the next occurrence of `-Bdynamic`. However, when you specify `-Bdynamic`, the link-editor will first look for shared objects and then archives in any given directory.

Thus in the previous example it would be more precise to say that the link-editor will first search for `libfoo.a`. It will then search for `libbar.so`, and if that fails, for `libbar.a`. Finally, it will search for `libc.so`, and if that fails, `libc.a`.

Another example of using these options is in the creation of an ABI-conforming application. For example:

```
$ cc -o prog main.o file.1.o -lsys -Bstatic
```

Here all the basic system routines defined in `libsys.so` will be bound to this shared object. Because the compiler driver appends a `-lc` to the options supplied to the link-editor, and because the `-Bstatic` has instructed the link-editor to search for archive libraries only, any remaining undefined symbols will be resolved by extracting the appropriate relocatable objects from `libc.a`.

Position of an Archive on the Command-Line

The position of an archive on the command-line may affect the output file being produced. The link-editor searches an archive only to resolve undefined or tentative external references it has previously seen. Once this search is completed and the required relocatable objects have been extracted, the archive will not be available to resolve any new symbols obtained from the input files that follow the archive on the command-line. For example, the command

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

directs the link-editor to search `libfoo.a` only to resolved symbol references that have been obtained from `file1.c; libfoo.a` will not be available to resolve symbol references from `file2.c` or `file3.c`.

Note – As a rule, it is best to 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 assumed that the link-editor knows where to search for the libraries listed on the command-line. By default the link-editor knows of only two standard places to look for libraries, `/usr/ccs/lib` and `/usr/lib`. All other directories to be searched must be added to the link-editor's search path explicitly.

There are two ways to change the link-editor search path: using a command-line option, or using an environment variable.

Using a Command-Line Option

The `-L` option can be used to *add* a new pathname to the library search path. This option affects the search path *at the point* it is encountered on the command-line. For example, the command

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

searches `path1` (then `/usr/ccs/lib` and `/usr/lib`) to find `libfoo`, but searches `path1` and then `path2` (and then `/usr/ccs/lib` and `/usr/lib`) to find `libbar`.

Pathnames defined using the `-L` option are used only by the link-editor. They are not recorded in the output file image created 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.

The `-Y` option can be used 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 command

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

searches for `libfoo` only in the directories `/opt/COMPILER/lib` and `/home/me/lib`. The directories specified using the `-Y` option can be supplemented by using the `-L` option.

Using an Environment Variable

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

Here is 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.o -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

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

If no semicolon were specified as part of the LD_LIBRARY_PATH definition the specified directory list would be interpreted *after* any -L options. For example:

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

Here the effective search path will be path1:path2... pathn:dir1:dir2:/usr/ccs/lib:/usr/lib.

Note – This environment variable may also be used to augment the search path of the runtime linker (refer to “Directories Searched by the Runtime Linker” on page 40). To prevent this environment variable from influencing the link-editor the -i option can be used.

Directories Searched by the Runtime Linker

The runtime linker knows of only one standard place to look for libraries, /usr/lib. 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, these shared objects are recorded as dependencies that must be located again during process execution by the runtime linker. During the link-edit, one or more pathnames can be recorded in the output file being built for the runtime linker to use to search for any shared object dependencies. These recorded pathnames are referred to as a *runpath*.

Note – No matter how you modify the runtime linker's library search path, its last element is always /usr/lib.

The -R option, which takes a colon-separated list of directories, can be used to record a runpath in a dynamic executable or shared library. For example:

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

will record the runpath /home/me/lib:/home/you/lib in the dynamic executable prog. The runtime linker will use these paths, and then the default location /usr/lib, to locate any shared object dependencies, in this case libfoo.so.1 and libbar.so.1.

The link-editor accepts multiple -R options and will concatenate each of these specifications, separated by a colon. Thus, the above example could also be expressed as:

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

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

The *.init* and *.fini* section types provide for runtime initialization and termination processing. These section types are concatenated from the input relocatable objects like any other sections. However, the compiler drivers may also supply *.init* and *.fini* sections as part of the additional files they add to the beginning and end of the user's input-file list. These files have the effect of encapsulating the *.init* and *.fini* code into individual functions that are identified by the reserved symbol names _init and _fini respectively. When building a dynamic executable or shared object, the link-editor records these symbol addresses in the output file's image so they may be called by the runtime linker during initialization and termination processing. Refer to the "Initialization and Termination Routines" on page 50 for more details on the runtime processing of these sections.

The creation of *.init* and *.fini* sections can be carried out directly using an assembler, or some compilers may offer special primitives to simplify their declaration. For example, the following code segments result in a *call* to the function `foo` being placed in an *.init* section, and a *call* to the function `bar` being placed in a *.fini* section:

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

foo()
{
    /* Perform some initialization processing. */
    ......
}

bar()
{
    /* Perform some termination processing. */
    .......
}
```

Care should be taken when designing initialization and termination code that may be included in both a shared object and archive library. If this code is spread throughout a number of relocatable objects within an archive library, then the link-edit of an application using this archive may only extract a portion of the modules, and hence only a portion of the initialization and termination code. At runtime only this portion of code will be executed. However, the same application built against the shared object will have *all* the accumulated initialization and termination code executed at runtime when the shared object is mapped in as one of the application's dependencies.

Symbol Processing

During input file processing, all *local* symbols from the input relocatable objects are passed through to the output file image. All other symbol entries are accumulated internally to the link-editor. Each time a symbol entry is processed, the link-editor determines if a symbol with the same name has already been encountered from a previous input file. If so, a symbol resolution process is called to determine which of the two entries is to be kept.

On completion of input file processing, providing no fatal error conditions have been encountered during symbol resolution, the link-editor determines if any unbound symbol references (undefined symbols) remain that will cause the link-edit to fail.

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

Symbol Resolution

Symbol resolution runs the entire spectrum, from simple and intuitive to complex and perplexing. Resolutions may be carried out silently by the link-editor, be accompanied by warning diagnostics, or result in a fatal error condition. The resolution of two symbols depends on the symbols' attributes, the type of file providing the symbol and the type of file being generated. For a complete description of symbol attributes refer to section "Symbol Table" on page 119, however, for the following discussions it is worth identifying three basic symbol types:

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

In its simplest form, resolution involves the use of a precedence relationship that has *defined* symbols dominating *tentative* symbols, which dominate *undefined* symbols.

The following C code example shows how these symbol types may be generated (undefined symbols are prefixed with `u_`, tentative symbols are prefixed with `t_`, and defined symbols are prefixed with `d_`):

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

int             t_bar;
int             d_bar = 1;

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

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

Simple Resolutions

These symbol resolutions are by far the most common, and result when two symbols with similar characteristics are detected, and one symbol takes precedence over the other. This symbol resolution is carried out silently by the link-editor. For example, for symbols with the same binding, a reference to an *undefined* symbol from one file will be bound to, or satisfied by, a *defined* or *tentative* symbol definition from another file. Or, a *tentative* symbol definition from one file will be bound to a *defined* symbol definition from another file.

Symbols that undergo resolution may have either a global or weak *binding*. Weak bindings have less precedence than global binding, and thus symbols with different bindings are resolved according to a slight alteration of the simple rules outlined above. But first, it is worth introducing how weak symbols may be produced.

Weak symbols may be defined individually, or as aliases to global symbols using a `pragma` definition:

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

int             bar = 1;

_foo()
{
        return (bar);
}
$ cc -o main.o -c main.c
$ nm -x main.o

[Index]   Value      Size      Type  Bind  Other Shndx   Name
...............
[7]     |0x00000000|0x00000004|OBJT |WEAK |0x0  |3      |bar
[8]     |0x00000000|0x00000028|FUNC |WEAK |0x0  |2      |foo
[9]     |0x00000000|0x00000028|FUNC |GLOB |0x0  |2      |_foo
```

Notice that the weak alias `foo` is assigned the same attributes as the global symbol `_foo`. This relationship will be maintained by the link-editor and will result in the symbols being assigned the same *value* in the output image.

In symbol resolution, weak defined symbols will be silently overridden by any global definition of the same name.

Another form of simple symbol resolution occurs between relocatable objects and shared objects, or between multiple shared objects, and is termed *interposition*. In these cases, if a symbol is multiply defined, the relocatable object, or the first definition between multiple shared objects, will be 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 may be used to override the functionality provided by one shared object by a dynamic executable or another shared object.

The combination of weak symbols and interposition provides a very useful programming technique. For example, the standard C library provides a number of services that users are allowed to redefine for themselves. However, ANSI C defines a set of standard services that must be present on the system and cannot be replaced in a strictly conforming program. The function `fread(3S)`, for example, is an ANSI C library function, whereas the system function `read(2)` is not. A conforming ANSI C program must be able to redefine `read(2)`, and still use `fread(3S)` in a predictable way.

The problem here is that `read(2)` underlies the `fread(3S)` implementation in the standard C library, and thus it would seem that a program that redefines `read(2)` could confuse the `fread(3S)` implementation. To guard against this, ANSI C states that an implementation cannot use a name that is not reserved to it, and by using the `pragma` directive shown below:

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

we are able to define just such a reserved name, and from it generate an alias for the function `read(2)`. A user may quite freely define their own `read()` function without compromising the `fread(3S)` implementation, which in turn is implemented to use the `_read()` function. The link-editor will not complain of a user's redefinition of `read()`, either when linking against the shared object or archive version of the standard C library. In the former case, interposition will take its course, whereas in the latter case, the fact that the C library's definition of `read(2)` is weak allows it to be quietly overridden.

By using the `-m` option, the link-editor will list 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 will select the most appropriate symbol and will generate a warning message indicating the symbol, the attributes that conflict, and the identity of the file from which the symbol definition is taken. For example:

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

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

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

Here, two files with a definition of the data item `array` have different size requirements. A similar diagnostic would be produced if the symbols' alignment requirements differed. In both of these cases the diagnostic may be suppressed by using the link-editor's `-t` option.

Another form of attribute difference is the symbol's *type*. For example:

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

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

Here the symbol `bar` has been defined as both a data item and a function.

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

In cases like this, the relocatable object definition will be taken when the resolution occurs between a relocatable object and a shared object, or, the first definition will be taken when the resolution occurs between two shared objects. When such resolutions occur between symbols of different bindings (*weak* or *global*), a warning will also be produced.

Inconsistences between symbol types are not suppressed by the `-t` option.

Fatal Resolutions

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

The most common fatal error condition exists when two relocatable objects both define symbols of the same name, and neither symbol is a weak definition:

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

$ cat bar.c
bar()
{
        return (0);
}

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

Here `foo.c` and `bar.c` have conflicting definitions for the symbol `bar`. Since the link-editor cannot determine which should dominate, it will normally give up. However, the link-editor's `-z muldefs` option can be used to suppress this error condition, and allows the first symbol definition to be taken.

Undefined Symbols

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

Generating an Executable

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

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

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

In a similar manner, a symbol reference within a shared object that is never matched to a symbol definition when the shared object is being used to build a dynamic executable, will also result in an undefined symbol:

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

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

Sometimes, developers wish to allow undefined symbols in cases like the previous example. In these cases the default fatal error condition can be suppressed by using the `-z nodefs` option.

Note – Care should be taken when using the `-z nodefs` option. If an unavailable symbol reference is required during the execution of a process, a fatal runtime relocation error will occur. Although this error may be detected during the initial execution and testing of an application, more complex execution paths may result in this error condition taking much longer to detect, which may 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  (./libfoo.so?)
ld: fatal: Symbol referencing errors. No output written to prog
```

Here `prog` is being built with an *explicit* reference to `libbar.so`, and because `libbar.so` has a dependency on `libfoo.so`, an *implicit* reference to `libfoo.so` from `prog` is established. Now, `main.c` made a specific reference to the interface provided by `libfoo.so`. This means that `prog` really has a dependency on `libfoo.so`. However, because *only* explicit shared object dependencies are recorded in the output file being generated, `prog` would fail to run should a new version of `libbar.so` be developed that no longer has a dependency on `libfoo.so`. For this reason, bindings of this type are deemed fatal, and the implicit reference should be made explicit by referencing the library directly during the link-edit of `prog` (the required reference is hinted at as “`(./libfoo.so?)`” in the fatal error message shown in this example).

Generating a Shared Object

When the link-editor is generating a shared object, it will by default allow undefined symbols to remain at the end of the link-edit. This allows the shared object to import symbols from either relocatable objects or other shared objects when it is used to build a dynamic executable. The `-z defs` option can be used to force a fatal error should any undefined symbols remain.

Weak Symbols

Weak symbol references that are not bound during a link-edit will not result in a fatal error condition, no matter what output file type is being generated. If a static executable is being generated, the symbol will be converted to an absolute symbol and assigned a value of zero. If a dynamic executable or shared object is being produced, the symbol will be left as an undefined weak reference. In this case, during process execution, the runtime linker will search for this symbol, and if it does not find a match, will bind the reference to an address of zero instead of generating a fatal runtime relocation error.

Within the confines of position-independent code (refer to section "Position-Independent Code" on page 85 for more information), these undefined weak referenced symbols may provide a useful mechanism for testing for the existence of functionality. For example, lets take the following C code fragment:

```
#pragma weak    foo

extern  void    foo(char *);

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

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

If, during the link-editing of an executable containing this code, a definition for the function `foo` was found (say, from binding with a shared object that defined the symbol), then during execution the function address will test nonzero, which will result in the function being called. However, if the symbol definition was not found, the executable would still have been built, but during execution the function address will test zero, and thus will not be called.

Tentative Symbol Order Within the Output File

Normally, contributions from input files appear in the output file in the order of their contribution. An exception occurs when processing tentative symbols and their associated storage. These symbols are not fully defined until their resolution is complete. If the resolution occurs as a result of encountering a *defined* symbol from a relocatable object, then the order of appearance will be that which would have occurred normally for the definition.

If it is desirable 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 have resulted in a reordering of the data items within the output file compared to the original order described in the source file `foo.c`:

```
$ cat foo.c
char A_array[0x10];
char B_array[0x20];
char C_array[0x30];
$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]    |0x00020754|0x00000010|OBJT |GLOB |0x0  |15  |A_array
[34]    |0x00020764|0x00000030|OBJT |GLOB |0x0  |15  |C_array
[42]    |0x00020794|0x00000020|OBJT |GLOB |0x0  |15  |B_array
```

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

```
$ cat foo.c
char A_array[0x10] = { 0 };
char B_array[0x20] = { 0 };
char C_array[0x30] = { 0 };
$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32]    |0x000206bc|0x00000010|OBJT |GLOB |0x0  |12  |A_array
[42]    |0x000206cc|0x00000020|OBJT |GLOB |0x0  |12  |B_array
[34]    |0x000206ec|0x00000030|OBJT |GLOB |0x0  |12  |C_array
```

Defining Additional Symbols

The `-u` option provides a mechanism to generate a symbol reference from the link-edit command line. This option is well suited for extracting objects from archive libraries. This option may be used to perform a link-edit entirely from archives, or to provide additional flexibility in selecting the objects to extract from multiple archives (refer to "Archive Processing" on page 12 for an overview of archive extraction).

For example, lets take the generation of a dynamic executable from the relocatable object `main.o` which makes reference to the symbols `foo` and `bar`. A developer wishes 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 defining the symbol `bar` (presumably of differing functionality to that provided in `lib2.a`). In order to specify the required archive extraction the following link-edit can be used:

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

Here, the `-u` option generates a reference to the symbol `foo`. This reference will cause extraction of the relocatable object `foo.o` from the archive `lib1.a`. As the first reference to the symbol `bar` occurs in `main.o`, which is encountered after `lib1.a` has been processed, the relocatable object `bar.o` will be 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 it did then the relocatable object `bar.o` would also be extracted from `lib1.a` during its processing (refer to "Archive Processing" on page 12 for a discussion of the link-editor's multi-pass processing of an archive).

Generating the Output Image

Once all input file processing and symbol resolution is completed with no fatal errors, the link-editor will start generating the output file image.

The link-editor establishes what additional *sections* must be generated to complete the output file image. These include the symbol tables that may contain local symbol definitions from the input files, together with the global and weak symbol information that has been collected in its internal symbol table, with any output relocation and dynamic information required by the runtime linker. Once all the output section information has been established, the total output file size is calculated and the output file image is created accordingly.

When building a dynamic executable or shared object, two symbol tables are normally generated. The *.dynsym*, and its associated string table *.dynstr*, contain only global, weak and section symbols. These sections are associated with the *.text* segment so that they are mapped as part of the process image at runtime, and made available to the runtime linker to perform any necessary relocations. The *.symtab*, 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, and can even be stripped from the image using the `-s` option, or after the link-edit using `strip(1)`.

During the generation of the symbol tables a number of *reserved* symbols are created. These have special meaning to the linking process and should not be defined in any user code:

- *_etext*, the first location after the text segment.
- *_edata*, the first location after initialized data.
- *_end*, the first location after all data.
- *_DYNAMIC*, the address of the dynamic information section (the *.dynamic* section).

- *_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 occurring in objects that have been compiled with the `-K pic` option (refer to the section "Position-Independent Code" on page 85 for more information).
- *_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 occurring in objects that have been compiled with the `-K pic` option (refer to the section "Position-Independent Code" on page 85 for more information).

If the link-editor is generating an executable, it will look for additional symbols to define the executable's entry point. If a symbol was specified using the `-e` option it will be used, otherwise the link-editor will look for the reserved symbol names `_start`, and then `main`. If none of these symbols exists, the first address of the *text* segment will be used.

Having created the output file, all data sections from the input files are copied to the new image. Any relocations specified in the input files are applied to the output image. Any new relocation information that must be generated, together with all the other link-editor generated information, is also written to the new image.

Debugging Aids

Provided with the SunOS operating system linkers is a debugging library that allows developers to trace the link-editing process in more detail. This library helps users understand, or debug, the link-edit of their own applications or libraries. This is a *visual* aid, and although the type of information displayed using this library is expected to remain constant, the exact format of the information may change slightly from release to release.

Much of the debugging output may be unfamiliar to those who do not have an intimate knowledge of ELF, however, some aspects may be of general interest to many developers.

Debugging is enabled by using the `-D` option, and all output produced is directed to the standard error. This option must be augmented with one or more tokens to indicate the type of debugging required. The tokens available can be displayed by using `-Dhelp`. For example:

```
$ ld -Dhelp
debug:
debug:          For debugging the link-editing of an application:
debug:                 LD_OPTIONS=-Doption1,option2 cc -o prog ...
debug:          or,
debug:                 ld -Doption1,option2 -o prog ...
debug:         where placement of -D on the command line is significant
debug:          and options can be switched off by prepending with `!'.
debug:
debug:
debug: args      display input argument processing
debug: detail    provide more information in conjunction with other
debug:           options
debug: entry     display entrance criteria descriptors
debug: files     display input file processing (files and libraries)
debug: help      display this help message
debug: libs      display library search paths; detail flag shows actual
debug:           library lookup (-l) processing
debug: map       display map file processing
debug: reloc     display relocation processing
debug: sections  display input section processing
debug: segments  display available output segments and address/offset
debug:           processing; detail flag shows associated sections
debug: symbols   display symbol table processing;
debug:           detail flag shows resolution and linker table addition
```

Note – The above is an example, and shows the options meaningful to the link-editor. The exact options may differ from release to release.

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

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

```
$ 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 (again) to verify that the extraction of `foo.o` did not warrant the extraction of additional relocatable objects. More than one "again" display indicates that the archive is a candidate for ordering using `lorder(1)` and `tsort(1)`.

By adding the `symbol's` token you can also determine what symbol caused this 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 ]
.............
```

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

Note – The above output has been simplified for this document.

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
```

Here, 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.

From the above example, it should be apparent that using some of the debugging tokens can produce a wealth of output. In cases where the developer is only interested in the activity around a subset of the input files, the `-D` option can be placed directly in the link-edit command-line, and toggled on and off (to obtain the link-edit command-line it may be necessary to expand the compilation line from any driver being used, refer to “Using a Compiler Driver” on page 9 for more details). For example:

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

Here the display of symbol processing will be switched on *only* during the processing of the library `libbar`.

Runtime Linker 3

Overview

As part of the initialization and execution of a *dynamic executable*, a runtime linker is called to complete the binding of the application to its shared object dependencies. The running application may also call the services of the runtime linker to extend its address space by mapping additional shared objects and binding to symbols within them.

During the link-editing of a dynamic executable, a special *.interp* section, together with an associated program header, were created. This section contains a pathname specifying the program's *interpreter*. The default name supplied by the link-editor is `/usr/lib/ld.so.1`. During the process of executing a dynamic executable (refer to `exec(2)`) the kernel maps the file (refer to `mmap(2)`), and using the program header information, locates the name of the required runtime linker. The kernel then maps this runtime linker and transfers control to it, passing sufficient information to allow the runtime linker to continue binding the application and then run it.

The following is a simple breakdown of the runtime linker's functionality, and introduces the topics covered in this chapter:

- It analyzes the executable's dynamic information section (*.dynamic*) and determines what shared object dependencies are required.
- It locates and maps in these dependencies, and analyzes their dynamic information sections to determine if any additional shared object dependencies are required.

- Once all shared object dependencies have been located and mapped, the runtime linker performs any necessary relocations to bind these objects in preparation for process execution.
- It calls any initialization functions provided by the shared object dependencies.
- It passes control to the application.
- During the application's execution, the runtime linker may be called upon to perform any delayed function binding.
- The application may also call upon the runtime linker's services to acquire additional shared libraries via `dlopen(3X)`, and bind to symbols within these libraries via `dlsym(3X)`.

Locating Shared Object Dependencies

Normally, during the link-edit of a dynamic executable, one or more shared objects were explicitly referenced. These shared objects would have been recorded as dependencies within the dynamic executable (refer to "Shared Object Processing" on page 13 for more information). The runtime linker first locates this dependency information and uses it to locate and map the associated shared objects. These shared object dependencies will be processed in the same order that they were referenced during the link-edit of the executable. Once all the dynamic executable's dependencies have been mapped, they too will be inspected, in the order they were mapped, to locate any additional shared object dependencies. This process will continue until all dependent shared objects have been located and mapped. This technique results in a breadth-first ordering of all dependent shared objects.

Directories Searched by the Runtime Linker

The runtime linker knows of only one standard place to look for shared object dependencies, `/usr/lib`. Any dependency specified as a simple filename will be prefixed with this default directory name and the resulting pathname will be used to locate the actual file.

The *actual* shared object dependencies of any 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
        libintl.so.1 =>  /usr/lib/libintl.so.1
        libw.so.1 =>     /usr/lib/libw.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
```

Here, the file /usr/bin/cat has a dependency, or *needs*, the files libintl.so.1, libw.so.1, libc.so.1 and libdl.so.1.

The shared object dependencies actually recorded in a file can be inspected by using the dump(1) command to display the file's *.dynamic* section, and referencing any entries with a NEEDED tag. For example:

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

/usr/bin/cat:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic :
[INDEX] Tag      Value
[1]     NEEDED   libintl.so.1
[2]     NEEDED   libw.so.1
[3]     NEEDED   libc.so.1
.........
```

Notice that the dependency libdl.so.1, displayed in the previous ldd(1) example, is not recorded in the file /usr/bin/cat. This is because ldd(1) shows the *total* dependencies of the specified file, and libdl.so.1 is actually a dependency of /usr/lib/libc.so.1.

In the previous dump(1) example the dependencies are expressed as simple filenames, in other words there is no '/' in the name. It is this use of a *simple* filename that requires the runtime linker to build the required pathname from a set of rules. Filenames that contain an embedded '/' will be used as-is. The simple filename recording is the standard, most flexible mechanism of recording dependencies, and is provided by using the `-l` option of the link-editor (refer to “Linking with Additional Libraries” on page 14, and “Naming Conventions” on page 68 for additional information on this topic).

Frequently, shared objects are distributed in a directory other than `/usr/lib`. 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. The recommended mechanism of indicating additional search paths to the runtime linker is to record a *runpath* during the link-edit of the dynamic executable or shared object (refer to “Directories Searched by the Runtime Linker” on page 18 for details on recording this information).

Any runpath recording can be displayed using `dump(1)` and referring to the entry with the `RPATH` tag. For example:

```
$ dump -Lv prog

prog:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic :
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so.1
[2]     NEEDED   libc.so.1
[3]     RPATH    /home/me/lib:/home/you/lib
.........
```

Here, `prog` has a dependency on `libfoo.so.1` and requires the runtime linker to search directories `/home/me/lib` and `/home/you/lib` before it looks in the default location `/usr/lib`.

An alternative mechanism of adding to the runtime linker’s search path is to set the environment variable `LD_LIBRARY_PATH`. This environment variable can be set to a colon-separated list of directories, and these directories will be searched by the runtime linker *before* any runpath specification or default directory. This environment variable is well suited for debugging purposes such as forcing an application to bind to a local shared library. For example:

```
$ LD_LIBRARY_PATH=. prog
```

Here the file `prog` from our previous example will be bound to `libfoo.so.1` found in the present working directory.

Although useful as a *temporary* mechanism of influencing the runtime linker's search path, the use of this environment variable is strongly discouraged in production software. Any dynamic executables that can reference this environment variable will have their search paths augmented, which may result in an overall degradation in performance. Also, as pointed out in "Using an Environment Variable" on page 17, and "Directories Searched by the Runtime Linker" on page 18, this environment variable effects the link-editor.

If a shared object dependency cannot be located, `ldd(1)` will indicate that the object cannot be found, and any attempt to execute the application will result in an appropriate error message from the runtime linker:

```
$ ldd prog
        libfoo.so.1 =>   (not found)
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
$ prog
ld.so.1: prog: fatal: libfoo.so.1: can't open file: errno=2
```

Note – Any runtime linker error that results from the failure of an underlying system call will result in the system error code value being displayed as part of the associated diagnostic message. This value may be interpreted more fully by referencing `/usr/include/sys/errno.h`.

Relocation Processing

Once the runtime linker has located and mapped all the shared object dependencies required by an application, it must then process each object and perform any necessary relocations.

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

For a more detailed description of the many relocation types, refer to “Relocation Types (Processor Specific)” on page 126. However, for the purposes of this discussion it is convenient to categorize relocations into one of two types:

- Non-symbolic relocations.
- Symbolic relocations.

The relocation records for an object can be displayed by using `dump(1)`. For example:

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

libbar.so.1:

    **** RELOCATION INFORMATION ****

.rela.got:
Offset      Symndx                Type              Addend

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

Here the file `libbar.so.1` contains two relocation records that indicate that the *global offset table* (the *.got* section) must be updated. The first relocation is a simple *relative* relocation that can be seen from its relocation type and from the fact that the symbol index (`Symndx`) field is zero. This relocation needs to use the base address at which the object was mapped into memory to update the associated *.got* offset. The second relocation requires the address of the symbol `foo`, and thus to complete this relocation the runtime linker must locate this symbol from the dynamic executable or shared objects that have so far been mapped.

Symbol Lookup

When the runtime linker needs to look up a symbol, it does so by searching in each object, starting with the dynamic executable, and progressing through each shared object in the same order in which the objects were mapped. As discussed in previous sections, `ldd(1)` will list the shared object dependencies of a dynamic executable in the order in which they are mapped. Therefore, if the shared object `libbar.so.1` requires the address of symbol `foo` to complete its relocation, and this shared object is a dependency of the dynamic executable `prog`:

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

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

Note – Symbol lookup can be an expensive operation, especially as the size of symbol names increase, and the numbers of shared object dependencies increase. This aspect of performance is discussed in more detail in the section “Performance Considerations” on page 81.

Interposition

The runtime linker’s mechanism of searching for a symbol first in the dynamic executable and then in each of the shared object dependencies means that the *first* occurrence of the required symbol will satisfy the search. Therefore, if more than one instance of the same symbol exists, the first instance will *interpose* on all others.

When Relocations are Performed

Having briefly described the relocation process, together with the simplification of relocations into the two types, non-symbolic and symbolic, it is also useful to distinguish relocations by when they are performed. This distinction arises due to the type of *reference* being made to the relocated offset, and can be either:

- A data reference.
- A function reference.

A *data reference* refers to an address that is used as a data item by the application's code. The runtime linker has no knowledge of the application's code, and thus does not know when this data item will be referenced. Therefore, all data relocations must be carried out during process initialization, prior to the application gaining control.

A *function reference* refers to the address of a function that will be called by the application's code. During the compilation and link-editing of any dynamic module, calls to global functions are relocated to become calls to a *procedure linkage table* entry (these entries make up the *.plt* section). These *.plt* entries are constructed so that when they are called, control is passed to the runtime linker. The runtime linker will look up the required symbol and then rewrite the *.plt* entry using the symbol's address. Thus, any future calls to this *.plt* entry will go directly to the function. This mechanism allows relocations of this type to be *deferred* until the first instance of a function being called, a process that is sometimes referred to as *lazy* binding.

The runtime linker's default mode of performing lazy binding 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 data reference and function reference relocations during process initialization, prior to transferring control to the application. For example:

```
$ LD_BIND_NOW=yes prog
```

Here, *all* relocations within the file `prog` and within its shared object dependencies will be processed before control is transferred to the application.

Relocation Errors

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

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

Here the symbol `bar`, which is referenced in the file `libfoo.so.1`, could not be located.

Note – During the link-edit of a dynamic executable any potential relocation errors of this sort will normally be flagged as fatal undefined symbols (see “Generating an Executable” on page 28 for examples). This runtime relocation error can occur if the link-edit of `main` used a different version of the shared object `libbar.so.1`, one that contained a symbol definition for `bar`. This runtime relocation error can also occur if the `-z nodefs` option was used as part of the link-edit.

If a relocation error of this type occurs because a symbol used as a data reference cannot be located, the error condition will occur immediately during process initialization. However, because of the default mode of lazy binding, if a symbol used as a function reference cannot be found, the error condition will occur after the application has gained control. This latter case could take minutes or months, or may 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 may be validated using `ldd(1)`.

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

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

When the -r option is specified with ldd(1), all data *and* function reference relocations will be processed, and if either cannot be resolved a diagnostic message will be produced.

Adding Additional Objects

The previous sections have described how the runtime linker initializes a process from the dynamic executable and its shared object dependencies as they were defined during the link-editing of each module. The runtime linker also provides an additional level of flexibility by allowing the user to introduce new objects during process initialization.

The environment variable LD_PRELOAD can be initialized to a shared object or relocatable object filename, or a string of filenames separated by white space. These objects will then be mapped *after* the dynamic executable and *before* any shared object dependencies. For example:

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

Here the dynamic executable `prog` will be mapped, followed by the shared object `newstuff.so.1`, and then by the shared object dependencies defined within `prog`. The order in which this object is processed can be displayed using `ldd(1)`:

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

Another example would be:

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

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

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

These mechanisms of inserting a shared object after a dynamic executable, takes the concept of *interposition*, introduced on page 45, to another level. Using these mechanisms, it is possible to experiment with a new implementation of a function that resides in a standard shared object. By preloading just that function it will interpose on the original. Here the intention is to completely hide the old functionality with the new preloaded version.

Another use of preloading is to augment the functionality of a function that resides in a standard shared object. Here the intention is to have the new symbol interpose on the original, allowing the new function to carry out some additional processing while still having it call through to the original function. This mechanism requires either a symbol alias to be associated with the original function (refer to “Simple Resolutions” on page 22), or the ability to lookup the original symbol’s address (refer to “Using Interposition” on page 60).

Initialization and Termination Routines

Prior to transferring control to the application, the runtime linker processes any initialization (*.init*) and termination (*.fini*) sections found in any of the shared object dependencies. These sections, and the symbols that describe them, were created during the link-editing of the shared objects (refer to “Initialization and Termination Sections” on page 19).

Any shared object dependency’s initialization routines are called in *reverse* load order, in other words, the reverse order of the shared objects displayed via `ldd(1)`.

Any shared object dependency’s termination routines are organized such that they can be recorded by `atexit(3C)`. This results in the termination routines being called in load order when the process calls `exit(2)`.

Although this initialization and termination calling sequence seems quite straightforward, be careful about placing too much emphasis on this sequence, as the ordering of shared objects can be effected by both shared library and application development (refer to section “Dependency Ordering” on page 77 for more details).

Note – Any *.init* or *.fini* sections within the dynamic executable will be called from the application itself via the process start-up and termination mechanism supplied by the compiler driver. The combined effect is that the dynamic executable’s *.init* section will be called last, after all the shared object dependency’s *.init* sections have been executed, and the dynamic executable’s *.fini* section will be called first, before the shared object dependency’s *.fini* sections are executed.

Runtime Linking Programming Interface

The previous discussions have described how the shared object 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 is able to extend its address space during its execution by binding to additional shared objects. This extensibility is provided by allowing the application to request the same services of the runtime linker that were used to process the shared object's dependencies specified during the link-edit of the application.

There are several advantages in this delayed binding of shared objects:

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

The following is a typical scenario that an application may perform to access an additional shared object, and introduces the topics covered in the next sections:

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

The services of the runtime linker are defined in the header file `dlfcn.h` and are made available to an application via the shared library `libdl.so.1`. For example:

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

Here the file `main.c` can make reference to any of the `dlopen(3X)` family of routines, and the application `prog` will be bound to these routines at runtime.

Adding Additional Objects

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

If the filename is specified as a simple filename, in other words, there is no '/' in the name, then the runtime linker will use a set of rules to build an appropriate pathname. Filenames that contain a '/' will be used as-is. These rules are exactly the same as were used to locate any initial shared library dependencies (refer to "Directories Searched by the Runtime Linker" on page 40). For example, lets take the file `main.c` that contains the following code fragment:

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

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

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

To locate the shared object `foo.so.1`, the runtime linker will use any `LD_LIBRARY_PATH` definition presently in effect, followed by any runpath specified during the link-edit of `prog`, and finally the default location `/usr/lib`. If the filename had been specified as:

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

then the runtime linker would have searched for the file only in the present working directory.

Note – It is recommended that any shared object specified using `dlopen(3X)` be referenced by its *versioned* filename (for more information on versioning refer to "Versioning" on page 73).

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

```
$ cc -o prog main.c -ldl
$ prog
dlopen: ld.so.1: prog: fatal: foo.so.1: can't open file: errno=2
```

The `errno` value can be referenced in `/usr/include/sys/errno.h`.

If the shared object being added by `dlopen(3X)` has dependencies on other shared objects, they too will be brought into the process's address space.

If the shared object specified by `dlopen(3X)`, or any of its dependencies, are already part of the process image, then the shared objects will not be processed any further, however a valid handle will still be returned to the application. This mechanism prevents the same shared object from being mapped more than once, and allows an application to obtain a handle to itself. For example, if our `main.c` example contained the following code:

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

then the handle returned from `dlopen(3X)` can be used by the application to locate symbols within itself, or in any of the shared object dependencies loaded as part of the process's initialization.

Relocation Processing

As described in the section "Relocation Processing" on page 43, after locating and mapping any shared objects, the runtime linker must then process each object and perform any necessary relocations. Any shared objects brought into the process's address space with `dlopen(3X)` must also be relocated in the same manner. For simple applications this process may be quite uninteresting. However, for users who have more complex applications with many `dlopen(3X)` calls involving numerous shared objects, possibly with common dependencies, this topic may be quite important.

Relocations can be categorized according to when they occur. The default behavior of the runtime linker is to process all data reference relocations at initialization and all function references during process execution, a mechanism commonly referred to as *lazy* binding. This same mechanism is applied to any shared objects added with `dlopen(3X)` when the *mode* is defined as `RTLD_LAZY`. The alternative to this is to require all relocations of a shared object to be performed immediately when the shared object is added, and this can be achieved by using a *mode* of `RTLD_NOW`.

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 may occur, with a focus on some of the subtleties of symbol lookup.

Symbol Lookup

If a shared object acquired by `dlopen(3X)` refers to a global symbol, the runtime linker will locate this symbol in the same manner as any other symbol lookup. The runtime linker will first look in the dynamic executable, and then look in each of the shared objects provided during the initialization of the process. However, if the symbol has still not been found, the runtime linker will continue the search and will look in the shared object acquired through the dlopen(3X) and in any of its dependencies. For example, lets take the dynamic executable prog, and the shared object B.so.1, each of which have the following (simplified) 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 via dlopen(3X), 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, in may be easier to think of the shared objects acquired through the dlopen(3X) as if they had been added to the end of the original link-edit of the application. For example, the objects referenced above can be expressed diagrammatically:

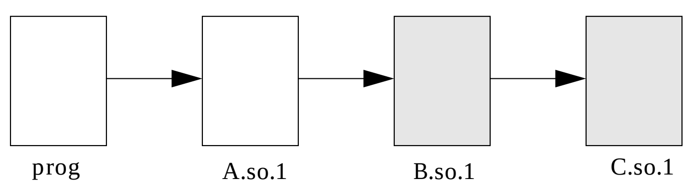

Figure 3-1 A Single dlopen(3X) Request

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

Note – Objects added to the process address space do not effect the normal symbol lookup required by either the application or its initial shared object dependencies. For example, if A.so.1 requires a function relocation *after* the above dlopen(3X) has occurred, the runtime linker's normal search for the relocation symbol will be to look in prog and then A.so.1, but not to follow through and look in B.so.1 or C.so.1.

This symbol lookup algorithm is established by assigning lookup *scopes* to each object. These scopes maintain associations between objects based on their introduction into the process address space, and any dependency relationships between the objects. All objects that were obtained during the process's initialization are assigned a *global* scope. Any object within the global scope can be used by any other object to provide symbols for relocation. On the other hand, the shared objects associated with a given `dlopen(3X)` are assigned a unique *local* scope that insures that only objects associated with the same `dlopen(3X)` 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(3X)`. For example, if the shared object `D.so.1` has the following dependency:

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

and the `prog` application was to `dlopen(3X)` this shared object in addition to the shared object `B.so.1`, then diagrammatically the symbol lookup relationship between the objects may be represented as:

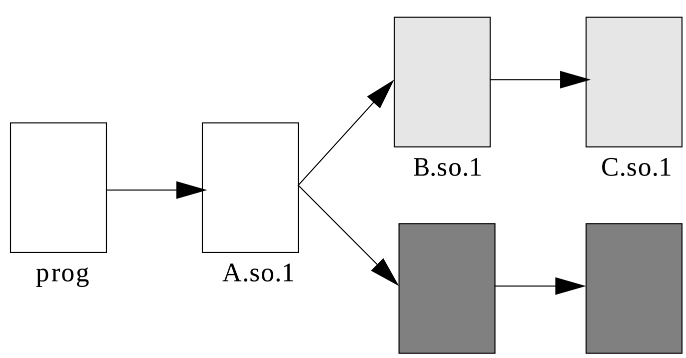

Figure 3-2 Multiple dlopen(3X) Requests

If 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, then because of the association of objects defined by the runtime linker, `C.so.1` will be bound to the definition in `B.so.1`, and `E.so.1` will be bound to the definition in `D.so.1`. This mechanism is intended to provide the most intuitive binding of shared objects obtained via multiple calls to `dlopen(3X)`.

When shared objects are used in the scenarios that have so far been described, the order in which each `dlopen(3X)` occurs has no effect on the resulting symbol binding. However, when shared objects have common dependencies the resultant bindings may be effected by the order in which the `dlopen(3X)` calls were made. Take for example the shared objects `O.so.1` and `P.so.1`, which 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(3X)` 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` it will be assigned both of the local scopes that are associated with the two `dlopen(3X)` calls. Diagrammatically this can be represent as:

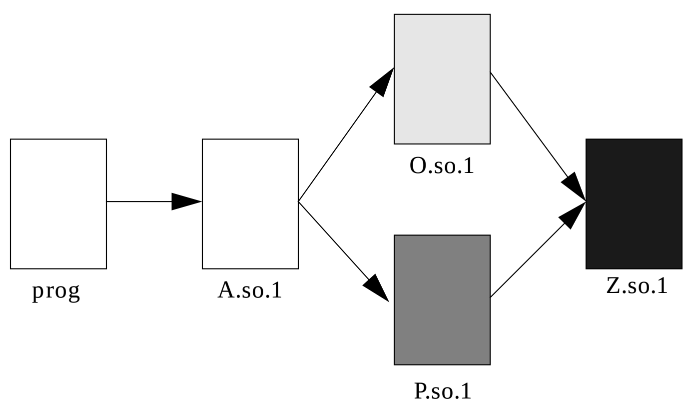

Figure 3-3 Multiple dlopen(3X) Requests With A Common Dependency

The result is that `Z.so.1` will be available for both `O.so.1` and `P.so.1` to look up symbols, but more importantly, as far as `dlopen(3X)` ordering is concerned, `Z.so.1` will 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 will be affected by the order of the dlopen(3X) calls. Thus, it should be obvious that 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 may vary depending on the application's dlopen(3x) ordering.

There is one final convolution involving the mode of a dlopen(3X). All previous examples have revolved around the shared objects obtained via a dlopen(3X) each having a unique local scope, or a combination of local scopes if a shared object is a common dependency. It is also possible to give a shared object a *global* scope by augmenting the mode argument with the RTLD_GLOBAL flag. Under this mode, any shared objects obtained through a dlopen(3X) may be used by any other objects to locate symbols.

Obtaining New Symbols

A process may obtain the address of a specific symbol using dlsym(3X). 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:

- The *handle* returned from a dlopen(3X) of a *named* shared object will allow symbols to be obtained from that shared object, or from any of its dependencies.
- The *handle* returned from a dlopen(3X) of a file whose value is 0 will allow symbols to be obtained from the dynamic executable, or from any of its initialization dependencies.
- The special *handle* RTLD_NEXT will allow symbols to be obtained from the *next* associated shared object.

The first example is probably the most common. Here an application will add additional shared objects to its address space and use `dlsym(3X)` to locate function or data symbols, and use these symbols to call upon services provided in these new shared objects. For example, let's take the file `main.c` that contains the following code:

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

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

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

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

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

Here the symbols `foo` and `bar` will be searched for in the file `foo.so.1` followed by any shared object dependencies that may be associated with this file. The function `foo` is then called with the single argument `bar` as part of the `return` statement.

If our application `prog` had been built using the above file `main.c`, and its initial shared object dependencies were:

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

then if the filename specified in the `dlopen(3X)` had the value 0, the symbols `foo` and `bar` would have been searched for in `prog`, followed by `/usr/lib/libdl.so.1`, and finally `/usr/lib/libc.so.1`.

Once the *handle* has indicated the root at which to start a symbol search, the search mechanism follows the same model as was described in the previous section “Symbol Lookup” on page 54”.

If the required symbol cannot be located, `dlsym(3X)` will return a NULL value. In this case `dlerror(3X)` can be used to indicate the true reason for the failure. For example;

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

Here the application `prog` was unable to locate the symbol `bar`.

Using Interposition

The special handle `RTLD_NEXT` allows an application to locate the next symbol in a symbol scope. For example, if our application `prog` were to contain the following code fragment:

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

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

then `foo` would have been searched for in the shared objects associated with `prog`, in this case, `/usr/lib/libdl.so.1` and then `/usr/lib/libc.so.1`. If this code fragment were contained in the file `B.so.1` from the example shown in Figure 3-2 on page 56, then `foo` would have been searched for in the associated shared object `C.so.1` only.

Using RTLD_NEXT provides a means to exploit symbol *interposition*. For example, a shared object function can be interposed upon by a preceding shared library, which can then augment the processing of the original function. If the following code fragment were placed in the shared object malloc.so.1:

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

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

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

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

Then by interposing this shared object between the system library /usr/lib/libc.so.1 where malloc(3C) normally resides, any calls to this function will be interposed on before the original function is called to complete the allocation:

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

Alternatively, this same interposition could be achieved via:

```
$ 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(3S)`, instead of using `printf(3S)` directly, to avoid any recursion caused by `printf(3S)`'s use of `malloc(3C)`.

Debugging Aids

Provided with the SunOS operating system linkers is a debugging library that allows developers to trace the runtime linking process in more detail. This library helps users understand, or debug, the execution of their own applications or libraries. This is a *visual* aid, and although the type of information displayed using this library is expected to remain constant, the exact format of the information may change slightly from release to release.

Much of the debugging output may be unfamiliar to those who do not have an intimate knowledge of the runtime linker, however, some aspects may be of general interest to many developers.

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

LD_DEBUG=help. Any dynamic executable can be used to solicit this information, as the process itself will terminate following the display of the information. For example:

```
$ LD_DEBUG=help prog
11693:
11693:           For debugging the run-time linking of an application:
11693:                  LD_DEBUG=option1,option2  prog
11693:           enables diagnostics to the stderr.  The additional
11693:           option:
11693:                  LD_DEBUG_OUTPUT=file
11693:           redirects the diagnostics to an output file created
11593:           using the specified name and the process id as a
11693:           suffix.  All output is prepended with the process id.
11693:
11693:
11693: bindings  display symbol binding; detail flag shows
11693:           absolute:relative addresses
11693: detail    provide more information in conjunction with other
11693:           options
11693: files     display input file processing (files and libraries)
11693: help      display this help message
11693: libs      display library search paths
11693: reloc     display relocation processing
11693: symbols   display symbol table processing;
11693:           detail flag shows resolution and linker table addition
$
```

Note – The above is an example, and shows the options meaningful to the runtime linker. The exact options may differ from release to release.

The environment variable LD_DEBUG_OUTPUT can be used to specify an output file for use instead of the standard error. The output file name will be suffixed with the process identifier.

Debugging of secure applications is not allowed.

One of the most useful debugging options is to display the symbol bindings that occur at runtime. For example, lets take 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 -Kpic -G bar.c

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

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

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

We can display the runtime symbol bindings 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: .......
```

Here, the symbol `bar`, which is required by a data relocation, is bound *prior* to the application gaining control. Whereas the symbol `foo`, which is required by a function relocation, is bound *after* the application gains control when the function is first called. This demonstrates the default mode of lazy binding. Had the environment variable `LD_BIND_NOW` been set, all symbol bindings would have occurred prior to the application gaining control.

Additional information regarding the real, and relative addresses of the actual binding locations can be obtained by setting `LD_DEBUG=bindings,detail`.

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

Another aspect of the runtime environment that can be displayed involves the various search paths used. For example, the search path mechanism used to locate any shared library dependencies can be displayed by setting `LD_DEBUG=libs`:

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

Here, the runpath recorded in the application `prog` effects the search for the two dependencies `foo.so.1` and `bar.so.1`.

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

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

Note – In the previous example the symbol `bar` is not searched for in the application `prog`. This is due to an optimization used when processing copy relocations (refer to section “Relocations” on page 90 for more details of this relocation type).

Shared Objects 4

Overview

Shared objects are one form of output created by the link-editor, and are generated by specifying the `-G` option. For example:

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

Here the shared object `libfoo.so` is generated from the input file `foo.c`.

Note – This is a simplified example of generating a shared object. Normally, additional options are recommended, and these will be discussed in subsequent sections of this chapter.

A shared object is an indivisible unit generated from one or more relocatable objects. Shared objects are intended to be bound with dynamic executables to form a runable process. As their name implies, shared objects may be *shared* by more than one application, and it is because of this potentially far-reaching effect that 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 made 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.

That is, *all* the functionality of the shared object is made available to the output file. These shared objects become *dependencies* of this output file. However, only a small amount of bookkeeping information is maintained to describe these dependencies, as it is the runtime linker that will finally interpret this information and complete 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 were introduced in "Shared Objects" on page 4). Issues that complement and help coordinate the use of shared objects within these environments are covered, with techniques that maximize the efficiency of the shared objects.

Naming Conventions

Neither the link-editor, nor the runtime linker interprets any file by virtue of its filename. All files are inspected to determine their ELF type (refer to section "ELF Header" on page 100) and from this information the processing requirements of the file are deduced. However, shared objects normally follow one of two naming conventions depending on whether they are being used as part of the compilation environment or the run-time environment.

When used as part of the compilation environment, shared objects are read and processed by the link-editor. Although these shared objects may be specified by filenames as part of the command-line passed to the link-editor, it is more common that the `-l` option be used to take advantage of the link-editor's library search capabilities (refer to "Shared Object Processing" on page 13). For a shared object to be applicable to this link-editor processing it should be designated with the prefix `lib` and the suffix `.so`. For example, `/usr/lib/libc.so` is the shared object representation of the standard C library made available to the compilation environment.

When used as part of the runtime environment, shared objects are read and processed by the runtime linker. Here it may be necessary to allow for change in the exported interface of the shared object over a series of software releases. This interface change can be anticipated and supported by providing the shared object as a *versioned* filename. This versioned filename commonly takes the form of a `.so` suffix followed by a version number. For example, `/usr/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 may drop the conventional `lib` prefix. However, a `.so` suffix is still recommended to indicate the actual file type, and a version number is strongly recommended to provide for the correct binding of the shared object across a series of software releases. Examples of shared objects that fall into this category are those used solely with `dlopen(3X)`.

Note – The shared object name used in a `dlopen(3X)` is normally represented as a *simple* filename, in other words there is no '/' in the name. This convention provides flexibility by allowing the runtime linker to use a set of rules to locate the actual file (refer to "Adding Additional Objects" on page 48 for more details).

Later, in the section "Versioning" on page 73, the concept of versioning is described in more detail and a mechanism for coordinating the naming conventions between shared objects used in both the compilation and runtime environments is presented. But first, a mechanism that allows a shared object to record its own runtime name is introduced.

Recording a Shared Object Name

When the link-editor records a *dependency* in a dynamic executable or shared object it is creating, this dependency will by default be the filename of the associated shared object as it was referenced by the link-editor. For example, the following dynamic executables, when built against the same shared object `libfoo.so`, result in different interpretations of the same dependency:

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

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

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

As these examples show, this mechanism of recording dependencies can result in inconsistencies due to different compilation techniques. Also, it may be the case that the location of a shared object as it is referenced during a link-edit is different than the eventual location of the shared object on an installed system. To provide a more straightforward means of specifying dependencies, shared objects may record within themselves the filename by which they should be referenced at runtime.

During the link-edit of a shared object, its eventual runtime name may be recorded within the shared object itself by using the `-h` option. For example:

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

Here, the shared object's runtime name `libfoo.so.1`, is recorded within the file itself. This identification is known as an *soname*, and its recording can be displayed using `dump(1)` and referring to the entry with the `SONAME` tag. For example:

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

../tmp/libfoo.so:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic :
[INDEX] Tag      Value
[1]     SONAME   libfoo.so.1
.........
```

When the link-editor processes a shared object that contains an *soname*, it is this name that will be recorded as the dependency within any output file being generated, rather than the filename of the shared object as it was referenced. Therefore, if this new version of `libfoo.so` was used during the creation of the dynamic executable prog from our previous example, all three methods of building the executable would have resulted in the same dependency recording:

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

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

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

In the examples shown above, the -h option is used to specify a *simple* filename, in other words there is no '/' in the name. This convention is also recommended, because it provides flexibility by allowing the runtime linker to use a set of rules to locate the actual file (refer to section "Locating Shared Object Dependencies" on page 40 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 via an archive library.

If an archive is built from one or more shared objects and this archive is then used to generate a dynamic executable or shared object, then any shared objects within the archive may be extracted to satisfy the requirements of the link-edit (refer to section "Archive Processing" on page 12 for more details on the criteria for archive extraction). However, unlike the processing of relocatable objects which are concatenated to the output file being created, any shared objects extracted from an archive will only be recorded as dependencies. The name of these dependencies is a concatenation of the archive name and the object within the archive. For example:

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

As it is highly unlikely that a file with this concatenated name will exist at runtime, providing an *soname* within the shared object is the only means of generating a meaningful runtime filename.

Note – The run-time linker does not extract objects from archives. Therefore, in the above example it will be necessary for the required shared object dependencies to be extracted from the archive and made available to the runtime environment.

Recorded Name Conflicts

When shared objects are used to build a dynamic executable or another shared object, the link-editor performs a number of consistency checks to insure that any dependency names that must be 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: file ./libbar.so: recording name `libsame.so.1' \
          matches that provided by file ./libfoo.so
ld: fatal: File processing errors.  No output written to prog
```

A similar error condition will occur if the filename of a shared object that does not have a recorded *soname* matches the *soname* of another shared object used during the same link-edit. Similarly, should the runtime name of a shared object being generated match one of its dependencies the link-editor will report a name conflict. For example:

```
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c -L. -lfoo
ld: fatal: file ./libfoo.so: recording name `libsame.so.1'  \
           matches that supplied with -h option
ld: fatal: File processing errors.  No output written to libfoo.so
```

Versioning

Versioning provides a mechanism by which a shared object's *interface* can be changed across a series of software releases.

If the shared object `libfoo.so.1` contains the function `foo()`, then an application can be built that refers to this function by defining this shared object as a dependency. At runtime this dependency will be processed by the runtime linker and added to the process address space. Thus, the application's reference to `foo()` will be satisfied by this shared library at runtime. If a later release of the shared object `libfoo.so.1` is provided that no longer contains the function `foo()`, then the old application will still bind to this shared object at runtime, but it will be unable to satisfy its reference to the function `foo()`. A modification of this kind has changed the shared object's interface, and by supplying this new interface without changing the file's versioned name, old applications are likely to misbehave or break entirely.

When a shared object's interface changes such that it will break old applications, the new shared object should be delivered with a new versioned filename. In our previous example, if the new shared object in which the function `foo()` no longer exists was made available as `libfoo.so.2`, then our original application would still bind to its dependency `libfoo.so.1` and execute correctly.

By providing shared objects as versioned filenames with the runtime environment, applications built over a series of software releases can be guaranteed that the interface against which they were built is available for them to bind during their execution.

The following section describes how to coordinate the binding of an interface between the compilation and runtime environments.

Coordination Of Binding Requirements

In the section "Naming Conventions" on page 68 it was stated that during a link-edit the most common method to input shared objects was to use the `-l` option. This option will use the link-editor's library search mechanism to locate shared objects that are prefixed with `lib` and suffixed with `.so`. In the section "Versioning" on page 73, it was also stated that at runtime any shared object dependencies should exist in their *versioned* name form. Instead of maintaining two distinct shared objects that follow these naming conventions, the most common mechanism of coordinating these objects involves creating file system links between the two filenames.

To make the runtime shared object `libfoo.so.1` available to the compilation environment it is necessary to provide a symbolic link from the compilation filename to the runtime filename. For example:

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ln -s libfoo.so.1 libfoo.so
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1991 libfoo.so -> libfoo.so.1
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
```

Note – Either a symbolic *or* hard link may be used. However, as a documentation and diagnostic aid, symbol links are more useful.

Here, the shared object `libfoo.so.1` has been generated for the runtime environment. Generating a symbolic link `libfoo.so`, has also enabled this file's use in a compilation environment. For example:

```
$ cc -o prog main.o -L. -lfoo
```

Here the link-editor will process the relocatable object `main.o` with the interface described by the shared object `libfoo.so.1` which it will find by following the symbolic link `libfoo.so`.

If over a series of software releases new versions of this shared object are distributed with changed interfaces, the compilation environment can be constructed to use the interface that is applicable by changing the symbolic link. For example:

```
$ ls -l libfoo*
lrwxrwxrwx  1 usr grp          11 1993 libfoo.so -> libfoo.so.3
-rwxrwxr-x  1 usr grp        3136 1991 libfoo.so.1
-rwxrwxr-x  1 usr grp        3237 1992 libfoo.so.2
-rwxrwxr-x  1 usr grp        3554 1993 libfoo.so.3
```

Here several versions of the shared object are available to maintain compatibility with the runtime requirements of new and existing applications. However, the compilation environment has been set up to use the latest shared object version.

Using this symbolic link mechanism is insufficient by itself to coordinate the correct binding of a shared object from its use in the compilation environment to its requirement in the runtime environment. As the example presently stands, the link-editor will record in the dynamic executable `prog` the *filename* of the shared object it has processed, which in this case will be the compilation environment filename:

```
$ dump -Lv prog

prog:

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

This means that when the application `prog` is executed, the runtime linker will search for the dependency `libfoo.so`, and consequently this will bind to whichever file this symbolic link is pointing. Therefore, to provide for the correct runtime name to be recorded as a dependency, the shared object libfoo.so.1 should be built with an *soname* definition. This definition can be provided using the -h option during the link-edit of the shared object itself. For example:

```
$ cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 foo.c
$ ln -s libfoo.so.1 libfoo.so
$ cc -o prog main.o -L. -lfoo
$ dump -Lv prog

prog:

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

This symbolic link and the *soname* mechanism has established a robust coordination between the shared object naming conventions of the compilation and runtime environments, one in which the interface processed during the link-edit is accurately recorded in the output file generated. This recording insures that the intended interface will be furnished at runtime.

Shared Objects With Dependencies

Although most of the examples presented so far in this chapter have shown how shared object dependencies are maintained in dynamic executables, it is also quite common for shared objects to have their own dependencies (this was introduced in section "Shared Object Processing" on page 13).

In the section "Directories Searched by the Runtime Linker" on page 40, the search rules used by the runtime linker to locate shared object dependencies were covered. If a shared object does not reside in the default directory /usr/lib, then the runtime linker must explicitly be told where to look. The preferred mechanism of indicating any requirement of this kind is to record a *runpath* in the object that has the dependencies by using the link-editor's -R option. For example:

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

libfoo.so:

  **** DYNAMIC SECTION INFORMATION ****
.dynamic :
[INDEX] Tag      Value
[1]     NEEDED   libbar.so
[2]     RPATH    /home/me/lib
.........
```

Here, the shared object libfoo.so has a dependency on libbar.so, which is expected to reside in the directory /home/me/lib at runtime.

It is the responsibility of the shared object to specify any runpath required to locate its dependencies. Any runpath specified in the dynamic executable will only be used to locate the dependencies of the dynamic executable, it will *not* be used to locate any dependencies of the shared objects.

However, the environment variable LD_LIBRARY_PATH has a more global scope, and any pathnames specified using this variable will be used by the runtime linker to search for *any* shared object dependencies. Although useful as a *temporary* mechanism of influencing the runtime linker's search path, the use of this environment variable is strongly discouraged in production software (refer to section "Directories Searched by the Runtime Linker" on page 40 for a more extensive discussion).

Dependency Ordering

In most of examples in this document, dependencies of dynamic executables and shared objects have been portrayed as unique and relatively simple (the breadth-first ordering of dependent shared objects was first described in the section "Locating Shared Object Dependencies" on page 40). From these examples, the ordering of shared objects as they are brought into the process address space may seem very intuitive and predictable. However, when dynamic executables and shared objects have dependencies on the same common shared objects, the order in which the objects are processed may 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 a developer of a dynamic executable uses this shared object, together with defining an explicit dependency on `libC.so.1`, then the resulting shared object order will be:

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

Therefore, if the developer of the shared object `libfoo.so.1` had placed a requirement on the order of processing of its dependencies, this requirement will have been compromised by the developer of the dynamic executable `prog`.

Developers who place special emphasis on symbol interposition (refer to section "Symbol Lookup" on page 45), and *.init* section processing (refer to section "Initialization and Termination Routines" on page 50), should be aware of this potential change in shared object processing order.

Shared Objects as Filters

A *filter* is a special form of shared object that is used to provide just a symbol table. At execution time, an application using the filter will "see" only the symbols provided by the filter. However, accesses to those symbols will be bound to the implementation identified by the filter. Filters are identified during their link-edit by the `-F` flag, which takes an associated filename indicating the shared object to be used to supply symbols at runtime.

Lets take, for example, the shared object `libbar.so.1`. This shared object may have been built from many relocatable objects, but one of these objects originated from the file `bar.c`, which supplies the symbols `foo` and `bar`:

```
$ cat bar.c
int bar = 2;

foo()
{
    return(printf("foo(): defined in bar.c: bar=%d\n", bar));
}
$ cc -o libbar.so.1 -G -K pic .... bar.c ....
$ nm -x libbar.so.1 | egrep "foo|bar"
[38]    |0x000104a0|0x00000004|OBJT |GLOB |0    |11     |bar
[40]    |0x00000418|0x00000038|FUNC |GLOB |0    |7      |foo
```

We can now generate a filter, `libfoo.so.1`, for just the symbols `foo` and `bar`, and indicate the association to the shared object `libbar.so.1`. For example:

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

foo()
{
    return (printf("foo(): defined in foo.c: bar=%d\n", bar));
}
$ LD_OPTIONS="-F libbar.so.1" \
    cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ ln -s libfoo.so.1 libfoo.so
$ dump -Lv libfoo.so.1 | egrep "SONAME|FILTER"
[1]     SONAME   libfoo.so.1
[2]     FILTER   libbar.so.1
```

Note – Here the environment variable `LD_OPTIONS` is used to circumvent this compiler driver from interpreting the `-F` option as one of its own.

By using the filter `libfoo.so.1` to build a dynamic executable, the link-editor will use the information from the symbol table of the filter during the symbol resolution process (see “Symbol Resolution” on page 21 for more details). However, at runtime the dynamic executable’s dependency on the filter will result in the additional loading of the associated shared object `libbar.so.1`. The runtime linker will use this association to resolve any symbols defined by `libfoo.so.1` from `libbar.so.1`. For example:

```
$ cc -o prog main.c -L. -lfoo
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libbar.so.1 =>   ./libbar.so.1
        ...........
$ prog
foo(): defined in bar.c: bar=2
```

Here the execution of the dynamic executable `prog` results in the function `foo()` being obtained from `libbar.so.1`, *not* from `libfoo.so.1`.

Note – In this example, the shared object `libbar.so.1` is uniquely associated to the filter `libfoo.so.1` and it is not available to satisfy symbol lookup from any other objects that may be loaded as a consequence of executing `prog`.

Filters therefore provide a convenient, generic mechanism for defining a subset interface of an existing shared object. This feature is used in the SunOS operating system to create the shared objects `/usr/lib/libsys.so.1` and `/usr/lib/libdl.so.1`. The former provides a subset of the standard C library `/usr/lib/libc.so.1`, which represents the ABI-conforming functions and data items that reside in the C library that *must* be imported by a conforming application. The latter defines the user interface to the runtime linker itself.

As the code in a filter is never actually referenced at runtime there is little point in adding content to any of the functions defined within the filter (our previous example filter `libfoo.so.1`, contains a `printf()` call within the function `foo()` to aid this explanation). Any code within a filter may require runtime relocations, which in turn will result in an unnecessary overhead when processing the filter at runtime. Functions are best defined as empty routines.

Care should also be taken when generating the data symbols within a filter. Some of the more complex symbol resolutions carried out by the link-editor require knowledge of a symbol’s attributes, including the section to which the symbol belongs (refer to section “Symbol Resolution” on page 21 for more details). Therefore, it is recommended that the symbols in the filter be generated so that their attributes match those of the symbols in the associated shared object, in other words distinguish between initialized and uninitialized data symbols, and ensure they have the correct size. This insures that the link-editing process will analyze the filter in a manner compatible with the symbol definitions that will actually be used at runtime.

Performance Considerations

A shared object may be used by multiple applications within the same system, therefore the performance of a shared object may have far reaching effects, not only on the applications that use it, but on the system as a whole.

Although the actual code within a shared object will directly effect the performance of a running process, the performance issues focused upon here relate more to the runtime processing of the shared object itself. The following sections investigate this processing in more detail by looking at aspects such as text size and purity, together with relocation overhead.

Useful Tools

Before discussing performance it is useful to be aware of some available tools and their use in analyzing the contents of an ELF file.

Frequently reference is made to the size of either the *sections* or the *segments* that are defined within an ELF file (for a complete description of the ELF format refer to Chapter 5, “Object Files”). The size of a file can be displayed using the `size(1)` command. For example:

```
$ size -x libfoo.so.1
59c + 10c + 20 = 0x6c8

$ size -xf libfoo.so.1
..... + 1c(.init) + ac(.text) + c(.fini) + 4(.rodata) + \
..... + 18(.data) + 20(.bss) .....
```

The first example indicates the size of the shared library’s *text*, *data* and *bss*, a categorization that has traditionally been used throughout previous releases of the SunOS operating system. However, the ELF format provides a finer granularity for expressing data within a file by organizing the data into *sections*. The second example shown above displays the size of each of the file's loadable sections.

Sections are allocated to units known as *segments*, some of which describe how portions of a file's image will be mapped into memory. These loadable segments can be displayed by using the `dump(1)` command and examining the LOAD entries. For example:

```
$ dump -ov libfoo.so.1

 ***** PROGRAM EXECUTION HEADER *****
Type        Offset      Vaddr       Paddr
Filesz      Memsz       Flags       Align

LOAD        0x94        0x94        0x0
0x59c       0x59c       r-x         0x10000

LOAD        0x630       0x10630     0x0
0x10c       0x12c       rwx         0x10000
```

Here, there are two segments in the shared object `libfoo.so.1`, commonly referred to as the *text* and *data* segments. The text segment is mapped to allow reading and execution of its contents (r-x), whereas the data segment is mapped to allow its contents to be modified (rwx). Notice that the memory size (*Memsz*) of the data segment differs from the file size (*Filesz*). This difference accounts for the *.bss* section, which is actually part of the data segment.

Programmers, however, usually think of a file in terms of the symbols that define the functions and data elements within their code. These symbols can be displayed using nm(1). For example:

```
$ nm -x libfoo.so.1

[Index]   Value      Size      Type  Bind  Other Shndx   Name
.........
[39]    |0x00000538|0x00000000|FUNC |GLOB |0x0  |7      |_init
[40]    |0x00000588|0x00000034|FUNC |GLOB |0x0  |8      |foo
[41]    |0x00000600|0x00000000|FUNC |GLOB |0x0  |9      |_fini
[42]    |0x00010688|0x00000010|OBJT |GLOB |0x0  |13     |data
[43]    |0x0001073c|0x00000020|OBJT |GLOB |0x0  |16     |bss
.........
```

The section that contains a symbol can be determined by referencing the section index (*Shndx*) field from the symbol table and by using dump(1) to display the sections within the file. For example:

```
$ dump -hv libfoo.so.1

           **** SECTION HEADER TABLE ****
[No]    Type    Flags   Addr      Offset    Size      Name
.........
[7]     PBIT    -AI     0x538     0x538     0x1c      .init

[8]     PBIT    -AI     0x554     0x554     0xac      .text

[9]     PBIT    -AI     0x600     0x600     0xc       .fini
.........
[13]    PBIT    WA-     0x10688   0x688     0x18      .data

[16]    NOBI    WA-     0x1073c   0x73c     0x20      .bss
.........
```

Using the output from both the nm(1) and dump(1), we can see that the functions _init, foo and _fini are associated with the sections *.init*, *.text* and *.fini* respectively, and that these sections are part of the text segment. And the data arrays data and bss are associated with the sections *.data* and *.bss* respectively, and that these sections are part of the data segment.

Note – The previous `dump(1)` display has been simplified for this example.

Armed with this tool information, a developer should be able to analyze the location of their code and data within any ELF file they have generated. This knowledge will be useful when following the discussions in later sections.

The Underlying System

When an application is built using a shared object, the entire contents of the object are mapped into the virtual address space of that process at run time. Each process that uses a shared object starts by referencing a single copy of the shared object in memory.

Relocations within the shared object are processed to bind symbolic references to their appropriate definitions. This results in the calculation of true virtual addresses which could not be derived at the time the shared object was generated by the link-editor. These relocations normally result in updates to entries within the process's data segment(s).

The memory management scheme underlying the dynamic linking of shared object's share memory among processes at the granularity of a page. Memory pages can be shared as long as they are not modified at runtime. If a process writes to a page of a shared object when writing a data item, or relocating a reference to a shared object, it generates a private copy of that page. This private copy will have no effect on other users of the shared object, however, this page will have lost any benefit of sharing between other processes. Text pages that become modified in this manner are sometimes referred to as *impure*.

The segments of a shared library 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 (refer to the previous section "Useful Tools" on page 81 on how to obtain this information from an ELF file). An overriding goal when developing a shared object is to maximize the text segment and minimize the data segment, thus optimizing the amount of code sharing while reducing the amount of processing needed to initialize and use a shared object. The following sections present mechanisms that can help achieve this goal.

Position-Independent Code

To create programs that require the smallest amount of page modification at run time, the compiler will generate position-independent code under the `-K pic` option. Whereas the code within a dynamic executable is normally tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of a process. Because the code is not tied to a specific address, it will execute correctly without page modification at a different address in each process that uses it.

When you use position-independent code, relocatable references are generated in the form of an indirection which will use data in the shared object's data segment. The result is that the text segment code will remain read-only, and all relocation updates will be applied to corresponding entries within the data segment. Refer to "Global Offset Table (Processor-Specific)" on page 155, "Procedure Linkage Table (SPARC)" on page 156, and "Procedure Linkage Table (x86)" on page 159 in the "Object Files" chapter for more details on the use of these two sections.

If a shared object is built from code that is not position-independent, the text segment will normally require a large number of relocations to be performed at runtime. Although the runtime linker is equipped to handle this, the system overhead this creates may cause serious performance degradation. A shared object that requires relocations against its text segment can be identified by using `dump(1)` and inspecting the output for any `TEXTREL` entry. For example:

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ dump -Lv libfoo.so.1 | grep TEXTREL
[9]     TEXTREL  0
```

Note – The value of the `TEXTREL` entry is irrelevant, its presence in a shared object indicates that text relocations exist.

A recommended practice to prevent the creation of a shared object that contains text relocations is to use the link-editor's `-z text` flag. This flag causes the link-editor to generate diagnostics indicating the source of any non position-independent code used as input, and results in a failure to generate the intended shared object. For example:

```
$ cc -o libfoo.so.1 -z text -G -R. foo.c
Text relocation remains                              referenced
    against symbol                     offset      in file
foo                                    0x0         foo.o
bar                                    0x8         foo.o
ld: fatal: relocations remain against allocatable but non-
writable sections
```

Here, two relocations would be generated against the text segment because of the non-position-independent code generated from the file `foo.o`. Where possible, these diagnostics will indicate any symbolic references that are required to carry out the relocations. In this case the relocations are against the symbols `foo` and `bar`.

Besides not using the `-K pic` option, the most common cause of creating text relocations when generating a shared object is by including hand written assembler code that has not been coded with the appropriate position-independent prototypes.

Note – By using the compiler's ability to generate an intermediate assembler file, the coding techniques used to enable position-independence can normally be revealed by experimenting with some simple test case source files.

A second form of the position-independence flag, `-K PIC`, is also available on some processors, and provides for a larger number of relocations to be processed at the cost of some additional code overhead (refer to `cc(1)` for more details).

Maximizing Shareability

As mentioned in the previous section "The Underlying System" on page 84, only a shared object's text segment is shared by all processes that use it, its data segment typically is not. Each process that uses a shared object usually generates a private memory copy of its entire data segment, as data items within the segment are written to. A goal then is to reduce the data segment, either by moving data elements that will never be written to the text segment, or by removing the data items completely.

The following sections cover a number of mechanisms that can be used to reduce the size of the data segment.

Move Read-Only Data to Text

Any data elements that are read-only should be moved into the text segment. This can be achieved using `const` declarations. For example, the following character string will reside in the *.data* section, which is part of the writable data segment:

```
char * rdstr = "this is a read-only string";
```

whereas, the following character string will reside 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";
```

Although reducing the data segment by moving read-only elements into the text segment is an admirable goal, moving data elements that require relocations may be counter productive. For example, given the array of strings:

```
char * rdstrs[] = { "this is a read-only string",
                    "this is another read-only string" };
```

it might at first seem that a better definition would be:

```
const char * const rdstrs[] = { ..... };
```

thereby insuring that the strings and the array of pointers to these strings are placed in a *.rodata* section. The problem with this definition is that even though the user perceives the array of addresses as read-only, these addresses must be relocated at runtime. This definition will therefore result in the creation of text relocations. This definition would be best represented as:

```
const char * rdstrs[] = { ..... };
```

so that the array strings are maintained in the read-only text segment, but the array pointers are maintained in the writable data segment where they can be safely relocated.

Note – Some compilers, when generating position-independent code, may be able to detect read-only assignments that will result in runtime relocations, and will arrange for placing such items in writable segments.

Collapse Multiply-Defined Data

Data can be reduced by collapsing multiply-defined data. For example, a program that has multiple occurrences of printing the same error messages may 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 library can be investigated using `strings(1)`. For example:

```
$ strings -10 libfoo.so.1 | sort | uniq -c | sort -rn
```

will generate 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.

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 will normally result in a corresponding reduction in the number of runtime relocations required.

Allocate Buffers Dynamically

Large data buffers should normally be allocated dynamically rather than being defined using permanent storage. Often this will result in an overall saving in memory, as only those buffers needed by the present invocation of an application will be allocated. Dynamic allocation also provides greater flexibility by allowing the buffer's size to change without effecting compatibility.

Minimizing Paging Activity

Many of the mechanisms discussed in the previous section "Maximizing Shareability" on page 86 will help reduce the amount of paging encountered when using shared objects. Here some additional generic software performance considerations are covered.

Any process that accesses a new page will cause a page fault. As this is an expensive operation, and because shared objects may be used by many processes, any reduction in the number of page faults generated by accessing a shared object will benefit the process and the system as a whole.

Organizing frequently used routines and their data to an adjacent set of pages will frequently improve performance because it improves the locality of reference. When a process calls one of these functions it may already be in memory because of its proximity to the other frequently used functions. Similarly, grouping interrelated functions will improve 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.

It is also advisable to isolate related functionality to its own shared object. The standard C library has historically been built containing many unrelated functions, and only rarely, for example, will any single executable use everything in this library. Because of its widespread use, it is also somewhat difficult to determine what set of functions are really the most frequently used. In contrast, when designing a shared object from scratch it is better to maintain only related functions within the shared object. This will improve locality of reference and usually has the side effect of reducing the object's overall size.

Relocations

In the section "Relocation Processing" on page 43 we covered the mechanisms by which the runtime linker must relocate dynamic executables and shared objects in order to create a runable process. The sections "Symbol Lookup" on page 45, and "When Relocations are Performed" on page 46 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, it does so by searching in each object, starting with the dynamic executable, and progressing through each shared object in the same order that the objects were mapped. In many instances the shared object that requires a symbolic relocation will turn out to be the provider of the symbol definition. If this is the case, and the symbol used for this relocation is not required as part of the shared object's interface, in other words no external objects reference this symbol, then this symbol is a strong candidate for conversion to a *static* or *automatic* variable. By making this conversion, the link-editor will incur just once the expense of processing any symbolic relocation against this symbol during the shared object's creation.

The only global data items that should be visible from a shared library are those that contribute to its user interface. However, frequently this is a hard goal to accomplish, as global data are often defined to allow reference from two or more functions located in different source files. Nevertheless, any reduction in the number of global symbols exported from a shared object will result in lower relocation costs and an overall performance improvement.

When Relocations are Performed

All *data* reference relocations must be carried out during process initialization prior to the application gaining control, whereas any *function* reference relocations can be *deferred* until the first instance of a function being called. By reducing the number of data relocations, the runtime initialization of a process will be reduced. Initialization relocation costs can also be deferred by converting data relocations into function relocations, for example, by returning data items via a functional interface. This conversion normally results in a perceived performance improvement as the initialization relocation costs are effectively spread throughout the process's lifetime. It is also possible that some of the functional interfaces will 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 next section "Copy Relocations" on page 91. This section examines a special, and somewhat expensive, relocation mechanism employed between dynamic executables and shared objects, and provides an example of how this relocation overhead can be avoided.

Copy Relocations

Shared objects are normally built with position-independent code. References to external data items from code of this type employs indirect addressing via a set of tables (refer to section "Position-Independent Code" on page 85 for more details). These tables are updated at runtime with the real address of the data items, which allows access to the data without the code itself being modified. Dynamic executables however, are generally not created from position-independent code. Therefore it would seem that any references to external data they make could only be achieved at runtime by modifying the code that makes the reference. Modifying any text segment is something to be avoided, and therefore a relocation technique is employed to solve this reference which is known as a *copy* relocation.

When the link-editor is used to build a dynamic executable, and a reference to a data item is found to reside in one of the dependent shared objects, space is allocated in the dynamic executable's *.bss* equivalent in size to the data item found in the shared object. This space is also assigned the same symbolic name as defined in the shared object. Along with this data allocation, the link-editor generates a special copy relocation record that will instruct the runtime linker to copy the data from the shared object to this allocated space within the dynamic executable. Because the symbol assigned to this space is global, it will be used to satisfy any references from any shared objects. The effect of this is that the dynamic executable inherits the data item, and any other objects within the process that make reference to this item will be bound to this copy. The original data from which the copy is made effectively becomes unused.

This mechanism is best explained with an example. This example 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]);
}
```

Here the application is using the function `error` to provide a focal point to obtain the system error message associated with the number `errnumb`.

Examining a dynamic executable built using this code, shows the implementation of the copy relocation in more detail:

```
$ cc -o prog main.c foo.c
$ nm -x prog | grep sys_
[36]  |0x00020910|0x00000260|OBJT |WEAK |0x0  |16 |sys_errlist
[37]  |0x0002090c|0x00000004|OBJT |WEAK |0x0  |16 |sys_nerr
$ dump -hv prog | grep bss
[16]    NOBI    WA-    0x20908   0x908    0x268   .bss
$ dump -rv prog

    **** RELOCATION INFORMATION ****

.rela.bss:
Offset      Symndx                Type              Addend

0x2090c     sys_nerr              R_SPARC_COPY      0
0x20910     sys_errlist           R_SPARC_COPY      0
..........
```

Here 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 will be copied from the C library by the runtime linker at process initialization. Thus, each application that uses these data will get a private copy of the data in its own data segment.

There are actually two problems with this technique. First, each application pays a performance penalty for the overhead of copying the data at run time, and secondly, the size of the data array `sys_errlist` has now become part of the C library's interface. If the size of this array were to change, presumably as new error messages are added, any dynamic executables that reference this array would 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)` illustrates this point. This function is implemented such that it will return 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]);
}
```

Our error routine in `foo.c` can now be simplified to use this functional interface, which in turn will remove 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, which allows the shared object the flexibility of changing the data without adversely effecting any dynamic executables that use it. Eliminating data items from a shared object’s interface will generally improve performance while making the shared object’s interface and code easier to maintain.

Although copy relocations should be avoided, `ldd(1)`, when used with either the `-d` or `-r` options, can be used to verify any that exist within a dynamic executable. For example, if the dynamic executable `prog` had originally been built against the shared library `libfoo.so.1` such that the following two copy relocations had been recorded:

```
$ nm -x prog | grep _size_
[36]    |0x000207d8|0x40|OBJT |GLOB |15  |_size_gets_smaller
[39]    |0x00020818|0x40|OBJT |GLOB |15  |_size_gets_larger
$ dump -rv size | grep _size_
0x207d8     _size_gets_smaller    R_SPARC_COPY      0
0x20818     _size_gets_larger     R_SPARC_COPY      0
```

and a new version of this shared object has been supplied which contains different data sizes for these symbols:

```
$ nm -x libfoo.so.1 | grep _size_
[26]    |0x00010378|0x10|OBJT |GLOB |8   |_size_gets_smaller
[28]    |0x00010388|0x80|OBJT |GLOB |8   |_size_gets_larger
```

then running `ldd(1)` against the dynamic executable will reveal:

```
$ ldd -d prog
    libfoo.so.1 =>   ./libfoo.so.1
    ...........
    copy relocation sizes differ: _size_gets_smaller
       (file prog size=40; file ./libfoo.so.1 size=10);
      ./libfoo.so.1 size used; possible insufficient data copied
    copy relocation sizes differ: _size_gets_larger
       (file prog size=40; file ./libfoo.so.1 size=80);
       ./prog size used; possible data truncation
```

Here `ldd(1)` informs us that the dynamic executable should copy as much data as the shared library has to offer, but can only accept as much as its allocated space allows.

Object Files 5

Introduction

This chapter describes the executable and linking format (ELF) of the object files produced by the assembler and link-editor. There are three main types of object files:

- A relocatable file holds code and data suitable to be linked with other object files to create an executable or shared object file.
- An executable file holds a program that is ready to execute. The file specifies how `exec(2)` creates a program's process image.
- A shared object file holds code and data suitable to be linked in two contexts. First, the link-editor can process it with other relocatable and shared object files to create other object files. Second, the runtime linker combines it with an executable file and other shared objects to create a process image.

The first section, "File Format" on page 98 focuses on the format of object files and how that pertains to building programs. The second section, "Dynamic Linking" on page 131 focuses on how the format pertains to loading programs. For background information, see "Link-Editor" on page 7.

Programs manipulate object files with the functions contained in the ELF access library, `libelf`. Refer to Section 3 of *SunOS Reference Manual* for a description of `libelf` contents.

5

File Format

As indicated, 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. Figure 5-1 shows an object file’s organization.

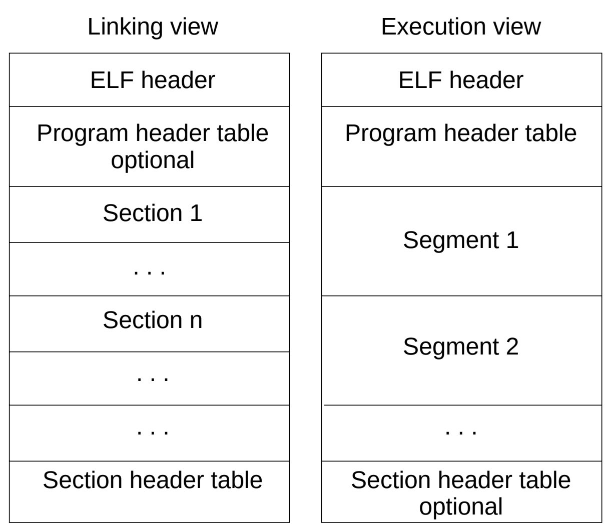

Figure 5-1 Object File Format

An *ELF header* resides at the beginning and holds a “road map” describing the file’s organization. *Sections* represent the smallest indivisible units that may be processed within an ELF file. *Segments* are a collection of sections that represent the smallest individual units that may be mapped to a memory image by `exec(2)` or by the runtime linker. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of *sections* appear in the first part of this chapter. The second part of this chapter discusses *segments* and the program execution view of the file.

A *section header table* contains information describing the file’s sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so forth. Files used during linking must have a section header table; other object files may or may not have one.

A *program header table*, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one.

Note – Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections; actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

Data Representation

As described here, the object file format supports various processors with 8-bit bytes and 32-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files therefore represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created.

Table 5-1 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 large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

All data structures that the object file format defines follow the “natural” size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, and so forth. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file.

Note – For portability, ELF uses no bit-fields.

ELF Header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore “extra” information. The treatment of “missing” information depends on context and will be specified if and when extensions are defined.

```
#define EI_NIDENT16
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
```

`e_ident`

The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file’s contents. Complete descriptions appear in “ELF Identification” on page 103 section.

`e_type`

This member identifies the object file type.

Table 5-2 ELF File Identifiers

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

Although the core file contents are unspecified, type `ET_CORE` is reserved to mark the file. Values from `ET_LOPROC` through `ET_HIPROC` (inclusive) are reserved for processor-specific semantics. Other values are reserved and will be assigned to new object file types as necessary.

`e_machine`

This member's value specifies the required architecture for an individual file.

Table 5-3 ELF Machines

Name	Value	Meaning
EM_NONE	0	No machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned below use the prefix EF_; a flag named WIDGET for the EM_XYZ machine would be called EF_XYZ_WIDGET.

e_version
This member identifies the object file version.

Table 5-4 ELF Versions

Name	**Value**	**Meaning**
EV_NONE	0	Invalid version
EV_CURRENT	>=1	Current version

The value 1 signifies the original file format; extensions will create new versions with higher numbers. The value of EV_CURRENT changes as necessary to reflect the current version number.

e_entry
This member gives 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
This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff
This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags
This member holds processor-specific flags associated with the file. Flag names take the form EF_*machine*_flag and are zero for both SPARC and x86.

e_ehsize
This member holds the ELF header's size in bytes.

e_phentsize
This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

`e_phnum`
This member holds the number of entries in the program header table. Thus the product of `e_phentsize` and `e_phnum` gives the table's size in bytes. If a file has no program header table, `e_phnum` holds the value zero.

`e_shentsize`
This member holds 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`
This member holds the number of entries in the section header table. Thus 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.

`e_shstrndx`
This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`. See "Section Header" on page 106 and "String Table" on page 118 for more information.

ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, 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 5-5 e_ident[ ] Identification Index *(1 of 2)*

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

Table 5-5 e_ident[ ] Identification Index *(2 of 2)*

Name	Value	Purpose
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size of e_ident[]

These indexes access bytes that hold the following values.

EI_MAG0 to EI_MAG3
A file's first 4 bytes hold a "magic number," identifying the file as an ELF object file.

Table 5-6 Magic Number

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

EI_CLASS
The next byte, e_ident[EI_CLASS], identifies the file's class, or capacity.

Table 5-7 File Class

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. Class ELFCLASS32 supports machines with files and virtual address spaces up to 4 gigabytes; it uses the basic types defined above.

Class ELFCLASS64 is reserved for 64-bit architectures. Its appearance here shows how the object file may change, but the 64-bit format is otherwise unspecified. Other classes will be defined as necessary, with different basic types and sizes for object file data.

EI_DATA
Byte e_ident[EI_DATA] specifies the data encoding of the processor-specific data in the object file. The following encodings are currently defined.

Table 5-8 Data Encoding

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	See below
ELFDATA2MSB	2	See below

More information on these encodings appears below. Other values are reserved and will be assigned to new encodings as necessary.

EI_VERSION
Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained in Table 5-4 on page 102 for e_version.

EI_PAD
This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and set to zero; programs that read object files should ignore them. The value of EI_PAD will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. As described above, class ELFCLASS32 files use objects that occupy 1, 2, and 4 bytes. Under the defined encodings, objects are represented as shown below. Byte numbers appear in the upper left corners.

Encoding ELFDATA2LSB specifies 2's complement values, with the least significant byte occupying the lowest address.

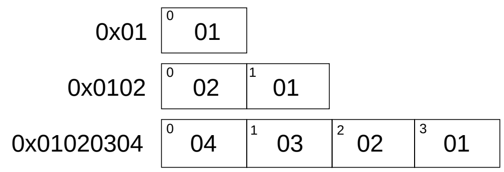

Figure 5-2 Data Encoding ELFDATA2LSB

Encoding `ELFDATA2MSB` specifies 2's complement values, with the most significant byte occupying the lowest address.

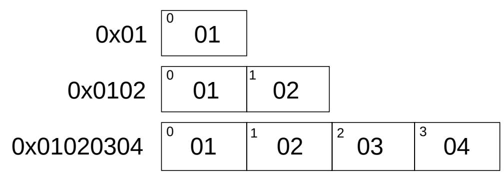

Figure 5-3 Data Encoding `ELFDATA2MSB`

Section Header

An object file's section header table lets you locate all file's sections. The section header table is an array of `Elf32_Shdr` structures as described below. A section header table index is a subscript into this array. The ELF header's `e_shoff` member gives the byte offset from the beginning of the file to the section header table; `e_shnum` tells how many entries the section header table contains; `e_shentsize` gives the size in bytes of each entry.

Some section header table indexes are reserved; an object file does not have sections for these special indexes.

Table 5-9 Special Section Indexes

Name	**Value**
`SHN_UNDEF`	`0`
`SHN_LORESERVE`	`0xff00`
`SHN_LOPROC`	`0xff00`

Table 5-9 Special Section Indexes

Name	Value
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff

SHN_UNDEF
This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol "defined" relative to section number SHN_UNDEF is an undefined symbol.

Note – Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. That is, if the e_shnum member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5. The contents of the initial entry are specified later in this section.

SHN_LORESERVE
This value specifies the lower bound of the range of reserved indexes.

SHN_LOPROC through SHN_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHN_ABS
This value specifies 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_HIRESERVE
This value specifies the upper bound 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. That is, 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, object files' sections satisfy several conditions:

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may 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:

```
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;
```

`sh_name`
: This member specifies the name of the section. Its value is an index into the section header string table section (see "String Table" on page 118), giving the location of a null-terminated string.

`sh_type`
: This member categorizes the section's contents and semantics. Section types and their descriptions are in Table 5-10 on page 110.

`sh_flags`
: Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions are in Table 5-10 on page 110.

`sh_addr`

If the section is to appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

`sh_offset`

This member gives the byte offset from the beginning of the file to the first byte in the section. Section type, `SHT_NOBITS` described below, occupies no space in the file, and its `sh_offset` member locates the conceptual placement in the file.

`sh_size`

This member gives 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` may have a nonzero size, but it occupies no space in the file.

`sh_link`

This member holds a section header table index link, whose interpretation depends on the section type. Table 5-14 on page 114 describes the values.

`sh_info`

This member holds extra information, whose interpretation depends on the section type. Table 5-14 on page 114 below describes the values.

`sh_addralign`

Some sections have address alignment constraints. For example, if a section holds a double-word, the system must ensure double-word alignment for the entire section. That is, the value of `sh_addr` must be congruent to 0, modulo the value of `sh_addralign`. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

`sh_entsize`

Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's sh_type member specifies the section's semantics.

Table 5-10 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_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL
This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS
The section holds information defined by the program, whose format and meaning are determined solely by the program.

SHT_SYMTAB and SHT_DYNSYM
: These sections hold a symbol table. Typically, SHT_SYMTAB provides symbols for link editing. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. Consequently, an object file may also contain a SHT_DYNSYM section, which holds a minimal set of dynamic linking symbols, to save space. See “Symbol Table” on page 119 for details.

SHT_STRTAB
: The section holds a string table. An object file may have multiple string table sections. See “String Table” on page 118 for details.

SHT_RELA
: The section holds relocation entries with explicit addends, such as type Elf32_Rela for the 32-bit class of object files. An object file may have multiple relocation sections. See “Relocation” on page 124 for details.

SHT_HASH
: The section holds a symbol hash table. All dynamically linkable object files must contain a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See “Hash Table” on page 161 for details.

SHT_DYNAMIC
: The section holds information for dynamic linking. Currently, an object file may have only one dynamic section, but this restriction may be relaxed in the future. See “Dynamic Section” on page 149 for details.

SHT_NOTE
: The section holds information that marks the file in some way. See “Note Section” on page 138 for details.

SHT_NOBITS
: A section of this type 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
: The section holds relocation entries without explicit addends, such as type Elf32_Rel for the 32-bit class of object files. An object file may have multiple relocation sections. See “Relocation” on page 124 for details.

SHT_SHLIB
This section type is reserved but has unspecified semantics. Programs that contain a section of this type do not conform to the ABI.

SHT_LOPROC through SHT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

SHT_LOUSER
This value specifies the lower bound of the range of indexes reserved for application programs.

SHT_HIUSER
This value specifies the upper bound of the range of indexes reserved for application programs. Section types between SHT_LOUSER and SHT_HIUSER may 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. This entry holds the following:

Table 5-11 Section Header Table Entry: Index 0

Name	Value	Note
sh_name	0	No name
sh_type	SHT_NULL	Inactive
sh_flags	0	No flags
sh_addr	0	No address
sh_offset	0	No file offset
sh_size	0	No size
sh_link	SHN_UNDEF	No link information
sh_info	0	No auxiliary information
sh_addralign	0	No alignment
sh_entsize	0	No entries

A section header's sh_flags member holds 1-bit flags that describe the section's attributes. Defined values are in Table 5-12 on page 113; other values are reserved.

Table 5-12 Section Attribute Flags

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xf0000000

If a flag bit is set in sh_flags, the attribute is "on" for the section. Otherwise, the attribute is "off" or does not apply. Undefined attributes are set to zero.

SHF_WRITE
The section contains data that should be writable during process execution.

SHF_ALLOC
The section 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
The section contains executable machine instructions.

SHF_MASKPROC
All bits included in this mask are reserved for processor-specific semantics.

Two members in the section header, sh_link and sh_info, hold special information, depending on section type.

Table 5-13 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 string 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.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
other	SHN_UNDEF	0

Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Table 5-14 Special Sections *(1 of 2)*

Name	Type	Attribute
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	None
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	See “.got” on page 116
.hash	SHT_HASH	SHF_ALLOC

Table 5-14 Special Sections *(2 of 2)*

Name	**Type**	**Attribute**
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	See ".interp" on page 116
.note	SHT_NOTE	None
.plt	SHT_PROGBITS	See ".plt" on page 116
.rel*name*	SHT_REL	See ".relname and .relaname" on page 116
.rela*name*	SHT_RELA	See ".relname and .relaname" on page 116
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	None
.strtab	SHT_STRTAB	See ".strtab" on page 117
.symtab	SHT_SYMTAB	See ".symtab" on page 117
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

`.bss`
This section holds 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`
This section holds version control information.

`.data` and `.data1`
These sections hold initialized data that contribute to the program's memory image.

`.dynamic`
This section holds dynamic linking information.

`.dynstr`
This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries.

`.dynsym`
This section holds the dynamic linking symbol table. See “Symbol Table” on page 119 for details.

`.fini`
This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

`.got`
This section holds the global offset table. See “Global Offset Table (Processor-Specific)” on page 155 for more information.

`.hash`
This section holds a symbol hash table. See “Hash Table” on page 161 for more information.

`.init`
This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the program entry point.

`.interp`
This section holds the path name of a program interpreter. See “Program Interpreter” on page 147 for more information.

`.note`
This section holds information in the format that “Note Section” on page 138 describes.

`.plt`
This section holds the procedure linkage table. See “Procedure Linkage Table (SPARC)” on page 156 and “Procedure Linkage Table (x86)” on page 159 for more information.

`.rel`*`name`* and `.rela`*`name`*
These sections hold relocation information, as “Relocation” on page 124 describes. If the file has a loadable segment that includes relocation, the sections’ attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for `.text` normally would have the name `.rel.text` or `.rela.text`.

`.rodata` and `.rodata1`
These sections hold read-only data that typically contribute to a non-writable segment in the process image. See “Program Header” on page 132 for more information.

`.shstrtab`
This section holds section names.

`.strtab`
This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section’s attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

`.symtab`
This section holds a symbol table, as “Symbol Table” on page 119 describes. If the file has a loadable segment that includes the symbol table, the section’s attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

`.text`
This section holds the “text” or executable instructions of a program.

Section names with a dot (`.`) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not in the list above. An object file may have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for `e_machine`. For example, .Foo.psect is the psect section defined by the FOO archictcture. Existing extensions use their historical names.

Preexisting Extensions

.conflict	.liblist	.lit8	.sdata
.debug	.line	.reginfo	.stab
.gptab	.lit4	.sbss	.tdesc

String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold 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; its section header's `sh_size` member would contain zero. Nonzero indexes are invalid for an empty string table.

A section header's `sh_name` member holds an index into the section header string table section, as designated by the `e_shstrndx` member of the ELF header. The following figures show a string table with 25 bytes and the strings associated with various indexes.

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Figure 5-4 String Table

Table below shows the strings of the string table above:

Table 5-15 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 may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified later in this section.

Table 5-16 Symbol Table Initial Entry

Name	**Value**
STN_UNDEF	0

A symbol table entry has the following format:

```
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;
```

`st_name`

This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is nonzero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

External C symbols have the same names in C and in object files' symbol tables.

`st_value`

This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, and so forth. See “Symbol Values” on page 124.

`st_size`

Many symbols have associated sizes. For example, a data object’s size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

`st_info`

This member specifies the symbol’s type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values.

```
#define ELF32_ST_BIND(i)    ((i)>>4)
#define ELF32_ST_TYPE(i)    ((i)&0xf)
#define ELF32_ST_INFO(b,t)  (((b)<<4+((t)&0xf))
```

`st_other`

This member currently holds 0 and has no defined meaning.

`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 5-10 on page 110

A symbol’s binding determines the linkage visibility and behavior.

Table 5-17 Symbol Binding, ELF32_ST_BIND

Name	**Value**
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL
Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

STB_GLOBAL
Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.

STB_WEAK
Weak symbols resemble global symbols, but their definitions have lower precedence.

STB_LOPROC through STB_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Global and weak symbols differ in two major ways, as described in "Generating the Output Image" on page 33.

- When the link-editor combines several relocatable object files, it does not allow multiple definitions of STB_GLOBAL symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link-editor honors the global definition and ignores the weak ones. Similarly, if a common symbol exists (that is, a symbol with the st_index field holding SHN_COMMON), the appearance of a weak symbol with the same name does not cause an error. The link-editor uses the common definition and ignores the weak one.
- When the link-editor searches archive libraries (see "Archive Processing" on page 12, it extracts archive members that contain definitions of undefined or tentative, global symbols. The member's definition may be either a global or a weak symbol. The link-editor does *not* extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with STB_LOCAL binding precede the weak and global symbols. As "Section Header" on page 106 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 provides a general classification for the associated entity.

Table 5-18 Symbol Types, ELF32_ST_TYPE

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

STT_NOTYPE
The symbol type is not specified.

STT_OBJECT
The symbol is associated with a data object, such as a variable, an array, and so forth.

STT_FUNC
The symbol is associated with a function or other executable code.

STT_SECTION
The 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 associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS, and it precedes the other STB_LOCAL symbols for the file, if it is present.

STT_LOPROC through STT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Function symbols (those with type STT_FUNC) in shared object files have special significance. When another object file references a function from a shared object, the link-editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than STT_FUNC will not be referenced automatically through the procedure linkage table.

If a symbol's value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, and references to the symbol continue to "point" to the same location in the program. Some special section index values give other semantics.

SHN_ABS
: The symbol has an absolute value that will not change because of relocation.

SHN_COMMON
: The 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. That is, the link-editor will allocate the storage for the symbol at an address that is a multiple of st_value. The symbol's size tells how many bytes are required.

SHN_UNDEF
: This section table index means the symbol is undefined. When the link-editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

As mentioned above, the symbol table entry for index 0 (STN_UNDEF) is reserved; it holds the following:

Table 5-19 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. That is, `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.

Relocation

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. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Relocation entries are these data.

```
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;
```

`r_offset`

This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

`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 would hold the symbol table index of the function being called. If the index is `STN_UNDEF`, the undefined symbol index, the relocation uses 0 as the symbol value. Relocation types are processor-specific; descriptions of their behavior appear below. When the text below refers to a relocation entry's relocation type or symbol table index, it means the result of applying `ELF32_R_TYPE` or `ELF32_R_SYM`, respectively, to the entry's `r_info` member.

```
#define ELF32_R_SYM(i)((i)>>8)
#define ELF32_R_TYPE(i)((unsigned char)(i))
#define ELF32_R_INFO(s,t)(((s)<<8)+(unsigned char)(t))
```

`r_addend`

This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As shown above, only `Elf32_Rela` entries contain an explicit addend. Entries of type `Elf32_Rel` store an implicit addend in the location to be modified. SPARC uses `Elf32_Rela` entries and x86 uses `Elf32_Rel` entries.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_info` and `sh_link` members, described in "Section Header" on page 106 earlier, specify these relationships. Relocation entries for different object files have slightly different interpretations for the `r_offset` member.

- In relocatable files, `r_offset` holds a section offset. That is, the relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.

- In executable and shared object files, `r_offset` holds a virtual address. To make these files' relocation entries more useful for the runtime linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of `r_offset` changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

Relocation Types (Processor Specific)

Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link-editor merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation:

`A`

means the addend used to compute the value of the relocatable field.

`B`

means the base address at which a shared object is loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address is different. See "Program Header" on page 132 for more information about the base address.

`G`

means 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 155 for more information.

`L`

means the place (section offset or address) of the *procedure linkage table* entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link-editor builds the initial procedure linkage table, and the runtime linker modifies the entries during execution. See "Procedure Linkage Table (SPARC)" on page 156 or "Procedure Linkage Table (x86)" on page 159 for more information.

P

means the place (section offset or address) of the storage unit being relocated (computed using r_offset).

S

means the value of the symbol whose index resides in the relocation entry.

Relocation entries apply to bytes (*byte8*), half-words (*half16*), or words (the others). In any case, 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. SPARC processor uses only Elf32_Rela relocation entries with explicit addends. Thus the r_addend member serves as the relocation addend.

SPARC Relocation Types

Field names in the following table tell whether the relocation type checks for overflow. A calculated relocation value may be larger than the intended field, and a relocation type may verify (V) the value fits or truncate (T) the result. As an example, V-simm13 means that the computed value may not have significant, nonzero bits outside the simm13 field.

Table 5-20 SPARC Relocation Types *(1 of 2)*

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

Table 5-20 SPARC Relocation Types *(2 of 2)*

Name	**Value**	**Field**	**Calculation**
R_SPARC_LO10	12	T-simm13	(S + A) & 0x3ff
R_SPARC_GOT10	13	T-simm13	G & 0x3ff
R_SPARC_GOT13	14	V-simm13	G
R_SPARC_GOT22	15	T-simm22	G >> 10
R_SPARC_PC10	16	T-simm13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	None	None
R_SPARC_GLOB_DAT	20	V-word32	S + A
R_SPARC_JMP_SLOT	21	None	See below
R_SPARC_RELATIVE	22	V-word32	B + A
R_SPARC_UA32	23	V-word32	S + A

Some relocation types have semantics beyond simple calculation:

R_SPARC_GOT10
: This relocation type resembles R_SPARC_LO10, except it refers to the address of the symbol's global offset table entry and additionally instructs the link-editor to build a global offset table.

R_SPARC_GOT13
: This relocation type resembles R_SPARC_13, except it refers to the address of the symbol's global offset table entry and additionally instructs the link-editor to build a global offset table.

R_SPARC_GOT22
: This relocation type resembles R_SPARC_22, except it refers to the address of the symbol's global offset table entry and additionally instructs the link-editor to build a global offset table.

R_SPARC_WPLT30
: This relocation type resembles R_SPARC_WDISP30, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link-editor to build a procedure linkage table

R_SPARC_COPY

The link-editor creates this relocation type for dynamic linking. Its 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 91

R_SPARC_GLOB_DAT

This relocation type resembles R_SPARC_32, except it sets a global offset table entry to the address of the specified symbol. The special relocation type allows you to determine the correspondence between symbols and global offset table entries.

R_SPARC_JMP_SLOT

The link-editor creates this relocation type for dynamic linking. Its 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

The link-editor creates this relocation type for dynamic linking. Its offset member gives the location within a shared object that contains a value representing a relative address. The runtime linker computes the corresponding virtual address by adding the virtual address at which the shared object is loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_SPARC_UA32

This relocation type resembles R_SPARC_32, except it refers to an unaligned word. That is, 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.

x86 Relocation Types

Field names in the following table tell whether the relocation type checks for `overflow`. A calculated relocation value may be larger than the intended field, and a relocation type may verify (V) the value fits or truncate (T) the result. As an example, V-simm13 means that the computed value may not have significant, nonzero bits outside the simm13 field.

Table 5-21 x86 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 - P
`R_386_PLT32`	`4`	word32	L + A - P
`R_386_COPY`	`5`	none	none
`R_386_GLOB_DAT`	`6`	word32	S
`R_386_JMP_SLOT`	`7`	word32	S
`R_386_RELATIVE`	`8`	word32	B + A
`R_386_GOTOFF`	`9`	word32	S + A - GOT
`R_386_GOTPC`	`10`	word32	GOT + A - P

Some relocation types have semantics beyond simple calculation:

`R_386_GOT32`

This relocation type computes the distance from the base of the global offset table to the symbol's global offset table entry. It also tells the link-editor to build a global offset table.

`R_386_PLT32`

This relocation type computes the address of the symbol's procedure linkage table entry and tells the link-editor to build a procedure linkage table.

`R_386_COPY`

The link-editor creates this relocation type for dynamic linking. Its 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 91

`R_386_GLOB_DAT`
: This relocation type is used to set a global offset table entry to the address of the specified symbol. The special relocation type lets one determine the correspondence between symbols and global offset table entries.

`R_386_JMP_SLOT`
: The link-editor creates this relocation type for dynamic linking. Its 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`
: The link-editor creates this relocation type for dynamic linking. Its offset member gives the location within a shared object that contains a value representing a relative address. The runtime linker computes the corresponding virtual address by adding the virtual address at which the shared object is loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

`R_386_GOTOFF`
: This relocation type computes the difference between a symbol's value and the address of the global offset table. It also tells the link-editor to build the global offset table.

`R_386_GOTPC`
: This relocation type resembles `R_386_PC32`, except it uses the address of the global offset table in its calculation. The symbol referenced in this relocation normally is `_GLOBAL_OFFSET_TABLE_`, which also tells the link-editor to build the global offset table.

Dynamic Linking

This section describes the object file information and system actions that create running programs. Some information here applies to all systems; information specific to one processor resides in sections marked accordingly.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images. A process image has segments that contain its text, data, stack, and so on. The major subsections of this section are:

- "Program Header" describes object file structures that are directly involved in program execution. The primary data structure, a program header table, locates segment images in the file and contains other information needed to create the memory image of the program.
- "Program Loading (Processor-Specific)" on page 140 describes the information used to load a program into memory.
- "Runtime Linker" on page 147 describes the information used to specify and resolve symbolic references among the object files of the process image.

Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information 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 137.

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. See "ELF Header" on page 100 for more information.

```
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;
```

`p_type`
This member tells what 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 5-22 on page 134.

`p_offset`
This member gives the offset from the beginning of the file at which the first byte of the segment resides.

`p_vaddr`
This member gives the virtual address at which the first byte of the segment resides in memory.

`p_paddr`
On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because the system ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

`p_filesz`
This member gives the number of bytes in the file image of the segment; it may be zero.

`p_memsz`
This member gives the number of bytes in the memory image of the segment; it may be zero.

`p_flags`
This member gives flags relevant to the segment. Defined flag values appear below.

`p_align`
As "Program Loading" describes later, 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`.

Some entries describe process segments; others give supplementary information and do not contribute to the process image. Segment entries may appear in any order, except as explicitly noted below. Defined type values follow; other values are reserved for future use.

Table 5-22 Segment Types, p_type

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL

The array element is unused; other members' values are undefined. This type lets the program header table contain ignored entries.

PT_LOAD

The array element specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the "extra" bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the p_vaddr member.

PT_DYNAMIC

The array element specifies dynamic linking information. See "Dynamic Section" on page 149 for more information.

PT_INTERP

The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See "Program Interpreter" on page 147 for further information.

PT_NOTE
The array element specifies the location and size of auxiliary information. See “Note Section” on page 138 below for details.

PT_SHLIB
This segment type is reserved but has unspecified semantics.

PT_PHDR
The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See “Program Interpreter” on page 147 for further information.

PT_LOPROC through 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. That is, a file’s program header table may contain only those elements relevant to its contents.

Base Address

Executable and shared object files have a *base address*, which is the lowest virtual address associated with the memory image of the program’s object file. One use of the base address is to relocate the memory image of the program during dynamic linking.

An executable or shared object file’s base address is calculated during execution from three values: the memory load address, the maximum page size, and the lowest virtual address of a program’s loadable segment. As “Program Loading (Processor-Specific)” on page 140 describes, the virtual addresses in the program headers might not represent the actual virtual addresses of the program’s memory image. To compute the base address, you determine the memory address associated with the lowest p_vaddr value for a PT_LOAD segment. You then obtain the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the kind of file being loaded into memory, the memory address might or might not match the p_vaddr values.

Segment Permissions

A program to be loaded by the system must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segments' memory images, it gives access permissions as specified in the p_flags member. All bits included in the PF_MASKPROC mask are reserved for processor-specific semantics.

Table 5-23 Segment Flag Bits, p_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 type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. In no case, however, will a segment have write permission unless it is specified explicitly. The following figure shows both the exact flag interpretation and the allowable flag interpretation.

Table 5-24 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 comprises one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The diagrams below illustrate segment contents in general terms. The order and membership of sections within a segment may vary; moreover, processor-specific constraints may alter the examples below.

Text segments contain read-only instructions and data, typically including the following sections described earlier in this chapter. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

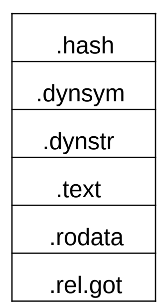

Figure 5-5 Text Segment

Data segments contain writable data and instructions, typically including the following sections.

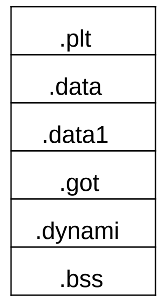

Figure 5-6 Data Segment

A PT_DYNAMIC program header element points at the .dynamic section, as explained in “Dynamic Section” on page 149 later. The .got and .plt sections also hold information related to position-independent code and dynamic linking. Although the .plt appears in a text segment above, it may reside in a text or a data segment, depending on the processor. See “Global Offset Table (Processor-Specific)” on page 155, “Procedure Linkage Table (SPARC)” on page 156 and “Procedure Linkage Table (x86)” on page 159 for details.

As previously described in “Section Header”, the .bss section has the type SHT_NOBITS. Although it occupies no space in the file, it contributes to the segment’s memory image. Normally, these uninitialized data reside at the end of the segment, thereby making p_memsz larger than p_filesz in the associated program header element.

Note Section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, and so forth. Sections of type SHT_NOTE and program header elements of type PT_NOTE can be used for this purpose. The note information in sections and program header elements holds any number of entries, each of which is an array of 4-byte words in the format of the target processor. Labels are shown on Figure 5-7 to help explain note information organization, but they are not part of the specification.

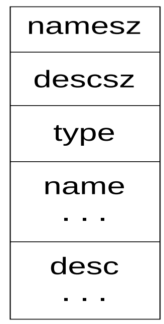

Figure 5-7 Note Information

`namesz` and `name`
: The first `namesz` bytes in `name` contain a null-terminated character representation of the entry's owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as "XYZ Computer Company," as the identifier. If no name is present, `namesz` contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in `namesz`.

`descsz` and `desc`
: The first `descsz` bytes in `desc` hold the note descriptor. If no descriptor is present, `descsz` contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in `descsz`.

`type`
: This word gives the interpretation of the descriptor. Each originator controls its own types; multiple interpretations of a single type value may exist. Thus, a program must recognize both the name and the type to "understand" a descriptor. Types currently must be nonnegative.

To illustrate, the following note segment holds two entries.

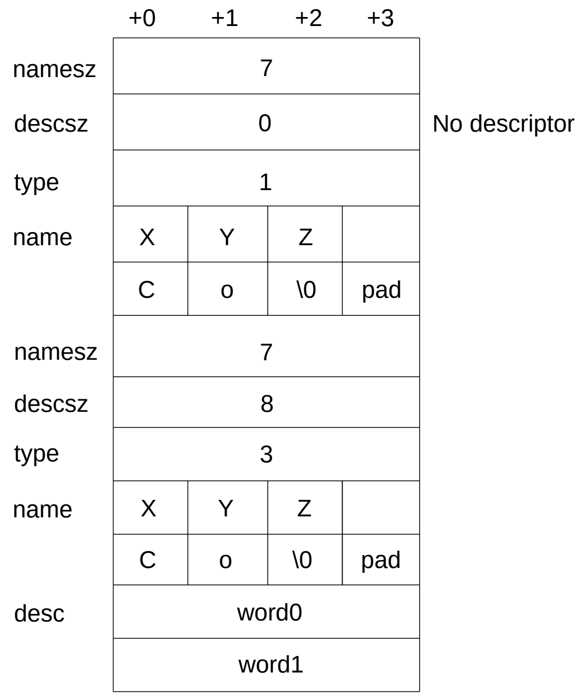

Figure 5-8 Example Note Segment

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.

Program Loading (Processor-Specific)

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When—and if—the system physically reads the file depends on the program's execution behavior, system load, and so forth. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for SPARC segments are congruent modulo 64 K (`0x10000`). Virtual addresses and file offsets for x86 segments are congruent modulo 4 K (`0x1000`). By aligning segments to the maximum page size, the files are suitable for paging regardless of physical page size. The following example presents the SPARC version.

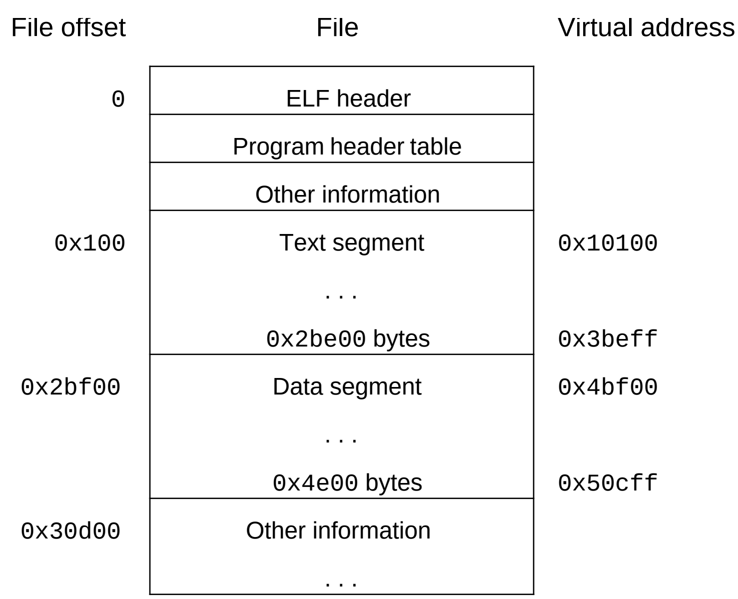

Figure 5-9 SPARC Executable File (64 K alignment)

Table 5-25 SPARC Program Header Segments (64 K alignment)

Member	**Text**	**Data**
`p_type`	`PT_LOAD`	`PT_LOAD`
`p_offset`	`0x100`	`0x2bf00`
`p_vaddr`	`0x10100`	`0x4bf00`
`p_paddr`	Unspecified	Unspecified
`p_filesize`	`0x2be00`	`0x4e00`

Table 5-25 SPARC Program Header Segments (64 K alignment)

Member	**Text**	**Data**
p_memsz	0x2be00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

The following example presents the x86 version.

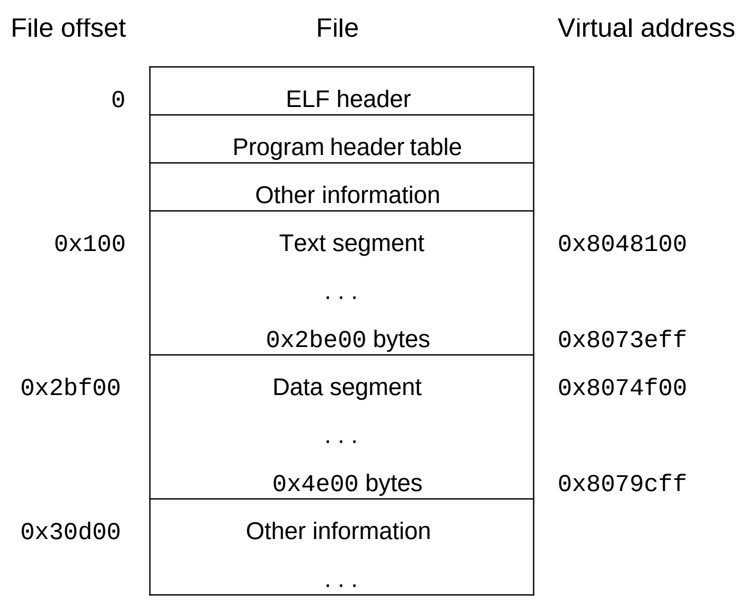

Figure 5-10 x86 Executable File (4 K alignment)

Table 5-26 x86 Program Header Segments (4 K alignment)

Member	**Text**	**Data**
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	Unspecified	Unspecified
p_filesize	0x2be00	0x4e00

Table 5-26 x86 Program Header Segments (4 K alignment)

Member	Text	Data
p_memsz	0x2be00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x1000	0x1000

Although 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 may contain file information not relevant to the running process. Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the examples above, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus, 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 them is unspecified. The memory image for this program follows, assuming 4 Kbyte (`0x1000`) pages. For simplicity, these examples illustrates only one page size.

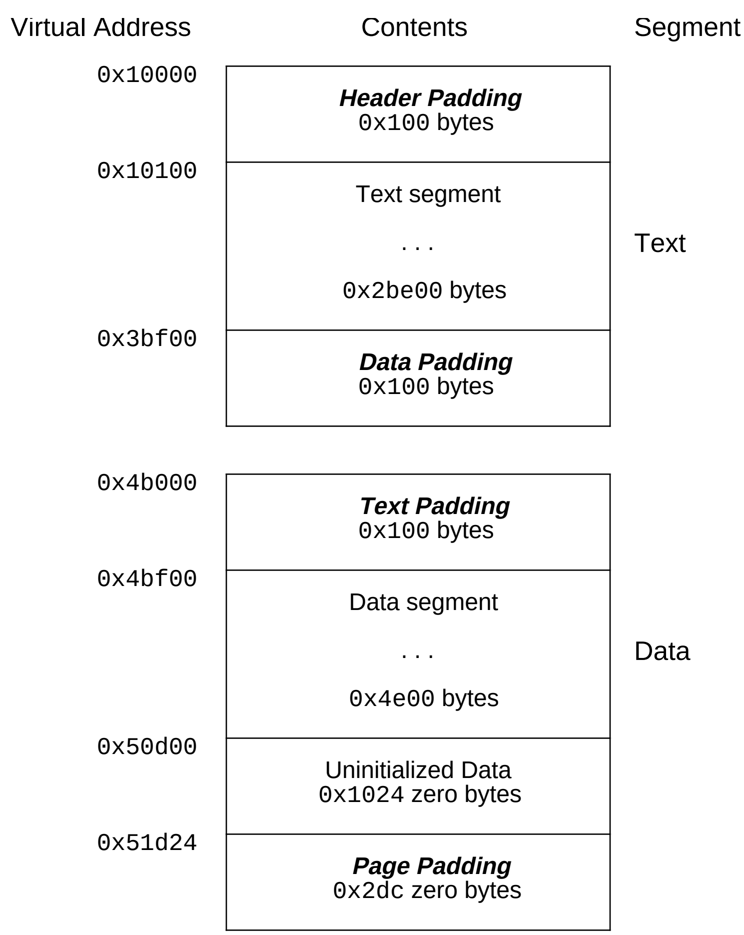

Figure 5-11 SPARC Process Image Segments

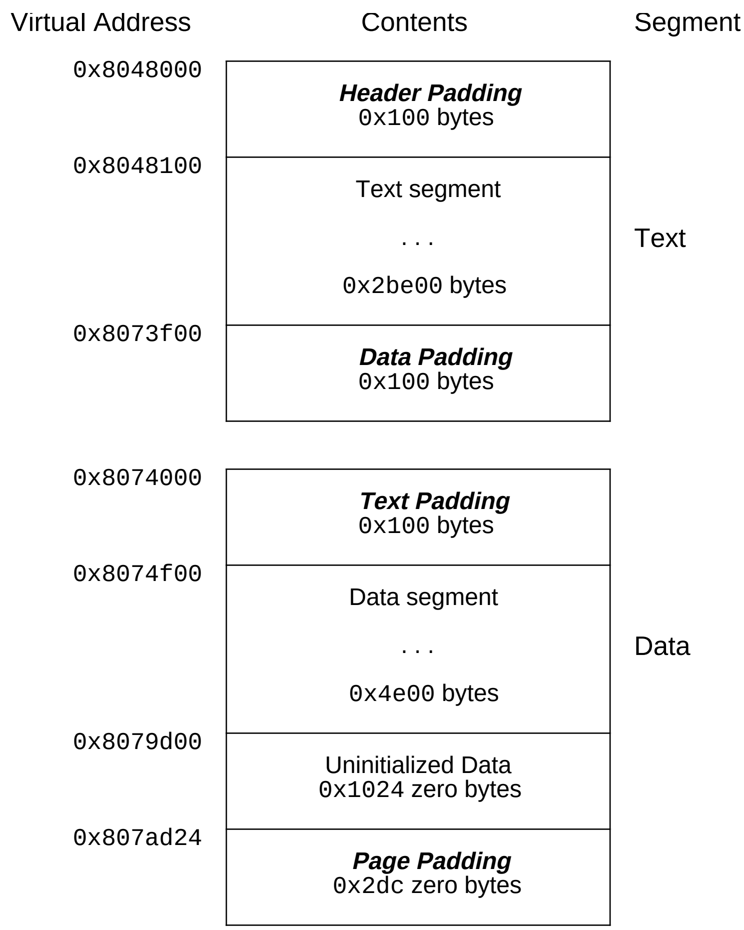

Figure 5-12 x86 Process Image Segments

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 build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. (For background, see “Link-Editor” on page 7.) This lets a segment’s virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments’ *relative positions*. 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 table also illustrates the base address computations.

Table 5-27 Example SPARC Shared Object Segment Addresses

Source	**Text**	**Data**	**Base Address**
File	0x200	0x2a400	0x0
Process 1	0xc0000200	0xc002a400	0xc0000000
Process 2	0xc0010200	0xc003c400	0xc0010000
Process 3	0xd0020200	0xd004a400	0xd0020000
Process 4	0xd0030200	0xd005a400	0xd0030000

Table 5-28 Example x86 Shared Object Segment Addresses

Source	**Text**	**Data**	**Base Address**
File	0x200	0x2a400	0x0
Process 1	0x80000200	0x8002a400	0x80000000
Process 2	0x80081200	0x800ab400	0x80081000
Process 3	0x900c0200	0x900ea400	0x900c0000
Process 4	0x900c6200	0x900f0400	0x900c6000

Program Interpreter

An executable file may 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. That is, instead of using segment images of the original executable files, the system composes a memory image for the interpreter. It then is the interpreter's responsibility to receive control from the system and provide an environment for the application program.

The interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by mmap(2) and related services. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.
- An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter's virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.

Runtime Linker

When building an executable file that uses dynamic linking, the link-editor adds a program header element of type PT_INTERP to an executable file, telling the system to invoke the runtime linker as the program interpreter. exec() and the runtime linker cooperate to create the process image for the program, which entails the following actions:

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;

- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the runtime linker;
- Calling any `.init` section provided in the objects mapped; see “Initialization and Termination Functions” on page 163
- Transferring control to the program, making it look as if the program had received control directly from `exec`().

The link-editor also constructs various data that assist the runtime linker for executable and shared object files. As shown above in “Program Header,” these data reside in loadable segments, making them available during execution. (Once again, recall the exact segment contents are processor-specific.)

- A `.dynamic` section with type `SHT_DYNAMIC` holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The `.hash` section with type `SHT_HASH` holds a symbol hash table.
- The `.got` and `.plt` sections with type `SHT_PROGBITS` hold two separate tables: the global offset table and the procedure linkage table. Sections below explain how the runtime linker uses and changes the tables to create memory images for object files.

As explained in “Program Loading (Processor-Specific)” on page 140, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file’s program header table. The runtime linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment (see `exec`(2)) contains a variable named `LD_BIND_NOW` with a non-null value, the runtime linker processes all relocation before transferring control to the program. For example, each of the environment entries

```
LD_BIND_NOW-=1
LD_BIND_NOW-=on
LD_BIND_NOW-=off
```

specifies this behavior. Otherwise, LD_BIND_NOW either is absent from the environment or has a null value. The runtime linker can evaluate procedure linkage table entries lazily, so avoiding resolution and relocation overhead for functions that are not called. See "Procedure Linkage Table (SPARC)" on page 156 and "Procedure Linkage Table (x86)" on page 159 for more information.

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.

```
typedef struct {
    Elf32_Sword    d_tag;
    union {
        Elf32_Word     d_val;
        Elf32_Addr     d_ptr;
    } d_un;
} Elf32_Dyn;

extern  Elf32_Dyn  _DYNAMIC[];
```

For each object with this type, d_tag controls the interpretation of d_un.

d_val
: These Elf32_Word objects represent integer values with various interpretations.

d_ptr
: These Elf32_Addr objects represent program virtual addresses. As mentioned previously, 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.

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 may appear but is not required.

Table 5-29 Dynamic Array Tags, d_tag *(1 of 2)*

Name	**Value**	**d_un**	**Executable**	**Shared Object**
DT_NULL	0	Ignored	Mandatory	Mandatory
DT_NEEDED	1	d_val	Optional	Optional
DT_PLTRELSZ	2	d_val	Optional	Optional
DT_PLTGOT	3	d_ptr	Optional	Optional
DT_HASH	4	d_ptr	Mandatory	Mandatory
DT_STRTAB	5	d_ptr	Mandatory	Mandatory
DT_SYMTAB	6	d_ptr	Mandatory	Mandatory
DT_RELA	7	d_ptr	Mandatory	Optional
DT_RELASZ	8	d_val	Mandatory	Optional
DT_RELAENT	9	d_val	Mandatory	Optional
DT_STRSZ	10	d_val	Mandatory	Mandatory
DT_SYMENT	11	d_val	Mandatory	Mandatory
DT_INIT	12	d_ptr	Optional	Optional
DT_FINI	13	d_ptr	Optional	Optional
DT_SONAME	14	d_val	Ignored	Optional
DT_RPATH	15	d_val	Optional	Ignored
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

Table 5-29 Dynamic Array Tags, d_tag *(2 of 2)*

Name	**Value**	**d_un**	**Executable**	**Shared Object**
DT_FILTER	24	d_ptr	Unspecified	Optional
DT_LOPROC	0x70000000	Unspecified	Unspecified	Unspecified
DT_HIPROC	0x7fffffff	Unspecified	Unspecified	Unspecified

DT_NULL
An entry with a DT_NULL tag marks the end of the _DYNAMIC array.

DT_NEEDED
This element holds the string table offset of a null-terminated string, giving the name of a needed dependency. The offset is an index into the table recorded in the DT_STRTAB entry. See “Shared Object Dependencies” on page 154 for more information about these names. The dynamic array may contain multiple entries with this type. These entries’ relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ
This element holds the total size, in bytes, of the relocation entries associated with the procedure linkage table. If an entry of type DT_JMPREL is present, a DT_PLTRELSZ must accompany it.

DT_PLTGOT
This element holds an address associated with the procedure linkage table and/or the global offset table.

DT_HASH
This element points to the symbol hash table, described in “Hash Table” on page 161. This hash table refers to the symbol table indicated by the DT_SYMTAB element.

DT_STRTAB
This element holds the address of the string table, described in the first part of this chapter. Symbol names, dependency names, and other strings required by the runtime linker reside in this table.

DT_SYMTAB
This element holds the address of the symbol table, described in the first part of this chapter, with Elf32_Sym entries for the 32-bit class of files.

DT_RELA
This element holds the address of a relocation table, described in the first part of this chapter. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class. An object file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link-editor catenates those sections to form a single table. Although the sections remain independent in the object file, the runtime linker sees a single table. When the runtime linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have DT_RELASZ and DT_RELAENT elements. When relocation is "mandatory" for a file, either DT_RELA or DT_REL may occur (both are permitted but not required).

DT_RELASZ
This element holds the total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT
This element holds the size, in bytes, of the DT_RELA relocation entry.

DT_STRSZ
This element holds the size, in bytes, of the string table.

DT_SYMENT
This element holds the size, in bytes, of a symbol table entry.

DT_INIT
This element holds the address of the initialization function, discussed in "Initialization and Termination Functions" on page 163 later.

DT_FINI
This element holds the address of the termination function, discussed in "Initialization and Termination Functions" on page 163 later.

DT_SONAME
This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. See Section , "Shared Object Dependencies," on page 154 for more information about these names.

DT_RPATH
This element holds the string table offset of a null-terminated search library search path string, discussed in "Shared Objects With Dependencies" on page 76. The offset is an index into the table recorded in the DT_STRTAB entry.

DT_SYMBOLIC
This element's presence in a shared object library alters the runtime linker's symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the runtime linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the runtime linker then searches the executable file and other shared objects as usual.

DT_REL
This element is similar to DT_RELA, except its table has implicit addends, such as Elf32_Rel for the 32-bit file class. If this element is present, the dynamic structure must also have DT_RELSZ and DT_RELENT elements.

DT_RELSZ
This element holds the total size, in bytes, of the DT_REL relocation table.

DT_RELENT
This element holds the size, in bytes, of the DT_REL relocation entry.

DT_PLTREL
This member specifies the type of relocation entry to which the procedure linkage table refers. The d_val member holds DT_REL or DT_RELA, as appropriate. All relocations in a procedure linkage table must use the same relocation.

DT_DEBUG
This member is used for debugging.

DT_TEXTREL
This member's absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the runtime linker can prepare accordingly.

DT_JMPREL
If present, this entry's d_ptr member holds the address of relocation entries associated solely with the procedure linkage table. Separating these relocation entries lets the runtime linker ignore them during process initialization, if lazy binding is enabled. If this entry is present, the related entries of types DT_PLTRELSZ and DT_PLTREL must also be present.

DT_FILTER
Holds the string table offset of a null-terminated string that names an object. The symbol table of this (shared) object acts as a filter for the symbol table of the named object.

DT_LOPROC through DT_HIPROC
Values in this inclusive range are reserved for processor-specific semantics.

Except for the DT_NULL element at the end of the array and the relative order of DT_NEEDED elements, entries may appear in any order. Tag values not appearing in the table are reserved.

Shared Object Dependencies

When the runtime linker creates the memory segments for an object file, the dependencies (recorded in DT_NEEDED entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the runtime linker builds a complete process image. When resolving symbolic references, the runtime linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the DT_NEEDED entries (in order), then at the second level DT_NEEDED entries, and so on.

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

Names in the dependency list are copies either of the DT_SONAME strings or the path names of the shared objects used to build the object file.

Global Offset Table (Processor-Specific)

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and shareability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries (see "Relocation" on page 124 for more information). After the system creates memory segments for a loadable object file, the runtime linker processes the relocation entries, some of which will be type `R_SPARC_GLOB_DAT` (for SPARC) or `R_386_GLOB_DAT` (for x86) referring to the global offset table. The runtime linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link-editor builds an object file, the runtime linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The runtime linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the 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 allows a program, such as the runtime linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the runtime linker, because it must initialize itself without relying on other programs to relocate its memory image.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For SPARC and x86 processors, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table.

```
extern Elf32_Addr_GLOBAL_OFFSET_TABLE_[];
```

The symbol _GLOBAL_OFFSET_TABLE_ may reside in the middle of the .got section, allowing both negative and nonnegative "subscripts" into the array of addresses.

Procedure Linkage Table (SPARC)

As the global offset table converts position-independent address calculations to absolute locations, the procedure linkage table converts position-independent function calls to absolute locations. The link-editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. So, the link-editor puts the program transfer control to entries in the procedure linkage table. On SPARC architectures, procedure linkage tables reside in private data. The runtime linker determines the destinations' absolute addresses 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.

The first four procedure linkage table entries are reserved. (The original contents of these entries are unspecified, despite the example, below.) 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 procedure linkage table entry. Except the first four entries, the relocation type is R_SPARC_JMP_SLOT, the relocation offset specifies the address of the first byte of the associated procedure linkage table entry, and the symbol table index refers to the appropriate symbol.

To illustrate procedure linkage tables, the figure below shows four entries: two of the four initial reserved entries, the third is a call to name1, and the fourth is a call to name2. The example assumes the entry for name2 is the table's last entry and shows the following nop instruction. The left column shows the instructions from the object file before dynamic linking. The right column demonstrates a possible way the runtime linker might *fix* the procedure linkage table entries.

Code Example 5-1 SPARC Procedure Linkage Table Example

Object File	**Memory Segment**

<pre>.PLT0:
    unimp
    unimp
    unimp
.PLT1:
    unimp
    unimp
    unimp
    ...</pre>
	<pre>.PLT0:
	    save %sp,-64,%sp
	    call runtime-linker
	    nop
	.PLT1:
	    .word identification
	    unimp
	    unimp
	    ...</pre>

<pre>   ...
.PLT101:
    sethi   (.-.PLT0),%g1
    ba,a   .PLT0
    nop
.PLT102:
    sethi   (.-.PLT0),%g1
    ba,a   .PLT0
    nop</pre>
	<pre> ...
	.PLT101:
	    sethi (.-.PLT0),%g1
	    sethi %hi(name1),%g1
	    jmp1 %g1+%lo(name1),%g0
	.plt102:
	    sethi (.-.PLT0),%g1
	    sethi %hi(name2),%g1
	    jmp1 %g1+%lo(name2),%g0</pre>

<pre> nop</pre>	<pre> nop</pre>

Following the steps below, the runtime linker and program jointly resolve the symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the runtime linker is not specified.

1. When first creating the memory image of the program, the runtime linker changes the initial procedure linkage table entries, making them transfer control to one of the runtime linker's own routines. It also stores a word of *identification* information in the second entry. When it receives control, it can examine this word to find what object called it.

2. All other procedure linkage table entries initially transfer to the first entry, letting the runtime linker gain control at the first execution of each table entry. For example, the program calls `name1`, 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. In this example, `&g1` contains `0x12f000` when the runtime linker receives control.

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 name1. Relocation entry 101 has type `R_SPARC_JMP_SLOT`, its offset specifies the address of .`PLT101`, and its symbol table index refers to `name1`. 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.

Although the runtime linker does not have to create the instruction sequences under the *Memory Segment* column, it might. If it did, some points deserve more explanation.

- To make the code reentrant, 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 two words to convert an entry. It updates each word automatically. Reentrancy is achieved by first overwriting the `nop` with the `jmp1` instruction, and then patching the `ba,a` to be `sethi`. If a reentrant function call happens between the two word updates, the `jmp1` resides in the delay slot of the `ba,a` instruction, and cancels the delay instruction. So, 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 above) needs a delay instruction for its jmp1. The required, trailing nop fills this delay slot.

The LD_BIND_NOW environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes R_SPARC_JMP_SLOT relocation entries (procedure linkage table entries) before transferring control to the program. If LD_BIND_NOW is null, the runtime linker evaluates linkage table entries on the first execution of each table entry.

Procedure Linkage Table (x86)

As for SPARC, the procedure linkage table redirects position-independent function calls to absolute locations. The link-editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. So, the link-editor has the program transfer control to entries in the procedure linkage table. On x86 architectures, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The runtime linker determines the destinations' absolute addresses 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.

```
.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
       ...
```

```
.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
       ...
```

Following the steps below, the runtime linker and program cooperate to resolve the symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the runtime linker sets the second and third entries in the global offset table to special values. Steps below explain these values.

2. If the procedure linkage table is position-independent, the address of the global offset table must be in `%ebx`. Each shared object file in the process image has its own procedure linkage table, and control transfers to a procedure linkage table entry only from within the same object file. So, the calling function must set the global offset table base register before it calls the procedure linkage table entry.

3. For example, the program calls `name1`, which transfers control to the label `.PLT1`.

4. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following `pushl` instruction, not the real address of `name1`.

5. So, 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. This is because the jmp instruction at .PLT1 jumps to name1 instead of falling through to the pushl instruction.

The LD_BIND_NOW environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes R_386_JMP_SLOT relocation entries (procedure linkage table entries) before transferring control to the program. If LD_BIND_NOW is null, the runtime linker evaluates linkage table entries on the first execution of each table entry.

Hash Table

A hash table of Elf32_Word objects supports symbol table access. Labels appear below to help explain the hash table organization, but they are not part of the specification.

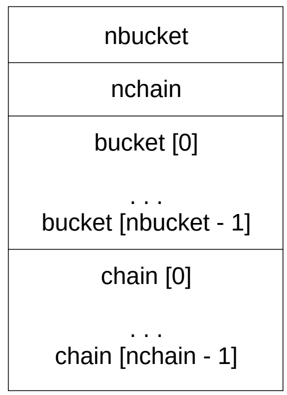

Figure 5-13 Symbol Hash Table

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 accepts a symbol name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value *x* for some name, `bucket [x%nbucket]` gives an index y into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[y]` gives the next symbol table entry with the same hash value. One can follow the `chain` links until either the selected symbol table entry holds the desired name or the `chain` entry contains the value `STN_UNDEF`.

```
unsigned long
elf_Hash(const unsigned char *name)
{
    unsigned long h = 0, g;

    while (*name)
    {
    h = (h << 4) + *name++;
    if (g = h & 0xf0000000)
        h ^= g >> 24;
            h &= ~g;
    }
    return h;
```

Initialization and Termination Functions

After the runtime linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. These initialization functions are called in the reverse of the order at which they are encountered.

Similarly, shared objects may have termination functions, which are executed with the `atexit(3C)` mechanism after the base process begins its termination sequence. Refer to `atexit(3C)` for more information. These termination functions are called in the order they are encountered.

Shared objects designate their initialization and termination functions through the `DT_INIT` and `DT_FINI` entries in the dynamic structure, described in "Dynamic Section" above. Typically, the code for these functions resides in the `.init` and `.fini` sections, mentioned in "Section Header" on page 106 earlier.

Note – Although the `atexit(3C)` termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls `_exit()` or if the process dies because it received a signal that it neither caught nor ignored.

Mapfile Option 6

Introduction

The link-editor automatically and intelligently maps input sections from relocatable objects to segments within the output file image. The `-M` option allows you to change the default mapping provided by the link-editor.

In particular, this mapfile option allows you to:

- Declare segments and specify values for segment attributes such as segment type, permissions, addresses, length, and alignment.
- Control mapping of input sections to segments by specifying the attribute values necessary in a section to map to a specific segment (the attributes are section name, section type, and permissions) and by specifying which object file(s) the input sections should be taken from, if necessary.
- Declare a global-absolute symbol that is assigned a value equal to the size of a specified segment (by the link-editor) and that can be referenced from object files.

The mapfile option allows users of *ifiles* (an option previously available to `ld(1)` that used link-editor command language directives) to convert to mapfiles. All other facilities previously available for *ifiles*, other than those mentioned above, are not available with the mapfile option.

Note – When using the mapfile option, be aware that you can easily create `a.out` files that do not execute. The link-editor knows how to produce a correct `a.out` without the use of the mapfile option. The mapfile option is intended for system programming use, not application programming use.

Using the Mapfile Option

To use the mapfile option, you must:

- Enter the mapfile directives into a file, for example *mapfile*
- Supply the following option on the `ld(1)` command line:

  ```
  -M mapfile
  ```

If the *mapfile* is not in your current directory, include the full path name; no default search path exists.

Mapfile Structure and Syntax

You can enter three types of directives into a mapfile:

- Segment declarations.
- Mapping directives.
- Size-symbol declarations.

Each directive can span more than one line and can have any amount of white space (including new-lines) as long as it is followed by a semicolon. You can enter zero or more directives in a mapfile. (Entering zero directives causes the link-editor to ignore the mapfile and use its own defaults.) Typically, segment declarations are followed by mapping directives, that is, you would 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, explained later), the segment is given default attributes as explained below. Such segment is then an “implicitly declared segment.”

Size-symbol declarations 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_name*s and *segment_name*s follow the same rules as C identifiers where a period (.) is treated as a letter (for example, `.bss` is a legal name).
- *section_name*s, *segment_name*s, *file_name*s, and *symbol_name*s are case sensitive; everything else is not case sensitive.
- Spaces (or new-lines) may appear anywhere except before a *number* or in the middle of a name or value.
- Comments beginning with # and ending at a new-line may appear anywhere that a space may appear.

Segment Declarations

A segment declaration creates a new segment in the `a.out` or changes the attribute values of an existing segment. (An existing segment is one that you previously defined or one of the three built-in segments described below.)

A segment declaration has the following syntax:

segment_name = {*segment_attribute_value*}`*;`

For each *segment_name*, you can specify any number of *segment_attribute_value*s 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 follows:

Table 6-1 Mapfile Segment Attributes

Attribute	Value
segment_type	LOAD NOTE
segment flags	?[R][W][X][O]
virtual_address	V*number*
physical_address	P*number*
length	L*number*
alignment	A*number*

There are three built-in segments with the following default attribute values:

- text (LOAD, ? RX, no *virtual_address*, *physical_address*, or *length* specified, *alignment* values set to defaults per CPU type)
- data (LOAD, ?RWX, no *virtual_address*, *physical_address*, or *length* specified, *alignment* values set to defaults per CPU type)
- note (NOTE)

The link-editor behaves as if these segments are declared before your mapfile is read in. See “Mapfile Option Defaults” on page 175 for more information.

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, or A and the *number.*
- The *segment_type* value can be either LOAD or NOTE.
- The *segment_type* value 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*.
- 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. Also, even though implicitly declared segments default to "no length limit," machine memory limitations still apply.

- `LOAD` segments can have an explicitly specified *virtual_address* value and / or *physical_address* value, as well as a maximum segment *length* value.
- If a segment has a *segment_flags* value of `?` with nothing following, the value defaults to not readable, not writable, and not executable.
- The *alignment* value is used in calculating the virtual address of the beginning of the segment. This alignment only affects the segment for which it is specified; other segments still have the default alignment unless their alignments are also changed.
- If any of the *virtual_address*, *physical_address*, or *length* attribute values are not set, the link-editor calculates these values as it builds the `a.out`.
- If an *alignment* value is not specified for a segment, it is set to the built-in default. (The default differs from one CPU to another and may even differ between kernel versions. You should check the appropriate documentation for these numbers).
- 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.

The `?O` flag lets the user control the order of sections in the final relocatable object, executable file or shared object. This flag should be used in conjunction with the -xF option to the compiler(s). 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 now called `.text%`*function_name*.

For example, a file containing three functions `main()`, `foo()` and `bar()` when compiled with the `-xF` option will yield an object file with text for the three functions in sections called `.text%main`, `.text%foo` and `.text%bar`. Because the -xF option forces one function per section, the use of `?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;
```

If the order of function definitions in the source file is `main`, `foo` and `bar`, then the final executable will contain 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 static function `bar()` exists in files `a.o` and `b.o`, and function `bar` from file `a.o` is to be placed before function `bar` from file `b.o`, then the mapfile entries should read:

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

Although the syntax allows for the entry:

```
text: .text%bar: a.o b.o;
```

This entry does not guarantee that function bar from file `a.o` will be placed before function bar from file `b.o`. Do not use the second format; the results are not reliable.

Note – If a *virtual_address* value is specified, the segment is placed at that virtual address. For the system kernel this creates a correct result. For files that start via `exec(2)`, this method creates an incorrect `a.out` file because the segments do not have correct offsets relative to their page boundaries.

Mapping Directives

A mapping directive tells 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 *a.out*, 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(s) via the *file_name* substitutable. The section attributes and their valid values are as follows:

Table 6-2 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 when entering mapping directives:

- You must choose at most one *section_type* from the *section_type*s listed above. The *section_type*s listed above are built-in types. For more information on *section_type*s, see “Section Header” on page 106.
- 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 linker checks to make sure that the flag is not set. No spaces are allowed between the question mark, exclamation mark(s), and the individual flags that make up the *section_flags* value.
- *file_name* may be any legal file name and can be of the form *archive_name(component_name)*, for example, `/usr/lib/usr/libc.a(printf.o)`. A file name may be of the form `*file_name` (see next bullet item). Note that the link-editor does not check the syntax of file names.
- If a *file_name* is of the form `*file_name`, the link-editor simulates a `basename`(1) on the file name from the command line and uses that to match against the mapfile *file_name*. In other words, the *file_name* from the mapfile only needs to match the last part of the file name from the command line. (See “Mapping Example” on page 173.)
- 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 may appear for a particular output segment, for example, the following set of directives is legal:

```
S1 : $PROGBITS;
S1 : $NOBITS;
```

Entering more than one mapping directive line for a segment is the only way to specify multiple values of a section attribute.

- A section can match more than one entrance criteria. In this case, the first segment encountered in the mapfile with that entrance criteria is used, for example, if a mapfile reads:

```
S1 : $PROGBITS;
S2 : $PROGBITS;
```

the `$PROGBITS` sections are mapped to segment `S1`.

Size-Symbol Declarations

Size-symbol declarations let you 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, although the link-editor does not check the syntax of the *symbol_name*.

Mapping Example

Following is an example of 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 would actually appear in the mapfile.

Code Example 6-1 User-Defined Mapfile

```
1.      elephant : .bss : peanuts.o *popcorn.o;
2.      monkey : $PROGBITS ?AX;
3.      monkey : .bss;
4.      monkey = LOAD V0x80000000 L0x4000;
5.      donkey : .bss;
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 `.bss` sections from the files `peanuts.o` and `popcorn.o`. Note that `*popcorn.o` matches any `popcorn.o` file that may have been supplied to the link-edit; the file need not be in the current directory. On the other hand, if `/var/tmp/peanuts.o` were supplied to the link-edit, it would not match `peanuts.o` because it is not preceded by a `*`.

The implicitly declared segment `monkey` (line 2) receives all sections that are both `$PROGBITS` and allocatable-executable (`?AX`), as well as all sections (not already in the segment `elephant`) with the name `.bss` (line 3). The `.bss` sections entering the `monkey` segment need not be `$PROGBITS` or allocatable-executable because the *section_type* and *section_flags* values were 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" `.bss` 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` (since the *segment_type* attribute value does not change, no warning is issued), *virtual_address* value to `0x80000000` and maximum *length* value to `0x4000`.

Line 5 implicitly declares the `donkey` segment. The entrance criteria are designed to route all `.bss` 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` (since 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 wanted to change the order of the directives to avoid warnings, the example would appear as follows:

```
1.      elephant : .bss : peanuts.o *popcorn.o;
4.      monkey = LOAD V0x80000000 L0x4000;
2.      monkey : $PROGBITS ?AX;
3.      monkey : .bss;
6.      donkey = ?RX A0x1000;
5.      donkey : .bss;
7.      text = V0x80008000;
```

This order eliminates all warnings.

Mapfile Option Defaults

The link-editor defines three built-in segments (`text`, `data`, and `note`) with default *segment_attribute_value*s and corresponding default mapping directives as described in “Segment Declarations” on page 167. Even though the link-editor does not use an actual “mapfile” to store the defaults, the model of a “default mapfile” helps to illustrate what happens when the link-editor encounters your mapfile.

The example below 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 : $PROGBITS ?A!W;
data = LOAD ?RWX;
data : $PROGBITS ?AW;
data : $NOBITS ?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 `LOAD`able segment, the segment is placed before any `LOAD`able 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 mentioned above, is used by the link-editor when the command is executed. Then, if the mapfile option is used, the link-editor parses the mapfile to augment and/or override certain values in the default map structure.

A typical (although somewhat simplified) map structure is illustrated in Figure6-1. The ‘‘Entrance Criteria’’ boxes correspond to the information in the default mapping directives and 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 in circles.

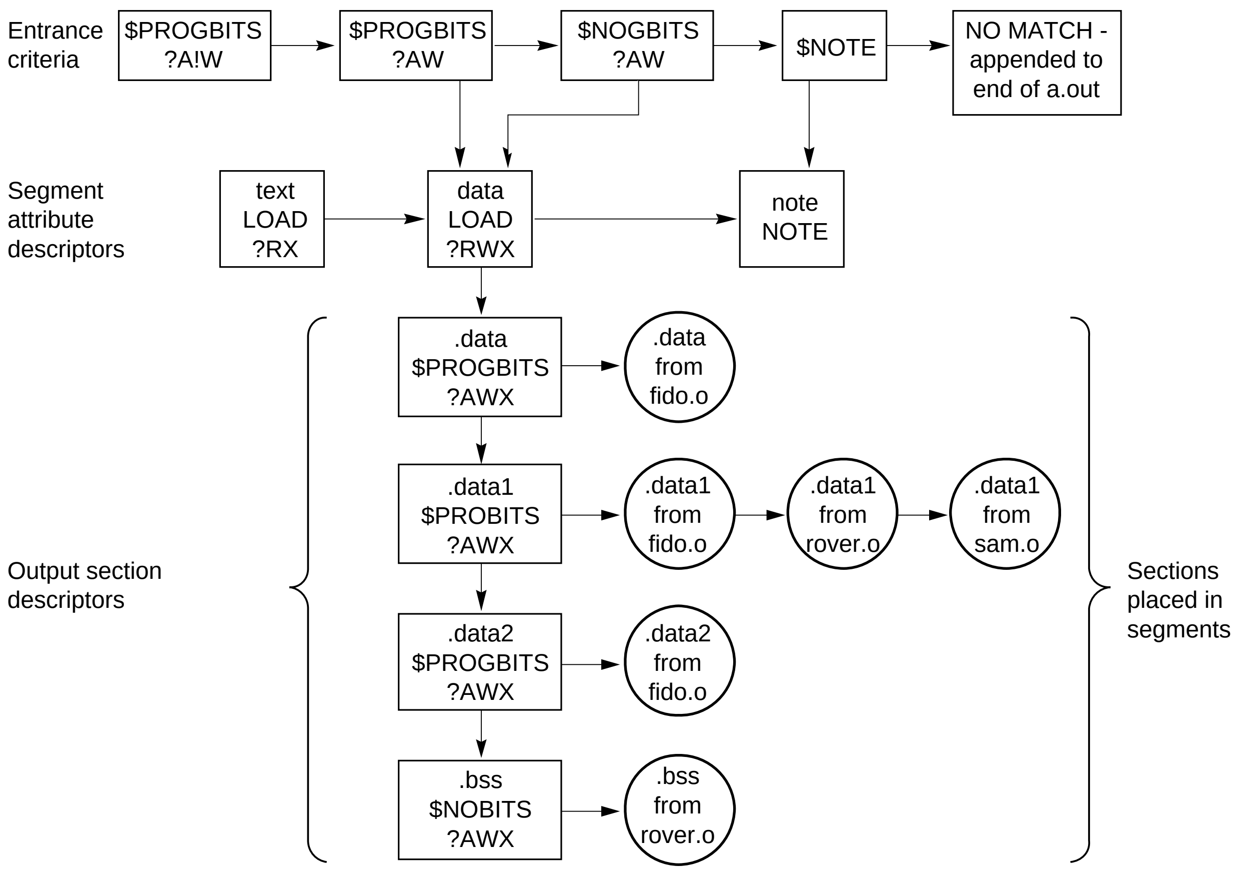

Figure 6-1 Simple Map Structure

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 6-1, for a section to fall into the `text` segment it 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 may be either `X` or `!X` (in the *section_flags* value) since 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 `a.out` file after all other segments. (No program header entry is created for this information. See "Program Header" on page 132 for more 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 `$PROGBITS`, and a *section_flags* value of `?AWX` falls into the second Entrance Criteria box in Figure 6-1, placing it in the `data` segment. The section matches the second Output Section Descriptor box exactly (`.data1`, `$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 6-1 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 (that is, between `SHT_LOUSER` and `SHT_HIUSER`, as described in the "Section Header" on page 106) it is treated as a `$PROGBITS` section. Note that 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 have been 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_type*s can only map output sections produced by the link-editor to segments.

Error Messages

When the mapfile option is used, the link-editor can return the following types of error messages:

warning:

Does not stop execution of the link-editor nor does it prevent the link-editor from producing a viable `a.out`.

fatal:

Stops execution of the link-editor at the point the fatal error occurred.

Warnings

The following conditions produce warnings:

- A *physical_address* or a *virtual_address* value or a *length* value appears for any segment other than a `LOAD` segment. (The directive is ignored.)
- A second declaration line exists for the same segment that changes an attribute value(s). (The second declaration overrides the original.)
- An attribute value(s) (*segment_type* and / or *segment_flags* for `text` and `data`; *segment_type* for `note`) was changed for one of the built-in segments

- An attribute value(s) (*segment_type*, *segment_flags*, *length* and / or *alignment*) was changed for a segment created by an implicit declaration. If only the `?O` flag has been added then the change of attribute value warning will not be generated.
- An entrance criteria was not met. If the `?O` flag has been turned on and if none of the input sections met an entrance criteria, the warning is generated.

Fatal Errors

The following conditions produce fatal errors:

- Specifying more than one `-M` option on the command line
- A mapfile cannot be opened or read
- A syntax error is found in the mapfile

Note – The link-editor does not return an error if a *file_name*, *section_name*, *segment_name* or *symbol_name* does not conform to the rules under the ‘‘Mapfile Structure and Syntax’’ section unless this condition produces a syntax error. For instance, if a name begins with a special character and this name is at the beginning of a directive line, the link-editor returns an error. If the name is a *section_name* (appearing within the directive), the link-editor does not return an error.

- More than one *segment_type*, *segment_flags*, *virtual_address*, *physical_address*, *length*, or *alignment* value appears on a single declaration line
- You attempt to manipulate either the `interp` segment or `dynamic` segment in a mapfile

---

Note – The `interp` and `dynamic` segments are special built-in segments that you cannot change in any way.

- A segment grows larger than the size specified by a your *length* attribute value
- A user-defined *virtual_address* value causes a segment to overlap the previous segment
- More than one *section_name*, *section_type*, or *section_flags* value appears on a single directive line

- A flag and its complement (for example, `A` and `!A`) appear on a single directive line.

Link-Editor Quick Reference A

The following sections provide a simple overview, or *cheat sheet*, of the most commonly used link-editor scenarios (refer to “Link-Editing” on page 2 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 the compiler driver `cc(1)`, this being the most common mechanism of invoking the link-editor (refer to “Using a Compiler Driver” on page 9).

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 (refer to “Input File Processing” on page 11). Shared objects that follow a naming convention of `lib`*x*`.so`, and archive libraries that follow a naming convention of `lib`*x*`.a`, may be input using the `-l` option (refer to “Library Naming Conventions” on page 14). This provides additional flexibility in allowing search paths to be specified using the `-L` option (refer to“Directories Searched by the Link-Editor” on page 16).

The link-editor basically operates in one of two modes, *static* or *dynamic*.

Static Mode

This mode is selected when the `-dn` option is used, and allows for the creation of relocatable objects and static executables. Under this mode only relocatable objects and archive libraries are acceptable forms of input. Use of the `-l` option will result in a search for archive libraries.

Building a Relocatable Object

- Use the `-dn` and `-r` options:

```
$ cc -dn -r -o temp.o file1.o file2.o file3.o .....
```

Building a Static Executable

- Use the `-dn` option *without* the `-r` option:

```
$ cc -dn -o prog file1.o file2.o file3.o .....
```

Note – The `-a` option is available to indicate the creation of a static executable, however, the use of `-dn` *without* a `-r` implies `-a`.

Dynamic Mode

This is the default mode of operation for the link-editor. It can be enforced by specifying the `-dy` option, but is implied when *not* using the `-dn` option. Under this mode relocatable objects, shared objects and archive libraries are acceptable forms of input. Use of the `-l` option will result in a directory search, where each directory is searched for a shared object, and if none is found the same directory is then searched for an archive library. A search for archive libraries only, can be enforced by using the `-B static` option (refer to "Linking with a Mix of Shared Objects and Archives" on page 15).

Building a Shared Object

- Use the `-dy` and `-G` option.
- Input relocatable objects should be built from position-independent code, and use the `-z text` option to enforce this requirement (refer to "Position-Independent Code" on page 85).
- Use a *versioned* name for the shared object to allow for future upgrades (refer to "Versioning" on page 73).

- If the shared object being generated has dependencies on any other shared objects, and these dependencies do not reside in `/usr/lib`, record their pathname in the output file using the `-R` option (refer to “Shared Objects With Dependencies” on page 76).

The following example combines the above points:

```
$ cc -c -o foo.o -Kpic foo.c
$ cc -dy -G -o libfoo.so.1 -z text -R /home/lib foo.o -L. -lbar
```

- If the shared object being generated will be used as input to another link-edit, record within it the shared object’s runtime name using the `-h` option (refer to “Recording a Shared Object Name” on page 69). Make the shared object available to the compilation environment by creating a file system link to a non-versioned shared object name (refer to “Coordination Of Binding Requirements” on page 74):

```
$ cc -dy -G -o libfoo.so.1 -z text -h libfoo.so.1 foo.o
$ ln -s libfoo.so.1 libfoo.so
```

- Consider the performance implications of the shared object; maximize shareability (refer to page 86) and minimize paging activity (refer to page 89), reduce relocation overhead, especially by minimizing symbolic relocations (refer to “Relocations” on page 90), and allow access to data via functional interfaces (refer to “Copy Relocations” on page 91).

Building a Dynamic Executable

- Use the `-dy` option without the `-G` option.
- If the dynamic executable being generated has dependencies on any other shared objects, and these dependencies do not reside in `/usr/lib`, record their pathname in the output file using the `-R` option (refer to “Directories Searched by the Runtime Linker” on page 18).

The following example combines the above points:

```
$ cc -dy -o prog -R /home/lib -L. -lfoo
```

Index

T

U

V

W