Linker and Libraries Guide

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043-1100
U.S.A.

Part No: 802-6319
August 1997

Copyright 1997 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, SunSoft, SunDocs, SunExpress, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227–14(g)(2)(6/87) and FAR 52.227–19(6/87), or DFAR 252.227–7015(b)(6/95) and DFAR 227.7202–3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 1997 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, Californie 94303-4900 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, SunSoft, SunDocs, SunExpress, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Please Recycle

Contents

Preface ix

1. Introduction 1

Overview 1

Link-Editing 2

Runtime Linking 3

Dynamic Executables 4

Shared Objects 4

Related Topics 4

Dynamic Linking 4

Application Binary Interfaces 5

Support Tools 5

New Features 5

2. Link-Editor 7

Overview 7

Invoking the Link-Editor 8

Direct Invocation 8

Using a Compiler Driver 9

Specifying the Link-Editor Options 9

Input File Processing 10

Archive Processing 11

Shared Object Processing 12

Linking with Additional Libraries 13

Initialization and Termination Sections 17

Symbol Processing 18

Symbol Resolution 19

Undefined Symbols 24

Tentative Symbol Order Within the Output File 27

Defining Additional Symbols 28

Reducing Symbol Scope 33

Generating the Output Image 37

Debugging Aids 38

3. Runtime Linker 43

Overview 43

Locating Shared Object Dependencies 44

Directories Searched by the Runtime Linker 44

Relocation Processing 46

Symbol Lookup 47

When Relocations are Performed 48

Relocation Errors 49

Loading Additional Objects 51

Initialization and Termination Routines 52

Security 53

Runtime Linking Programming Interface 54

Loading Additional Objects 55

Relocation Processing 57

Obtaining New Symbols 63

Debugging Aids 67

4. Shared Objects 71

Overview 71

Naming Conventions 72

Recording a Shared Object Name 73

Shared Objects with Dependencies 76

Dependency Ordering 77

Shared Objects as Filters 78

Generating a Standard Filter 78

Generating an Auxiliary Filter 81

Filtee Processing 83

Performance Considerations 83

Useful Tools 83

The Underlying System 86

Position-Independent Code 86

Maximizing Shareability 88

Minimizing Paging Activity 90

Relocations 90

Profiling Shared Objects 95

5. Versioning 97

Overview 97

Interface Compatibility 98

Internal Versioning 99

Creating a Version Definition 99

Binding to a Version Definition 105

Specifying a Version Binding 109

Relocatable Objects 114

External Versioning 114

Coordination of Versioned Filenames 114

6. Support Interfaces 117

Overview 117

Link-Editor Support Interface 117

Invoking the Support Interface 118

Support Interface Functions 118

Support Interface Example 120

Runtime Linker Auditing Interface 122

Establishing a Name-space 123

Building an Audit Library 123

Invoking the Auditing Interface 124

Audit Interface Functions 124

Audit Interface Example 128

Audit Interface Demonstrations 128

Audit Interface Limitations 129

Runtime Linker Debugger Interface 130

Interaction Between Controlling and Target Process 130

Debugger Interface Agents 132

Debugger Exported Interface 132

Debugger Import Interface 141

7. Object Files 145

Overview 145

File Format 146

Data Representation 147

ELF Header 148

ELF Identification 152

Sections 155

Special Sections 166

String Table 170

Symbol Table 171
Relocation 177
Versioning Information 187
Note Section 193
Dynamic Linking 195
Program Header 195
Program Loading (Processor-Specific) 201
Runtime Linker 208
Hash Table 223
Initialization and Termination Functions 224
8. Mapfile Option 225
Overview 225
Using the Mapfile Option 226
Mapfile Structure and Syntax 226
Segment Declarations 227
Mapping Directives 230
Size-Symbol Declarations 232
File Control Directives 233
Mapping Example 233
Mapfile Option Defaults 235
Internal Map Structure 236
Error Messages 238
Warnings 238
Fatal Errors 238
A. Link-Editor Quick Reference 241
Overview 241
Static Mode 241
Building a Relocatable Object 242

Building a Static Executable 242

Dynamic Mode 242

Building a Shared Object 242

Building a Dynamic Executable 244

B. Versioning Quick Reference 245

Overview 245

Naming Conventions 246

Defining a Shared Object's Interface 247

Versioning a Shared Object 248

Versioning an Existing (Non-versioned) Shared Object 248

Updating a Versioned Shared Object 249

Adding New Symbols 250

Internal Implementation Changes 250

New Symbols and Internal Implementation Changes 251

Migrating Symbols to a Standard Interface 251

Index 255

Preface

Solaris™ 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**(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 Solaris link-editor and runtime linker. Special emphasis is placed on the generation and use of shared objects because of their importance in a dynamic runtime environment.

Intended Audience

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

- Beginners learn the principle operations of the link-editor and runtime linker.
- Intermediate programmers learn to 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 gives an overview of the linking processes under `Solaris`, together with an introduction of new features added with this release. This chapter is intended for all programmers.

Chapter 2 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 describes the execution environment and program-controlled runtime binding of code and data. This chapter is intended for all programmers.

Chapter 4 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 describes how to manage the evolution of an interface provided by a dynamic object. This chapter is intended for all programmers.

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

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

Chapter 8 describes the mapfile directives to the link-editor, which specify the layout of the output file. This chapter is intended for advanced programmers.

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

Appendix B gives naming conventions and guidelines for versioning shared objects, 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.

Note - The term “x86” refers to the Intel 8086 family of microprocessor chips, including the Pentium and Pentium Pro processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term “x86” refers to the overall platform architecture, whereas “*Intel Platform Edition*” appears in the product name.

Ordering Sun Documents

The SunDocs[SM] program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

CHAPTER **1**

Introduction

Overview

This manual describes the operations of the `Solaris` link-editor and runtime linker, together with the objects on which they operate. The basic operation of the `Solaris` 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:

Link-Editor

The link-editor, **`ld`**(1), concatenates 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)).

Runtime Linker

The runtime linker, **`ld.so.1`**(1)[1], processes dynamic executables and shared objects at runtime, and binds them to create a runable process.

1. ld.so.1 is a special case of a shared object and therefore allows itself to be versioned. Here a version number of 1 is used, however later releases of Solaris might provide higher version numbers.

Shared Objects

Shared objects (sometimes referred to as *Shared Libraries*) 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 `Solaris` linkers work with files that conform to the executable and linking format (ELF).

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

Link-Editing

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

Relocatable object

A concatenation of input relocatable objects, which can 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 runtime linker to produce a runable process. Its symbolic references might still need to be bound at runtime, and it might have one or more dependencies in the form of shared objects.

Shared object

A concatenation of input relocatable objects that provides services that might be bound to a dynamic executable at runtime. The shared object might also have dependencies on other shared objects.

These output files, and the key link-editor options used to create them, are shown in Figure 1–1.

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

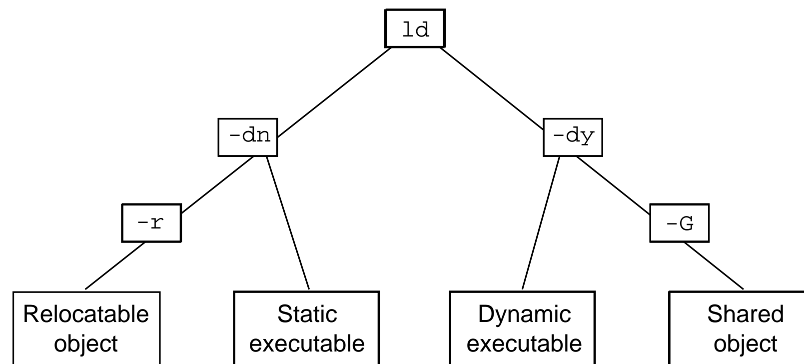


Figure 1–1 Static or Dynamic Link-editing

Runtime Linking

Runtime linking involves the binding of objects, usually 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 can be used to extend the process' address space by adding additional shared objects on demand. The two most common components involved in runtime linking are *dynamic executables* and *shared objects*.

Dynamic Executables

Dynamic executables are applications that are executed under the control of a runtime linker. These applications usually have dependencies in the form of shared objects, which are located and bound by the runtime linker to create a runable process. Dynamic executables are the default output file 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 have not yet been assigned a virtual address.

Dynamic executables usually 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 can be used by many applications, 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:

***compilation* environment**

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

***runtime* environment**

Shared objects are processed by the runtime linker, together with a dynamic executable, to produce a 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 `Solaris` applications.

Application Binary Interfaces

To enable the asynchronous evolution of system and application components, binary interfaces between these facilities are defined. The `Solaris` linkers operate upon these interfaces to assemble applications for execution. Although all components handled by the `Solaris` 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*. `Solaris` provides for the generation and execution of ABI-conforming 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 see the appropriate ABI manuals.

Support Tools

Together with the objects mentioned in the previous sections come several 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), **pvs**(1), **elf**(3E), and a linker debugging support library. Throughout this document many discussions are augmented with examples of these tools use.

New Features

This section gives an overview of new features and/or updates to this document that are available with the `Solaris 2.6` release:

- Weak symbol references can trigger archive member extraction by using the link-editor `-z weakextract` option. Extracting all archive members can be achieved using the `-z allextract` option. See “Archive Processing” on page 11.

- Shared objects specified as part of a link-edit that are not referenced by the object being built, can be ignored, and hence their dependency recording suppressed, using the `-z ignore` option. See "Shared Object Processing" on page 12.
- The link-editor generates the reserved symbols `_START_` and `_END_` to provide a means of establishing an objects address range. See "Generating the Output Image" on page 37.
- Changes have been made to the runtime ordering of initialization and finalization code to better accommodate dependency requirements. See "Initialization and Termination Routines" on page 52.
- Symbol resolution semantics have been expanded for **dlopen**(3X). See "Symbol Lookup" on page 57, `RTLD_GROUP` in "Isolating a Group" on page 62, and `RTLD_PARENT` in "Object Hierarchies" on page 62.
- Symbol lookup semantics have been expanded with a new **dlsym**(3X) handle `RTLD_DEFAULT`. See "Obtaining New Symbols" on page 63.
- Extensions have been made to *filter* processing that allow more than one *filtee* to be defined, and provide for forcibly loading *filtees*. An example of creating a platform specific filter is also provided. See "Shared Objects as Filters" on page 78.
- Recording additional version dependencies can be achieved using the `mapfile` file control directive `$ADDVERS`. See "Binding to Additional Version Definitions" on page 111.
- A runtime linker audit interface provides support for monitoring and modifying a dynamically linked application from within the process. See "Runtime Linker Auditing Interface" on page 122.
- A runtime linker debugger interface provides support for monitoring and modifying a dynamically linked application from an external process. See "Runtime Linker Debugger Interface" on page 130.
- Additional section types are supported. See Table 7–9 for `SHN_BEFORE` and `SHN_AFTER`, and see Table 7–12 for `SHF_ORDERED` and `SHF_EXCLUDE`.
- A new dynamic section tag, `DT_1_FLAGS`, is supported. See Table 7–35 for the various flag values.
- A package of demonstration ELF programs is provided. See Chapter 7.
- The link-editors now support internationalized messages. All system errors are reported using **strerror**(3C).

CHAPTER **2**

Link-Editor

Overview

The link-editing process builds 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 see Chapter 7. For this introduction, however, it is first necessary to introduce two ELF structures, *sections* and *segments*.

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

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

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

The following simple breakdown of link-editor functionality introduces the topics 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 can 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 usually 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 might 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 object dependencies to the runtime linker.

The process of concatenating like *sections* and associating *sections* to *segments* is carried out using default information within the link-editor. The default *section* and *segment* handling provided by the link-editor is usually sufficient for most link-edits. However, these defaults can be manipulated using the `–M` option with an associated `mapfile` (see Chapter 8 for more details).

Invoking the Link-Editor

You can either run the link-editor directly from the command-line or have a compiler driver invoke it for you. In the following two sections both 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 builds 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 can be language or operating system specific, and is usually provided through files supplied by the compiler drivers.

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

In practice, when creating runtime objects such as executables and shared objects, it is recommended that a compiler driver be used to invoke the link-editor. Invoking the link-editor directly is only recommended when creating intermediate relocatable objects using the −r option.

Using a Compiler Driver

The conventional way to use the link-editor is through a language-specific compiler driver. You supply the compiler driver, **cc**(1), **f77**(1), etc., with the input files that make up your application, and the compiler driver adds additional files and default libraries to complete the link-edit. These additional files can be seen by expanding the compilation invocation, for example:

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

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

Specifying the Link-Editor Options

Most options to the link-editor can be passed through the compiler driver command-line. For the most part the compiler and the link-editor options do not conflict. Where a conflict arises, the compiler drivers usually provide a command-line syntax that allows you to pass specific options to the link-editor. However, you can instead provide options to the link-editor by setting 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. When either of these cases is found, a suitable error message is generated, and when 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 inconsistencies. For example:

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

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

See Appendix A for the most commonly used link-editor options, and 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 in which they appear on the command-line. Each file is opened and inspected to determine its ELF file type and so 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* mode the link-editor will accept only relocatable objects or archive libraries as input files. Under *dynamic* mode 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 places. Symbols are gathered into a special internal symbol table that allows for their verification and resolution, and eventually for 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 (see "Linking with Additional Libraries" on page 13 for coverage 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 can 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 usually consist of a collection of relocatable objects with an archive symbol table. This symbol table provides an association of symbol definitions with the objects that supply these definitions. By default, the link-editor provides *selective* extraction of archive members. 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. It is also possible to explicitly extract all member of an archive.

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

- The archive 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.
- The archive 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 is a `FORTRAN COMMON` block definition which will cause the extraction of a relocatable object that defines the same `DATA` symbol.
- The link-editors `-z allextract` is in effect. This option suspends selective archive extraction and causes all archive members to be extracted from the archive being processed.

Under selective archive extraction, a *weak* symbol reference will not cause the extraction of an object from an archive unless the `-z weakextract` option is in effect. Weak symbols are expanded upon in section "Simple Resolutions" on page 20.

Note - The options `-z weakextract`, `-z allextract` and `-z defaultextract` provide a means to toggle the archive extraction mechanism among multiple archives.

Under selective archive extraction 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-editor 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 extracting from the archive only the relocatable objects needed *at the time* the archive was encountered means that the position of the archive within the input file list can be significant (see "Position of an Archive on the Command-Line" on page 14 for more details).

Note - Although the link-editor will make multiple passes through an archive to resolve symbols, this mechanism can 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 so reduce the number of passes the link-editor must carry out.

Shared Object Processing

Shared objects are indivisible, whole units that have been generated by a previous link-edit of one or more input files. When the link-editor processes a shared object the entire contents of the shared object become a *logical* part of the resulting output file image. 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.

By default, *all* shared objects specified as part a link-edit are recorded as dependencies in the object being built. This recording is made regardless of whether the object being built actually references symbols offered by the shared object. To minimize runtime linking overhead only those dependencies required to resolve symbol references from the object being built should be specified as part of the link-edit. Alternatively, the link-editor's `-z ignore` option can be used to suppress the dependency recording of unused shared objects.

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 can have a global effect. Multiple symbols of the same name are allowed to occur between relocatable objects and shared objects, and between multiple shared objects (see “Symbol Resolution” on page 19 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 ensure that appropriate libraries are specified to the link-editor, it is frequently necessary for you to supply your own. Shared objects and archives can be specified by explicitly naming the input files required to the link-editor, but a more common and more flexible method involves using the link-editor’s `−l` option.

Library Naming Conventions

By convention, shared objects are usually designated by the prefix `lib` and the suffix `.so`, 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. This option is commonly used to supply additional libraries to a link-edit. The following 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`, before moving on to the next directory to be searched.

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. See "Naming Conventions" on page 72 and "Coordination of Versioned Filenames" on page 114 for more details.

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

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.c -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.

In the previous example it is more precise to say that the link-editor first searches for `libfoo.a`, and then for `libbar.so`, and if that fails, for `libbar.a`. Finally, it will search for `libc.so`, and if that fails, `libc.a`.

Position of an Archive on the Command-Line

The position of an archive on the command-line can affect the output file being produced. The link-editor searches an archive only to resolve undefined or tentative external references 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.c -lfoo file2.c -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.c -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.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

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

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

```
$ LD_LIBRARY_PATH=dir1:dir2
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -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 can also be used to augment the search path of the runtime linker (see "Directories Searched by the Runtime Linker" on page 44 for more details). 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. These pathnames will be used by the runtime linker 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.c -R/home/me/lib:/home/you/lib -Lpath1 \ -Lpath2 file1.c file2.c -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, this runpath will be used to locate 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 previous example can also be expressed as:

```
$ cc -o prog main.c -R/home/me/lib -Lpath1 \ -R/home/you/lib -Lpath2 file1.c file2.c -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 can also supply .init and .fini sections as part of the additional files they add to the beginning and end of the your 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 can be called by the runtime linker during initialization and termination processing. See "Initialization and Termination Routines" on page 52 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 can 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 can be included in both a shared object and archive library. If this code is spread throughout several relocatable objects within an archive library, then the link-edit of an application using this archive can extract only a portion of the modules, and therefore only a portion of the initialization and termination code. At runtime, only this portion of code will be executed.

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.

It is also recommended that data initialization be idempotent if the `.init` code is involved with a dynamic object whose memory may be dumped using **`dldump`**(3X).

Symbol Processing

During input file processing, all *local* symbols from the input relocatable objects are passed through to the output file image. All *global* symbols are accumulated internally within the link-editor. This internal symbol table is searched for each new

global symbol entry processed to determine 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, and 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.

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

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

Symbol Resolution

Symbol resolution runs the entire spectrum, from simple and intuitive to complex and perplexing. Resolutions can 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 see "Symbol Table" on page 171. For the following discussions, however, it is worth identifying three basic symbol types:

Undefined

These symbols have been referenced in a file but have not been assigned a storage address.

Tentative

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

These symbols have been created and assigned storage addresses and space within the file.

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

The following C code example shows how these symbol types can 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 can have either a global or weak *binding*. Weak bindings have less precedence than global binding, and so symbols with different bindings are resolved according to a slight alteration of the basic rules. But first, it is worth introducing how weak symbols can be produced.

Weak symbols can 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;

```

(continued)

```
_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 can 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 several services that you are allowed to redefine. 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 so it would seem that a program that redefines **read**(2) might 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
```

you can define just such a reserved name, and from it generate an alias for the function **read**(2). Thus, you can quite freely define your 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 your 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 link-editor's `-m` option, a list of all interposed symbol references, along with section load address information, is written 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 is produced if the symbols' alignment requirements differed. In both of these cases the diagnostic can be suppressed by using the link-editor's `-t` option.

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

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

(continued)

```
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.

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

Fatal Resolutions

Symbol conflicts that cannot be resolved result in a fatal error condition. 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);
```

(continued)

```
}

$ 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 usually 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
```

If you wish to allow undefined symbols, as in cases like the previous example, then the default fatal error condition can be suppressed by using the link-editor's `-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 can be detected during the initial execution and testing of an application, more complex execution paths can result in this error condition taking much longer to detect, which can be time consuming and costly.

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

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

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

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

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.

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

For this reason, bindings of this type are deemed fatal, and the implicit reference must be made explicit by referencing the library directly during the link-edit of `prog` (the required reference is hinted at in the fatal error message 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 link-editor's `−z defs` option can be used to force a fatal error if any undefined symbols remain.

In general a self contained shared object, in which all references to external symbols are satisfied by named dependencies, provides maximum flexibility. In this case the shared object can be employed by many users, without those users having to determine and establish dependencies to satisfy the shared objects requirements.

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. 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.

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

```
#pragma weak    foo

extern  void    foo(char *);
```

(continued)

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

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

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

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

Users are discouraged from using undefined weak references in this manner, and instead are encouraged to use **dlsym**(3X) with the `RTLD_DEFAULT` flag as a means of testing for a symbols existence (see “Testing for Functionality” on page 64).

Tentative Symbol Order Within the Output File

Contributions from input files usually appear in the output file in the order of their contribution. An exception occurs when processing tentative symbols and their associated storage. These symbols are not fully defined until their resolution is complete. If the resolution 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 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 result 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

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

The –u option provides a mechanism for generating a symbol reference from the link-edit command line. This option can be used to perform a link-edit entirely from archives, or to provide additional flexibility in selecting the objects to extract from multiple archives (see section “Archive Processing” on page 11 for an overview of archive extraction).

For example, let's take the generation of a dynamic executable from the relocatable object main.o which makes reference to the symbols foo and bar. You wish 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). 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 will also be extracted from lib1.a during its processing (see section "Archive Processing" on page 11 for a discussion of the link-editor's multi-pass processing of an archive).

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

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

name

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

scope

Indicates the visibility of the symbols' binding within the output file being generated. This can have either the value local or global. All symbols defined with a mapfile are treated as global in scope during the link-edit process. That is, they will be resolved against any other symbols of the same name obtained from any of the input files. However, symbols defined as local scope will be reduced to symbols with a local binding within any executable or shared object file being generated.

symbol

The name of the symbol required. If the name is not followed by any symbol attributes then the result will be the creation of a symbol reference. This reference is exactly the same as would be generated using the −u option discussed earlier in this section. If the symbol name is followed by an optional "=" character then a symbol definition will be generated using the associated attributes.

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

type

Indicates the symbols' type attribute and can be either `data`, `function` or `common`. The former two type attributes result in an*absolute* symbol definition (see "Symbol Table" on page 171). The latter type attribute results in a *tentative* symbol definition.

value

Indicates the symbols' value attribute and takes the form of `Vnumber`.

size

Indicates the symbols' size attribute and takes the form of `Snumber`.

dependency

Represents a *version definition* that will be inherited by this definition. See Chapter 5 for more details.

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

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

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

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

The remainder of this section presents several examples of using this `mapfile` syntax.

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

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

(continued)

(Continuation)

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

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

The significance of reducing symbol scope from *global* to *local* is covered in more detail in the section "Reducing Symbol Scope" on page 33.

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

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

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = FUNCTION V0x400;
```

(continued)

(Continuation)

```
                bar = DATA V0x800;
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 400
&bar = 800
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00000800|0x00000000|OBJT |GLOB |0x0  |ABS    |bar
[42]    |0x00000400|0x00000000|FUNC |GLOB |0x0  |ABS    |foo
```

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

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

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

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

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

main()
{
        (void) printf("&foo = %x\n", &foo);
        (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
        global:
                foo = COMMON V0x4 S0x200;
                bar = COMMON V0x100 S0x40;
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol 'bar' has differing alignments:
        (file mapfile value=0x100; file main.o value=0x4);
        largest value applied
```

(continued)

```
$ prog
&foo = 20940
&bar = 20900
$ nm -x prog | egrep "foo$|bar$"
[37]    |0x00020900|0x00000040|OBJT |GLOB |0x0  |16     |bar
[42]    |0x00020940|0x00000200|OBJT |GLOB |0x0  |16     |foo
```

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

Reducing Symbol Scope

In the previous section it was shown how symbol definitions defined to have local scope within a `mapfile` can be used to reduce the symbol's eventual binding. This mechanism can play an important role in reducing the symbol's visibility to future link-edits that use the generated file as part of their input. In fact, this mechanism can provide for the precise definition of a file's interface, and so restrict the functionality made available to others.

For example, let's take the generation of a simple shared object from the files `foo.c` and `bar.c`. The file `foo.c` contains the global symbol `foo` which provides the service that you wish to make available to others. The file `bar.c` contains the symbols `bar` and `str` which provide the underlying implementation of the shared object. A simple build of the shared object will usually result in all three of these symbols having global scope:

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

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

const char * bar()
{
        return (str);
}
$ cc -o lib.so.1 -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
```

(continued)

```
[29]    |0x000104d0|0x00000004|OBJT |GLOB |0x0  |12     |str
[32]    |0x00000418|0x00000028|FUNC |GLOB |0x0  |6      |bar
[33]    |0x000003f0|0x00000028|FUNC |GLOB |0x0  |6      |foo
```

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

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

Another consequence of the global binding of the symbols `bar` and `str` is that they can be *interposed* upon by symbols of the same name (the interposition of symbols within shared objects is covered in section "Simple Resolutions" on page 20). This interposition can be intentional and be used as a means of circumventing the intended functionality offered by the shared object. On the other hand, this interposition can be unintentional, being the result of the application and the shared object using the same common symbol name.

When developing the shared object you can protect against this type of scenario by reducing the scope of the symbols `bar` and `str` to a local binding, for example:

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

Here the symbols `bar` and `str` are no longer available as part of the shared objects interface. Thus these symbols cannot be referenced, or interposed upon, by an external object. You have effectively defined an interface for the shared object. This interface can be managed while hiding the details of the underlying implementation.

This symbol scope reduction has an additional performance advantage. The symbolic relocations against the symbols `bar` and `str` that would have been necessary at runtime are now reduced to relative relocations. This reduces the runtime overhead

of initializing and processing the shared object (see section “When Relocations are Performed” on page 91 for details of symbolic relocation overhead).

As the number of symbols processed during a link-edit gets large, the ability to define each local scope reduction within a mapfile becomes harder to maintain. An alternative, and more flexible mechanism, allows you to define the shared objects interface in terms of the global symbols that should

be maintained, and instructs the link-editor to reduce all other symbols to local binding. This mechanism is achieved using the special *auto-reduction* directive “*”. For example, the previous mapfile definition can be rewritten to define foo as the only global symbol required in the output file generated:

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

This example also defines a version name, lib.so.1.1, as part of the mapfile directive. This version name establishes an internal *version definition* that defines the files symbolic interface. The creation of a version definition is recommended, and forms the foundation of an internal versioning mechanism that can be used throughout the evolution of the file. See Chapter 5 for more details on this topic.

Note - If a version name is not supplied the output filename will be used to label the version definition. The versioning information created within the output file can be suppressed using the link-editors' -z noversion option.

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

```
$ cat mapfile
lib.so.1.1 {
        global:
                foo;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
Undefined           first referenced
 symbol                 in file
```

(continued)

```
str                     bar.o  (symbol has no version assigned)
bar                     bar.o  (symbol has no version assigned)
ld: fatal: Symbol referencing errors. No output written \
to lib.so.1
```

When generating an executable or shared object, any symbol reduction results in the recording of version definitions within the output image, together with the reduction of the appropriate symbols. By default, when generating a relocatable object, the version definitions are created, but the symbol reductions are not processed. The result is that the symbol entries for any symbol reductions will still remain global. For example, using the previous `mapfile` and associated relocatable objects, an intermediate relocatable object is created which shows no symbol reduction:

```
$ ld -o lib.o -M mapfile -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[17]    |0x00000000|0x00000004|OBJT |GLOB |0x0  |3       |str
[19]    |0x00000028|0x00000028|FUNC |GLOB |0x0  |1       |bar
[20]    |0x00000000|0x00000028|FUNC |GLOB |0x0  |1       |foo
```

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

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

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

Symbol reductions can be forced to occur when building a relocatable object by using the link-editor's `–B reduce` option:

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

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 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.

Also included are any output relocation and dynamic information sections 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 usually generated. The `.dynsym`, and its associated string table `.dynstr`, contain only global, weak and section symbols. These sections become part of the `text` segment which is mapped as part of the process image at runtime. This allows the runtime linker to read these sections and 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 link-editor's `–s` option, or after the link-edit using **strip**(1).

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

`_etext`

The first location after the text segment.

`_edata`

The first location after initialized data.

`_end`

The first location after all data.

`_DYNAMIC`

The address of the dynamic information section (the `.dynamic` section).

`_END_`

The same as `_end`, but the symbol has local scope (see `_START_`).

`_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 (see "Position-Independent Code" on page 86 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 (see "Position-Independent Code" on page 86 for more information).

`_START_`

The first location within the text segment. The symbol has local scope, and together with `_END_`, provide a means of establishing an objects address range.

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 link-editor's `-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 `Solaris` linkers is a debugging library that allows you to trace the link-editing process in more detail. This library helps you understand, or debug, the link-edit of your 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 might change slightly from release to release.

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

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=-Dtoken1,token2 cc -o prog ...
debug:           or,
debug:                 ld -Dtoken1,token2 -o prog ...
debug:           where placement of -D on the command line is significant
debug:           and options can be switched off by prepending with '!'.
debug:
debug:
debug: args      display input argument processing
debug: basic     provide basic trace information/warnings
debug: 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: support   display support library processing
debug: symbols   display symbol table processing;
debug:           detail flag shows resolution and linker table addition
debug: versions  display version processing
```

Note - This listing is an example, and shows the options meaningful to the link-editor. The exact options might 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 using the `symbols` token you can determine what symbol caused an archive member to be extracted, and which object made the initial symbol reference:

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

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 - This output has been simplified for this document.

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

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

(continued)

```
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 previous example, it should be apparent that using some of the debugging tokens can produce a wealth of output. In cases where are interested only 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. 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`.

Note - To obtain the link-edit command-line it might be necessary to expand the compilation line from any driver being used. See “Using a Compiler Driver” on page 9 for more details.

CHAPTER **3**

Runtime Linker

Overview

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

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

During the process of executing a dynamic executable (see **exec**(2)) the kernel maps the file (see **mmap**(2)), and using the program header information (see "Program Header" on page 195) locates the name of the required interpreter. The kernel maps this interpreter and transfers control to it, passing sufficient information to allow the interpreter to continue binding the application and then run it.

In addition to initializing an application the runtime linker provides services that allow the application to extend its address space by mapping additional objects and binding to symbols within them.

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

- It analyzes the executable's dynamic information section (`.dynamic`) and determines what dependencies are required.
- It locates and maps in these dependencies, and analyzes their dynamic information sections to determine if any additional dependencies are required.
- Once all dependencies are 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 dependencies.
- It passes control to the application.
- During the application's execution, the runtime linker can be called upon to perform any delayed function binding.
- The application can also call upon the runtime linker's services to acquire additional objects by **dlopen**(3X), and bind to symbols within these objects with **dlsym**(3X).

Locating Shared Object Dependencies

Usually, during the link-edit of a dynamic executable, one or more shared objects are explicitly referenced. These objects are recorded as dependencies within the dynamic executable (see "Shared Object Processing" on page 12 for more information).

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

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

Directories Searched by the Runtime Linker

The runtime linker knows of only one standard place to look for 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* 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
        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 `libc.so.1` and `libdl.so.1`.

The 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 that have a NEEDED tag. For example:

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

/usr/bin/cat:
[INDEX] Tag      Value
[1]     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. The use of a simple filename 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 –h option of the link-editor to record a simple name within the dependency (see "Naming Conventions" on page 72 and "Recording a Shared Object Name" on page 73 for additional information on this topic).

Frequently, dependencies are distributed in directories 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 way to indicate additional search paths to the runtime linker is to record a *runpath* during the link-edit of the dynamic executable or shared object (see "Directories Searched by the Runtime Linker" on page 16 for details on recording this information).

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

```
$ dump -Lvp prog

prog:
[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`.

Another way to add to the runtime linker's search path is to set the environment variable `LD_LIBRARY_PATH`. This environment variable (which is analyzed once at process start-up) 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 dependency. For example:

```
$ LD_LIBRARY_PATH=. prog
```

Here the file `prog` from the 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 can result in an overall degradation in performance. Also, as pointed out in "Using an Environment Variable" on page 16 and "Directories Searched by the Runtime Linker" on page 16, this environment variable affects the link-editor.

If a 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 =>   (file 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: open failed: No such file or \
directory
```

Relocation Processing

Once the runtime linker has located and mapped all the dependencies required by an application, it processes each object and performs any necessary relocations.

During the link-editing of an object, any relocation information supplied with the input relocatable objects is applied to the output file. However, when 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 known only 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, see “Relocation Types (Processor-Specific)” on page 179. 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 -rvp libbar.so.1

libbar.so.1:

.rela.got:
Offset      Symndx                Type              Addend

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

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 the symbol index (`Symndx`) field being 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`. To complete this relocation the runtime linker must locate this symbol from the dynamic executable and its dependencies that have been mapped so far.

Symbol Lookup

When an object requires a symbol, the runtime linker searches for that symbol based upon the requesting objects’ symbol *search scope*, and the symbol *visibility* offered by each object within the process. These attributes are applied as defaults to an object at the time it is loaded, as specific modes to **dlopen**(3X), and in some cases can be recorded within the object at the time it is built.

Typically, an average user becomes familiar with the default symbol search models that are applied to a dynamic executable and its dependencies, and to objects obtained through **dlopen**(3X). The former is outlined in this section and the latter,

which is also able to exploit the various symbol lookup attributes, is discussed in "Symbol Lookup" on page 57.

A dynamic executable and all the dependencies loaded with it, are assigned *world* search scope, and *global* symbol visibility (see "Symbol Lookup" on page 57). Thus, when the runtime linker looks up a symbol for a dynamic executable or for any of the dependencies loaded with the executable, it does so by searching each object, starting with the dynamic executable, and progressing through each dependency in the same order in which the objects were mapped.

As discussed in previous sections, **ldd**(1) will list the 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
```

Then, 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 increases, and the number of dependencies increases. This aspect of performance is discussed in more detail in" Performance Considerations" on page 83.

Interposition

The runtime linkers mechanism of searching for a symbol first in the dynamic executable and then in each of the 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 (see also "Shared Object Processing" on page 12).

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 code. The runtime linker has no knowledge of the application code, and so does not know when this data item will be referenced. Therefore, all data relocations must be carried out during process initialization, before the application gains control.

A *function reference* refers to the address of a function that will be called by the application 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).

Procedure linkage table entries are constructed so that when first called control is passed to the runtime linker (see "Procedure Linkage Table (Processor-Specific)" on page 219). The runtime linker will look up the required symbol and rewrite information in the application so that 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, before transferring control to the application. For example:

```
$ LD_BIND_NOW=yes prog
```

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

Individual objects can also be built using the link-editors' `-z now` option to indicate that they require complete relocation processing at the time they are loaded. This relocation requirement is also propagated to any dependencies of the marked object at runtime.

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
```

(continued)

```
ld.so.1: prog: fatal: relocation error: file ./libfoo.so.1: \
symbol bar: referenced symbol not found
```

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

Note - During the link-edit of a dynamic executable any potential relocation errors of this sort will be flagged as fatal undefined symbols (see “Generating an Executable” on page 24 for examples). This runtime relocation error can occur if the link-edit of `main` used a different version of the shared object `libbar.so.1` that contained a symbol definition for `bar`, or if the `-z nodefs` option was used as part of the link-edit.

If a relocation error of this type occurs because a symbol used as 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 can take minutes or months, or might never occur, depending on the execution paths exercised throughout the code. To guard against errors of this kind, the relocation requirements of any dynamic executable or shared object can be validated using **`ldd`**(1).

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

```
$ 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.

Loading Additional Objects

The previous sections have described how the runtime linker initializes a process from the dynamic executable and its dependencies as they were defined during the link-editing of each module. The runtime linker also provides an additional level of flexibility by allowing you 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 are mapped *after* the dynamic executable and *before* any dependencies, and are assigned *world* search scope, and *global* symbol visibility (see “Symbol Lookup” on page 57). 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 dependencies defined within prog. The order in which these objects are processed can be displayed using **ldd**(1):

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

Another example is:

```
$ 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 to generate a shared object that is maintained in memory. This memory image is then inserted between the dynamic executable and its 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 an object after a dynamic executable take the concept of *interposition* introduced in “Interposition” on page 48 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 an object containing this

function it will interpose on the original. Thus the old functionality can be completely hidden with the new preloaded version.

Another use of preloading is to augment a function that resides in a standard shared object. 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 (see "Simple Resolutions" on page 20) or the ability to look up the original symbol's address (see "Using Interposition" on page 65).

Initialization and Termination Routines

Before transferring control to the application, the runtime linker processes any initialization (`.init`) and termination (`.fini`) sections found in the applications dependencies. These sections, and the symbols that describe them, are created during the link-editing of the dependencies (see "Initialization and Termination Sections" on page 17).

Prior to the `Solaris 2.6` release, any initialization routines from dependencies were called in *reverse* load order - in other words, the reverse order of the dependencies displayed with **`ldd`**(1).

Starting with the `Solaris 2.6` release, the runtime linker constructs a dependency ordered list of initialization routines from the dependencies that have been loaded. This list is built from the dependency relationships expressed by each object, in addition to any bindings that occur outside of the expressed dependencies.

The `.init` sections are executed in the reverse topological order of the dependencies. If any cyclic dependencies are found, the objects that form the cycle cannot be topologically sorted, and thus their `.init` sections will be executed in the order they are loaded.

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

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

(continued)

```
        A.so.1 =>         ./A.so.1
        B.so.1 =>         ./B.so.1
        libc.so.1 =>      /usr/lib/libc.so.1
        C.so.1 =>         ./C.so.1
        libdl.so.1 =>     /usr/lib/libdl.so.1

    init library=./A.so.1
    init library=./C.so.1 (cyclic dependency on ./B.so.1)
    init library=./B.so.1 (cyclic dependency on ./C.so.1)
    init library=/usr/lib/libc.so.1
```

The environment variable `LD_BREADTH` can be set to a non-null value to force the runtime linker to execute `.init` sections in pre-`2.6` order.

Initialization processing is repeated for any objects added to the running process with **`dlopen`**(3X).

Any termination routines for an applications dependencies are organized such that they can be recorded by **`atexit`**(3C). These routines are called when the process calls **`exit`**(2), or when objects are removed from the running process with **`dlclose`**(3X).

Starting with the `Solaris 2.6` release, termination routines are called in the topological order of dependencies. Prior to the `Solaris 2.6` release, or when the `LD_BREADTH` environment variable is in effect, termination routines were called in load order.

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

Note - Any `.init` or `.fini` sections within the dynamic executable are called from the application itself by the process start-up and termination mechanism supplied by the compiler driver. The dynamic executable’s `.init` section is called last, after all its dependencies `.init` sections are executed. The dynamic executable’s `.fini` section is called first, before its dependencies `.fini` sections are executed.

Security

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

The runtime linker categorizes a process as secure if the user is not the *root*, and either the real users and effective users identifiers are not equal (see **getuid**(2) and **geteuid**(2)), or the real group and effective group identifiers are not equal (see **getgid**(2) and **getegid**(2)).

If an LD_LIBRARY_PATH environment variable is in effect (see "Directories Searched by the Runtime Linker" on page 44) for a secure process, then only the *trusted* directories specified by this variable will be used to augment the runtime linker's search rules. Presently, the only trusted directory known to the runtime linker is /usr/lib.

In a secure process, any *runpath* specifications provided by the application or any of it's dependencies (see "Directories Searched by the Runtime Linker" on page 44) will be used provided they are full pathnames - in other words the pathname starts with a '/'.

Additional objects may be loaded with a secure process using the LD_PRELOAD environment variable (see "Loading Additional Objects" on page 55) provided the objects are specified as *simple* filenames - in other words there is no '/' in the name. These objects will be located subject to the search path restrictions previously described.

In a secure process, any dependencies that consist of simple filenames will be processed using the pathname restrictions outlined. Dependencies that are expressed as full or relative pathnames will be used as is. Therefore, the developer of a secure process should insure that the target directory referenced as a full or relative pathname dependency is suitably protected from malicious intrusion.

When creating a secure process, it is recommended that relative pathnames *not* be used to express dependencies or to construct **dlopen**(3X) pathnames. This restriction should be applied to the application and to *all* dependencies.

Runtime Linking Programming Interface

The previous discussions described how the dependencies specified during the link-edit of an application are processed by the runtime linker during process initialization. In addition to this mechanism, the application can extend its address space during its execution by binding to additional objects. This extensibility is provided by allowing the application to request the same services of the runtime linker as used to process the dependencies specified during the link-edit of the application.

This delayed object binding has several advantages:

- By processing an object when it is required rather than during the initialization of an application, start-up time can be greatly reduced. In fact, the object might not

be required if its services are not needed during a particular run of the application, such as for help or debugging information.

- The application can choose between several different objects depending on the exact services required, such as for a networking protocol.
- Any objects added to the process address space during execution can be freed after use.

The following is a typical scenario that an application can 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 has are located and added at this time.
- The added shared object and its dependencies are relocated, and any initialization sections within these objects are called.
- The application locates symbols within the added objects 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 objects, the address space can be freed using **dlclose**(3X). Any termination sections within the objects 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 by the shared object `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` can bind to these routines at runtime.

Loading Additional Objects

Additional 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 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 search path rules are exactly the same as are used to locate any initial dependencies (see "Directories Searched by the Runtime Linker" on page 44). For example, if the file `main.c` contains the following code fragment:

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

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

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

then to locate the shared object `foo.so.1`, the runtime linker will use any `LD_LIBRARY_PATH` definition present at process initialization, followed by any runpath specified during the link-edit of `prog`, and finally the default location `/usr/lib`.

If the filename is specified as:

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

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

Note - It is recommended that any shared object specified using **dlopen**(3X) be referenced by its *versioned* filename (for more information on versioning see “Coordination of Versioned Filenames” on page 114).

If the required 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: open failed: No such \
file or directory
```

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

If the object specified by **dlopen**(3X), or any of its dependencies, are already part of the process image, then the objects will not be processed any further, but a valid handle will still be returned to the application. This mechanism prevents the same object from being mapped more than once, and allows an application to obtain a

handle to itself. For example, if the previous main.c example contained the following dlopen( ) call:

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

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

Relocation Processing

As described in "Relocation Processing" on page 46, after locating and mapping any objects, the runtime linker must process each object and perform any necessary relocations. Any objects brought into the process's address space with **dlopen**(3X) must also be relocated in the same manner.

For simple applications this process might be quite uninteresting. However, for users who have more complex applications with many **dlopen**(3X) calls involving many objects, possibly with common dependencies, this topic can 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 objects added with **dlopen**(3X) when the *mode* is defined as RTLD_LAZY. An alternative is to require all relocations of an object to be performed immediately when the object is added. This can be achieved by using a *mode* of RTLD_NOW, or by recording this requirement in the object when it was built using the link-editors' -z now option. This relocation requirement is propagated to any dependencies of the object being opened.

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

Symbol Lookup

If an object acquired by **dlopen**(3X) refers to a global symbol, the runtime linker must locate this symbol from the pool of objects that makeup the process. A default symbol search model is applied to objects obtained by **dlopen**(3X), and this is described in the following sections. However, the mode of a **dlopen**(3X), combined with the attributes of the objects that makeup the process, provide for alternative symbol search models.

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

`world`

The object can look in any other *global* object within the process.

`group`

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

The visibility of a symbol from an object can be:

`global`

The objects symbols can be referenced from any object having *world* search scope.

`local`

The objects symbols can only be referenced from other objects that comprise the same *group*.

By default, objects obtained with **dlopen**(3X) are assigned *world* symbol search scope, and *local* symbol visibility. The following section "Default Symbol Lookup Model" on page 58, uses this default model to illustrate typical object group interactions. Sections "Defining a Global Object" on page 61, "Isolating a Group" on page 62 and "Object Hierarchies" on page 62 show examples of using **dlopen**(3X) modes and file attributes to extend the default symbol lookup model.

Default Symbol Lookup Model

For each object added by **dlopen**(3X) the runtime linker will first look for the symbol in the dynamic executable, and then look in each of the objects provided during the initialization of the process. However, if the symbol is still not found, the runtime linker will continue the search, looking in the object acquired through the **dlopen**(3X) and in any of its dependencies.

For example, let's take the dynamic executable `prog`, and the shared object `B.so.1`, each of which has 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` by **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, it might 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 in the previous listing can be expressed diagrammatically:

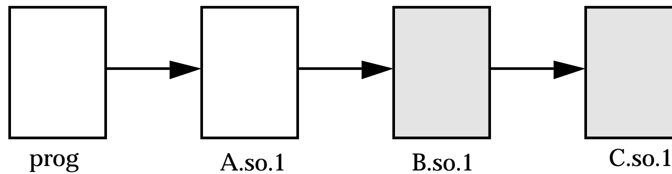


Figure 3–1 A Single dlopen(3X) Request

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`.

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

These new objects do not affect the normal symbol lookup required by either the application or its initial object dependencies. For example, if `A.so.1` requires a function relocation *after* the above **dlopen**(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 is again a result of the attributes assigned to the objects as they were loaded. The *world* symbol search scope assigned the dynamic executable and all the dependencies loaded with it, does not allow them to look for symbols in the new objects that only offer *local* symbol visibility.

These symbol search and symbol visibility attributes thus maintain associations between objects based on their introduction into the process address space, and on any dependency relationships between the objects. Assigning the objects associated with a given **dlopen**(3X) a unique *group* 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 can be represented as:

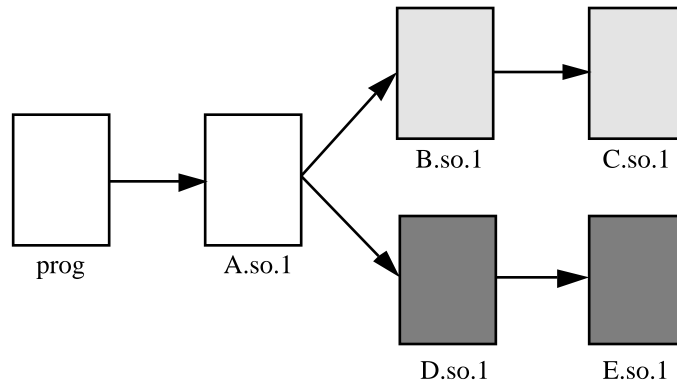


Figure 3–2 Multiple dlopen(3X) Requests

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 to a unique group, `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 objects obtained from multiple calls to **dlopen**(3X).

When 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 objects have common dependencies the resultant bindings can be affected by the order in which the **dlopen**(3X) calls are 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.1and P.so.1, it will be assigned to both of the groups that are associated with the two **dlopen**(3X) calls. Diagrammatically this can be represented as:

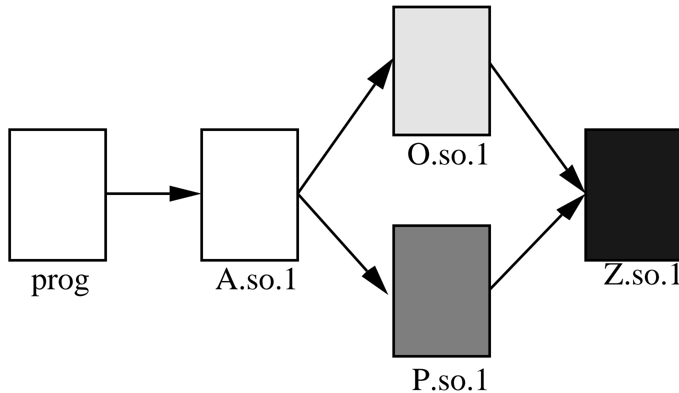


Figure 3–3 Multiple dlopen(3X) Requests With A Common Dependency

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. If the functionality of symbol foo differs between the two shared objects in which it is defined, the overall outcome of executing code within Z.so.1 might vary depending on the application's **dlopen**(3X) ordering.

Defining a Global Object

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

In addition, any object obtained by **dlopen**(3X) with the RTLD_GLOBAL flag will also be available for symbol lookup using dlopen(0) (see "Loading Additional Objects" on page 55).

Note - If a member of a group, having *local* symbol visibility, is referenced by another group requiring *global* symbol visibility, the objects visibility will become a concatenation of both local and global. This promotion of attributes will remain, even if the global group reference is later removed.

Isolating a Group

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

Objects can have the *group* symbol search scope assigned to them when they are built by using the link-editors' `-B group` option.

Note - If a member of a group, having *group* search capability, is referenced by another group requiring *world* search capability, the objects search capability will become a concatenation of both group and world. This promotion of attributes will remain, even if the world group reference is later removed.

Object Hierarchies

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

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

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

If the initial object is an explicit dependency of the secondary object, it will also be assigned to the secondary objects' group, and thus will be able to provide symbols for the secondary objects' relocation.

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

There is one small difference between these two techniques. In the case of specifying an explicit dependency, the dependency itself becomes part of the secondary objects'

dlopen(3X) dependency tree, and thus becomes available for symbol lookup with **dlopen**(3X). In the case of obtaining the secondary object with RTLD_PARENT, the initial object does not become available for symbol lookup with **dlopen**(3X).

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

Obtaining New Symbols

A process can obtain the address of a specific symbol using **dlsym**(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* object will allow symbols to be obtained from that objects dependency tree.
- The *handle* returned from a **dlopen**(3X) of a file whose value is 0 will allow symbols to be obtained from the dynamic executable, from any of its initialization dependencies, or from any object obtained by a **dlopen**(3X) with the RTLD_GLOBAL mode.
- The special *handle* RTLD_DEFAULT will allow symbols to be obtained from the dynamic executable, from any of its initialization dependencies, or from any object obtained by a **dlopen**(3X) that belongs to the same group as the caller (see "Testing for Functionality" on page 64).
- The special *handle* RTLD_NEXT will allow symbols to be obtained from the *next* associated object (see "Using Interposition" on page 65).

The first example is probably the most common. Here an application will add additional objects to its address space and use **dlsym**(3X) to locate function or data symbols. The application then uses these symbols to call upon services provided in these new 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);
```

(continued)

```
    }

    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 dependencies that are associated with this file. The function `foo` is then called with the single argument `bar` as part of the `return()` statement.

If the application `prog` is built using the above file `main.c`, and its initial dependencies are:

```
$ 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` will be 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 “Symbol Lookup” on page 47.

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: bar: can't find symbol
```

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

Testing for Functionality

The special handle `RTLD_DEFAULT` allows an application to test for the existence of another symbol. The symbol search follows the same model as used to relocate the

calling object (see “Relocation Processing” on page 57). For example, if the application `prog` contained the following code fragment:

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

then `foo` will be searched for in `prog`, followed by `/usr/lib/libdl.so.1`, and then `/usr/lib/libc.so.1`. If this code fragment was contained in the file `B.so.1` from the example shown in Figure 3–2, then the search for `foo` will continue into `B.so.1` and then `C.so.1`.

This mechanism provides a robust and flexible alternative to the use of undefined weak references discussed in “Weak Symbols” on page 26.

Using Interposition

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

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

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

then `foo` will be 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 was contained in the file `B.so.1` from the example shown in Figure 3–2, then `foo` will be searched for in the associated shared object `C.so.1` only.

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

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

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

(continued)

```
    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) usually 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 can be achieved by:

```
$ 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).

The use of `RTLD_NEXT` within a dynamic executable or preloaded object provides a predictable and useful interpositioning technique. However, care should be taken when using this technique in a generic object dependency, as the actual load order of objects is not always predictable (see "Dependency Ordering" on page 77).

Debugging Aids

Provided with the `Solaris` linkers is a debugging library that allows you to trace the runtime linking process in more detail. This library helps you understand, or debug, the execution of applications and dependencies. 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 might change slightly from release to release.

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

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

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

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

Note - This is an example, and shows the options meaningful to the runtime linker. The exact options might differ from release to release.

The environment variable `LD_DEBUG_OUTPUT` can be used to specify an output file for use instead of the standard error. The 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, let's 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
```

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

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

Here, the symbol `bar`, which is required by a data relocation, is bound *before* the application gains 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. If the environment variable LD_BIND_NOW is set, all symbol bindings will occur before the application gains 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 data associated with the functions .plt so that any subsequent calls will go directly to the function. The environment variable LD_BIND_NOT can be set to any value to prevent this data update. By using this variable together with the debugging request for detailed bindings, you can get a complete runtime account of all function binding. The output from this combination can be excessive, 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 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 affects the search for the two dependencies foo.so.1 and bar.so.1.

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

(continued)

```
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 (see "Copy Relocations" on page 91 for more details of this relocation type).

CHAPTER **4**

Shared Objects

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.1 -G -K pic foo.c
```

Here the shared object `libfoo.so.1` is generated from the input file `foo.c`.

Note - This is a simplified example of generating a shared object. Usually, 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 can be bound with dynamic executables to form a runable process. As their name implies, shared objects can be *shared* by more than one application. Because of this potentially far-reaching effect, this chapter describes this form of link-editor output in greater depth than has been covered in previous chapters.

For a shared object to be bound to a dynamic executable or another shared object, it must first be available to the link-edit of the required output file. During this link-edit, any input shared objects are interpreted as if they had been added to the logical address space of the output file being produced. 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. A small amount of bookkeeping information is maintained within the output file to describe these

dependencies. The runtime linker interprets this information and completes the processing of these shared objects as part of creating a runable process.

The following sections expand upon the use of shared objects within the *compilation* and *runtime* environments (these environments are introduced in “Shared Objects” on page 4). Issues that complement and help coordinate the use of shared objects within these environments are covered, together with techniques that maximize the efficiency of 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 (see “ELF Header” on page 148). From this information the processing requirements of the file are deduced. However, shared objects usually follow one of two naming conventions depending on whether they are being used as part of the compilation environment or the runtime environment.

When used as part of the compilation environment, shared objects are read and processed by the link-editor. Although these shared objects can be specified by explicit 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 (see “Shared Object Processing” on page 12).

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 might 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.

A 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 might drop the conventional `lib` prefix. Examples of shared objects that fall into this category are those used solely with **dlopen**(3X). A suffix of `.so` 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.

Note - The shared object name used in a **dlopen**(3X) is usually 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 (see "Loading Additional Objects" on page 51 for more details).

In Chapter 5 the concept of versioning a shared objects interface over a series of software releases is described in more detail. In addition, 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 described.

Recording a Shared Object Name

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

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

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

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

As these examples show, this mechanism of recording dependencies can result in inconsistencies due to different compilation techniques. Also, the location of a shared object as referenced during the link-edit might differ from the eventual location of the shared object on an installed system.

To provide a more consistent means of specifying dependencies, shared objects can record within themselves the filename by which they should be referenced at runtime.

During the link-edit of a shared object, its runtime name can be recorded within the shared object itself by using the `–h` option. 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 that has the SONAME tag. For example:

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

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

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

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

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

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

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

In the 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 recommended, as it provides flexibility by allowing the runtime linker to use a set of rules to locate the actual file (see "Locating Shared Object Dependencies" on page 44 for more details).

Inclusion of Shared Objects in Archives

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

An archive can be built from one or more shared objects and then used to generate a dynamic executable or shared object. Shared objects can be extracted from the archive to satisfy the requirements of the link-edit (see "Archive Processing" on page 11 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 the archive will be recorded as dependencies.

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

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

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 for the dependency.

Note - The runtime 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 several consistency checks to insure that any dependency names that will 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.

If the runtime name of a shared object being generated matches one of its dependencies the link-editor will also 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
```

Shared Objects with Dependencies

Although most of the examples presented in this chapter so far have shown how shared object dependencies are maintained in dynamic executables, it is quite common for shared objects to have their own dependencies (this was introduced in "Shared Object Processing" on page 12).

In "Directories Searched by the Runtime Linker" on page 44 the search rules used by the runtime linker to locate shared object dependencies are 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 (see "Directories Searched by the Runtime Linker" on page 44 for a more extensive discussion).

Dependency Ordering

In most of the examples in this document, dependencies of dynamic executables and shared objects are portrayed as unique and relatively simple (the breadth-first ordering of dependent shared objects is described in “Locating Shared Object Dependencies” on page 44). From these examples, the ordering of shared objects as they are brought into the process address space might 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 can become less predictable.

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

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

If you create a dynamic executable, `prog`, using this shared object, and also define 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, had the developer of the shared object `libfoo.so.1` placed a requirement on the order of processing of its dependencies, this requirement will be compromised by the construction of the dynamic executable `prog`.

Developers who place special emphasis on symbol interposition (see “Symbol Lookup” on page 47, “Symbol Lookup” on page 57 and “Using Interposition” on page 65) and `.init` section processing (see “Initialization and Termination Routines” on page 52) 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 used to provide indirection to an alternative shared object. Two forms of shared object filter exist:

- a *standard* filter
- an *auxiliary* filter

A *standard* filter, in essence, consists solely of a symbol table, and provides a mechanism of *abstracting* the compilation environment from the runtime environment. A link-edit using the filter will reference the symbols provided by the filter itself, however the implementation of the symbol reference is provided from an alternative source at runtime.

Standard filters are identified using the link-editor's `-F` flag. This flag takes an associated filename indicating the shared object that will supply symbol references at runtime. This shared object is referred to as the *filtee*. Multiple use of the `-F` flag allows multiple *filtees* to be recorded.

If the filtee cannot be processed at runtime, or any symbol defined by the filter cannot be located within the filtee, the filter is ignored and symbol resolution continues to the next associated dependency.

An *auxiliary* filter has a similar mechanism, however the filter itself contains an implementation corresponding to its symbols. A link-edit using the filter will reference the symbols provided by the filter itself, however the implementation of the symbol reference can be provided from an alternative source at runtime.

Auxiliary filters are identified using the link-editor's `-f` flag. This flag takes an associated filename indicating the shared object that can be used to supply symbols at runtime. This shared object is referred to as the *filtee.* Multiple use of the `-f` flag allows multiple *filtees* to be recorded.

If the filtee cannot be processed at runtime, or any symbol defined by the filter cannot be located within the filtee, the implementation of the symbol within the filter will be used.

Generating a Standard Filter

First let's define a filtee, `libbar.so.1`, on which this filter technology will be applied. This filtee might be built from several relocatable objects. One of these objects originates from the file `bar.c`, and supplies the symbols `foo` and `bar`:

```
$ cat bar.c
char * bar = "bar";
```

(continued)

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

A standard filter, `libfoo.so.1`, is generated for the symbols `foo` and `bar`, and indicates the association to the filtee `libbar.so.1`. For example:

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

char * foo(){}

$ 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.

If the link-editor references the standard filter `libfoo.so.1` to build a dynamic executable or shared object, it will use the information from the filters symbol table during symbol resolution (see “Symbol Resolution” on page 19 for more details).

At runtime, any reference to the symbols of the filter will result in the additional loading of the filtee `libbar.so.1`. The runtime linker will use this filtee to resolve any symbols defined by `libfoo.so.1`.

For example, the following dynamic executable, `prog`, references the symbols `foo` and `bar` which are resolved during link-edit from the filter `libfoo.so.1`:

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

main(){
    (void) printf("foo() is %s: bar=%s\n", foo(), bar);
}
$ cc -o prog main.c -R. -L. -lfoo
$ prog
```

```
foo() is defined in bar.c: bar=bar
```

The execution of the dynamic executable `prog` results in the function `foo( )`, and the data item `bar`, being obtained from the filtee `libbar.so.1`, *not* from the filter `libfoo.so.1`.

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

Standard filters provide a mechanism for defining a subset interface of an existing shared object, or an interface group spanning a number of existing shared objects. Two filters used in `Solaris` are `/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`. This subset represents the ABI-conforming functions and data items that reside in the C library that *must* be imported by a conforming application.

The latter defines the user interface to the runtime linker itself. This interface provides an abstraction between the symbols referenced in a compilation environment (from `libdl.so.1`) and the actual implementation binding produced within the runtime environment (from `ld.so.1`).

An example of a filter that uses mutiple filtees is `/usr/lib/libxnet.so.1`. This library provides socket and XTI interfaces from `/usr/lib/libsocket.so.1`, `/usr/lib/libnsl.so.1`, and `/usr/lib/libc.so.1`.

As any code in a standard filter is never referenced at runtime, there is no point in adding content to any functions defined within the filter. Any filter code might require relocation, which 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. Data items should always be initialized to insure that they result in references from dynamic executables.

Some of the more complex symbol resolutions carried out by the link-editor require knowledge of a symbol's attributes, including the symbols size (see "Symbol Resolution" on page 19 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 filtee. This insures that the link-editing process will analyze the filter in a manner compatible with the symbol definitions used at runtime.

Generating an Auxiliary Filter

The creation of an *auxiliary* filter is essentially the same as for a *standard* filter (see "Generating a Standard Filter" on page 78 for more details). First let's define a filtee, libbar.so.1, on which this filter technology will be applied. This filtee might be built from several relocatable objects. One of these objects originates from the file bar.c, and supplies the symbol foo:

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

An auxiliary filter, libfoo.so.1, is generated for the symbols foo and bar, and indicates the association to the filtee libbar.so.1. For example:

```
$ cat foo.c
char * bar = "foo";

char * foo()
{
    return ("defined in foo.c");
}
$ 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|AUXILIARY"
[1]     SONAME    libfoo.so.1
[2]     AUXILIARY 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.

If the link-editor references the auxiliary filter libfoo.so.1 to build a dynamic executable or shared object, it will use the information from the filters symbol table during symbol resolution (see "Symbol Resolution" on page 19 for more details).

At runtime, any reference to the symbols of the filter will result in a search for the filtee libbar.so.1. If this filtee is found the runtime linker will use this filtee to resolve any symbols defined by libfoo.so.1. If the filtee is not found, or a symbol from the filter is not found in the filtee, then the original value of the symbol within the filter is used.

For example, the following dynamic executable, prog, references the symbols foo and bar which are resolved during link-edit from the filter libfoo.so.1:

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

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

The execution of the dynamic executable `prog` results in the function `foo()` being obtained from the filtee `libbar.so.1`, ***not*** from the filter `libfoo.so.1`. However, the data item `bar` is obtained from the filter `libfoo.so.1`, as this symbol has no alternative definition in the filtee `libbar.so.1`.

Auxiliary filters provide a mechanism for defining an alternative interface of an existing shared object. This mechanism is used in `Solaris` to provide optimized functionality within platform specific shared objects.

Note - The environment variable `LD_NOAUXFLTR` can be set to disable the runtime linkers ***auxiliary*** filter processing. This may be useful in evaluating a ***filtees*** use and performance impact.

Platform Specific Shared Objects

An extension to the auxiliary filter naming mechanism provides for the use of the reserved token `$PLATFORM`. This token is expanded at runtime to reflect the underlying hardware implementation as displayed by the utility **`uname`**(1) with the `-i` option.

The following example shows how the auxiliary filter `libfoo.so.1` can be designed to access a platform specific filtee `libbar.so.1`:

```
$ LD_OPTIONS='-f /usr/platform/$PLATFORM/lib/libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
[1]     SONAME    libfoo.so.1
[2]     AUXILIARY /usr/platform/$PLATFORM/lib/libbar.so.1
```

This mechanism is used on `Solaris` to provide platform specific extensions to the shared object `/usr/lib/libc.so.1`.

Filtee Processing

The runtime linkers processing of a filter defers the loading of a filtee until a reference to a symbol within the filter has occurred. This implementation is analogous to the filter performing a **dlopen**(3X) on each of its filtees as they are required. This implementation accounts for differences in dependency reporting that can be produced by tools such as **ldd**(1).

By default, **ldd**(1) lists the dependencies of a dynamic object, and thus reports a filter as it would any other dependency. However, use of the `-d` or `-r` options may result in relocation processing that will cause a symbol lookup in the filter. In this case filtee processing will be triggered which can result in additional dependencies being reported by **ldd**(1). To insure immediate processing of any filtee the `-l` option can be used.

The link-editors' `-z loadfltr` option can be used when creating a filter to cause the immediate processing of its filtees at runtime. In addition, the immediate processing of any filtees within a process can be triggered by setting the `LD_LOADFLTR` environment variable.

Performance Considerations

A shared object can be used by multiple applications within the same system. The performance of a shared object therefore can 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 affect the performance of a running process, the performance issues focused upon here target the runtime processing of the shared object itself. The following sections investigate this processing in more detail by looking at aspects such as text size and purity, together with relocation overhead.

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 see Chapter 7). The size of a file can be displayed using the **size**(1) command. For example:

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

(continued)

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

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

Sections are allocated to units known as *segments*, some of which describe how portions of a file 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

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

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 previous **nm**(1) and **dump**(1) examples, the association of the functions `_init`, `foo` and `_fini` to the sections `.init`, `.text` and `.fini` can be seen. These sections, because of their read-only nature, are part of the *text* segment.

Similarly, it can be seen that the data arrays `data` and `bss` are associated with the sections `.data` and `.bss` respectively. These sections, because of their writable nature, are part of the *data* segment.

Note - The previous **dump**(1) display has been simplified for this example.

Armed with this tool information, you can analyze the location of code and data within any ELF file you generate. 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 loadable 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 usually 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 referred to as *impure*.

The segments of a shared object that are mapped into memory fall into two basic categories; the *text* segment, which is read-only, and the *data* segment which is read-write (see "Useful Tools" on page 83 on how to obtain this information from an ELF file). An overriding goal when developing a shared object is to maximize the text segment and minimize the data segment. This optimizes the amount of code sharing while reducing the amount of processing needed to initialize and use a shared object. The following sections present mechanisms that can help achieve this goal.

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 usually tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of a process. Because the code is not tied to a specific address, it will execute correctly without page modification at a different address in each process that uses it.

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. See "Global Offset Table (Processor-Specific)" on page 218 and "Procedure Linkage Table (Processor-Specific)" on page 219 for more details on the use of these two sections.

If a shared object is built from code that is not position-independent, the text segment will usually require a large number of relocations to be performed at

runtime. Although the runtime linker is equipped to handle this, the system overhead this creates can cause serious performance degradation.

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 are 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 usually 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 (see **cc**(1) for more details).

Maximizing Shareability

As mentioned in "The Underlying System" on page 86 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 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 several 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 can 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 is:

```
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 is 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, can detect read-only assignments that will result in runtime relocations, and will arrange for placing such items in writable segments (for example `.picdata`).

Collapse Multiply-Defined Data

Data can be reduced by collapsing multiply-defined data. A program with multiple occurrences of the same error messages can be better off by defining one global datum, and have all other instances reference this. For example:

```
const char * Errmsg = "prog: error encountered: %d";

foo()
{
    ......
    (void) fprintf(stderr, Errmsg, error);
    ......
```

The main candidates for this sort of data reduction are strings. String usage in a shared object can be investigated using **strings**(1). 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 usually result in a corresponding reduction in the number of runtime relocations required.

Allocate Buffers Dynamically

Large data buffers should usually be allocated dynamically rather than being defined using permanent storage. Often this 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 88 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 can be used by many processes, any reduction in the number of page faults generated by accessing a shared object will benefit the process and the system as a whole.

Organizing frequently used routines and their data to an adjacent set of pages will frequently improve performance because it improves the locality of reference. When a process calls one of these functions it might 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 "Relocation Processing" on page 46 the mechanisms by which the runtime linker relocates dynamic executables and shared objects to create a runable process was covered. "Symbol Lookup" on page 47 and "When Relocations are Performed" on page 48 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 are 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, then this symbol is a strong candidate for conversion to a *static* or *automatic* variable. A symbol reduction can also be applied to removed symbols from a shared objects interface (see "Reducing Symbol Scope" on page 33 for more details). By making these conversions the link-editor will incur the expense of processing any symbolic relocation against these symbols during the shared object's creation.

The only global data items that should be visible from a shared object are those that contribute to its user interface. 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 before the application gains 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 by a functional interface. This conversion usually 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 usually built with position-independent code. References to external data items from code of this type employs indirect addressing through a set of tables (see "Position-Independent Code" on page 86 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 can only be achieved at runtime by modifying the code that makes the reference. Modifying any text segment is something to be avoided, and so 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 ****
```

(continued)

```
.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 downsides to this technique. First, each application pays a performance penalty for the overhead of copying the data at run time. 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 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]);
}
```

The 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 object `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 is 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 will copy as much data as the shared object has to offer, but only accepts as much as its allocated space allows.

Profiling Shared Objects

The runtime linker is capable of generating profiling information for any shared objects processed during the running of an application. This is possible because the runtime linker is responsible for binding shared objects to an application and is therefore able to intercept any *global* function bindings (these bindings take place through `.plt` entries - see “When Relocations are Performed” on page 48 for details of this mechanism).

The profiling of a shared object is enabled by specifying its name with the `LD_PROFILE` environment variable. You can analyze one shared object at a time using this environment variable. However, the setting of the environment variable can be used to analyze one or more applications use of the shared object. In the following example the use of `libc` by the single invocation of the command **ls**(1) is analyzed:

```
$ LD_PROFILE=libc.so.1  ls -l
```

In the following example the environment variable setting will cause any applications use of `libc` to accumulate the analyzed information for the duration that the environment variable is set:

```
$ LD_PROFILE=libc.so.1; export LD_PROFILE
$ ls -l
$ make
$ ...
```

When profiling is enabled, a profile data file is created, if it doesn’t already exist, and is mapped by the runtime linker. In the above examples this data file is `/var/tmp/libc.so.1.profile`. You can also specify an alternative directory to store the profile data using the `LD_PROFILE_OUTPUT` environment variable.

This profile data file is used to deposit **profil**(2) data and call count information related to the specified shared objects use. This profiled data can be directly examined with **gprof**(1).

Note - **gprof**(1) is most commonly used to analyze the `gmon.out` profile data created by an executable that has been compiled with the `–xpg` option of **cc**(1). The runtime linkers profile analysis does *not* require any code to be compiled with this option. Applications whose dependent shared objects are being profiled should not make calls to **profil**(2), because this system call does not provide for multiple invocations within the same process. For the same reason, these applications must not be compiled with the `–xpg` option of **cc**(1), as this compiler generated mechanism of profiling is also built on top of **profil**(2).

One of the most powerful features of this profiling mechanism is to allow the analysis of a shared object as used by multiple applications. Frequently profiling analysis is carried out using one or two applications. However, a shared object, by its very nature, can be used by a multitude of applications. Analyzing how these applications use the shared object can offer insights into where energy might be spent to improvement the *overall* performance of the shared object.

The following example shows a performance analysis of `libc` over a build of several applications within a source hierarchy:

```
$ LD_PROFILE=libc.so.1 ; export LD_PROFILE
$ make
$ gprof -b /usr/lib/libc.so.1 /var/tmp/libc.so.1.profile
.....

granularity: each sample hit covers 4 byte(s) ....

                                  called/total     parents
index  %time    self descendents  called+self    name      index
                                  called/total     children
.....
-----------------------------------------------
                0.33        0.00      52/29381     _gettxt [96]
                1.12        0.00     174/29381     _tzload [54]
               10.50        0.00    1634/29381     <external>
               16.14        0.00    2512/29381     _opendir [15]
              160.65        0.00   25009/29381     _endopen [3]
[2]     35.0  188.74        0.00   29381         _open [2]
-----------------------------------------------
.....
granularity: each sample hit covers 4 byte(s) ....

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 35.0     188.74   188.74    29381     6.42     6.42  _open [2]
 13.0     258.80    70.06    12094     5.79     5.79  _write [4]
  9.9     312.32    53.52    34303     1.56     1.56  _read [6]
  7.1     350.53    38.21     1177    32.46    32.46  _fork [9]
 ....
```

The special name *<external>* indicates a reference from outside of the address range of the shared object being profiled. Thus, in the above example, 1634 calls to the function **`open`**(2) within `libc` occurred from the dynamic executables, or from other shared objects, bound with `libc` while the profiling analysis was in progress.

Note - The profiling of shared objects is multi-threaded safe except in the case where one thread calls **`fork`**(2) while another thread is updating the profile data information. The use of **`fork`**(2) removes this restriction.

CHAPTER **5**

Versioning

Overview

ELF objects processed by the link-editors provide many global symbols to which other objects can bind. These symbols describe the object's *application binary interface* (ABI). During the evolution of an object this interface can change due to the addition or deletion of global symbols. In addition, the objects evolution can involve internal implementation changes.

Versioning refers to several techniques that can be applied to an object to indicate interface and implementation changes. These techniques provide for the objects controlled evolution while maintaining backward compatibility.

This chapter describes how an object's ABI can be defined, classifies how changes to this interface can affect backward compatibility, and presents models by which interface and implementation changes can be incorporated into new releases of the object.

The focus of this chapter is the runtime interfaces of dynamic executables and shared objects. The techniques used to describe and manage changes within these dynamic objects are presented in generic terms. A common set of naming conventions and versioning scenarios, as applied to shared objects, can be found in Appendix B.

It is important that developers of dynamic objects be aware of the ramifications of an interface change, and understand how such changes can be managed, especially in regards to maintaining backward compatibility with previously shipped objects.

The global symbols made available by any dynamic object represent the object's *public* interface. Frequently, the number of global symbols remaining in an object at the end of a link-edit are more than you would like to make public. These global

symbols derive from the interrelationships required between the relocatable objects used to build the object, and represent *private* interfaces within the object itself.

A precursor to defining an object's binary interface is to first define only those global symbols you wish to make publicly available from the object being created. These public symbols can be established using the link-editor's `–M` option and an associated `mapfile` as part of the final link-edit. This technique is introduced in "Reducing Symbol Scope" on page 33. This public interface establishes one or more *version definitions* within the object being created, and forms the foundation for the addition of new interfaces as the object evolves.

The following sections build upon this initial public interface. First though, it is useful to understand how various changes to an interface can be categorized so that they can be managed appropriately.

Interface Compatibility

There are many types of change that can be made to an object. In their simplest terms these changes can be categorized into one of two groups:

- *compatible* updates. These updates are *additive*, in that all previously available interfaces remain intact.
- *incompatible* updates. These updates have changed the existing interface in such a way that existing users of the interface can fail or behave incorrectly.

The following list attempts to clarify some common object changes into one of the above categorizations:

- the addition of a symbol - a *compatible* update.
- the removal of a symbol - an *incompatible* update.
- the addition of an argument to a non- **`varargs`**(5) function - an *incompatible* change.
- the removal of an argument from a function - an *incompatible* update.
- the change of size, or content, of a data item to a function or as an external definition - an *incompatible* change.
- a bug fix, or internal enhancement to a function - a *compatible* change providing the semantic properties of the object remain unchanged, otherwise, this is an *incompatible* change.

Note - It is possible, because of interposition, that the addition of a symbol can constitute an *incompatible* update, such that the new symbol might conflict with an applications use of that symbol. However, this does seem rare in practice as source level name space management is commonly used.

Compatible updates can be accommodated by maintaining version definitions *internal* to the object being generated. Incompatible updates can be accommodated by producing a new object with a new *external* versioned name. Both of these versioning techniques allow for the selective binding of applications as well as verification of correct version binding at runtime. These two techniques are explored in more detail in the following sections.

Internal Versioning

A dynamic object can have associated with it one or more internal *version definitions*. Each version definition is commonly associated with one or more symbol names. A symbol name can only be associated with *one* version definition, however a version definition can inherit the symbols from other version definitions. Thus, a structure exists to define one or more independent, or related, version definitions within the object being created. As new changes are made to the object, new version definitions can be added to express these changes.

There are two consequences of providing version definitions within a shared object:

- Dynamic objects that are built against this shared object can record their dependency on the version definitions they bind to. These version dependencies will be verified at runtime to insure that the appropriate interfaces, or functionality, are available for the correct execution of an application.
- Dynamic objects can select only those version definitions of a shared object that they wish to bind to during their link-edit. This mechanism allows developers to control their dependency on a shared object to the interfaces, or functionality, that provide the most flexibility.

Creating a Version Definition

Version definitions commonly consist of an association of symbol names to a unique version *name*. These associations are established within a `mapfile` and supplied to the final link-edit of an object using the link-editor's `-M` option (this technique was introduced in the section "Reducing Symbol Scope" on page 33).

A version definition is established whenever a version name is specified as part of the `mapfile` directive. In the following example two source files are combined, together with `mapfile` directives, to produce an object with a defined public interface:

```
$ cat foo.c
extern  const char * _foo1;
```

(continued)

```
void foo1()
{
        (void) printf(_foo1);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";

$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x0001058c|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[35]    |0x00000454|0x00000034|FUNC |GLOB |0x0  |9    |foo1
```

Here, the symbol `foo1` is the *only* global symbol defined to provide the shared object's public interface. The special *auto-reduction* directive "*" causes the reduction of all other global symbols to have local binding within the object being generated (this directive is introduced in "Defining Additional Symbols" on page 28). The associated version name, `SUNW_1.1`, causes the generation of a version definition. Thus, the shared object's public interface consists of the internal version definition `SUNW_1.1`, associated with the global symbol `foo1`.

Whenever a version definition, or the *auto-reduction* directive, are used to generate an object, a *base version* definition is also created. This base version is defined using the name of the file itself, and is used to associate any reserved symbols generated by the link-editor (see "Generating the Output Image" on page 37 for a list of these reserved symbols).

The version definitions contained within an object can be displayed using **pvs**(1) with the `-d` option:

```
$ pvs -d libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
```

Here, the object `libfoo.so.1` has an internal version definition named `SUNW_1.1`, together with a base version definition `libfoo.so.1`.

Note - The link-editor's `-z noversion` option allows `mapfile` directed symbol reduction to be performed but suppresses the creation of version definitions.

Starting with this initial version definition, it is possible for the object to evolve by adding new interfaces and updated functionality. For example, a new function, `foo2`, together with its supporting data structures, can be added to the object by updating the source files `foo.c` and `data.c`:

```
$ cat foo.c
extern  const char * _foo1;
extern  const char * _foo2;

void foo1()
{
        (void) printf(_foo1);
}

void foo2()
{
        (void) printf(_foo2);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";
const char * _foo2 = "string used by foo2()\n";
```

A new version definition, `SUNW_1.2`, can be created to define a new interface representing the symbol `foo2`. In addition, this new interface can be defined to inherit the original version definition `SUNW_1.1`.

The creation of this new interface is important as it identifies the evolution of the object and enables users to verify and select the interfaces to which they bind. These concepts are covered in more detail in “Binding to a Version Definition” on page 105 and in “Specifying a Version Binding” on page 109.

The following example shows the `mapfile` directives that create these two interfaces:

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                  # Release X+1
        global:
                foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]    |0x00010644|0x00000004|OBJT |LOCL |0x0  |17   |_foo1
[34]    |0x00010648|0x00000004|OBJT |LOCL |0x0  |17   |_foo2
[36]    |0x000004bc|0x00000034|FUNC |GLOB |0x0  |9    |foo1
```

(continued)

```
[37]    |0x000004f0|0x00000034|FUNC |GLOB |0x0  |9    |foo2
```

Here, the symbols `foo1` and `foo2` are both defined to be part of the shared object's public interface. However, each of these symbols is assigned to a different version definition; `foo1` is assigned to `SUNW_1.1`, and `foo2` is assigned to `SUNW_1.2`.

These version definitions, their inheritance, and their symbol association can be displayed using **pvs**(1) together with the `-d`, `-v` and `-s` options:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1}:
                foo2;
                SUNW_1.2
```

Here, the version definition `SUNW_1.2` has a dependency on the version definition `SUNW_1.1`.

The inheritance of one version definition by another is a useful technique that reduces the version information that will eventually be recorded by any object that binds to a version dependency. Version inheritance is covered in more detail in the section "Binding to a Version Definition" on page 105.

Any internal version definition will have an associated *version definition symbol* created. As shown in the previous **pvs**(1) example, these symbols are displayed when using the `-v` option.

Creating a Weak Version Definition

Internal changes to an object that do not require the introduction of a new interface definition, can be defined by creating a *weak* version definition. Examples of such changes are bug fixes or performance improvements.

Such a version definition is empty, in that it has no global interface symbols associated with it.

For example, if the data file `data.c`, used in the previous examples, is updated to provide more detailed string definitions:

```
$ cat data.c
const char * _foo1 = "string used by function foo1()\n";
const char * _foo2 = "string used by function foo2()\n";
```

then a weak version definition can be introduced to identify this change:

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                  # Release X+1
        global:
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;    # Release X+2

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:      {SUNW_1.2};
```

Here, the empty version definition is signified by the *weak* label. These *weak* version definitions allow applications to verify the existence of a particular implementation by binding to the version definition associated with that functionality. The section "Binding to a Version Definition" on page 105 illustrates how these definitions can be used in more detail.

Defining Unrelated Interfaces

The previous examples have shown how new version definitions added to an object have inherited any existing version definitions. It is also possible to create version definitions that are unique and independent. In the following example, two new files, `bar1.c` and `bar2.c`, are added to the object `libfoo.so.1`. These files contribute two new symbols, `bar1` and `bar2`, respectively:

```
$ cat bar1.c
extern  void foo1();

void bar1()
{
        foo1();
}
$ cat bar2.c
extern  void foo2();

void bar2()
{
        foo2();
}
```

These two symbols are intended to define two new public interfaces. Neither of these new interfaces are related to each other, however each expresses a dependency on the original `SUNW_1.2` interface.

The following `mapfile` definition creates this required association:

```
$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};

SUNW_1.2 {                  # Release X+1
        global:
                foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2;     # Release X+2

SUNW_1.3a {                 # Release X+3
        global:
                bar1;
} SUNW_1.2;

SUNW_1.3b {                 # Release X+3
        global:
                bar2;
} SUNW_1.2;
```

Again, the version definitions created in `libfoo.so.1` using this `mapfile`, and their related dependencies, can be inspected using **`pvs`**(1):

```
$ cc -o libfoo.so.1 -M mapfile -G foo.o bar.o data.o
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1};
        SUNW_1.2.1 [WEAK]:       {SUNW_1.2};
        SUNW_1.3a:               {SUNW_1.2};
        SUNW_1.3b:               {SUNW_1.2};
```

The following sections explore how these version definition recordings can be used to verify runtime binding requirements and control the binding requirements of an object during its creation.

Binding to a Version Definition

When a dynamic executable or shared object is built against other shared objects, these dependencies are recorded in the resulting object (see "Shared Object Processing" on page 12 and "Recording a Shared Object Name" on page 73 for more details). If these shared object dependencies also contain version definitions then an associated *version dependency* will be recorded in the resulting object.

The following example takes the data files from the previous section and generates a shared object suitable for a compile time environment. This shared object, `libfoo.so.1`, will be used in following binding examples:

```
$ cc -o libfoo.so.1 -h libfoo.so.1 -M mapfile -G foo.o bar.o \ data.o
$ ln -s libfoo.so.1 libfoo.so
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1}:
                foo2;
                SUNW_1.2;
        SUNW_1.2.1 [WEAK]:       {SUNW_1.2}:
                SUNW_1.2.1;
        SUNW_1.3a:               {SUNW_1.2}:
                bar1;
                SUNW_1.3a;
        SUNW_1.3b:               {SUNW_1.2}:
                bar2;
```

(continued)

```
                 SUNW_1.3b
```

In effect, there are six public interfaces being offered by this shared object. Four of these interfaces (SUNW_1.1, SUNW_1.2, SUNW_1.3a and SUNW_1.3b) define a set of functions, one interface (SUNW_1.2.1) describes an internal implementation change to the shared object, and one interface (libfoo.so.1) defines several reserved labels. Dynamic objects that build with this object will record which of these interfaces they bind to.

The following example builds an application that references both symbols foo1 and foo2. The versioning dependency information recorded in the application can be examined using **pvs**(1) with the −r option:

```
$ cat prog.c
extern void foo1();
extern void foo2();

main()
{
        foo1();
        foo2();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2, SUNW_1.2.1);
```

In this example, the application prog has really bound to the two interfaces SUNW_1.1 and SUNW_1.2, as these interfaces have provided the global symbols foo1 and foo2 respectively.

However, since version definition SUNW_1.1 is defined within libfoo.so.1 as being inherited by the version definition SUNW_1.2, it is only necessary to record the latter version dependency. This *normalization* of version definition dependencies reduces the amount of version information that must be maintained within an object and processed at runtime.

Since the application prog was built against the shared object's implementation containing the *weak* version definition SUNW_1.2.1, this dependency is also recorded. Even though this version definition is defined to inherit the version definition SUNW_1.2, the version's weak nature precludes its normalization with SUNW_1.1, and results in a separate dependency recording.

Had there been multiple weak version definitions that inherit from each other then these definitions will be normalized in the same manner as non-weak version definitions are.

Note - The recording of a version dependency can be suppressed by the link-editor's –z noversion option.

Having recorded these version definition dependencies, the runtime linker validates the existence of the required version definitions in the objects bound to when the application is executed. This validation can be displayed using **ldd**(1) with the –v option. For example, by running **ldd**(1) on the application prog, the version definition dependencies are shown to be found correctly in the shared object libfoo.so.1:

```
$ ldd -v prog

   find library=libfoo.so.1; required by prog
        libfoo.so.1 =>   ./libfoo.so.1
   find version=libfoo.so.1;
        libfoo.so.1 (SUNW_1.2) =>           ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>         ./libfoo.so.1
   ....
```

Note - **ldd**(1) with the –v option implies *verbose* output, in that a recursive list of all dependencies, together with all versioning requirements, will be generated.

If a non-weak version definition dependency cannot be found, a fatal error will occur during application initialization. Any weak version definition dependency that cannot be found is silently ignored. For example, if the application prog was run in an environment in which libfoo.so.1 only contained the version definition SUNW_1.1, then the following fatal error will occur:

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
$ prog
ld.so.1: prog: fatal: libfoo.so.1: version 'SUNW_1.2' not \
found (required by file prog)
```

Had the application prog not recorded any version definition dependencies, the nonexistence of the required interface symbol foo2 will have manifested itself sometime during the execution of the application as a fatal relocation error (see "Relocation Errors" on page 49). This relocation error might occur at process initialization, during process execution, or might not occur at all if the execution path of the application did not call the function foo2.

Recording version definition dependencies provides an alternative, and immediate indication of the availability of the interfaces required by the application.

If the application prog was run in an environment in which libfoo.so.1 only contained the version definitions SUNW_1.1 and SUNW_1.2, then all non-weak version definition requirements will be satisfied. The absence of the weak version definition SUNW_1.2.1 is deemed nonfatal, and so no runtime error condition will be generated. However, **ldd**(1) can be used to display *all* version definitions that cannot be found:

```
$ pvs -dv libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
        SUNW_1.2:                {SUNW_1.1};
$ prog
string used by foo1()
string used by foo2()
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>         (version not found)
        ...........
```

Note - If an object requires a version definition from a given dependency, and at runtime an implementation of that dependency is found that contains no version definition information, the version verification of the dependency will be silently ignored. This policy provides a level of backward compatibility as the transition from non-versioned to versioned shared objects is taken. **ldd**(1) however, can still be used to display any version requirement discrepancies.

Verifying Versions in Additional Objects

Version definition symbols also provide a mechanism for verifying the version requirements of an object obtained by **dlopen**(3X). Any object added to the process's address space using this function will have no automatic version dependency verification carried out by the runtime linker. Thus, it is the responsibility of the caller of this function to verify that any versioning requirements are met.

The presence of a required version definition can be verified by looking up the associated version definition symbol using **dlsym**(3X). The following example shows the shared object libfoo.so.1 being added to a process by **dlopen**(3X) and verified to insure that the interface SUNW_1.2 is available:

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

main()
{
    void *       handle;
```

(continued)

```
const char * file = "libfoo.so.1";
const char * vers = "SUNW_1.2";
....

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

if (dlsym(handle, vers) == NULL) {
        (void) printf("fatal: %s: version ‘%s’ not found\n",
             file, vers);
        exit (1);
}
....
```

Specifying a Version Binding

When building a dynamic object against a shared object containing version definitions, it is possible to instruct the link-editor to limit the binding to specific version definitions. Effectively, the link-editor allows you to control an object's binding to specific interfaces.

An object's binding requirements can be controlled using a *file control directive*. This directive is supplied using the link-editor's `-M` option and an associated `mapfile`. The syntax for these file control `mapfile` directives is shown below:

```
name - version [ version ... ] [ $ADDVERS=version ];
```

- *name* - represents the name of the shared object dependency. This name should match the shared object's compilation environment name as used by the link-editor (see “Library Naming Conventions” on page 13).
- *version* - represents the version definition name within the shared object that should be made available for binding. Multiple version definitions can be specified.
- *$ADDVERS* - allows for additional version definitions to be recorded.

There are a couple of scenarios where this binding control can be useful:

- If a shared object has been versioned to define unique and independent versions, possibly defining different standards interfaces, then the application can insure that its bindings meet the requirements of a specific interface.
- If a shared object has been versioned over several software releases, application developers can restrict themselves to the interfaces that were available in a previous software release. Thus, an application can be built using the latest release

of the shared object in the knowledge that the application's interface requirements can be met by a previous release of the shared object.

The following is an example of using the version control mechanism. This example uses the shared object `libfoo.so.1` containing the following version interface definitions:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:
                foo1;
                foo2;
                SUNW_1.1;
        SUNW_1.2:            {SUNW_1.1}:
                bar;
```

The version definitions `SUNW_1.1` and `SUNW_1.2` represent interfaces within `libfoo.so.1` that were made available in software `Release X` and `Release X+1` respectively.

An application can be built to bind only to the interfaces available in `Release X` by using the following version control `mapfile` directive:

```
$ cat mapfile
libfoo.so - SUNW_1.1;
```

For example, if you develop an application, `prog`, and wish to insure that the application will run on `Release X`, then the application can only use the interfaces available in that release. If the application mistakenly references the symbol `bar`, then the application's noncompliance to the required interface will be signalled by the link-editor as an undefined symbol error:

```
$ cat prog.c
extern void foo1();
extern void bar();

main()
{
        foo1();
        bar();
}
$ cc -o prog prog.c -M mapfile -L. -R. -lfoo
```

(continued)

```
Undefined           first referenced
 symbol                 in file
bar                     prog.o  (symbol belongs to unavailable \
                        version ./libfoo.so (SUNW_1.2))
ld: fatal: Symbol referencing errors. No output written to prog
```

To be compliant with the SUNW_1.1 interface, you must remove the *reference* to bar. This can be achieved either by reworking the application to remove the requirement on bar, or by adding an implementation of bar to the build of the application.

Binding to Additional Version Definitions

Sometimes it is desirable to record additional version dependencies than would be produced from the normal symbol binding of an object. This can be achieved using the $ADDVERS file control directive. This section describes a couple of scenarios where this additional binding may be useful.

Continuing with the libfoo.so.1 example, assume that in Release X+2 the version definition SUNW_1.1 is subdivided into two standard releases STAND_A and STAND_B. To preserve compatibility, the SUNW_1.1 version definition must be maintained, but now it is expressed as inheriting the two standard definitions:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:           {STAND_A, STAND_B}:
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
        STAND_A:
                foo1;
                STAND_A;
        STAND_B:
                foo2;
                STAND_B;
```

If we continued to build our application prog so that its only requirement is the interface symbol foo1 it will have a single dependency on the version definition STAND_A. This precludes running prog on a system where libfoo.so.1 is less

than Release X+2 as the version definition STAND_A did not exist in previous releases, even though the interface foo1 did.

Therefore, the application prog can be built to align its requirement with previous releases by creating a dependency on SUNW_1.1 by using the following file control directive:

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cat prog
extern void foo1();

main()
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.1);
```

This explicit dependency is sufficient to encapsulate the true dependency requirements and satisfy compatibility with older releases.

In "Creating a Weak Version Definition" on page 102 it was described how weak version definitions can be used to mark an internal implementation change. These version definitions are well suited to indicate bug fixes and performance improvements made to an object. If the existence of a *weak* version is required for the correct execution of an application, then an explicit dependency on this version definition can be generated.

Establishing such a dependency can be important when a bug fix, or performance improvement, is critical for the application to function correctly.

Continuing with the libfoo.so.1 example, assume a bug fix is incorporated as the weak version definition SUNW_1.2.1 in software Release X+3:

```
$ pvs -dsv libfoo.so.1
        libfoo.so.1:
                _end;
                _GLOBAL_OFFSET_TABLE_;
                _DYNAMIC;
                _edata;
                _PROCEDURE_LINKAGE_TABLE_;
                _etext;
        SUNW_1.1:           {STAND_A, STAND_B}:
                SUNW_1.1;
        SUNW_1.2:           {SUNW_1.1}:
                bar;
        STAND_A:
                foo1;
                STAND_A;
```

(continued)

```
        STAND_B:
                foo2;
                STAND_B;
        SUNW_1.2.1 [WEAK]:  {SUNW_1.2}:
                SUNW_1.2.1;
```

Normally, if an application is built against this shared object, it will record a weak dependency on the version definition `SUNW_1.2.1`. This dependency is informational only, in that it will not cause termination of the application should the version definition not be found in the `libfoo.so.1` used at runtime.

The file control directive `$ADDVERS` can be used to generate an explicit dependency on a version definition. If this definition is weak, then this explicit reference also causes the version definition to be ***promoted*** to a strong dependency.

Therefore, the application `prog` can be built to enforce the requirement that the `SUNW_1.2.1` interface be available at runtime by using the following file control directive:

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.2.1;
$ cat prog
extern void foo1();

main()
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2.1);
```

Here, `prog` has been built with an explicit dependency on the interface `STAND_A`. Because the version definition `SUNW_1.2.1` is promoted to a strong version, it is also normalized with the dependency `STAND_A`. At runtime, if the version definition `SUNW_1.2.1` cannot be found, a fatal error will be generated.

Note - When working with one or two dependencies, explicit binding to a version definition can also be achieved by referencing the version definition symbol. This is most easily achieved by using the link-editors `–u` option. However, a symbol reference is nonselective, and when working with multiple dependencies, which may contain similarly named version definitions, this technique is insufficient to create explicit bindings.

Relocatable Objects

The preceding sections have described how version information can be recorded and used within dynamic objects. Relocatable objects can maintain versioning information in a similar manner, however there are one or two subtle differences in how this information is used.

Any version definitions supplied to the link-edit of a relocatable object are recorded in exactly the same format as has been described in previous examples. However, by default, symbol reduction is not carried out on the object being created. Instead, when the relocatable object is finally used as input to the generation of a dynamic object, the version recording itself will be used to determine the symbol reductions to apply.

In addition, any version definitions found in relocatable objects will be propagated to the dynamic object. For an example of version processing in relocatable objects see "Reducing Symbol Scope" on page 33.

External Versioning

Runtime references to a shared object should always refer to the files *version* filename. Commonly this is expressed as a filename suffixed with a version number. When a shared object's interface changes in an *incompatible* manner - such that it will break old applications - a new shared object should be distributed with a new versioned filename. In addition, the original versioned filename must still be distributed to provide the interfaces required by the old applications.

By providing shared objects as separate versioned filenames within 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 Versioned Filenames

In the section "Naming Conventions" on page 72 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`.

However, at runtime any shared object dependencies should exist in their *versioned* name form. Instead of maintaining two distinct shared objects that follow these naming conventions, 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 can be used. However, as a documentation and diagnostic aid, symbolic 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, three major versions of the shared object are available. Two of these shared objects, `libfoo.so.1` and `libfoo.so.2`, provide the dependencies for existing applications. `libfoo.so.3` offers the latest major release for building and running new applications.

Using this symbolic link mechanism itself is insufficient to coordinate the correct binding of a shared object from its use in the compilation environment to its requirement in the runtime environment. As the example presently stands, the link-editor 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.

To provide the correct runtime name to be recorded as a dependency, the shared object `libfoo.so.1` should be built with an *soname* definition. This definition identifies the shared objects runtime name, and is used as the dependency name by any object that links against this shared object. This definition can be provided using the `-h` option during the link-edit of the shared object itself. For example:

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

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

This symbolic link and the *soname* mechanism 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 ensures that the intended interface will be furnished at runtime.

CHAPTER 6

Support Interfaces

Overview

The link-editors provide a number of support interfaces that allow for the monitoring, and in some cases modification, of link-editor and runtime linker processing. These interfaces typically require a more advanced understanding of link-editing concepts than has been described in previous chapters. The following interfaces are described in this chapter:

- Link-editor support interface (*ld-support*).
- Runtime linker auditing interface (*rtld-audit*).
- Runtime linker debugger interface (*rtld-debugger*).

Link-Editor Support Interface

As described in Chapter 2 the link-editor performs many operations including the opening of files and the concatenation of sections from these files. Monitoring, and sometimes modifying these operations can often be beneficial to components of a compilation system.

This section describes the supported interface for input file inspection, and to some degree, input file data modification of those files that compose a link-edit. This interface is referred to as the *ld-support* interface. Two applications that employ this interface are the link-editor itself which uses it to process debugging information within relocatable objects, and the **make**(1) utility which uses it to save state information.

The *ld-support* interface is composed of a support library that offers one or more support interface routines. This library is loaded as part of the link-edit process, and any support routines found are called at various stages of link-editing.

You should be familiar with the **elf**(3E) structures and file format when using this interface.

Invoking the Support Interface

The link-editor accepts one or more support libraries provided by either the `SGS_SUPPORT` environment variable or with the link-editors' `–S` option. The environment variable consists of a colon separated list of support libraries:

```
$ SGS_SUPPORT=./support.so.1:libldstab.so.1 cc ...
```

The `–S` option specifies a single support library. Multiple `–S` options can be specified:

```
$ ld -S ./support.so.1 -S libldstab.so.1 ...
```

A support library is a shared object. The link-editor performs a **dlopen**(3X) on each shared object, in the order they are specified. If both the environment variable and `–S` option are encountered, then the shared objects specified with the environment variable are processed first. Each support library is then searched, using **dlsym**(3X), for any support interface routines. These support routines are then called at various stages of link-editing.

Note - By default, the `Solaris` support library `libldstab.so.1` is used by the link-editor to process, and compact, compiler generated debugging information supplied within input relocatable objects. This default processing is suppressed if you invoke the link-editor with any support libraries specified using the `–S` option. If the default processing of `libldstab.so.1` is required in addition to your support library services, then `libldstab.so.1` should be explicitly added to the list of support libraries supplied to the link-editor.

Support Interface Functions

All *ld-support* interfaces are defined in the header file `link.h`. All interface arguments are basic C types or ELF types. The ELF data types can be examined with the ELF access library `libelf` (see **elf**(3E) for a description of `libelf` contents). The following interface functions are provided by the *ld-support* interface, and are described in their expected order of use:

```
void ld_start(const char * name, const Elf32_Half type,
        const char * caller);
```

`ld_start()`

This function is called after initial validation of the link-editor command line, and indicates the start of input file processing.

name is the output filename being created. *type* is the output file type, which is either `ET_DYN`, `ET_REL`, or `ET_EXEC` (as defined in `sys/elf.h`). *caller* is the application calling the interface, which is normally `/usr/ccs/bin/ld`.

```
void ld_file(const char * name, const Elf_Kind kind, int flags,
        Elf * elf);
```

`ld_file()`

This function is called for each input file before any processing of the files data is carried out.

name is the input file about to be processed. *kind* indicates the input file type, which is either `ELF_K_AR`, or `ELF_K_ELF` (as defined in `libelf.h`). *flags* indicates how the link-editor obtained the file, and can be one or more of the following definitions:

- `LD_SUP_DERIVED` - the file name was not explicitly named on the command-line. It was either derived from a `-l` expansion or it identifies an extracted archive member.
- `LD_SUP_INHERITED` - the file was obtained as a dependency of a command-line shared object.
- `LD_SUP_EXTRACTED` - the file was extracted from an archive.

If no *flags* values are specified then the input file has been explicitly named on the command-line. *elf* is a pointer to the files ELF descriptor.

```
void ld_section(const char * name, Elf32_Shdr * shdr,
        Elf32_Word sndx, Elf_Data * data, Elf * elf);
```

`ld_section()`

This function is called for each section of the input file before any processing of the section data is carried out.

name is the input section name. *shdr* is a pointer to the associated section header. *sndx* is the section index within the input file. *data* is a pointer to the associated data buffer. *elf* is a pointer to the files ELF descriptor.

Modification of the data is permitted by reallocating the data itself and reassigning the `Elf_Data` buffers `d_buf` pointer. Any modification to the data should insure the correct setting of the `Elf_Data` buffers `d_size` element. For input sections that will become part of the output image, setting the `d_size` element to zero will effectively remove the data from the output image.

Note - Any sections that are stripped by use of the link-editors `-s` option are not reported to `ld_section()`.

```
void ld_atexit(int status);
```

`ld_atexit()`

This function is called on completion of the link-edit.

status is the **`exit`**(2) code that will be returned by the link-editor and is either `EXIT_FAILURE` or `EXIT_SUCCESS` (as defined in `stdlib.h`).

Support Interface Example

The following example creates a support library that prints the section name (see Table 7–14) of any relocatable object file processed as part of a link-edit.

```
$ cat support.c
#include        <link.h>

static int      indent = 0;

void
ld_start(const char * name, const Elf32_Half type,
    const char * caller)
{
        (void) printf("output image: %s\n", name);
}

void
ld_file(const char * name, const Elf_Kind kind, int flags,
    Elf * elf)
{
        if (flags & LD_SUP_EXTRACTED)
                indent = 4;
        else
                indent = 2;
```

(continued)

(Continuation)

```
        (void) printf("%*sfile: %s\n", indent, "", name);
}

void
ld_section(const char * name, Elf32_Shdr * shdr, Elf32_Word sndx,
    Elf_Data * data, Elf * elf)
{
        Elf32_Ehdr *    ehdr = elf32_getehdr(elf);

        if (ehdr->e_type == ET_REL)
                (void) printf("%*s   section [%ld]: %s\n", indent,
                    "", sndx, name);
}
```

This support library is dependent upon `libelf` to provide the ELF access function **`elf32_getehdr`**(3E) used to determine the input file type. The support library is built using:

```
$ cc -o support.so.1 -G -K pic support.c -lelf -lc
```

The following example shows the section diagnostics resulting from the construction of a trivial application from a relocatable object and a local archive library. The invocation of the support library, in addition to default debugging information processing, is brought about by the `–S` option usage:

```
$ LD_OPTIONS=-S./support.so.1:libldstab.so.1 cc -o prog \ main.c -L. -lfoo
output image: prog
  file: /opt/COMPILER/crti.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /opt/COMPILER/crt1.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: /opt/COMPILER/values-xt.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: main.o
     section [1]: .shstrtab
     section [2]: .text
     .......
  file: ./libfoo.a
    file: ./libfoo.a(foo.o)
       section [1]: .shstrtab
       section [2]: .text
       .......
```

(continued)

```
file: /usr/lib/libc.so
file: /opt/COMPILER/crtn.o
   section [1]: .shstrtab
   section [2]: .text
   .......
file: /usr/lib/libdl.so.1
```

Note - The number of sections displayed in this example have been reduced to simplify the output. Also, the files included by the compiler driver can vary.

Runtime Linker Auditing Interface

As described in Chapter 3 the runtime linker performs many operations including the mapping of objects into memory and the binding of symbols. Monitoring, and sometimes modifying these operations from *within* the same process, can enable powerful process monitoring tools.

This section describes the supported interface for a process to access runtime linking information regarding itself. This interface is referred to as the *rtld-audit* interface. One example of the use of this mechanism is the runtime profiling of shared objects described in “Profiling Shared Objects” on page 95.

The *rtld-audit* interface is implemented as an audit library that offers one or more auditing interface routines. If this library is loaded as part of a process, then the audit routines are called by the runtime linker at various stages of process execution. These interfaces allow the audit library to access:

- Information regarding loaded objects.
- Symbol bindings that occur between these loaded objects. These bindings may be altered by the audit library.
- Exploitation of the *lazy* binding mechanism provided by procedure linker table entries (see “Procedure Linkage Table (Processor-Specific)” on page 219), to allow auditing of function calls and their return values. The arguments to a function and its return value may be modified by the audit library.

Some of these facilities can be achieved by *preloading* specialized shared objects (see “Loading Additional Objects” on page 55). However, a preloaded object exists within the same *name-space* as the objects of a process, and this often restricts, or complicates the implementation of the preloaded shared object. The *rtld-audit* interface offers the user a unique name-space in which to execute their audit library, that insures the

audit library does not intrude upon the normal bindings that occur within the process.

Establishing a Name-space

When the runtime linker binds a dynamic executable with its dependencies it builds a linked list of *link-maps* to describe the process. The link-map structure describes each object within the process and is defined in `/usr/include/sys/link.h`. The symbol search mechanism required to bind objects of an application traverses this list of link-maps. This link-map list is said to provide the *name-space* for process symbol resolution.

The runtime linker itself is also described by a link-map, and this link-map is maintained on a different list from that of the application objects. This results in the runtime linker residing in its own unique name-space, which prevents any direct binding of the application to services within the runtime linker. (An application can only call upon the public services of the runtime linker via the filter `libdl.so.1`).

The *rtld-audit* interface employs its own link-map list on which it maintains any audit libraries. This results in the audit libraries being isolated from the symbol binding requirements of the application. Inspection of the application link-map list is possible with **dlmopen**(3X), which when used with the `RTLD_NOLOAD` flag allows the audit library to query an objects existence without causing its loading.

Two identifiers are defined in `/usr/include/link.h` to define the application and runtime linker link-map lists:

```
#define LM_ID_BASE      0     /* application link-map list */
#define LM_ID_LDSO      1     /* runtime linker link-map list */
```

Each *rtld-audit* support library is assigned a unique free link-map identifier.

Building an Audit Library

An audit library is built like any other shared object, however its unique name-space within a process requires some additional care:

- It must provide all dependency requirements.
- It should not use system interfaces that do not provide for multiple instances of the interface within a process.

If the audit library calls **printf**(3C), then the audit library must define a dependency on `libc` (see "Generating a Shared Object" on page 26). Because the audit library has a unique name-space, symbol references cannot be satisfied by the `libc` present in the application being audited. If an audit library has a dependency

on `libc` then two versions of `libc.so.1` will be loaded into the process. One satisfies the binding requirements of the application link-map list, and the other satisfies the binding requirements of the audit link-map list.

To insure audit libraries are built with all dependencies recorded, the link-editors `-z defs` option should be used (see "Generating a Shared Object" on page 26).

Some system interfaces exist assuming they are the only instance of their implementation within a process. For example, threads, signals and **malloc**(3C). Audit libraries should avoid using such interfaces, as doing so may inadvertently alter the behavior of the application.

Note - The allocation of memory using **mapmalloc**(3X) by an audit library is acceptable as this can exist with any allocation scheme normally employed by the application.

Invoking the Auditing Interface

The *rtld_audit* interface is enabled using the runtime linker environment variable `LD_AUDIT`. This environment variable is set to a colon separated list of shared objects that will be loaded by **dlopen**(3X). Each object is loaded onto its own audit link-map list, and each object is searched for audit routines using **dlsym**(3X). Audit routines that are found will be called at various stages during the applications execution.

The *rtld_audit* interface allows for multiple audit libraries to be supplied. Audit libraries that expect to be employed in this fashion should not alter the bindings that would normally be returned by the runtime linker, doing so may produce unexpected results from audit libraries that follow.

Secure applications (see "Security" on page 53) may only obtain audit libraries from *trusted* directories. Presently, the only trusted directories available for audit libraries are `/usr/lib` and `/usr/ccs/lib`.

Audit Interface Functions

The following functions are provided by the *rtld-audit* interface, and are described in their expected order of use:

```
uint_t la_version(uint_t version);
```

`la_version()`

This function provides the initial handshake between the runtime linker and the audit library. This interface *must* be provided by the audit library for it to be loaded.

The runtime linker calls this interface with the highest *version* of the *rtld-audit* interface it is capable of supporting. The audit library can verify that this version is sufficient for its use, and return the version it expects to use. This version is normally `LAV_CURRENT` which is defined in `/usr/include/link.h`.

If the audit library returns a version of zero, or a value greater than the *rtld-audit* interface the runtime linker supports, the audit library will not be used.

```
uint_t la_objopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie);
```

`la_objopen()`

This function is called each time a new object is loaded by the runtime linker.

lmp provides the link-map structure describing the new object. *lmid* identifies the link-map list to which the object has been added (see “Establishing a Name-space” on page 123). *cookie* provides a pointer to an identifer. This identifier is initialized to the objects *lmp*, but may be modified by the audit library to better identify the object to other *rtld-audit* interface routines

This function returns a value indicating the symbol bindings of interest for this object, which will result in later calls to `la_symbind32()`. The return value is a mask of the following values defined in`/usr/include/link.h`:

- `LA_FLG_BINDTO` - audit symbol bindings *to* this object.

- `LA_FLG_BINDFROM` - audit symbol bindings *from* this object.

See `la_symbind()` for more details on the use of these two flags.

A return value of zero indicates that binding information is of no interest for this object.

```
void la_preinit(uintptr_t cookie);
```

`la_preinit()`

This function is called once after all objects have been loaded for the application, but before transfer of control to the application occurs.

cookie identifies the primary object that started the process, normally the dynamic executable.

```
uintptr_t la_symbind32(Elf32_Sym * sym, uint ndx,
        uintptr_t * refcook, uintptr_t * defcook, uint_t * flags);
```

`la_symbind32()`

This function is called when a binding occurs between two objects that have been tagged for binding notification (see `la_objopen()`).

sym is a constructed symbol structure (see `/usr/include/sys/elf.h`), whose *sym->st_value* indicates the address of the symbol definition being bound, and *sym->st_name* points to a symbol name string.

ndx indicates the symbol index within the bound objects dynamic symbol table. *refcook* describes the object making reference to this symbol. This identifier is the same as passed to the `la_objopen()` that returned `LA_FLG_BINDFROM`. *defcook* describes the object defining this symbol. This identifier is the same as passed to the `la_objopen()` that returned `LA_FLG_BINDTO`.

flags points to a data item that can be used to modify the continued auditing of procedure linkage table symbol entries. This value is a mask of the following flags defined in `/usr/include/link.h`:

- `LA_SYMB_NOPLTENTER` - the `la_*_pltenter()` function will *not* be called for this symbol.
- `LA_SYMB_NOPLTEXIT` - the `la_pltexit()` function will *not* be called for this symbol.
- `LA_SYMB_DLSYM` - the symbol binding occured as a result of calling **dlsym**(3X).

By default, if the `la_*_pltenter()` or `la_pltexit()` functions exist within the audit library, these will be called (after `la_symbind32()`) for procedure linkage table symbols each time the symbol is referenced (see also "Audit Interface Limitations" on page 129).

The return value indicates the address to which control should be passed following this call. An audit library that simply monitors symbol binding should return the value of *sym->st_value* so that control is passed to the bound symbol definition. An audit library may intentionally redirect a symbol binding by returning a different value.

```
uint_t la_sparcv8_pltenter(Elf32_Sym * sym, uint ndx,
        uintptr_t * refcook, uintptr_t * defcook,
        La_sparcv8_regs * regs, uint_t * flags);

uint_t la_i86_pltenter(Elf32_Sym * sym, uint ndx,
        uintptr_t * refcook, uintptr_t * defcook,
        La_i86_regs * regs, uint_t * flags);
```

`la_sparcv8_pltenter()`, `la_i86_pltenter()`

These functions are called on a SPARC and x86 system respectively, when a procedure linkage symbol entry between two objects that have been tagged for binding notification is called (see `la_objopen()` and `la_symbind32()`).

sym, *ndx*, *refcook* and *defcook* provide the same information as passed to `la_symbind32()`.

regs points to the *out* registers on a SPARC system, and the *stack* and *frame* registers on a x86 system, as defined in `/usr/include/link.h`.

flags points to a data item that can be used to modify the continuing auditing of this procedure linkage table entry. This data item is the same as pointed to by the *flags* from `la_symbind32()`. This value is a mask of the following flags defined in `/usr/include/link.h`:

- `LA_SYMB_NOPLTENTER` - the `la_sparcv8_pltenter()` or `la_i86_pltenter()` function will *not* be called subsequently for this symbol.
- `LA_SYMB_NOPLTEXIT` - the `la_pltexit()` function will *not* be called for this symbol.

The return value indicates the address to which control should be passed following this call. An audit library that simply monitors symbol binding should return the value of *sym->st_value* so that control is passed to the bound symbol definition. An audit library may intentionally redirect a symbol binding by returning a different value.

```
uint_t la_pltexit(Elf32_Sym * sym, uint ndx, uintptr_t * refcook,
        uintptr_t * defcook, uintptr_t retval);
```

`la_pltexit()`

This function is called when a procedure linkage symbol entry between two objects that have been tagged for binding notification (see `la_objopen()` and `la_symbind32()`) returns, but before control reaches the caller.

sym, *ndx*, *refcook* and *defcook* provide the same information as passed to `la_symbind32()`. *retval* is the return code from the bound function.

An audit library that simply monitors symbol binding should return *retval*. An audit library may intentionally return a different value.

Note - This interface function is *experimental*, see “Audit Interface Limitations” on page 129.

```
uint_t la_objclose(uintptr_t * cookie);
```

`la_objclose()`

This function is called after any termination code for an object has been executed (see "Initialization and Termination Routines" on page 52) and prior to the object being unloaded.

cookie was obtained from a previous `la_objopen()` and identifies the object. Any return value is presently ignored.

Audit Interface Example

The following simple example creates an audit library that prints the name of each shared object dependency loaded by the dynamic executable **`date`**(1):

```
$ cat audit.c
#include        <link.h>
#include        <stdio.h>

uint_t
la_version(uint_t version)
{
        return (LAV_CURRENT);
}

uint_t
la_objopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie)
{
        if (lmid == LM_ID_BASE)
                (void) printf("file: %s loaded\n", lmp->l_name);
        return (0);
}
$ cc -o audit.so.1 -G -K pic -z defs audit.c -lmapmalloc -lc
$ LD_AUDIT=./audit.so.1 date
file: date loaded
file: /usr/lib/libc.so.1 loaded
file: /usr/lib/libdl.so.1 loaded
file: /usr/locale/en_US/en_US.so.1 loaded
Fri Mar  8 10:03:55 PST 1997
```

Audit Interface Demonstrations

A number of demonstration applications that use the *rtld-audit* interface are provided in the `SUNWosdem` package under `/usr/demo/link_audit`:

`sotruss`

This demo provides tracing of procedure calls between the dynamic objects of a named application.

`whocalls`

This demo provides a stack trace for a specified function whenever called by a named application.

`perfcnt`

This demo traces the amount of time spent in each function for a named application.

`symbindrep`

This demo reports all symbol bindings performed to load a named application.

`sotruss`(1) and **`whocalls`**(1) are also included in the `SUNWtoo` package. `perfcnt` and `symbindrep` are example programs only and are not intended for use in a production environment.

Audit Interface Limitations

There are some limitations regarding the use of the `la_pltexit()` function. These limitations stem from the need to insert an extra stack frame between the caller and callee to provide a means of acquiring the `la_pltexit()` return value. This is not a problem when calling just the `la_*_pltenter()` routines, as any intervening stack can be cleaned up prior to transferring control to the destination function.

Because of these limitations `la_pltexit()` should be considered an ***experimental*** interface, that may change in future releases to better address these limitations. When in doubt, the use of `la_pltexit()` should be avoided.

Functions that Directly Inspect the Stack

A small number of functions exist that directly inspect the stack or make assumptions regarding its state. Some examples of these functions are the **`setjmp`**(3C) family, **`vfork`**(2) and any function that returns a structure (not a pointer to a structure). These functions will be compromised by the extra stack created to support `la_pltexit()`.

The runtime linker cannot detect functions of this type, and thus it becomes the responsibility of the audit library creator to disable `la_pltexit()` for such routines.

Runtime Linker Debugger Interface

As described in Chapter 3 the runtime linker performs many operations including the mapping of objects into memory and the binding of symbols. Debugging programs often need to access information that describes these runtime linker operations as part of analyzing an application. These debugging programs run as a *separate* process to the application they are analyzing.

This section describes the supported interface for monitoring and modifying a dynamically linked application from another process. This interface is referred to as the *rtld-debugger* interface. The architecture of this interface follows the model used in **libthread_db**(3T).

When using the *rtld-debugger* interface at least two processes are involved:

- One or more *target* processes. The target processes must be dynamically linked and use `/usr/lib/ld.so.1` as their runtime linker.
- A *controlling* process links with the *rtld-debugger* interface library and uses it to inspect the dynamic aspects of the target process(es).

The most anticipated use for *rtld-debugger* is that the controlling process is a debugger and its target is a dynamic executable.

The *rtld-debugger* interface enables the following for a target process:

- Initial rendezvous with the runtime linker.
- Notification of the loading and unloading of dynamic objects.
- Retrieval of information regarding any loaded objects.
- Stepping over procedure linkage table entries.
- Enabling object padding.

Interaction Between Controlling and Target Process

To be able to inspect and manipulate a target process the *rtld-debugger* interface employs an *exported* interface, an *imported* interface and *agents* for communicating between these interfaces.

The controlling process is linked with the *rtld-debugger* interface provided by `librtld_db.so.1`, and makes requests of the interface exported from this library.

This interface is defined in /usr/include/rtld_db.h. In turn librtld_db.so.1 makes requests of the interface imported from the controlling process. This interaction allows the *rtld-debugger* interface to:

- Look up symbols in a target process.
- Read and write memory in the target process.

The imported interface consists of a number of proc_service routines (which are described in “Debugger Import Interface” on page 141), which most debuggers already employ to analyze processes.

The *rtld-debugger* interface assumes that the process being analyzed is stopped when requests are made of the *rtld-debugger* interface. If this is not the case, data structures within the runtime linker of the target process may not be in a consistent state for examination.

The flow of information between librtld_db.so.1, the controlling process (debugger) and the target process (dynamic executable) is diagrammed below:

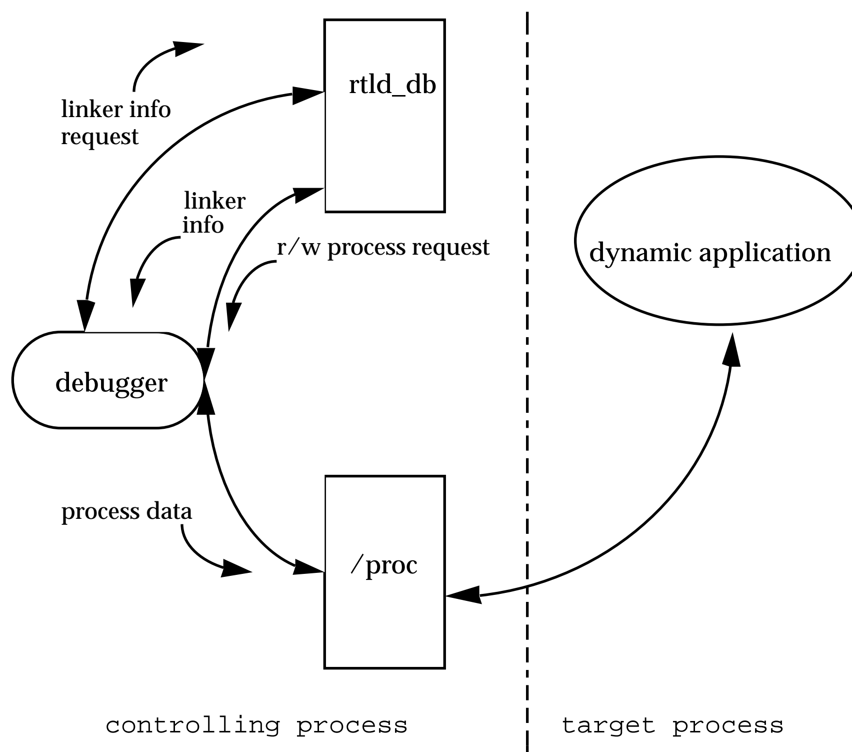


Figure 6–1 *rtld-debugger* information flow

Note - The *rtld-debugger* interface is dependent upon the `proc_service` interface (`/usr/include/proc_service.h`) which is considered experimental. The *rtld-debugger* interface may have to track changes in the `proc_service` interface as it evolves.

A sample implementation of a controlling process that uses the *rtld-debugger* interface is provided in the `SUNWosdem` package under `/usr/demo/librtld_db`. This debugger, `rdb`, provides an example of using the `proc_service` imported interface, and shows the required calling sequence for all `librtld_db.so.1` exported interfaces. The following sections describe the *rtld-debugger* interfaces, and more detailed information can be obtained by examining the sample debugger.

Debugger Interface Agents

An agent provides an opaque handle that can describe internal interface structures, and provides a mechanism of communication between the exported and imported interfaces. As the *rtld-debugger* interface is intended to be used by a debugger which can manipulate several processes at the same time, these agents are used to identify the process.

```
struct ps_prochandle;
```

`struct ps_prochandle`

An opaque structure created by the controlling process to identify the target process which is passed between the exported and imported interface.

```
struct rd_agent;
```

`struct rd_agent`

An opaque structure created by the *rtld-debugger* interface identifying the target process which is passed between the exported and imported interface.

Debugger Exported Interface

This section describes the various interfaces exported by the `/usr/lib/librtld_db.so.1` audit library, and is broken down into functional groups.

Agent Manipulation

```
rd_err_e rd_init(int version);
```

`rd_init()`

This function establishes the *rtld-debugger* version requirements. The current *version* is defined by `RD_VERSION`.

If the version requirement of the controlling process is greater than the *rtld-debugger* interface available, then `RD_NOCAPAB` is returned.

```
rd_agent * rd_new(struct ps_prochandle * php);
```

`rd_new()`

This function creates a new exported interface agent. *php* is a cookie created by the controlling process to identify the target process. This cookie is used by the imported interface offered by the controlling process to maintain context, and is opaque to the *rtld-debugger* interface.

```
rd_err_e rd_reset(struct rd_agent * rdap);
```

`rd_reset()`

This function resets the information within the agent based off the same `ps_prochandle` structure given to `rd_new()`. This function is called when a target process is restarted.

```
void rd_delete(struct rd_agent * rdap);
```

`rd_delete()`

This function deletes an agent and frees any state associated with it.

Error Handling

The following error states can be returned by the *rtld-debugger* interface (defined in `rtld_db.h`):

```
typedef enum {
        RD_ERR,
        RD_OK,
        RD_NOCAPAB,
        RD_DBERR,
        RD_NOBASE,
        RD_NODYNAM,
        RD_NOMAPS
```

(continued)

```
} rd_err_e;
```

The following interfaces can be used to gather the error information:

```
char * rd_errstr(rd_err_e rderr);
```

`rd_errstr()`

This function returns a descriptive error string describing the error code *rderr*.

```
void rd_log(const int onoff);
```

`rd_log()`

This function turns logging on (1) or off (0). When logging is turned on, the imported interface function `ps_plog()` provided by the controlling process, is called with more detailed diagnostic information.

Scanning Loadable Objects

Obtaining information for each object maintained on the runtime linkers link-map (see "Establishing a Name-space" on page 123) is achieved using the following structure (defined in `rtld_db.h`):

```
typedef struct rd_loadobj {
        psaddr_t        rl_nameaddr;
        unsigned        rl_flags;
        psaddr_t        rl_base;
        psaddr_t        rl_data_base;
        unsigned        rl_lmident;
        psaddr_t        rl_refnameaddr;
        psaddr_t        rl_plt_base;
        unsigned        rl_plt_size;
        psaddr_t        rl_bend;
        psaddr_t        rl_padstart;
        psaddr_t        rl_padend;
} rd_loadobj_t;
```

Note that all addresses given in this structure, including string pointers, are addresses in the target process and not in the address space of the controlling process itself:

`rl_nameaddr`

Pointer to a string which contains the name of the dynamic object.

`rl_flags`

Reserved for future use.

`rl_base`

Base address of dynamic object.

`rl_data_base`

Base address of data segment of dynamic object.

`rl_lmident`

The link-map identifier (see “Establishing a Name-space” on page 123).

`rl_refnameaddr`

If the dynamic object is a *filter* (see “Shared Objects as Filters” on page 78) then this points to the name of the *filtee*(s).

`rl_plt_base`, `rl_plt_size`

These elements are present for backward compatibility and are currently unused.

`rl_bend`

End address of object (`text + data + bss`).

`rl_padstart`

Base address of padding before dynamic object (refer to “Dynamic Object Padding” on page 141).

`rl_padend`

Base address of padding after dynamic object (refer to “Dynamic Object Padding” on page 141).

The following routine uses this object data structure to access information from the runtime linkers link-map lists:

```
typedef int rl_iter_f(const rd_loadobj_t *, void *);

rd_err_e rd_loadobj_iter(rd_agent_t * rap, rl_iter_f * cb,
```

```
        void * clnt_data);
```

`rd_loadobj_iter()`

This function iterates over all dynamic objects currently loaded in the target process. On each iteration the *imported* function specified by *cb* is called. *clnt_data* can be used to pass data to the *cb* call. Information about each object is returned via a pointer to a volatile (stack allocated) `rd_loadobj_t` structure.

Return codes from the *cb* routine are examined by `rd_loadobj_iter()` and have the following meaning:

- 1 - continue processing link-maps.
- 0 - stop processing link-maps and return control to the controlling process.

`rd_loadobj_iter()` returns `RD_OK` on success. A return of `RD_NOMAPS` indicates the runtime linker has not yet loaded the initial link-maps.

Event Notification

There are certain events that occur within the scope of the runtime linker that a controlling process can track. These events are:

`RD_PREINIT`

The runtime linker has loaded and relocated all the dynamic objects and is about to start calling the `.init` sections of each object loaded (see "Initialization and Termination Routines" on page 52).

`RD_POSTINIT`

The runtime linker has finished calling all of the `.init` sections and is about to transfer control to the primary executable.

`RD_DLACTIVITY`

The runtime linker has been invoked to either load or unload a dynamic object (see "Loading Additional Objects" on page 51).

These events can be monitored using the following interface (defined in `sys/link.h` and `rtld_db.h`):

```
typedef enum {
        RD_NONE = 0,
        RD_PREINIT,
        RD_POSTINIT,
        RD_DLACTIVITY
} rd_event_e;

/*
 * ways that the event notification can take place:
 */
typedef enum {
        RD_NOTIFY_BPT,
        RD_NOTIFY_AUTOBPT,
        RD_NOTIFY_SYSCALL
} rd_notify_e;

/*
 * information on ways that the event notification can take place:
 */
typedef struct rd_notify {
        rd_notify_e     type;
        union {
                psaddr_t        bptaddr;
                long            syscallno;
        } u;
} rd_notify_t;
```

```
rderr_e rd_event_enable(struct rd_agent * rdap, int onoff);
```

`rd_event_enable()`

This function enables (1) or disables (0) event monitoring.

Note - Presently, for performance reasons, the runtime linker ignores event disabling. The controlling process should not assume that a given break-point will not be reached because of the last call to this routine.

```
rderr_e rd_event_addr(rd_agent_t * rdap, rd_event_e event,
        rd_notify_t * notify);
```

`rd_event_addr()`

This function specifies how the controlling program will be notified of a given event.

Depending on the event type, the notification of the controlling process will take place by calling a benign, cheap system call which is identified by *notify->u.syscallno*, or executing a break point at the address specified by *notify->u.bptaddr*. It is the responsibility of the controlling process to trace the system call or place the actual break-point.

When an event has occurred additional information can be obtained via this interface (defined in `rtld_db.h`):

```
typedef enum {
        RD_NOSTATE = 0,
        RD_CONSISTENT,
        RD_ADD,
        RD_DELETE
} rd_state_e;

typedef struct rd_event_msg {
        rd_event_e      type;
        union {
                rd_state_e      state;
        } u;
} rd_event_msg_t;
```

`rd_state_e` values have the following meaning:

`RD_NOSTATE`

There is no additional state information available.

`RD_CONSISTANT`

The link-maps are in a stable state and may be examined.

`RD_ADD`

A dynamic object is in the process of being loaded and the link-maps are not in a stable state. They should not be examined until the `RD_CONSISTANT` state is reached.

`RD_DELETE`

A dynamic object is in the process of being deleted and the link-maps are not in a stable state. They should not be examined until the `RD_CONSISTANT` state is reached.

```
rderr_e rd_event_getmsg(struct rd_agent * rdap,
        rd_event_msg_t * msg);
```

rd_event_getmsg( )

This function provides additional information concerning an event.

The following table shows the possible state for each of the different event types:

RD_PREINIT	RD_POSTINIT	RD_DLACTIVITY
RD_NOSTATE	RD_NOSTATE	RD_CONSISTANT
		RD_ADD
		RD_DELETE

Procedure Linkage Table Skipping

The *rtld-debugger* interface offers the ability to help skip over procedure linkage table entries (refer to "Procedure Linkage Table (Processor-Specific)" on page 219). When a controlling process, such as a debugger, is asked to step into a function for the first time, they often wish to skip the actual procedure linkage table processing as this results in control being passed to the runtime linker to search for the function definition.

The following interface allows a controlling process to step over the runtime linkers procedure linkage table processing. It is assumed that the controlling process can determine when a procedure linkage table entry is encountered based on external information provided in the ELF file.

Once a target process has stepped into a procedure linkage table entry it calls the following interface:

```
rd_err_e rd_plt_resolution(rd_agent_t * rdap, paddr_t pc,
        lwpid_t lwpid, paddr_t plt_base, rd_plt_info_t * rpi);
```

rd_plt_resolution( )

This function returns resolution state of the current procedure linkage table entry and information on how to skip it.

pc represents the first instruction of the procedure linkage table entry. *lwpid* privides the `lwp` identifier and *plt_base* provides the base address of the procedure linkage table. These three variables provide information sufficient for various architectures to process the procedure linkage table.

rpi provides detailed information regarding the procedure linkage table entry as defined in the following data structure (defined in `rtld_db.h`):

```
typedef enum {
    RD_RESOLVE_NONE,
    RD_RESOLVE_STEP,
    RD_RESOLVE_TARGET,
    RD_RESOLVE_TARGET_STEP
} rd_skip_e;

typedef struct rd_plt_info {
        rd_skip_e       pi_skip_method;
        long            pi_nstep;
        psaddr_t        pi_target;
} rd_plt_info_t;
```

The following scenarios are possible from the `rd_plt_info_t` return values:

- This is the first call through this procedure linkage table so it must be resolved by the runtime linker. `rd_plt_info_t` will contain:

```
{RD_RESOLVE_TARGET_STEP, M, <BREAK>}
```

  The controlling process sets a break-point at `BREAK` and continues the target process. When the break-point is reached the procedure linkage table entry processing has finish, and the controlling process can step `M` instructions to the destination function.

- This is the `N`th time through this procedure linkage table. `rd_plt_info_t` will contain:

```
{RD_RESOLVE_STEP, M, 0}
```

  The procedure linkage table entry has already been resolved and the controlling process can step `M` instructions to the destination function.

Note - Future implementations may employ `RD_RESOLVE_TARGET` as a means of setting a break point directly in the target function, however, this capability is not yet available in this version of the *rtld-debugger* interface.

Dynamic Object Padding

The default behavior of the runtime linker relies on the operating system to load dynamic objects where they can be most efficiently referenced. Some controlling processes benefit from the existence of padding around the objects loaded into memory of the target process. This interface allows a controlling process to request this padding.

```
rd_err_e rd_objpad_enable(struct rd_agent * rdap, size_t padsize);
```

`rd_objpad_enable()`

This function enables or disables the padding of any subsequently loaded objects with the target process. Padding occurs on both sides of the loaded object.

padsize specifies the size of the padding, in bytes, to be preserved both before and after any objects loaded into memory. This padding is reserved as a memory mapping using **mmap**(2) with `PROT_NONE` permissions and the `MAP_NORESERVE` flag. Effectively the runtime linker reserves areas of the virtual address space of the target process adjacent to any mapped objects. These areas can later be utilized by the controlling process.

A *padsize* of 0 disables any object padding for later objects.

Note - Reservations obtained using **mmap**(2) from `/dev/zero` with `MAP_NORESERVE` can be reported using the **proc**(1) facilities and by referring to the link-map information provided in `rd_loadobj_t`.

Debugger Import Interface

The imported interface a controlling process must provide to `librtld_db.so.1` is defined in `/usr/include/proc_service.h`. A sample implementation of these `proc_service` functions can be found in the `rdb` demonstration debugger. The *rtld-debugger* interface uses only a subset of the `proc_service` interfaces available. Future versions of the *rtld-debugger* interface may take advantage of additional `proc_service` interfaces without creating an incompatible change.

The following interfaces are currently being used by the *rtld-debugger* interface:

```
ps_err_e ps_pauxv(const struct ps_prochandle * ph, auxv_t ** aux);
```

`ps_pauxv()`

This function returns a pointer to a copy of the `auxv` vector. As the `auxv` vector information is copied to an allocated structure, the life time of this pointer is as long as the `prochandle` is valid.

```
ps_err_e ps_pread(const struct ps_prochandle * ph, paddr_t addr,
        char * buf, int size);
```

ps_pread()

This function reads *size* bytes from the target process at address *addr* and copies them into *buf*.

```
ps_err_e ps_pwrite(const struct ps_prochandle * ph, paddr_t addr,
          char * buf, int size);
```

ps_pwrite()

This function writes *size* bytes from *buf* into the target process at address *addr*.

```
void ps_plog(const char * fmt, ...);
```

ps_plog()

This function is called with additional diagnostic information from the *rtld-debugger* interface. It is up to the controlling process to decide where (or if) to log this diagnostic information. The arguments to ps_plog() follow the **printf**(3S) format.

```
ps_err_e ps_pglobal_sym(const struct ps_prochandle * ph,
        const char * obj, const char * name, Elf32_Sym * sym);
```

ps_pglobal_sym()

This function searches for the symbol named *name* within the object named *obj* within the target process *ph*. If the symbol is found the descriptor *sym* is filled in.

In the event that the *rtld-debugger* interface needs to find symbols within the application or runtime linker prior to any link-map creation, the following reserved values for *obj* are available:

```
#define PS_OBJ_EXEC ((const char *)0x0)  /* application id */
#define PS_OBJ_LDSO ((const char *)0x1)  /* runtime linker id */
```

One mechanism the controlling process can use to find the symbol table for these objects is through the `procfs` file system using the following pseudo code:

```
ioctl(.., PIOCNAUXV, ...)       - obtain AUX vectors
ldsoaddr = auxv[AT_BASE];
ldsofd = ioctl(..., PIOCOPENM, &ldsoaddr);

/* process elf information found in ldsofd ... */

execfd = ioctl(.., PIOCOPENM, 0);

/* process elf information found in execfd ... */
```

Once the file descriptors are found, the ELF files can be examined for their symbol information by the controlling program.

CHAPTER **7**

Object Files

Overview

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

The first section in this chapter, "File Format" on page 146, focuses on the format of object files and how that pertains to building programs. The second section, "Dynamic Linking" on page 195, focuses on how the format pertains to loading programs.

Programs manipulate object files with the functions contained in the ELF access library, `libelf`. Refer to **`elf`**(3E) for a description of `libelf` contents. Sample source code that uses `libelf` is provided in the `SUNWosdem` package under the `/usr/demo/ELF` directory.

File Format

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 7–1 shows an object file's organization.

Linking view	Execution view
ELF header	ELF header
Program header table optional	Program header table
Section 1	Segment 1
. . .	
Section n	Segment 2
. . .	
. . .	. . .
Section header table	Section header table optional

Figure 7–1 Object File Format

An ELF header resides at the beginning of an object file and holds a *road map* describing the file's organization.

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 program header table, if present, tells the system how to create a process image. Files used to build a process image (executables and shared objects) must have a program header table; relocatable objects do not need one.

A section header table contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so forth. Files used in link-editing must have a section header table; other object files may or may not have 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 7–1 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.

The ELF header has the following structure (defined in `sys/elf.h`):

```
#define EI_NIDENT       16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

`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 152.

`e_type`

This member identifies the object file type.

TABLE 7–2 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 7–3 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

TABLE 7–3 ELF Machines *(continued)*

Name	Value	Meaning
EM_88K	5	Motorola 88000
EM_486	6	Intel 80486
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000 Big-Endian
EM_MIPS_RS3_LE	10	MIPS RS3000 Little-Endian
EM_RS6000	11	RS6000
EM_PA_RISC	15	PA-RISC
EM_nCUBE	16	nCUBE
EM_VPP500	17	Fujitsu VPP500
EM_SPARC32PLUS	18	Sun SPARC 32+
EM_PPC	20	PowerPC

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 7–4 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*. This member is presently zero for 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 "Sections" on page 155 and "String Table" on page 170 for more information.

ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encoding, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the e_ident member.

TABLE 7–5 e_ident[ ] Identification Index

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification

TABLE 7–5 e_ident[ ] Identification Index *(continued)*

Name	Value	Purpose
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size ofe_ident[]

These indexes access bytes that hold the following values:

EI_MAG0 - EI_MAG3

A file's first 4 bytes hold a *magic number*, identifying the file as an ELF object file.

TABLE 7–6 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 7–7 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 7–8 Data Encoding

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	See Figure 7–2.
ELFDATA2MSB	2	See Figure 7–3.

More information on these encodings appears 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 7–4 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.

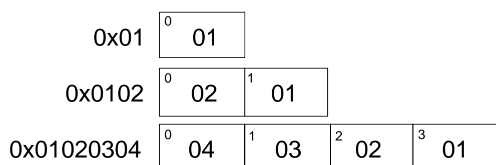


Figure 7–2 Data Encoding ELFDATA2LSB

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

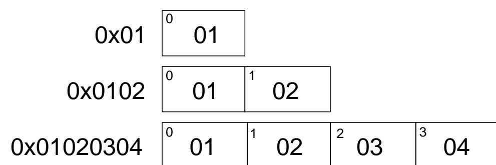


Figure 7–3 Data Encoding ELFDATA2MSB

Sections

An object file's section header table 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 7–9 Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
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 - SHN_HIPROC

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

SHN_BEFORE, SHN_AFTER

These values provide for initial and final section ordering in conjunction with the SHF_ORDERED section flag (see Table 7–12).

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 (defined in sys/elf.h):

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

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 170) giving the location of a null-terminated string. Section names and their descriptions are in Table 7–14.

sh_type

This member categorizes the section's contents and semantics. Section types and their descriptions are in Table 7–10.

sh_flags

Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions are in Table 7–12.

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 7–13 describes the values.

`sh_info`

This member holds extra information, whose interpretation depends on the section type. Table 7–13 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 7–10 Section Types, *sh_type*

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3

TABLE 7–10 Section Types, *sh_type* *(continued)*

Name	Value
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_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
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, SHT_DYNSYM

These sections hold a symbol table. Typically a SHT_SYMTAB section 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 171 for details.

SHT_STRTAB, SHT_DYNSTR

These sections hold a string table. An object file may have multiple string table sections. See "String Table" on page 170 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 177 for details.

SHT_HASH

The section holds a symbol hash table. All dynamically linked 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 223 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 209 for details.

SHT_NOTE

The section holds information that marks the file in some way. See "Note Section" on page 193 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 177 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_SUNW_verdef

The section contains definitions of fine-grained versions defined by this file.

SHT_SUNW_verneed

The section contains descriptions of fine-grained dependencies required for the execution of an image.

SHT_SUNW_versym

The section contains a table describing the relationship of symbols to the version definitions offered by the file.

SHT_LOPROC - 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 7–11 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:

TABLE 7–12 Section Attribute Flags

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4

TABLE 7–12 Section Attribute Flags *(continued)*

Name	Value
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000
SHF_MASKPROC	0xf0000000

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

SHF_WRITE

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_ORDERED

The section requires ordering in relation to other sections of the same type. Ordered sections are combined within the section pointed to by the `sh_link` entry. The `sh_link` entry of an ordered section may point to itself.

If the `sh_info` entry of the ordered section is a valid section within the same input file, the ordered section will be sorted based on the relative ordering within the output file of the section pointed to by the `sh_info` entry. The special `sh_info` values `SHN_BEFORE` and `SHN_AFTER` (see Table 7–9) imply that the sorted section is to precede or follow, respectively, all other sections in the set being ordered. Input file link-line order is preserved if multiple sections in an ordered set have one of these special values.

In the absence of the `sh_info` ordering information, sections from a single input file combined within one section of the output file shall be contiguous and have the same relative ordering as they did in the input file. The contributions from multiple input files shall appear in link-line order.

SHF_EXCLUDE

The section is excluded from input to the link-edit of an executable or shared object. This flag is ignored if the SHF_ALLOC flag is also set, or if relocations exist against the section.

SHF_MASKPROC

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

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

TABLE 7–13 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 symbol table.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies. See also Table 7–14.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
SHT_SUNW_verdef	The section header index of the associated string table.	The number of version definitions within the section.
SHT_SUNW_verneed	The section header index of the associated string table.	The number of version dependencies within the section.
SHT_SUNW_versym	The section header index of the associated symbol table.	0
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 7–14 Special Sections

Name	Type	Attribute
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	None
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	See "Global Offset Table (Processor-Specific)" on page 218
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	See "Program Interpreter" on page 207
.note	SHT_NOTE	None
.plt	SHT_PROGBITS	See "Procedure Linkage Table (Processor-Specific)" on page 219
.rel*name*	SHT_REL	See "Relocation" on page 177
.rela*name*	SHT_RELA	See "Relocation" on page 177

TABLE 7–14 Special Sections *(continued)*

Name	Type	Attribute
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	None
.strtab	SHT_STRTAB	See description below
.symtab	SHT_SYMTAB	See "Symbol Table" on page 171
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_reloc	SHT_rel SHT_rela	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef SHT_SUNW_verneed SHT_SUNW_versym	SHF_ALLOC

.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 comment information.

.data, .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 171 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 218 for more information.

`.hash`

This section holds a symbol hash table. See "Hash Table" on page 223 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 207 for more information.

`.note`

This section holds information in the format that "Note Section" on page 193 describes.

`.plt`

This section holds the procedure linkage table. See “Procedure Linkage Table (Processor-Specific)” on page 219 for more information.

`.rel`***name***, `.rela`***name***

These sections hold relocation information, as “Relocation” on page 177 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 will have the name `.rel.text` or `.rela.text`.

`.rodata`, `.rodata1`

These sections hold read-only data that typically contribute to a non-writable segment in the process image. See “Program Header” on page 195 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 171 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.

`.SUNW_heap`

This section holds the *heap* of a dynamic executable created from **`dldump`**(3X).

`.SUNW_reloc`

This section holds relocation information, as “Relocation” on page 177 describes. This section is a concatenation of relocation sections that provides better locality of reference of the individual relocation records. Only the offset of the relocation record itself is meaningful and thus the section `sh_info` value is zero.

`.SUNW_version`

Sections of this name hold versioning information. See “Versioning Information” on page 187 for more information.

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` architecture.

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 will 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 7–4 String Table

The table below shows the strings of the string table above:

TABLE 7–15 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 7–16 Symbol Table Initial Entry

Name	Value
STN_UNDEF	0

A symbol table entry has the following format (defined in sys/elf.h):

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

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.

Note - 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 177.

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 (defined in `sys/elf.h`):

```
#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 7–10.

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

TABLE 7–17 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` - `STB_HIPROC`

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

Global and weak symbols differ in two major ways:

- When the link-editor combines several relocatable object files, it does not allow multiple definitions of `STB_GLOBAL` symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name 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 11) 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, by default, does not extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value. The use of `-z weakextract` overrides this default behavior, and allows weak references to cause the extraction of archive members.

In each symbol table, all symbols with `STB_LOCAL` binding precede the weak and global symbols. As "Sections" on page 155 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 7–18 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. Symbol index 1 of the SHT_SYMTAB is an STT_FILE symbol representing the file itself.

Conventionally, this is followed by the files STT_SECTION symbols, and any global symbols that have been reduced to locals (see “Reducing Symbol Scope” on page 33, and Chapter 5 for more details).

STT_LOPROC - 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 bound to the actual definition.

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

TABLE 7–19 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.

Relocation entries can have the following structure (defined in sys/elf.h):

```
typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;
```

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 will 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

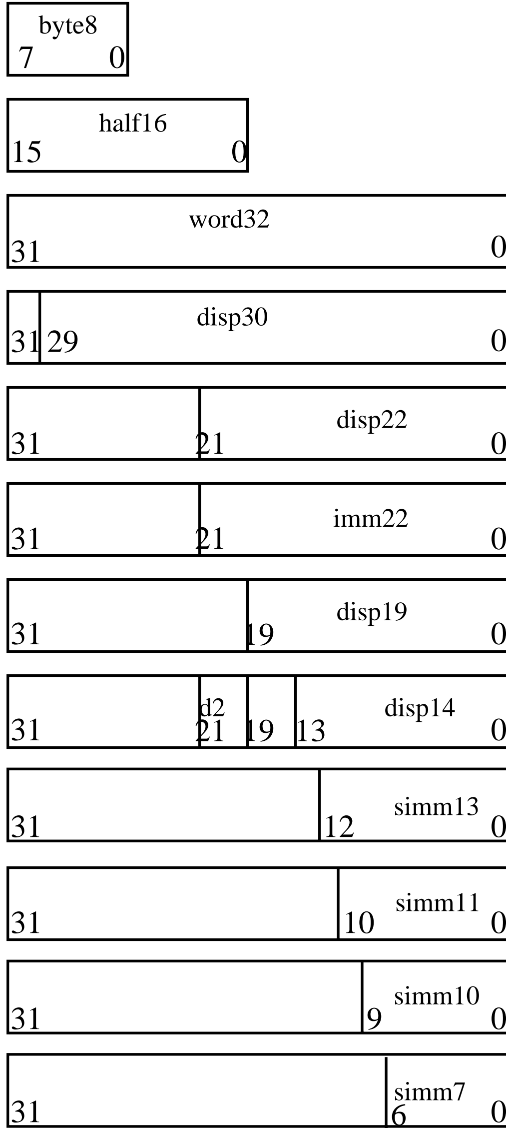
"Sections" on page 155 earlier, specify these relationships. Relocation entries for different object files have slightly different interpretations for the `r_offset` member.

- 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.

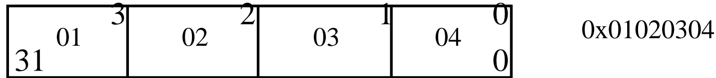
Relocation Types (Processor-Specific)

On SPARC, relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners):



On x86, relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners):

`word32` specifies a 32-bit field occupying 4 bytes with an arbitrary byte alignment. These values use the same byte order as other word values in the x86 architecture):



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 195 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 218 for more information.

`GOT` — means the address of the *global offset table*. See "Global Offset Table (Processor-Specific)" on page 218 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 (Processor-Specific)" on page 219 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.

SPARC relocation entries apply to bytes (byte8), half-words (half16), or words (the others). x86 relocation entries apply to words. 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 uses only `Elf32_Rela` relocation entries with explicit addends. Thus the `r_addend` member serves as the relocation addend. x86 uses only `Elf32_Rel` relocation entries, the field to be relocated holds the addend. In all cases the addend and the computed result use the same byte order.

SPARC: Relocation Types

Note - 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 7–20 SPARC: Relocation Types

Name	Value	Field	Calculation
`R_SPARC_NONE`	0	None	None
`R_SPARC_8`	1	`V-byte8`	`S + A`
`R_SPARC_16`	2	`V-half16`	`S + A`
`R_SPARC_32`	3	`V-word32`	`S + A`
`R_SPARC_DISP8`	4	`V-byte8`	`S + A - P`
`R_SPARC_DISP16`	5	`V-half16`	`S + A - P`
`R_SPARC_DISP32`	6	`V-disp32`	`S + A - P`
`R_SPARC_WDISP30`	7	`V-disp30`	`(S + A - P) >> 2`
`R_SPARC_WDISP22`	8	`V-disp22`	`(S + A - P) >> 2`
`R_SPARC_HI22`	9	`T-imm22`	`(S + A) >> 10`
`R_SPARC_22`	10	`V-imm22`	`S + A`
`R_SPARC_13`	11	`V-simm13`	`S + A`

TABLE 7–20 SPARC: Relocation Types *(continued)*

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 R_SPARC_JMP_SLOT,
R_SPARC_RELATIVE	22	V-word32	B + A
R_SPARC_UA32	23	V-word32	S + A
R_SPARC_PLT32	24	V-word32	L + A
R_SPARC_HIPLT22	25	T-imm22	(L + A) >> 10
R_SPARC_LOPLT10	26	T-simm13	(L + A) & 0x3ff
R_SPARC_PCPLT32	27	V-word32	L + A - P
R_SPARC_PCPLT22	28	V-disp22	(L + A - P) >> 10
R_SPARC_PCPLT10	29	V-simm12	(L + A - P) & 0x3ff
R_SPARC_10	30	V_simm10	S + A

TABLE 7–20 SPARC: Relocation Types *(continued)*

Name	Value	Field	Calculation
R_SPARC_11	31	V_simm11	S + A
R_SPARC_WDISP16	40	V-d2/disp14	(S + A - P) >> 2
R_SPARC_WDISP19	41	V-disp19	(S + A - P) >> 2
R_SPARC_7	43	V-imm7	S + A
R_SPARC_5	44	V-imm5	S + A
R_SPARC_6	45	V-imm6	S + A

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 for more details.

`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

TABLE 7–21 **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
R_386_PLT32	4	word32	L + A - P
R_386_COPY	5	none	none
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P
R_386_32PLT	11	word32	L + A

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.

Versioning Information

Objects created by the link-editor may contain two types of versioning information:

- *version definitions* provide associations of global symbols and are implemented using sections of type `SHT_SUNW_verdef` and `SHT_SUNW_versym`.
- *version dependencies* indicate the version definition requirements from other object dependencies and are implemented using sections of type `SHT_SUNW_verneed`.

The structures that form these sections are defined in `sys/link.h`. Sections that contain versioning information are named `.SUNW_version`.

Version Definition Section

This section is defined by the type `SHT_SUNW_verdef`. If this section exists a `SHT_SUNW_versym` section must also exist. Using these two structures an association of symbols to version definitions is maintained within the file (see "Creating a Version Definition" on page 99 for more details). Elements of this section have the following structure:

```
typedef struct {
        Elf32_Half      vd_version;
        Elf32_Half      vd_flags;
        Elf32_Half      vd_ndx;
        Elf32_Half      vd_cnt;
        Elf32_Word      vd_hash;
        Elf32_Word      vd_aux;
        Elf32_Word      vd_next;
} Elf32_Verdef;

typedef struct {
        Elf32_Word      vda_name;
        Elf32_Word      vda_next;
} Elf32_Verdaux;
```

`vd_version`

This member identifies the version of the structure itself.

TABLE 7–22 Version Definition Structure Versions

Name	Value	Meaning
`VER_DEF_NONE`	0	Invalid version
`VER_DEF_CURRENT`	>=1	Current version

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

vd_flags

This member holds version definition specific information.

TABLE 7–23 Version Definition Section Flags

Name	Value	Meaning
VER_FLG_BASE	0x1	Version definition of the file itself
VER_FLG_WEAK	0x2	Weak version identifier

The ***base*** version definition is always present when version definitions, or symbol ***auto-reduction*** has been applied to the file. The base version provides a default version for the files ***reserved*** symbols (see "Generating the Output Image" on page 37). A ***weak*** version definition has no symbols associated with it (see "Creating a Weak Version Definition" on page 102 for more details).

vd_ndx

This member holds the version index. Each version definition has a unique index that is used to associate SHT_SUNW_versym entries to the appropriate version definition.

vd_cnt

This member indicates the number of elements in the Elf32_Verdaux array.

vd_hash

This member holds the hash value of the version definition name (this value is generated using the same hashing function described in "Hash Table" on page 223).

vd_aux

This member holds the byte offset, from the start of this Elf32_Verdef entry, to the Elf32_Verdaux array of version definition names. The first element of the array ***must*** exist and points to the version definition string this structure defines. Additional elements may be present, the number being indicated by the vd_cnt value. These

elements represent the dependencies of this version definition. Each of these dependencies will have its own version definition structure.

vd_next

This member holds the byte offset, from the start of this Elf32_Verdef structure, to the next Elf32_Verdef entry.

vda_name

This member holds a string table offset to a null-terminated string, giving the name of the version definition.

vda_next

This member holds the byte offset, from the start of this Elf32_Verdaux entry, to the next Elf32_Verdaux entry.

Version Symbol Section

This section is defined by the type SHT_SUNW_versym, and consists of an array of elements having the following structure:

```
typedef Elf32_Half      Elf32_Versym;
```

The number of elements of the array *must* equal the number of symbol table entries contained in the associated symbol table (determined by the sections sh_link value). Each element of the array contains a single index that may have the following values:

TABLE 7–24 Version Dependency Indexes

Name	Value	Meaning
VER_NDX_LOCAL	0	Symbol has local scope
VER_NDX_GLOBAL	1	Symbol has global scope (assigned to base version definition)
	>1	Symbol has global scope (assigned to user-defined version definition)

Any index values greater than VER_NDX_GLOBAL must correspond to the vd_ndx value of an entry in the SHT_SUNW_verdef section. If no index values greater than VER_NDX_GLOBAL exist then no SHT_SUNW_verdef section need be present.

Version Dependency Section

This section is defined by the type SHT_SUNW_verneed. This section compliments the dynamic dependency requirements of the file by indicating the version definitions required from these dependencies. Only if a dependency contains version definitions will a recording be made in this section. Elements of this section have the following structure:

```
typedef struct {
        Elf32_Half      vn_version;
        Elf32_Half      vn_cnt;
        Elf32_Word      vn_file;
        Elf32_Word      vn_aux;
        Elf32_Word      vn_next;
} Elf32_Verneed;

typedef struct {
        Elf32_Word      vna_hash;
        Elf32_Half      vna_flags;
        Elf32_Half      vna_other;
        Elf32_Word      vna_name;
        Elf32_Word      vna_next;
} Elf32_Vernaux;
```

vn_version

This member identifies the version of the structure itself.

TABLE 7–25 Version Dependency Structure Versions

Name	Value	Meaning
VER_NEED_NONE	0	Invalid version
VER_NEED_CURRENT	>=1	Current version

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

vn_cnt

This member indicates the number of elements in the Elf32_Vernaux array.

`vn_file`

This member holds a string table offset to a null-terminated string, giving the filename having a version dependency. This name will match one of the `.dynamic` dependencies (refer to “Dynamic Section” on page 209) found in the file.

`vn_aux`

This member holds the byte offset, from the start of this `Elf32_Verneed` entry, to the `Elf32_Vernaux` array of version definitions required from the associated file dependency. There must exist at least one version dependency. Additional version dependencies may be present, the number being indicated by the `vn_cnt` value.

`vn_next`

This member holds the byte offset, from the start of this `Elf32_Verneed` entry, to the next `Elf32_Verneed` entry.

`vna_hash`

This member holds the hash value of the version dependency name (this value is generated using the same hashing function described in “Hash Table” on page 223).

`vna_flags`

This member holds version dependency specific information.

TABLE 7–26 Version Dependency Structure Flags

Name	Value	Meaning
`VER_FLG_WEAK`	`0x2`	Weak version identifier

A *weak* version dependency indicates an original binding to a*weak* version definition. See “Creating a Version Definition” on page 99 for more details.

`vna_other`

This member is presently unused.

`vna_name`

This member holds a string table offset to a null-terminated string, giving the name of the version dependency.

`vna_next`

This member holds the byte offset, from the start of this `Elf32_Vernaux` entry, to the next `Elf32_Vernaux` entry.

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.

namesz
descsz
type
name . . .
desc . . .

Figure 7–5 Note Information

`namesz` **and** `name`

The first `namesz` bytes in name contain a null-terminated character representation of the entry's owner or originator. 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.

	+0	+1	+2	+3	
namesz	7				
descsz	0				No descriptor
type	1				
name	X	Y	Z		
	C	o	\0	pad	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	pad	
desc	word0				
	word1				

Figure 7–6 Example Note Segment

Note - The system reserves note information with no name (`namesz==0`) and with a zero-length name (`name[0]=='\0'`) but currently defines no types. All other names must have at least one non-null character.

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” on page 195 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 201 describes the information used to load a program into memory.
- “Runtime Linker” on page 208 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 200.

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 148 for more information.

A program header has the following structure (defined in `sys/elf.h`):

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

`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 7–27.

`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 (Processor-Specific)" on page 201 describes, 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 7–27 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 209 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 207 for further information.

PT_NOTE

The array element specifies the location and size of auxiliary information. See "Note Section" on page 193 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 207 for further information.

PT_LOPROC - PT_HIPROC

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

Note - Unless specifically required elsewhere, all program header segment types are optional. 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 201 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 7–28 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 7–29 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, in sections described earlier in this chapter. Data segments contain writable data and instructions. See Table 7–14 for a list of all special sections. Use **dump**(1) to see which sections are in a particular executable file.

A PT_DYNAMIC program header element points at the .dynamic section, as explained in “Dynamic Section” on page 209 later. The .got and .plt sections also hold information related to position-independent code and dynamic linking.

The .plt may reside in a text or a data segment, depending on the processor. See “Global Offset Table (Processor-Specific)” on page 218 and “Procedure Linkage Table (Processor-Specific)” on page 219 for details.

As previously described on Table 7–10, the .bss section has the type SHT_NOBITS. Although it occupies no space in the file, it contributes to the segment’s memory image. Normally, these uninitialized data reside at the end of the segment, thereby making p_memsz larger than p_filesz in the associated program header element.

Program Loading (Processor-Specific)

As the system creates or augments a process image, it logically copies a file’s segment to a virtual memory segment. When, and if, the system physically reads the file depends on the program’s execution behavior, system load, and so forth.

A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for SPARC segments are congruent modulo 64K (0x10000). Virtual addresses and file offsets for x86 segments are congruent modulo 4K (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.

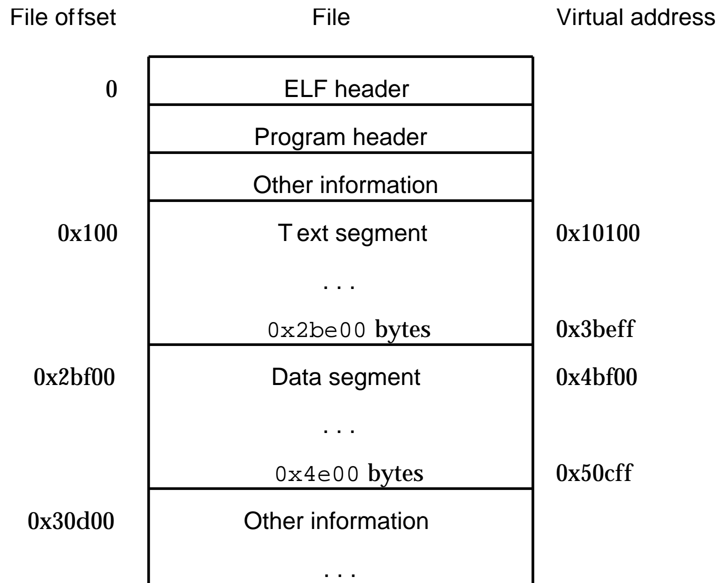


Figure 7–7 SPARC: Executable File (64 K alignment)

TABLE 7–30 **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
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.

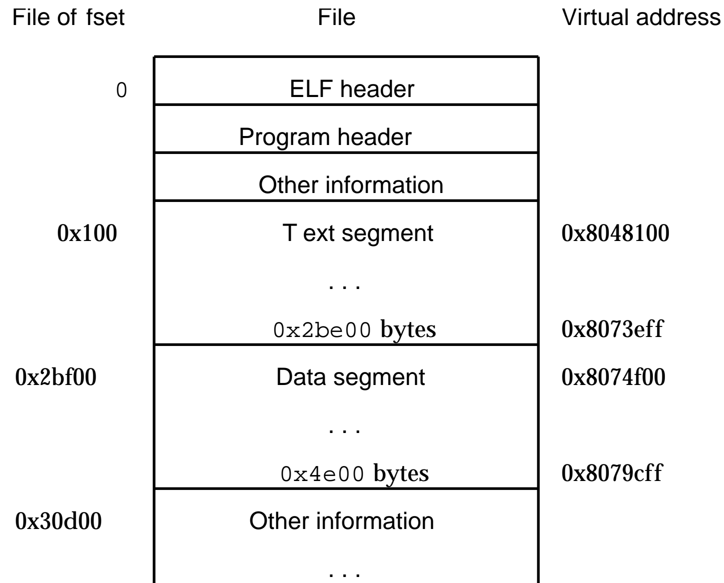


Figure 7–8 x86: Executable File (4 K alignment)

TABLE 7–31 **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
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 is 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 Kilobyte (0x1000) pages. For simplicity, these examples illustrates only one page size.

Virtual Address	Contents	Segment
0x10000	*Header Padding* `0x100` bytes	Text
0x10100	Text segment ... `0x2be00` bytes	Text
0x3bf00	*Data Padding* `0x100` bytes	Text
0x4b000	*Text Padding* `0x100` bytes	Data
0x4bf00	Data segment ... `0x4e00` bytes	Data
0x50d00	Uninitialized Data `0x1024` zero bytes	Data
0x51d24	*Page Padding* 0x2dc zero bytes	Data

Figure 7–9 SPARC: Process Image Segments

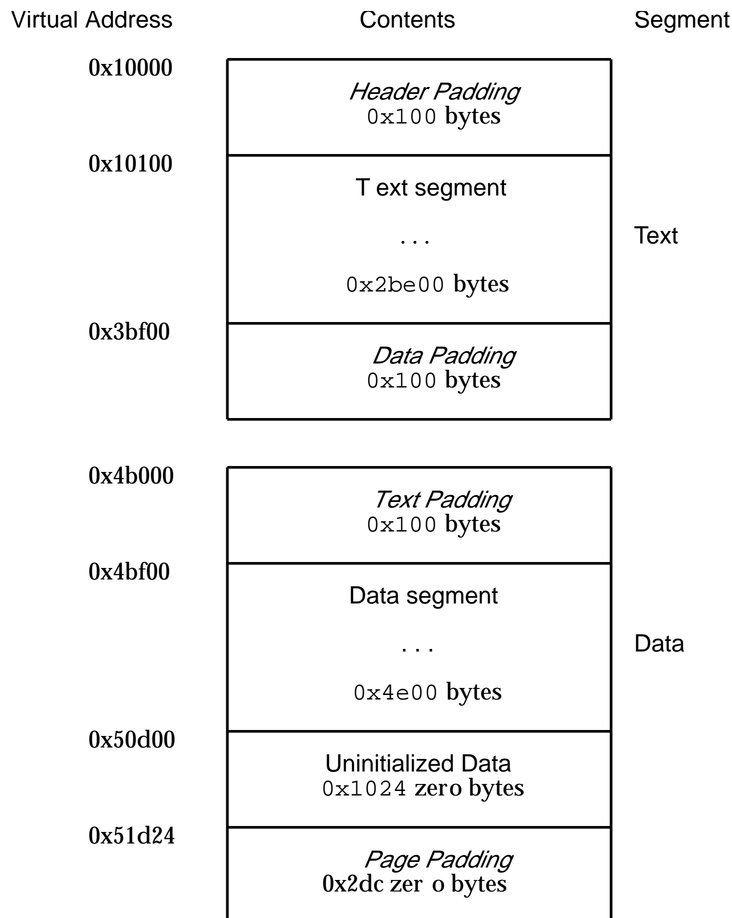


Figure 7–10 x86: Process Image Segments

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 Chapter 2.) 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 7–32 **SPARC:** Example 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 7–33 **x86:** Example 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 has 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**(2) 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 224.
- Transferring control to the program, making it look as if the program had received control directly from **exec**(2).

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 201, 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 will be correct if the library is 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. 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 (Processor-Specific)" on page 219 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 (defined in sys/link.h):

```
typedef struct {
        Elf32_Sword d_tag;
        union {
                Elf32_Word      d_val;
                Elf32_Addr      d_ptr;
                Elf32_Off       d_off;
        } d_un;
} Elf32_Dyn;
```

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 7–34 Dynamic Array Tags, d_tag

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

TABLE 7–34 Dynamic Array Tags, d_tag *(continued)*

Name	Value	d_un	Executable	Shared Object
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
DT_FLAGS_1	0x6ffffffb	d_val	Optional	Optional
DT_VERDEF	0x6ffffffc	d_ptr	Optional	Optional
DT_VERDEFNUM	0x6ffffffd	d_val	Optional	Optional
DT_VERNEED	0x6ffffffe	d_ptr	Optional	Optional
DT_VERNEEDNUM	0x6fffffff	d_val	Optional	Optional

TABLE 7–34 Dynamic Array Tags, d_tag *(continued)*

Name	Value	d_un	Executable	Shared Object
DT_AUXILIARY	0x7ffffffd	d_val	Unspecified	Optional
DT_USED	0x7ffffffe	d_val	Optional	Optional
DT_FILTER	0x7fffffff	d_val	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 218 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 223. 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 224 later.

DT_FINI

This element holds the address of the termination function, discussed in "Initialization and Termination Functions" on page 224 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 "Shared Object Dependencies" on page 218 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_FLAGS_1

If present, this entry's d_val member holds various state flags. See Table 7–35.

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_VERDEF

Holds the address of the version definition table, described in the first part of this chapter, with Elf32_Verdef entries for the 32-bit class of files. See section "Version Definition Section" on page 188for more information. Elements within these entries contain indexes into the table recorded in the DT_STRTAB entry.

DT_VERDEFNUM

This element specifies the number of entries in the version definition table.

DT_VERNEED

Holds the address of the version dependency table, described in the first part of this chapter, with Elf32_Verneed entries for the 32-bit class of files. See section

"Version Dependency Section" on page 191 for more information. Elements within these entries contain indexes into the table recorded in the DT_STRTAB entry.

DT_VERNEEDNUM

This element specifies the number of entries in the version dependency table.

DT_AUXILIARY

Holds the string table offset of a null-terminated string that names an object. The offset is an index into the table recorded in the DT_STRTAB entry. Symbols in the auxiliary object will be used in preference to the symbols within this object.

DT_FILTER

Holds the string table offset of a null-terminated string that names an object. The offset is an index into the table recorded in the DT_STRTAB entry. The symbol table of this object acts as a filter for the symbol table of the named object.

DT_LOPROC - DT_HIPROC

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

The following dynamic state flags are presently available:

TABLE 7–35 Dynamic Tags, DT_FLAGS_1

Name	Value	Meaning
DF_1_NOW	0x1	Perform complete relocation processing.
DT_1_GLOBAL	0x2	Unused.
DT_1_GROUP	0x4	Indicate object is a member of a group.
DT_1_NODELETE	0x8	Object can not be deleted from a process.
DT_1_LOADFLTR	0x10	Insure immediate loading of filtee(s)..
DT_1_INITFIRST	0x20	Run objects' initialization first.
DT_1_NOOPEN	0x40	Object can not be used with **dlopen**(3X).

DF_1_NOW

When the object is loaded *all* relocation processing is completed, see "When Relocations are Performed" on page 48. This state is recorded in the object using the link-editors' −z now option.

DF_1_GROUP

Indicates that the object is a member of a group, see "Symbol Lookup" on page 57. This state is recorded in the object using the link-editors' −B group option.

DF_1_NODELETE

Indicates that the object can not be deleted from a process. Thus if the object is loaded in a process, either directly or as a dependency, with **dlopen**(3X), it can not be unloaded with **dlclose**(3X). This state is recorded in the object using the link-editors' −z nodelete option.

DF_1_LOADFLTR

This state is only meaningful for *filters* (See "Shared Objects as Filters" on page 78). When the filter is loaded all associated *filtees* are immediately processed, see "Filtee Processing" on page 83. This state is recorded in the object using the link-editors' −z loadfltr option.

DF_1_INITFIRST

When the object is loaded its initialization section is run before any other objects loaded with it, see "Initialization and Termination Routines" on page 52. This specialized state is intended for libthread.so.1. This state is recorded in the object using the link-editors' −z initfirst option.

DF_1_NOOPEN

Indicates that the object can not be added to a running process with **dlopen**(3X). This state is recorded in the object using the link-editors' −z nodlopen option.

Except for the DT_NULL element at the end of the dynamic 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 177 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 (Processor-Specific)

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.

SPARC: Procedure Linkage Table

On SPARC architectures, procedure linkage tables reside in private data. The runtime linker determines the destinations' absolute addresses and modifies the procedure linkage 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.

TABLE 7–36 **SPARC:** Procedure Linkage Table Example

Object File	Memory Segment

.PLT0:
    unimp
    unimp
    unimp
.PLT1:
    unimp
    unimp
    unimp
    ...
	.PLT0:
	    save %sp,-64,%sp
	    call runtime-linker
	    nop
	.PLT1:
	    .word identification
	    unimp
	    unimp
	    ...

...
.PLT101:
    sethi (.-.PLT0),%g1
    ba,a .PLT0
    nop
.PLT102:
    sethi (.-.PLT0),%g1
    ba,a .PLT0
    nop
	...
	.PLT101:
	    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

nop	nop

Following the steps below, the runtime linker and program jointly resolve the symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the runtime linker is not specified.

1. When first creating the memory image of the program, the runtime linker changes the initial procedure linkage table entries, making them transfer control to one of the runtime linker's own routines. 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 `jmpl` instruction, and then patching the `ba,a` to be `sethi`. If a reentrant function call happens between the two word updates, the `jmpl` 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 `jmpl` instruction. Although the `sethi` changes the value of the `%g1` register, the previous contents can be safely discarded.
- After conversion, the last procedure linkage table entry (`.PLT102` above) needs a delay instruction for its `jmpl`. 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.

x86: 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.

TABLE 7–37 x86: Procedure Linkage Table Example

```
.PLT0:  pushl   got_plus_4
        jmp     *got_plus_8
        nop;    nop
        nop;    nop
.PLT1:  jmp     *name1_in_GOT
        pushl   $offset
        jmp     .PLT0@PC
.PLT2:  jmp     *name2_in_GOT
        pushl   $offset
        jmp     .PLT0@PC
        ...
```

```
.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. The symbol table to which the hashing is associated is specified in the sh_link entry of the hash table's section header (refer to Table 7–13). Labels appear below to help explain the hash table organization, but they are not part of the specification.

nbucket
nchain
bucket [0] . . . bucket [nbucket - 1]
chain [0] . . . chain [nchain - 1]

Figure 7–11 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.

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.

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 “Sections” on page 155 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.

CHAPTER **8**

Mapfile Option

Overview

The link-editor automatically and intelligently maps input sections from relocatable objects to segments within the output file object. The `–M` option with an associated `mapfile` 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 the link-editor that used 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 link-editor command line using `–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 four basic types of directives into a `mapfile`:

- Segment declarations.
- Mapping directives.
- Size-symbol declarations.
- File control directives.

Each directive can span more than one line and can have any amount of white space (including new-lines) as long as 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 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, and file control directives can appear anywhere in a `mapfile`.

The following sections describe each directive type. For all syntax discussions, the following notations apply:

- All entries in constant width, all colons, semicolons, equal signs, and at (@) signs are typed in literally.
- All entries in *italics* are substitutable.
- { ... }* means "zero or more."
- { ... }+ means "one or more."

- [...]means "optional."
- section_names and segment_names follow the same rules as C identifiers where a period (.) is treated as a letter (for example, .bss is a legal name).
- section_names, segment_names, file_names, and symbol_names are case sensitive; everything else is not case sensitive.
- Spaces (or new-lines) 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_values in any order, each separated by a space. (Only one attribute value is allowed for each segment attribute.) The segment attributes and their valid values are as follows:

TABLE 8–1 Mapfile Segment Attributes

Attribute	Value
segment_type	LOAD | NOTE
segment_flags	? [E] [N] [O] [R] [W] [X]
virtual_address	V*number*
physical_address	P*number*
length	L*number*
rounding	R*number*
alignment	A*number*

There are 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 235 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`, `R`, 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.
- If the rounding value is set for a segment, that segments virtual address will be rounded to the next address that conforms to the value given. This value only effects the segments that it is specified for. If no value is given no rounding is performed.

The `?E` flag allows the creation of an empty segment, this is a segment that has no sections associated with it. This segment can only be specified for executables, and must be of type `LOAD` with a specified size and alignment. Multiple segment definitions of this type are permitted.

The `?N` flag lets you control whether the ELF header, and any program headers, will be included as part of the first loadable segment. By default, the ELF header and program headers are included with the first segment, as the information in these headers is used within the mapped image (commonly by the runtime linker). The use of the `?N` option causes the virtual address calculations for the image to start at the first *section* of the first segment.

The `?O` flag lets you control the order of sections in the final relocatable object, executable file or shared object. This flag is intended for use 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 the `?O` flag to control the order of sections in effect controls the order of functions.

Consider the following user-defined `mapfile`:

```
text = LOAD ?RXO;
text: .text%foo;
text: .text%bar;
text: .text%main;
```

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 a static function bar() exists in files a.o and b.o, and function bar() from file a.o is to be placed before function bar() from file b.o, then the mapfile entries should read:

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

Although the syntax allows for the entry:

```
text: .text%bar: a.o b.o;
```

this entry does not guarantee that function bar() from file a.o will be placed before function bar() from file b.o. The second format is not recommended as 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 8–2 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_types listed above. The section_types listed above are built-in types. For more information on section_types, see “Sections” on page 155.
- The section_flags values are A for allocatable, W for writable, or X for executable. If an individual flag is preceded by an exclamation mark (!), the link-editor checks 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 `*filename` (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 `*filename`, the link-editor simulates a **basename**(1) on the file name from the command line and uses that to match against the specified `filename`. In other words, the `filename` from the `mapfile` only needs to match the last part of the file name from the command line. (See “Mapping Example” on page 233.)
- 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`.

Section-within-Segment Ordering

By using the following notation it is possible to specify the order that sections will be placed within a segment:

```
segment_name | section_name1;
segment_name | section_name2;
segment_name | section_name3;
```

The sections that are named in the above form will be placed before any unnamed sections, and in the order they are listed in the `mapfile`.

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.

File Control Directives

File control directives allow users to specify which version definitions within shared objects are to be made available during a link-edit. The file control definition has the following syntax:

```
shared_object_name - version_name [ version_name ... ];
```

`version_name` is a version definition name contained within the specified `shared_object_name`. For more information on version control see “Specifying a Version Binding” on page 109.

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 actually appears in the `mapfile`.

CODE EXAMPLE 8–1 User-Defined Mapfile

```
1.     elephant : .data : peanuts.o *popcorn.o;
2.     monkey : $PROGBITS ?AX;
3.     monkey : .data;
4.     monkey = LOAD V0x80000000 L0x4000;
5.     donkey : .data;
6.     donkey = ?RX A0x1000;
7.     text = V0x80008000;
```

Four separate segments are manipulated in this example. The implicitly declared segment `elephant` (line 1) receives all of the `.data` sections from the files `peanuts.o` and `popcorn.o`. Note that `*popcorn.o` matches any `popcorn.o` file that may be supplied to the link-edit; the file need not be in the current directory. On the other hand, if `/var/tmp/peanuts.o` was supplied to the link-edit, it will not match `peanuts.o` because it is not preceded by a `*`.

The implicitly declared segment `monkey` (line 2) receives all sections that are both `$PROGBITS` and allocatable-executable (`?AX`), as well as all sections (not already in the segment `elephant`) with the name `.data` (line 3). The `.data` sections entering the `monkey` segment need not be `$PROGBITS` or allocatable-executable because the section_type and section_flags values are entered on a separate line from the section_name value. (An “and” relationship exists between attributes on the same line as illustrated by `$PROGBITS` “and” `?AX` on line 2. An “or” relationship exists between attributes for the same segment that span more than one line as illustrated by `$PROGBITS ?AX` on line 2 “or” `.data` on line 3.)

The `monkey` segment is implicitly declared in line 2 with segment_type value `LOAD`, segment_flags value `RWX`, and no virtual_address, physical_address, length or alignment values specified (defaults are used). In line 4 the segment_type value of `monkey` is set to `LOAD` (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 `.data` sections to this segment. Actually, no sections fall into this segment because the entrance criteria for `monkey` in line 3 capture all of these sections. In line 6, the segment_flags value is set to `?RX` and the alignment value is set to 0x1000 (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 want to change the order of the directives to avoid warnings, use the following example:

```
1.      elephant : .data : peanuts.o *popcorn.o;
4.      monkey = LOAD V0x80000000 L0x4000;
2.      monkey : $PROGBITS ?AX;
3.      monkey : .data;
6.      donkey = ?RX A0x1000;
5.      donkey : .data;
7.      text = V0x80008000;
```

The following `mapfile` example uses the segment within section ordering:

```
1.      text = LOAD ?RXN V0xf0004000;
2.      text | .text;
3.      text | .rodata;
4.      text : $PROGBITS ?A!W;
5.      data = LOAD ?RWX R0x1000;
```

The `text` and `data` segments are manipulated in this example. Line 1 declares the `text` segment to have a `virtual_address` of 0xf0004000 and to *not* include the ELF header or any program headers as part of this segments address calculations. Lines 2 and 3 turn on section-within-segment ordering and specify that the `.text` and `.rodata` sections will be the first two sections in this segment. The result is that the `.text` section will have a virtual address of 0xf0004000, and the `.rodata` section will immediately follow that.

Any other `$PROGBITS` section that make up the `text` segment will follow the `.rodata` section. Line 5 declares the `data` segment and specifies that its virtual address must begin on a 0x1000 byte boundary. The first section that constitutes the `data` segment will also reside on a 0x1000 byte boundary within the file image.

Mapfile Option Defaults

The link-editor defines three built-in segments (`text`, `data`, and `note`) with default segment_attribute_values and corresponding default mapping directives as described in “Segment Declarations” on page 227. Even though the link-editor does not use an actual `mapfile` to provide the defaults, the model of a default `mapfile` helps illustrate what happens when the link-editor encounters your `mapfile`.

The 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 : ?A!W;
data = LOAD ?RWX;
data : ?AW;
note = NOTE;
note : $NOTE;
```

As each segment declaration in your `mapfile` is read in, it is compared to the existing list of segment declarations as follows:

1. If the segment does not already exist in the `mapfile`, but another with the same segment-type value exists, the segment is added before all of the existing segments of the same segment_type.
2. If none of the segments in the existing `mapfile` has the same segment_type value as the segment just read in, then the segment is added by segment_type value to maintain the following order:

   `INTERP`

   `LOAD`

   `DYNAMIC`

   `NOTE`
3. If the segment is of segment_type `LOAD` and you have defined a virtual_address value for this `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 Figure 8–1. 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.

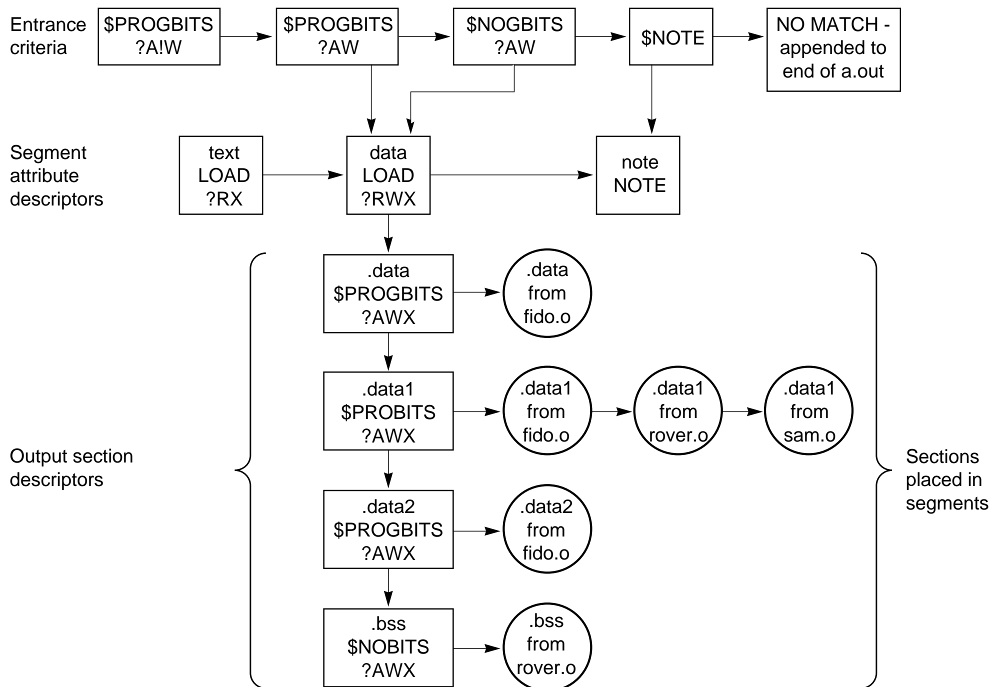


Figure 8–1 Simple Map Structure

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 8–1, 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 195 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 8–1, 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 8–1 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 "Sections" on page 155) 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 are read in, no program header entry is produced for that segment.

Note - Input sections of type $SYMTAB, $STRTAB, $REL, and $RELA are used internally by the link-editor. Directives that refer to these section_types can only map output sections produced by the link-editor to segments.

Error Messages

Warnings

Errors within this category do not stop execution of the link-editor nor do they prevent the link-editor from producing a viable `a.out`. 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

Errors within this category stop execution of the link-editor at the point the fatal error occurred. The following conditions produce fatal errors:

- 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 "Mapfile Structure and Syntax" on page 226 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.

APPENDIX **A**

Link-Editor Quick Reference

Overview

The following sections provide a simple overview, or *cheat sheet*, of the most commonly used link-editor scenarios (see “Link-Editing” on page 2for 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 (see “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 (see “Input File Processing” on page 10).

Shared objects that follow a naming convention of `lib`*x*`.so`, and archive libraries that follow a naming convention of `lib`*x*`.a`, can be input using the `-l` option (see “Library Naming Conventions” on page 13). This provides additional flexibility in allowing search paths to be specified using the `-L` option (see “Directories Searched by the Link-Editor” on page 15).

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

The use of static executables is limited. Static executables usually contain platform specific implementation details which restricts the ability of the executable to be run on an alternative platform. Also, many implementations of Solaris libraries depend on dynamic linking capabilities such as **dlopen**(3X) and **dlsym**(3X). (see "Loading Additional Objects" on page 55). These capabilities are not available to static executables.

- 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 (see "Linking with a Mix of Shared Objects and Archives" on page 14).

Building a Shared Object

- Use the –G option (–dy is optional as it is implied by default).

- Input relocatable objects should be built from position-independent code. Use the `–z text` option to enforce this requirement (see "Position-Independent Code" on page 86).
- Establish the shared objects *public interface* by defining the global symbols that should be visible from this shared object, and reducing any other global symbols to local scope. This definition is provided by the `–M` option together with an associated `mapfile`, and is covered in more detail inAppendix B.
- Use a *versioned* name for the shared object to allow for future upgrades (see "Coordination of Versioned Filenames" on page 114).
- 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 (see "Shared Objects with Dependencies" on page 76).
- Self contained shared objects offer maximum flexibility, and are produced when the object expresses all dependency needs. Use the `–z defs` to enforce this self containment (see "Generating a Shared Object" on page 26)

The following example combines the above points:

```
$ cc -c -o foo.o -Kpic foo.c
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs \
-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 (see "Recording a Shared Object Name" on page 73).
- Make the shared object available to the compilation environment by creating a file system link to a non-versioned shared object name (see "Coordination of Versioned Filenames" on page 114).

The following example combines the above points:

```
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs \
-R /home/lib-h libfoo.so.1 foo.o
$ ln -s libfoo.so.1 libfoo.so
```

- Consider the performance implications of the shared object; maximize shareability (see "Maximizing Shareability" on page 88) and minimize paging activity (see "Minimizing Paging Activity" on page 90), reduce relocation overhead, especially by minimizing symbolic relocations (see "Profiling Shared Objects" on page 95), and allow access to data via functional interfaces (see "Copy Relocations" on page 91).

Building a Dynamic Executable

- Don't use the –G, or –dn options.
- 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 (see "Directories Searched by the Runtime Linker" on page 16).

The following example combines the above points:

```
$ cc -o prog -R /home/lib -L. -lfoo file1.o file2.o file3.o .....
```

APPENDIX **B**

Versioning Quick Reference

Overview

ELF objects make available global symbols to which other objects can bind. Some of these global symbols can be identified as providing the object's *public interface.* Other symbols are part of the objects internal implementation and are not intended for external use. An objects interface can evolve from one software release to another, and thus the ability to identify this evolution is desirable.

In addition, identifying the *internal implementation* changes of an object from one software release to another might be desirable.

Both interface and implementation identifications can be recorded within an object by establishing internal *version definitions* (see "Overview" on page 97 for a more complete introduction to the concept of internal versioning).

Shared objects are prime candidates for internal versioning as this technique defines their evolution, provides for interface validation during runtime processing (see "Binding to a Version Definition" on page 105), and provides for the selective binding of applications (see "Specifying a Version Binding" on page 109). Shared objects will be used as the examples throughout this chapter.

The following sections provide a simple overview, or *cheat sheet*, of the internal versioning mechanism provided by the link-editors as applied to shared objects. The examples recommend conventions and mechanisms for versioning shared objects, from their initial construction through several common update scenarios.

Naming Conventions

A shared object follows a naming convention that includes a *major* number file suffix (see "Naming Conventions" on page 72). Within this shared object, one or more *version definitions* can be created. Each version definition corresponds to one of the following categories:

- It defines an industry *standard* interface (for example, the *System V Application Binary Interface*).
- It defines a vendor specific *public* interface.
- It defines a vendor specific *private* interface.
- It defines a vendor specific change to the internal implementation of the object.

The following *version definition* naming conventions help indicate which of these categories the definition represents.

The first three of these categories indicate interface definitions. These definitions consist of an association of the global symbol names that comprise the interface, with a version definition name (see "Creating a Version Definition" on page 99). Interface changes within a shared object are often referred to as *minor* revisions. Therefore, version definitions of this type, are suffixed with a *minor* version number which is based off of the filenames *major* version number suffix.

The last category indicates a change having occurred within the object. This definition consists of a version definition acting as a label and has no symbol names associated with it. This definition is referred to as being a *weak* version definition (see "Creating a Weak Version Definition" on page 102). Implementation changes within a shared object are often referred to as *micro* revisions. Therefore, version definitions of this type are suffixed with a *micro* version number based off of the previous *minor* number to which the internal changes have been applied.

Any industry standard interfaces should use a version definition name that reflects the standard. Any vendor interfaces should use a version definition name unique to that vendor (the company's stock symbol is often appropriate).

Private version definitions indicate symbols that have restricted or uncommitted use, and should have the word *private* clearly visible.

All version definitions result in the creation of associated version symbol names. Therefore, the use of unique names and the *minor/micro* suffix convention reduce the chance of symbol collision within the object being built.

The following version definition examples show the use of these naming conventions:

```
SVABI.1
```

defines the *System V Application Binary Interface* standards interface.

```
SUNW_1.1
```

defines a SunSoft public interface.

```
SUNWprivate_1.1
```

defines a SunSoft private interface.

```
SUNW_1.1.1
```

defines a SunSoft internal implementation change.

Defining a Shared Object's Interface

When establishing a shared object's interface the first task is to determine which global symbols provided by the shared object can be associated to one of the three interface version definition categories:

- Industry standard interface symbols conventionally are defined in publicly available header files and associated manual pages supplied by the vendor, and are also documented in recognized standards literature.
- Vendor public interface symbols conventionally are defined in publicly available header files and associated manual pages supplied by the vendor.
- Vendor private interface symbols can have little or no public definition.

By defining these interfaces, a vendor is indicating the commitment level of each interface of the shared object. Industry standard and vendor public interfaces remain stable from release to release. You are free to bind to these interfaces safe in the knowledge that your application will continue to function correctly from release to release.

Industry standard interfaces might be available on systems provided by other vendors, and thus you can achieve a higher level of binary compatibility by restricting your applications to use these interfaces.

Vendor public interfaces might not be available on systems provided by other vendors, however these interfaces will remain stable during the evolution of the system on which they are provided.

Vendor private interfaces are very unstable, and can change, of even be deleted, from release to release. These interfaces provide for uncommitted or experimental functionality, or are intended to provide access for vendor specific applications only. If you who wish to achieve *any* level of binary compatibility you should avoid using these interfaces.

Any global symbols that do not fall into one of the above categories should be reduced to local scope so that they are no longer visible for binding (see “Reducing Symbol Scope” on page 33).

Versioning a Shared Object

Having determined a shared object’s available interfaces, the associated version definitions are created using a `mapfile` and the link-editors `-M` option (see “Defining Additional Symbols” on page 28 for an introduction of this `mapfile` syntax).

The following example defines a vendor public interface in the shared object `libfoo.so.1`:

```
$ cat mapfile
SUNW_1.1 {                      # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
$ cc -G -o libfoo.so.1 -h libfoo.so.1 -z text -M mapfile foo.c
```

Here the global symbols `foo1` and `foo2` are assigned to the shared objects public interface `SUNW_1.1`. Any other global symbols supplied from the input files are reduced to local by the *auto-reduction* directive “*” (see “Reducing Symbol Scope” on page 33).

Note - Each version definition `mapfile` entry should be accompanied by a comment reflecting the release or date of the update. This information helps coordinate multiple updates of a shared object, possibly by different developers, into one version definition suitable for delivery of the shared object as part of a software release.

Versioning an Existing (Non-versioned) Shared Object

Versioning an existing, non-versioned shared object requires extra care, as the shared object delivered in a previous software release has made available *all* its global symbols for others to bind with. Although it can be possible to determine the shared objects *intended* interfaces, it can be the case that others have discovered and bound

to other symbols. Therefore, the removal of any symbols *might* result in an applications failure on delivery of the new versioned shared object.

The internal versioning of an existing, non-versioned shared object can be achieved if the interfaces can be determined, and applied, without breaking any existing applications. The runtime linker's debugging capabilities can be useful to help verify the binding requirements of various applications (see "Debugging Aids" on page 67). However, this determination of existing binding requirements assumes that all users of the shared object are known.

If the binding requirements of an existing, non-versioned shared object can not be determined then it is necessary to create a new shared object file using a new versioned name (see "Coordination of Versioned Filenames" on page 114). In addition to this new shared object, the original shared object must also be delivered so as to satisfy the dependencies of any existing applications.

If the implementation of the original shared object is to be frozen then maintaining and delivering the shared object *binary* might be sufficient. If however, the original shared object might require updating - for example, through patches, or because its implementation must evolve to remain compatible with new platforms - then an alternative source tree from which to generate the shared object can be more applicable.

Updating a Versioned Shared Object

The only changes that can be made to a shared object that can be absorbed by internal versioning are *compatible* changes (see "Interface Compatibility" on page 98). Any *incompatible* changes require producing a new shared object with a new *external* versioned name (see "Coordination of Versioned Filenames" on page 114).

Compatible updates that can be accommodated by internal versioning fall into three basic categories:

- adding new symbols.
- creating new interfaces from existing symbols.
- internal implementation changes.

The first two categories are achieved by associating an interface version definition with the appropriate symbols. The latter is achieved by creating a *weak* version definition that has no associated symbols.

Adding New Symbols

Any compatible new release of a shared object that contains new global symbols should assign these symbols to a new version definition. This new version definition should inherit the previous version definition.

The following `mapfile` example assigns the new symbol `foo3` to the new interface version definition `SUNW_1.2`. This new interface inherits the original interface `SUNW_1.1`:

```
$ cat mapfile
SUNW_1.2 {                    # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1 {                    # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
```

The inheritance of version definitions reduces the amount of version information that must be recorded in any user of the shared object.

Internal Implementation Changes

Any compatible new release of the shared object that consists of an update to the implementation of the object - for example, a bug fix or performance improvement - should be accompanied by a *weak* version definition. This new version definition should inherit the latest version definition present at the time the update occurred.

The following `mapfile` example generates a weak version definition `SUNW_1.1.1`. This new interface indicates that the internal changes were made to the implementation offered by the previous interface `SUNW_1.1`:

```
$ cat mapfile
SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.

SUNW_1.1 {                    # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
```

(continued)

```
};
```

New Symbols and Internal Implementation Changes

If both internal changes and the addition of a new interface has occurred during the same release, both a weak version and a interface version definition should be created. The following example shows the addition of a version definition `SUNW_1.2` and an interface change `SUNW_1.1.1` which are added during the same release cycle. Both interfaces inherit the original interface `SUNW_1.1`:

```
$ cat mapfile
SUNW_1.2 {                    # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;      # Release X+1.

SUNW_1.1 {                    # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
```

Note - The comments for the `SUNW_1.1` and `SUNW_1.1.1` version definitions indicate that they have both been applied to the same release.

Migrating Symbols to a Standard Interface

Occasionally, symbols offered by a vendors interface become absorbed into a new industry standard. When creating a new standard interface it is important to maintain the original interface definitions provided by the shared object. To accomplish this it is necessary to create intermediate version definitions on which the new standard, and original interface definitions, can be built.

The following `mapfile` example shows the addition of a new industry standard interface `STAND.1`. This interface contains the new symbol `foo4` and the existing

symbols `foo3` and `foo1` which were originally offered through the interfaces `SUNW_1.2` and `SUNW_1.1` respectively:

```
$ cat mapfile
STAND.1 {                       # Release X+2.
        global:
                foo4;
} STAND.0.1 STAND.0.2;

SUNW_1.2 {                      # Release X+1.
        global:
                SUNW_1.2;
} STAND.0.1 SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;        # Release X+1.

SUNW_1.1 {                      # Release X.
        global:
                foo2;
        local:
                *;
} STAND.0.2;
                                # Subversion - providing for
STAND.0.1 {                     # SUNW_1.2 and STAND.1 interfaces.
        global:
                foo3;
};
                                # Subversion - providing for
STAND.0.2 {                     # SUNW_1.1 and STAND.1 interfaces.
        global:
                foo1;
};
```

Here the symbols `foo3` and `foo1` are pulled into their own intermediate interface definitions which are used to build the original and new interface definitions.

Note - The new definition of the `SUNW_1.2` interface has referenced its own version definition symbol. Without this reference the `SUNW_1.2` interface would have contained no immediate symbol references and hence would be categorized as a weak version definition.

When migrating symbol definitions to a standards interface the requirement is that any original interface definitions continue to represent the same symbol list. This requirement can be validated using **pvs**(1). The following example shows the symbol list of the `SUNW_1.2` interface as it existed in the software release `X+1`:

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
                foo3;
        SUNW_1.1:
                foo2;
```

(Continuation)

```
                foo1;
```

Although the introduction of the new standards interface in software release X+2 has changed the interface version definitions available, the list of symbols provided by each of the original interfaces remains constant. The following example shows that interface SUNW_1.2 still provides symbols foo1, foo2 and foo3:

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
        SUNW_1.2:
        STAND.0.1:
                foo3;
        SUNW_1.1:
                foo2;
        STAND.0.2:
                foo1;
```

It is possible that an application might only reference one of the new subversions, in which case any attempt to run the application on a previous release will result in a runtime versioning error (see “Binding to a Version Definition” on page 105).

In this case an applications version binding can be promoted by directly referencing an existing version name (see “Binding to Additional Version Definitions” on page 111).

For example, if an application only references the symbol foo1 from the shared object libfoo.so.1, then its version reference will be to STAND.0.2. To allow this application to be run on previous releases, the version binding can be promoted to SUNW_1.1 using a version control mapfile directive:

```
$ cat prog.c
extern void foo1();

main()
{
        foo1();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (STAND.0.2);

$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
```

(continued)

```
        libfoo.so.1 (SUNW_1.1);
```

In practice it is rarely necessary to promote a version binding in this manner, as the introduction of new standards binary interfaces is rare, and most applications reference many symbols from an interface family.

Index

A

ABI (see Application Binary Interface and System V Application Binary Interface), 5
$ADDVERS, *see* versioning,
Application Binary Interface, 5, 80, 97
ar(1), 11
archives, 13
- inclusion of shared objects in, 74
- link-editor processing, 11
- multiple passes through, 12
- naming conventions, 13

as(1), 2
auxiliary filters, 78, 81

B

base address, 199
binding, 1
- dependency ordering, 77
- lazy, 49, 57, 69
- to shared object dependencies, 73, 105
- to version definitions, 105
- to weak version definitions, 112

C

cc(1), 1, 2, 9
COMMON, 19, 30, 32, 157
compilation environment, 4, 14, 72
- *See also* link-editing and link-editor,

D

data representation, 147
debugging aids
- link-editing, 38
- runtime linking, 67

demonstrations
- prefcnt, 129
- sotruss, 129
- symbindrep, 129
- whocalls, 129

dependency
- groups, 56, 58

dependency ordering, 77
dlclose(3X), 55
dldump(3X), 18
dlerror(3X), 55
dlfcn.h, 55
dlmopen(3X), 123
dlopen(3X), 44, 55, 61, 83, 108, 118
- effects of ordering, 60
- group, 56, 58
- modes
  - RTLD_GLOBAL, 57, 61
  - RTLD_GROUP, 62
  - RTLD_LAZY, 57
  - RTLD_NOLOAD, 123
  - RTLD_NOW, 57
  - RTLD_PARENT, 62, 63
- of a dynamic executable, 57, 61
- shared object naming conventions, 72

dlsym(3X), 44, 55, 63, 66, 108, 118

special handle
RTLD_DEFAULT, 27, 63
RTLD_NEXT, 63
dump(1), 5, 45, 47, 85, 87
dynamic executables, 2, 4
dynamic information tags
NEEDED, 45, 73
RPATH, 45
SONAME, 74
TEXTREL, 87
dynamic linking, 4
implementation, 177, 185, 204

E

ELF, 2, 7, 83
(see also object files), 145
elf(3E), 5
environment variables
LD_AUDIT, 124
LD_BIND_NOT, 69
LD_BIND_NOW, 49, 69, 209, 221, 223
LD_BREADTH, 53
LD_DEBUG, 67
LD_DEBUG_OUTPUT, 68
LD_LIBRARY_PATH, 16, 46, 54, 56, 76
LD_LOADFLTR, 83
LD_NOAUXFLTR, 82
LD_OPTIONS, 9, 39
LD_PRELOAD, 51, 54
LD_PROFILE, 95
LD_PROFILE_OUTPUT, 95
LD_RUN_PATH, 17
SGS_SUPPORT, 118
error messages
link-editor
illegal argument to option, 10
illegal option, 10
incompatible options, 10
multiple instances of an option, 10
multiply defined symbols, 24
relocations against non-writable sections, 87
shared object name conflicts, 75
soname conflicts, 75
symbol not assigned to version, 35
symbol warnings, 22
undefined symbols, 24, 25
undefined symbols from an implicit reference, 26
version unavailable, 110
runtime linker
copy relocation size differences, 94
relocation errors, 49, 107
unable to find shared object, 46, 56
unable to find version definition, 107
unable to locate symbol, 64
exec(2), 7, 43, 146
executable and linking format (see ELF), 2

F

f77(1), 9
filters, 78
auxiliary, 78, 81
platform specific, 82
standard, 78

G

generating a shared object, 26
generating an executable, 24
generating the output file image, 37
global offset table, 38, 47, 86, 185, 218, 219
global symbols, 19, 98, 173, 174

I

initialization and termination, 9, 17, 52
input file processing, 10
interface
private, 98
public, 98, 246

interposition, 21, 22, 34, 48, 52, 65, 98
interpreter (see also runtime linker), 43

L

lazy binding, 49, 57, 69, 122
ld(1), 1
ld.so.1(1) (see also runtime linker), 7, 43
ld.so.1(1) (see runtime linker), 2
ldd(1), 5, 44, 46, 48, 50, 83, 107, 108
ldd(1) options, 52
 -d, 50, 83, 94
 -i, 52
 -l, 83
 -r, 83, 94
 -v, 107
LD_AUDIT, 124
LD_BIND_NOT, 69
LD_BIND_NOW, 49, 69, 209, 221, 223
LD_BREADTH, 53
LD_DEBUG, 67
LD_DEBUG_OUTPUT, 68
LD_LIBRARY_PATH, 16, 46, 54, 56, 76
LD_LOADFLTR, 83
LD_NOAUXFLTR, 82
LD_OPTIONS, 10, 39
LD_PRELOAD, 51, 54
LD_PROFILE, 95
LD_PROFILE_OUTPUT, 95
LD_RUN_PATH, 17
libdl.so.1, 55
libelf.so.1, 118, 145
libldstab.so.1, 118
libraries
 archives, 13
 naming conventions, 13
 shared, 177, 185, 204
link-editing, 2, 171, 185, 204
 adding additional libraries, 13
 archive processing, 11
 binding to a version definition, 105, 109
 dynamic, 177, 185, 204
 input file processing, 10
 library input processing, 11
 library linking options, 11
 mixing shared objects and archives, 14
 multiply defined symbols, 174
 position of files on command line, 14
 search paths, 15
 shared object processing, 12
link-editor, 1, 7
 debugging aids, 38
 invoking directly, 8
 invoking using compiler driver, 9
 overview, 7
 sections, 7
 segments, 7
 specifying options, 9
link-editor options
 -a, 242
 -B dynamic, 14
 -B group, 58, 62, 217
 -B reduce, 30, 37
 -B static, 14, 242
 -D, 39
 -d, 242
 -e, 38
 -F, 78
 -f, 78
 -G, 71
 -h, 45, 73, 116, 243
 -i, 16
 -l, 11, 13
 -L, 15
 -l, 72, 114
 -L, 241
 -l, 241
 -M, 8
 -m, 13, 22
 -M, 28, 29, 98, 99, 109, 225, 243, 248
 -r, 9
 -R, 17, 76
 -r, 242
 -R, 243, 244
 -s, 37
 -S, 118
 -t, 22, 23
 -u, 28
 -Y, 15
 -z allextract, 11
 -z defaultextract, 12
 -z defs, 26, 124, 243
 -z ignore, 12
 -z initfirst, 217
 -z loadfltr, 83, 217

-z muldefs, 24
-z nodefs, 25, 50
-z nodelete, 217
-z nodlopen, 217
-z noversion, 35, 100, 107
-z now, 49, 57, 217
-z text, 87, 243
-z weakextract, 11, 174
link-editor output
dynamic executables, 3
relocatable objects, 2
shared objects, 3
static executables, 2
link-editor support interface (ld-support), 117
ld_atexit, 120
ld_file, 119
ld_section, 120
ld_start, 119
link-editor, *see* error messages
error messages,
local symbols, 19, 173, 174
lorder(1), 12, 40

M

mapfiles, 225, 241
defaults, 235
error messages, 238
example, 233
map structure, 236
mapping directives, 230
segment declarations, 227
size-symbol declarations, 232
structure, 226
syntax, 226
usage, 226
mmap(2), 43
multiply defined symbols, 37, 173, 174

N

Name-space, 123
naming conventions
archives, 13
libraries, 13
shared objects, 13, 72
NEEDED, 45, 73
nm(1), 5, 85

O

object files, 2
base address, 199
data representation, 147
global offset table (see global offset table), 218
note section, 193, 194
preloading at runtime, 51
procedure linkage table (see procedure linkage table), 219, 221
program header, 195, 198
program interpretor, 207
program loading, 201
relocation, 177, 185, 218
section alignment, 159
section attributes, 163, 170
section header, 156, 170
section names, 170
section types, 159, 170
segment contents, 200, 201
segment permissions, 199, 200
segment types, 196, 199
string table, 170, 171
symbol table, 171, 177

P

packages
SUNWosdem, 129, 132, 145
SUNWtool, 129
paging, 204, 201
performance
allocating buffers dynamically, 89
collapsing multiple definitions, 89
improving locality of references, 90, 95
maximizing shareability, 88
minimizing data segment, 88
position-independent code (see position-dependent code), 86
relocations, 90, 95
the underlying system, 86
using automatic variables, 89
platform specific auxiliary filters, 82
$PLATFORM, *see* search paths,
position-independent code, 86, 206, 217, 218
preloading objects (see LD_PRELOAD also), 51

procedure linkage table, 38, 49, 86, 185, 219, 221
SPARC, 219, 221
profil(2), 95
program interpreter, 43, 207, 208
(see also runtime linker), 43
pvs(1), 5, 100, 102, 105, 106

R

relocatable objects, 2
relocation, 46, 90, 94, 177, 185
copy, 91
data references, 48
function references, 49
non-symbolic, 47, 90
runtime linker
symbol lookup, 48, 49, 57, 69
symbolic, 47, 90
RPATH (see also runpath), 45
RTLD_DEFAULT, 27
See also dependency ordering,
RTLD_GLOBAL, 57, 61
RTLD_GROUP, 62
RTLD_LAZY, 57
RTLD_NEXT
See also dependency ordering,
RTLD_NOLOAD, 123
RTLD_NOW, 57
RTLD_PARENT, 62, 63
runpath, 17, 45, 54, 56, 69, 76
runtime environment, 4, 14, 72
runtime linker, 2, 3, 43, 208
initialization and termination routines, 52
lazy binding, 49, 57, 69
link-maps, 123
loading additional objects, 51
name-space, 123
programming interface (see also dlopen(3X) family of routines), 54
relocation processing, 46
search paths, 17, 44
security, 53, 124
shared object processing, 44
version definition verification, 107
runtime linker support interfaces (rtld-audit), 117, 122
la_i86_pltenter, 127
la_objclose, 128
la_objopen, 125
la_pltexit, 127
la_preinit, 125
la_sparcv8_pltenter, 127
la_symbind32, 126
la_version, 125
runtime linker support interfaces (rtld-debugger), 117, 130
ps_global_sym, 142
ps_pglobal_sym, 142
ps_plog, 142
ps_pread, 142
ps_pwrite, 142
rd_delete, 133
rd_errstr, 134
rd_event_addr, 138
rd_event_enable, 137
rd_event_getmsg, 139
rd_init, 133
rd_loadobj_iter, 136
rd_log, 134
rd_new, 133
rd_objpad_enable, 141
rd_plt_resolution, 139
rd_reset, 133
runtime linking, 3

S

SCD (see SPARC Compliance Definition), 5
search paths
link-editing, 15
runtime linker, 17, 44
$PLATFORM token, 82
section types, 17, 45, 92
.bss, 7
.data, 7, 88
.dynamic, 38, 43
.dynstr, 37
.dynsym, 37
.fini, 17, 52
.got, 38, 47
.init, 17, 52
.interp, 43
.picdata, 89

.plt, 38, 49, 95
.rela.text, 7
.rodata, 88
.strtab, 7, 37
.SUNW_version, 188
.symtab, 7, 37
.text, 7
sections, 7, 84
(see also section types), 7
security, 53, 124
segments, 7, 84
data, 84, 86
text, 84, 86
SGS_SUPPORT, 118
shared libraries (see shared objects), 2
shared objects, 2 to 4, 44
as filters (see filters), 78
building (see also performance), 71
dependency ordering, 77
explicit definition, 26
implementation, 177, 185, 204
implicit definition, 25
link-editor processing, 12
naming conventions, 13, 72
recording a runtime name, 73
with dependencies, 76
size(1), 83
SONAME, 74
SPARC Compliance Definition, 5
standard filters, 78
static executables, 2
strings(1), 89
strip(1), 37
SUNWosdem, 129, 132, 145
SUNWtoo, 129
support interfaces
link-editor (ld-support), 117
runtime linker (rtld-audit), 117, 122
runtime linker (rtld-debugger), 117, 130
symbol reserved names, 37
_DYNAMIC, 38
_edata, 37
_end, 38
END, 38
_etext, 37
_fini, 17
_GLOBAL_OFFSET_TABLE_, 38, 219
_init, 17
main, 38
_PROCEDURE_LINKAGE_TABLE_, 38
_start, 38
START, 38
symbol resolution, 18, 19, 37
complex, 22
fatal, 23
interposition (see interposition), 48
multiple definitions, 13
search scope
group, 58
simple, 20
symbol visibility
global, 58
local, 58
symbols
absolute, 30, 157
archive extraction, 11
auto-reduction, 30, 35, 100, 248
COMMON, 19, 30, 32, 157
defined, 19
definition, 11, 24
existence test, 26
global, 19, 20, 98, 173, 174
local, 19, 173, 174
private interface, 98
public interface, 98
reference, 11, 24
runtime lookup, 57, 66
deferred, 49, 57, 69
scope, 57, 61
tentative, 11, 19, 27, 30, 32, 157
ordering in the output file, 27
realignment, 32
undefined, 11, 19, 24, 26
weak, 11, 20, 26, 174
System V Application Binary Interface, 246

T

tentative symbols, 11, 19, 30, 32
TEXTREL, 87
tsort(1), 12, 40

U

undefined symbols, 24

/usr/ccs/bin/ld, *see* link-editor,
/usr/lib, 16, 44, 45, 54, 56, 124
/usr/lib/ld.so.1, 43, 130

V

versioning, 97
  base version definition, 100
  binding to a definition, 105, 109
    $ADDVERS, 109
  defining a public interface, 35, 99
  definitions, 98, 99, 105
  file control directive, 109
  filename, 99, 249
  generating definitions within an image, 29, 35, 99
  normalization, 106
  overview, 97
  runtime verification, 107, 108
virtual addressing, 201

W

weak symbols, 20, 173, 174
  undefined, 26