Object Files 6

Introduction

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

- A relocatable file holds code and data suitable to be linked with other object files to create an executable or shared object file, 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 140, focuses on the format of object files and how that pertains to building programs. The second section, "Dynamic Linking" on page 189, 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 *man Pages(3): Library Routines*" for a description of `libelf` contents.

6

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 6-1 shows an object file's organization.

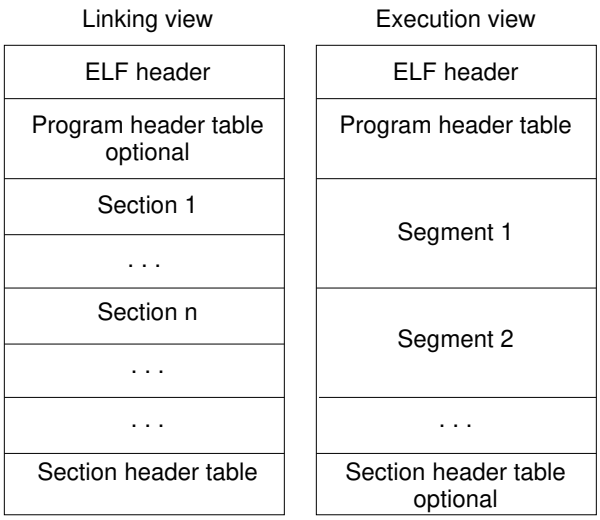


Figure 6-1 Object File Format

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 6-1 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 146.

`e_type`
This member identifies the object file type.

Table 6-2 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 6-3 ELF Machines

Name	Value	Meaning
`EM_NONE`	`0`	No machine
`EM_M32`	`1`	AT&T WE 32100
`EM_SPARC`	`2`	SPARC
`EM_386`	`3`	Intel 80386
`EM_68K`	`4`	Motorola 68000
`EM_88K`	`5`	Motorola 88000
`EM_486`	`6`	Intel 80486
`EM_860`	`7`	Intel 80860
`EM_MIPS`	`8`	MIPS RS3000 Big-Endian

Table 6-3 ELF Machines

Name	Value	Meaning
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 6-4 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, x86, and PowerPC.

`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 148 and "String Table" on page 161 for more information.

6

ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the `e_ident` member.

Table 6-5 e_ident[ ] Identification Index

Name	Value	Purpose
`EI_MAG0`	`0`	File identification
`EI_MAG1`	`1`	File identification
`EI_MAG2`	`2`	File identification
`EI_MAG3`	`3`	File identification
`EI_CLASS`	`4`	File class
`EI_DATA`	`5`	Data encoding
`EI_VERSION`	`6`	File version
`EI_PAD`	`7`	Start of padding bytes
`EI_NIDENT`	`16`	Size of `e_ident[]`

These indexes access bytes that hold the following values:

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

Table 6-6 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 6-7 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 6-8 Data Encoding

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

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

EI_VERSION
Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained in Table 6-4 on page 144 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.

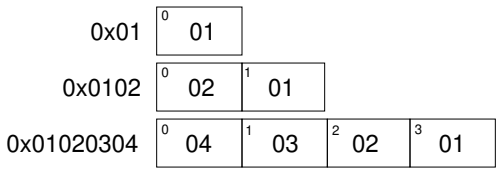


Figure 6-2 Data Encoding `ELFDATA2LSB`

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

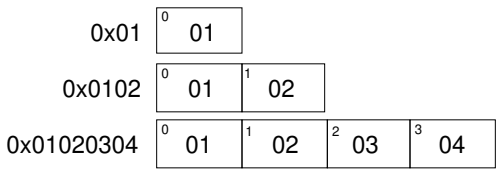


Figure 6-3 Data Encoding `ELFDATA2MSB`

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 6-9 Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
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_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 161), giving the location of a null-terminated string. Section names and their descriptions are in Table 6-14 on page 157.

`sh_type`
This member categorizes the section’s contents and semantics. Section types and their descriptions are in Table 6-10 on page 152.

`sh_flags`
Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions are inTable 6-12 on page 155.

`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 6-13 on page 156 describes the values.

`sh_info`
This member holds extra information, whose interpretation depends on the section type. Table 6-13 on page 156 below describes the values.

`sh_addralign`
Some sections have address alignment constraints. For example, if a section holds a double-word, the system must ensure double-word alignment for the entire section. That is, the value of `sh_addr` must be congruent to 0,

modulo the value of `sh_addralign`. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

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

A section header's `sh_type` member specifies the section's semantics:

Table 6-10 Section Types, *sh_type*

Name	`Value`
`SHT_NULL`	`0`
`SHT_PROGBITS`	`1`
`SHT_SYMTAB`	`2`
`SHT_STRTAB`	`3`
`SHT_RELA`	`4`
`SHT_HASH`	`5`
`SHT_DYNAMIC`	`6`
`SHT_NOTE`	`7`
`SHT_NOBITS`	`8`
`SHT_REL`	`9`
`SHT_SHLIB`	`10`
`SHT_DYNSYM`	`11`
`SHT_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 162 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 161 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 167 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 224 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 207 for details.

`SHT_NOTE`
The section holds information that marks the file in some way. See "Note Section" on page 187 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 167 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 6-11 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 6-12 Section Attribute Flags

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

If a flag bit is set in sh_flags, the attribute is *on* for the section. Otherwise, the attribute is *off* or does not apply. Undefined attributes are 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_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 6-13 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.
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 6-14 Special Sections *(1 of 2)*

Name	Type	Attribute
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	None
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	See "`.got`" on page 158
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	See ".interp" on page 159
.note	SHT_NOTE	None
.plt	SHT_PROGBITS	See "`.plt`" on page 159
.rel*name*	SHT_REL	See ".relname, .relaname" on page 159
.rela*name*	SHT_RELA	See ".relname, .relaname" on page 159
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	None
.strtab	SHT_STRTAB	See ".strtab" on page 160

Table 6-14 Special Sections *(2 of 2)*

Name	Type	Attribute
.symtab	SHT_SYMTAB	See ".symtab" on page 160
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.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 162 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 213 for more information.

`.hash`
This section holds a symbol hash table. See “Hash Table” on page 224 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 204 for more information.

`.note`
This section holds information in the format that “Note Section” on page 187 describes.

`.plt`
This section holds the procedure linkage table. For PowerPC only, the `SHT_NOBITS` type may also be used. See “Procedure Linkage Table” on page 215, “Procedure Linkage Table” on page 218, and “Procedure Linkage Table” on page 221 for more information.

`.rel`*`name`*`, .rela`*`name`*
These sections hold relocation information, as “Relocation” on page 167 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 189 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 162 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_version`
Sections of this name hold versioning information. See "Versioning Information" on page 181 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 6-4 String Table

The table below shows the strings of the string table above:

Table 6-15 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 6-16 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 167.

`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 6-10 on page 152

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

Table 6-17 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 12), it extracts archive members that contain definitions of undefined or tentative, global symbols. The member's definition may be either a global or a weak symbol. The link-editor does *not* extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with STB_LOCAL binding precede the weak and global symbols. As "Sections" on page 148 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 6-18 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 38, and Chapter 5, "Versioning" 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 6-19 Symbol Table Entry: Index 0

Name	Value	Note
st_name	0	No name
st_value	0	Zero value
st_size	0	No size

Table 6-19 Symbol Table Entry: Index 0

Name	Value	Note
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 and PowerPC use `Elf32_Rela` entries and x86 uses `Elf32_Rel` entries.

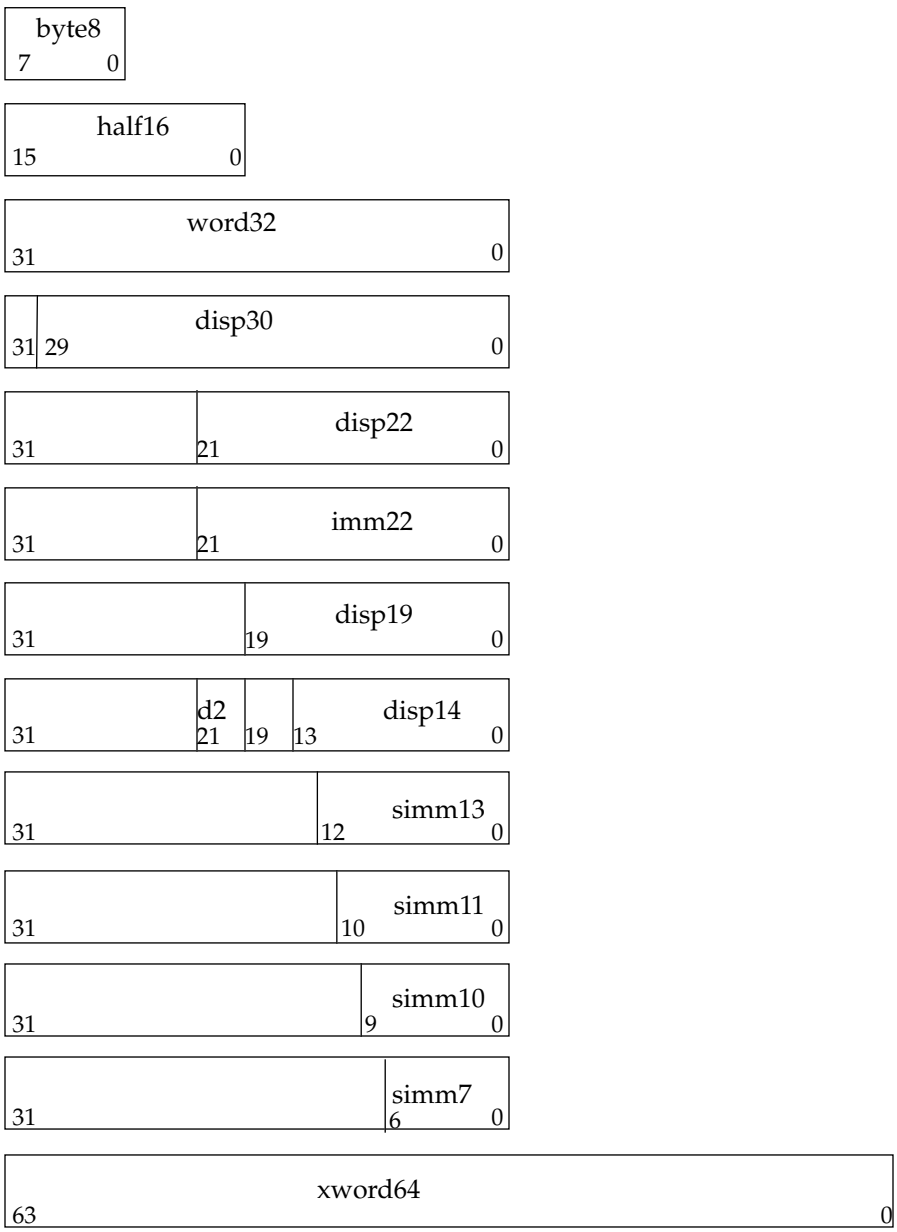
A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_info` and `sh_link` members, described in "Sections" on page 148 earlier, specify these relationships. Relocation entries for different object files have slightly different interpretations 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).

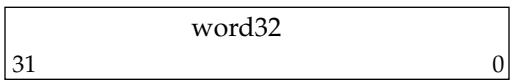
Although the interpretation of `r_offset` changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

Relocation Types (Processor Specific)

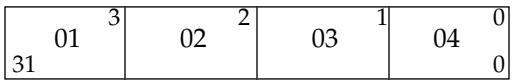
On SPARC, relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners):



On x86, relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners):

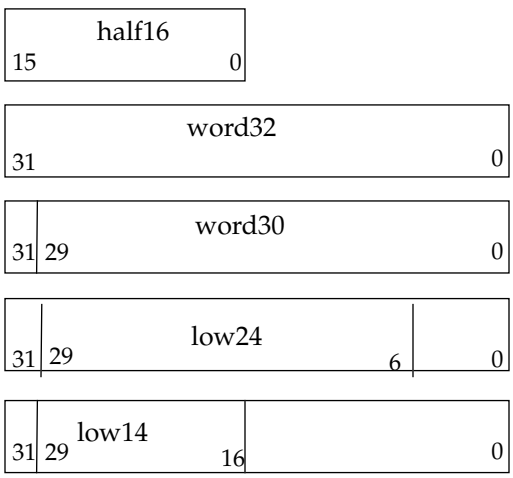


`word32` specifies a 32-bit field occupying 4 bytes with an arbitrary byte alignment. These values use the same byte order as other word values in the x86 architecture):



0x01020304

On PowerPC, relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners)



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 189 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 213 for more information.

GOT
: means the address of the *global offset table*. See "Global Offset Table (Processor-Specific)" on page 213 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" on page 215, "Procedure Linkage Table" on page 218, or "Procedure Linkage Table" on page 221 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. PowerPC relocation entries apply to half-words and 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 and PowerPC use 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 6-20 SPARC Relocation Types *(1 of 2)*

Name	Value	Field	Calculation
R_SPARC_NONE	0	None	None
R_SPARC_8	1	V-byte8	S + A
R_SPARC_16	2	V-half16	S + A
R_SPARC_32	3	V-word32	S + A
R_SPARC_DISP8	4	V-byte8	S + A - P
R_SPARC_DISP16	5	V-half16	S + A - P
R_SPARC_DISP32	6	V-disp32	S + A - P
R_SPARC_WDISP30	7	V-disp30	(S + A - P) >> 2
R_SPARC_WDISP22	8	V-disp22	(S + A - P) >> 2
R_SPARC_HI22	9	T-imm22	(S + A) >> 10
R_SPARC_22	10	V-imm22	S + A
R_SPARC_13	11	V-simm13	S + A
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

Table 6-20 SPARC Relocation Types *(2 of 2)*

Name	Value	Field	Calculation
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	None	None
R_SPARC_GLOB_DAT	20	V-word32	S + A
R_SPARC_JMP_SLOT	21	None	See below
R_SPARC_RELATIVE	22	V-word32	B + A
R_SPARC_UA32	23	V-word32	S + A
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
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 107 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 6-21 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 107

`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.

PowerPC: Relocation Types

The following general rules apply to the interpretation of the relocation types in Table 6-22:

- "+" and "−" denote 32-bit modulus addition and subtraction, respectively. ">>" denotes arithmetic right shifting (shifting with sign copying) of the value of the left operand by the number of bits given by the right operand.
- For relocation types in which the names contain the string `14` or the string `16`, the upper 17 bits of the value computed before shifting must all be the same. For relocation types whose names contain the string `24`, the upper 7 bits of the value computed before shifting must all be the same. For relocation types whose names contain the string `14` or the string `24`, the low 2 bits of the value computed before shifting must all be zero.
- `#hi(`*value*`)` and `#lo(`*value*`)` denote the most and least significant 16 bits, respectively, of the indicated value. That is,

  `#lo(`*x*`) = (`*x*` & 0xFFFF) and #hi(`*x*`) = ((`*x*` >> 16)& 0xFFFF)`

  The "high adjusted" value, #ha(*value*), compensates for #lo() being treated as a signed number.

  `#ha(`*x*`) = (((`*x*` >> 16) + ((`*x*` & 0x8000) ? 1 : 0)) & 0xFFFF).`
- Reference in a calculation to the value `G` implicitly creates a `GOT` entry for the indicated symbol.

Table 6-22 Relocation Types

Name	Value	Field	Calculation
R_PPC_NONE	0	none	none
R_PPC_ADDR32	1	word32	S + A
R_PPC_ADDR24	2	low24*	(S + A) >> 2
R_PPC_ADDR16	3	half16*	S + A
R_PPC_ADDR16_LO	4	half16	#lo(S + A)
R_PPC_ADDR16_HI	5	half16	#hi(S + A)
R_PPC_ADDR16_HA	6	half16	#ha(S + A)
R_PPC_ADDR14	7	low14*	(S + A) >> 2
R_PPC_ADDR14_BRTAKEN	8	low14*	(S + A) >> 2
R_PPC_ADDR14_BRNTAKEN	9	low14*	(S + A) >> 2
R_PPC_REL24	10	low24*	(S + A − P) >> 2
R_PPC_REL14	11	low14*	(S + A − P) >> 2

Table 6-22 Relocation Types *(Continued)*

Name	Value	Field	Calculation
R_PPC_REL14_BRTAKEN	12	low14*	(S + A – P) >> 2
R_PPC_REL14_BRNTAKEN	13	low14*	(S + A – P) >> 2
R_PPC_GOT16	14	half16*	G + A
R_PPC_GOT16_LO	15	half16	#lo(G + A)
R_PPC_GOT16_HI	16	half16	#hi(G + A)
R_PPC_GOT16_HA	17	half16	#ha(G + A)
R_PPC_PLTREL24	18	low24*	(L + A – P) >> 2
R_PPC_COPY	19	none	none
R_PPC_GLOB_DAT	20	word32	S + A
R_PPC_JMP_SLOT	21	none	see below
R_PPC_RELATIVE	22	word32	B + A
R_PPC_LOCAL24PC	23	low24*	see below
R_PPC_UADDR32	24	word32	S + A
R_PPC_UADDR16	25	half16*	S + A

Relocation types with special semantics are described below. Relocation values not in the above table and less than 101 or greater than 200 are reserved. Values in the range 101-200 and names beginning with "R_PPC_EMB_" have been assigned for embedded system use.

The relocation types whose Field column entry contains an asterisk are subject to failure if the value computed does not fit in the allocated bits.

The relocation types in which the names include _BRTAKEN or _BRNTAKEN specify whether the branch prediction bit (bit 10) should indicate that the branch will be taken or not taken, respectively. For an unconditional branch, the branch prediction bit must be 0.

R_PPC_GOT16*
: These relocation types resemble the corresponding R_PPC_ADDR16* types, except that they refer to the address of the symbol's global offset table entry and additionally instruct the link-editor to build a global offset table.

R_PPC_PLTREL24

This relocation type refers to the address of the symbol's procedure linkage table entry and additionally instructs the link-editor to build a procedure linkage table. There is an implicit assumption that the procedure linkage table for a module will be within +/- 32 megabytes of an instruction that branches to it, so that the `R_PPC_PLTREL24` relocation type is the only one needed for relocating branches to procedure linkage table entries.

`R_PPC_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 dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

`R_PPC_GLOB_DAT`
This relocation type resembles `R_PPC_ADDR32`, except that it sets a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

`R_PPC_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 dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address (see the section “Procedure Linkage Table” on page 221).

`R_PPC_LOCAL24PC`
This relocation type resembles `R_PPC_REL24`, except that it uses the value of the symbol within the object, not an interposed value, for S in its calculation. The symbol referenced in this relocation normally is `_GLOBAL_OFFSET_TABLE_`, which additionally instructs the link-editor to build the global offset table.

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

R_PPC_UADDR*
These relocation types are the same as the corresponding R_PPC_ADDR* types, except that the datum to be relocated is allowed to be unaligned.

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 118 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_Addr      vda_name;
        Elf32_Word      vda_next;
} Elf32_Verdaux;
```

`vd_version`
This member identifies the version of the structure itself.

*Table 6-23*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 6-24*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 42). A *weak* version definition has no symbols associated with it (see "Creating a Weak Version Definition" on page 122 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 224).

`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 6-25 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_Addr      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_Addr      vna_name;
        Elf32_Word      vna_next;
} Elf32_Vernaux;
```

`vn_version`
This member identifies the version of the structure itself.

Table 6-26 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 "DT_NEEDED" 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 224).

`vna_flags`
This member holds version dependency specific information.

Table 6-27 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 118 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.

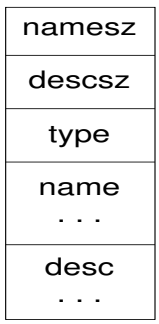


Figure 6-5 Note Information

`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.

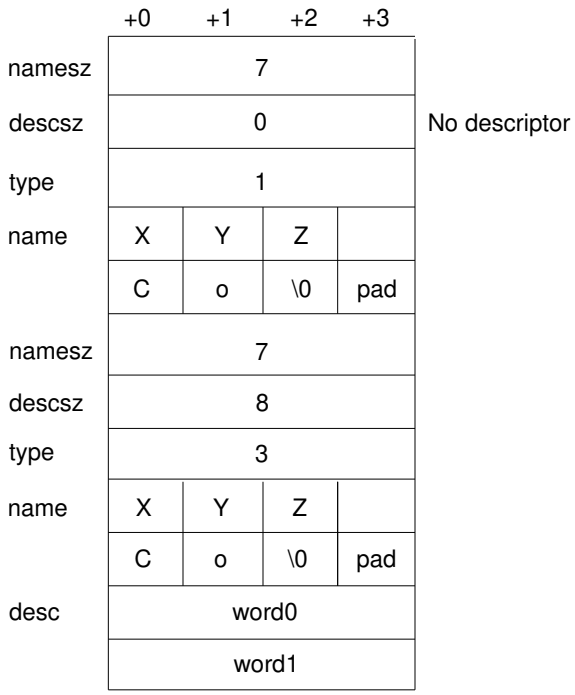


Figure 6-6 Example Note Segment

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" 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)" describes the information used to load a program into memory.
- "Runtime Linker" 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 194.

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 142 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 6-28 on page 191.

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 195 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 6-28 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 207 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 204 for further information.

`PT_NOTE`
The array element specifies the location and size of auxiliary information. See "Note Section" on page 187 below for details.

`PT_SHLIB`
This segment type is reserved but has unspecified semantics.

`PT_PHDR`
The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See "Program Interpreter" on page 204 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 195 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 6-29 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 6-30 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 “Special Sections” on page 157 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 207 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 213, "Procedure Linkage Table" on page 215, "Procedure Linkage Table" on page 218, and "Procedure Linkage Table" on page 221 for details.

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

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 and PowerPC 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.

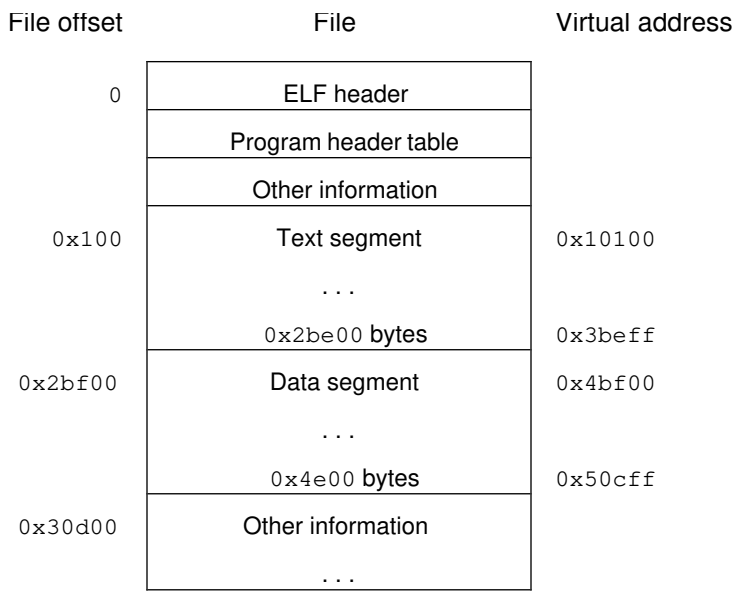


Figure 6-7 SPARC: Executable File (64 K alignment)

Table 6-31 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.

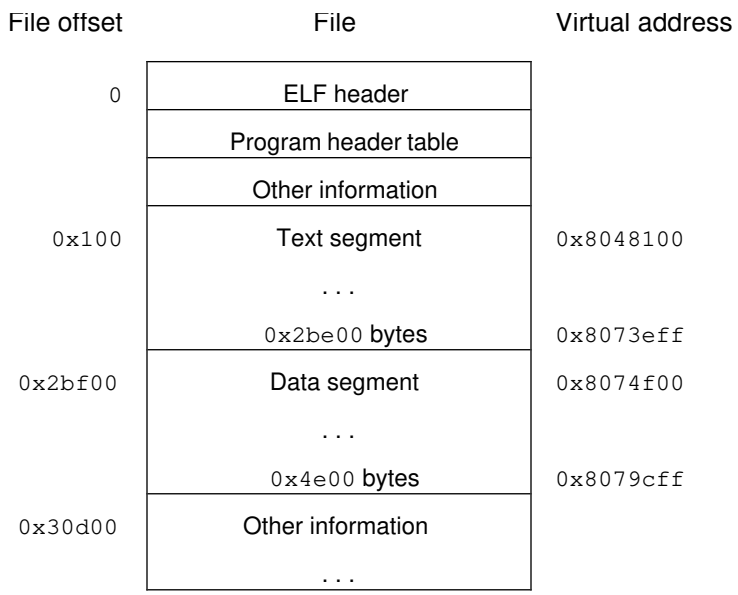


Figure 6-8 *x86:* Executable File (4 K alignment)

Table 6-32 *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

The following example presents the PowerPC version.

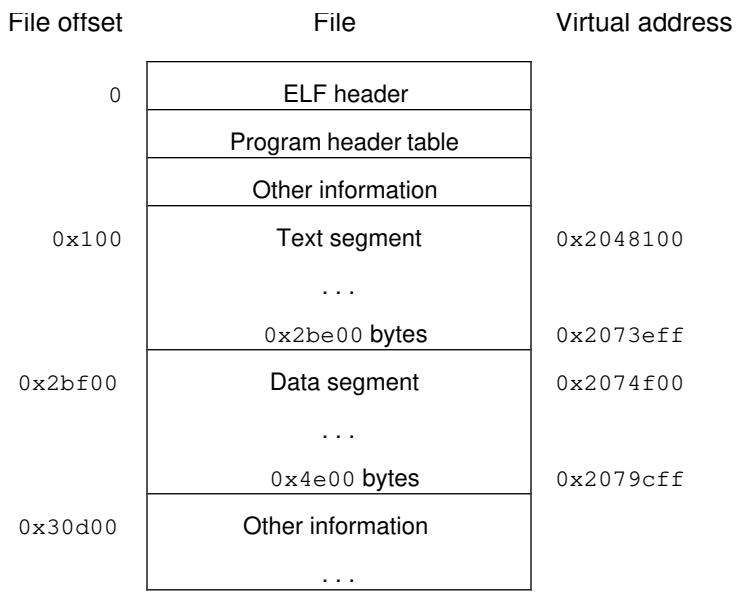


Figure 6-9 PowerPC: Executable File (4 K alignment)

Table 6-33 PowerPC: Program Header Segments (4 K alignment)

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x2048100	0x2074f00
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.

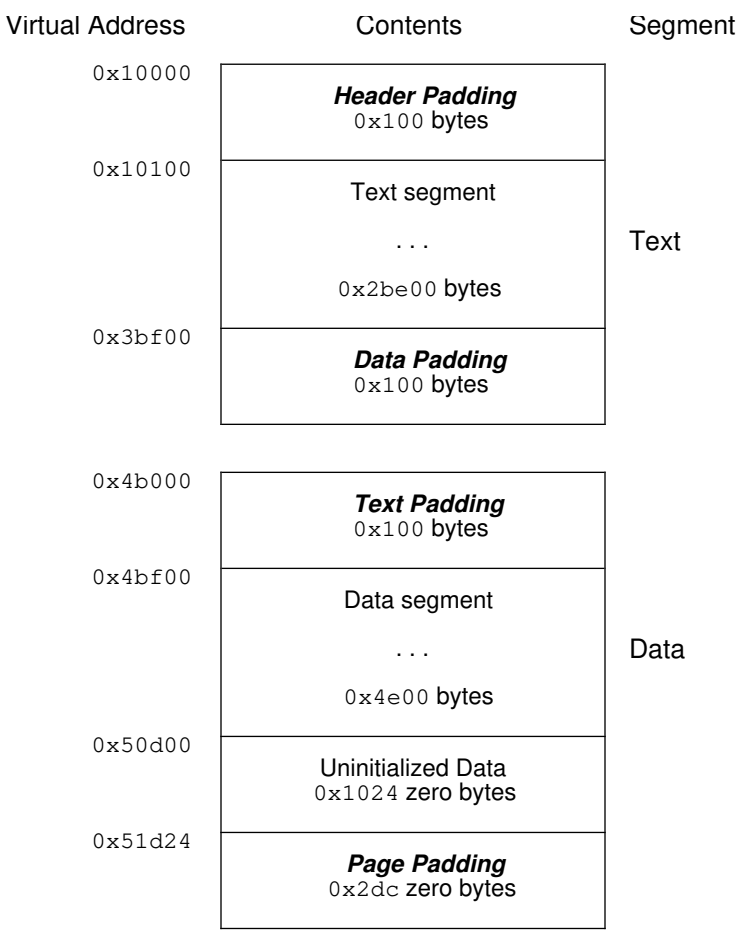


Figure 6-10 SPARC: Process Image Segments

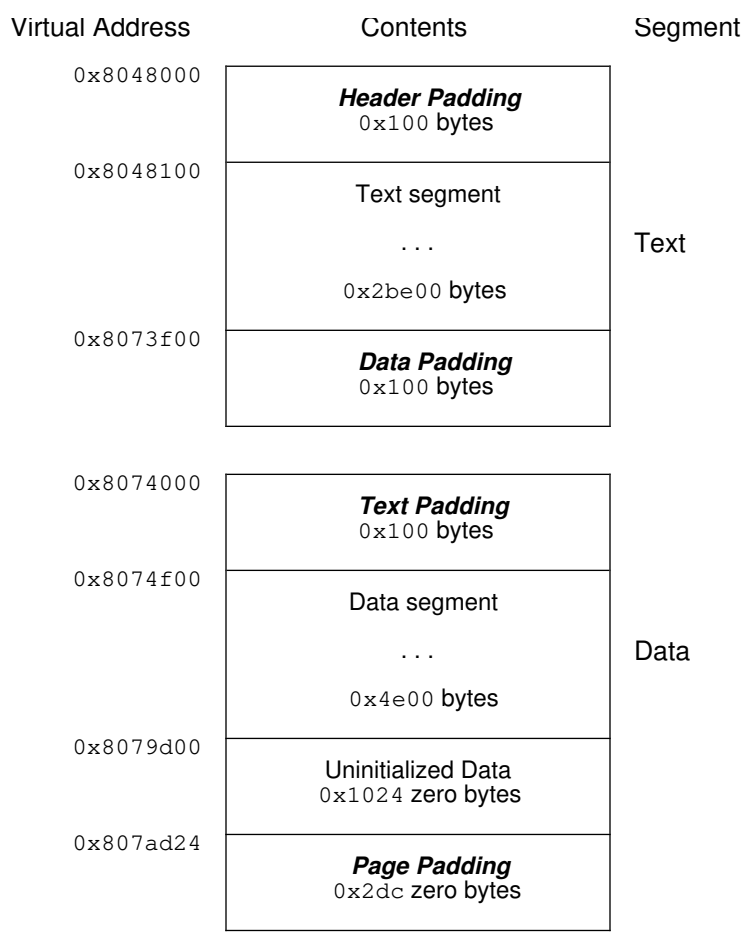


Figure 6-11 x86: Process Image Segments

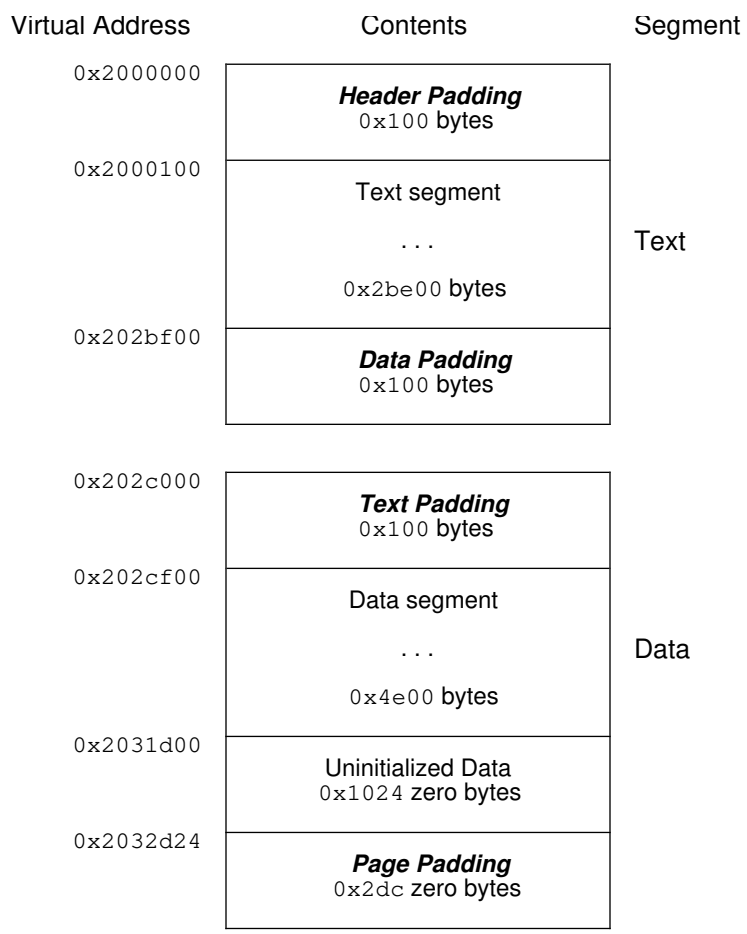


Figure 6-12 PowerPC: Process Image Segments

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. For the process to execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. (For background, see “Link-Editor” on page 7.) This lets a segment’s virtual address change from one process to another, without invalidating execution behavior.

Though the system chooses virtual addresses for individual processes, it maintains the segments’ *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file.

The following tables show possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

Table 6-34 Example SPARC Shared Object Segment Addresses

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

Table 6-35 Example x86 Shared Object Segment Addresses

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

Unlike the SPARC and x86, which load shared object segments in a single region, the PowerPC can load shared object segments in two regions. The PowerPC Application Binary Interface states that region 1 (0x10000 to USRTEXT) and region 2 (user_stack_limit to program_break_address) can be used for dynamic segments. Using the address space depicted by region 1 is an optimization that causes the relative addresses to fall within a 32M range, limited by the relative address branch instructions on PowerPC. The allocation is done in the following order:

Region 1: 0x10000 to USERTEXT

Allocation is from high end to low end.

Region 2: program_break_address to user_stack_limit

Allocation is from high end to low end.

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 225
- 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 195, 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" on page 215, "Procedure Linkage Table" on page 218, and "Procedure Linkage Table" on page 221 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 6-36 Dynamic Array Tags, d_tag *(1 of 2)*

Name	Value	d_un	Executable	Shared Object
DT_NULL	0	Ignored	Mandatory	Mandatory
DT_NEEDED	1	d_val	Optional	Optional
DT_PLTRELSZ	2	d_val	Optional	Optional
DT_PLTGOT	3	d_ptr	Optional	Optional
DT_HASH	4	d_ptr	Mandatory	Mandatory
DT_STRTAB	5	d_ptr	Mandatory	Mandatory
DT_SYMTAB	6	d_ptr	Mandatory	Mandatory
DT_RELA	7	d_ptr	Mandatory	Optional
DT_RELASZ	8	d_val	Mandatory	Optional
DT_RELAENT	9	d_val	Mandatory	Optional
DT_STRSZ	10	d_val	Mandatory	Mandatory
DT_SYMENT	11	d_val	Mandatory	Mandatory
DT_INIT	12	d_ptr	Optional	Optional
DT_FINI	13	d_ptr	Optional	Optional
DT_SONAME	14	d_val	Ignored	Optional
DT_RPATH	15	d_val	Optional	Ignored
DT_SYMBOLIC	16	Ignored	Ignored	Optional
DT_REL	17	d_ptr	Mandatory	Optional
DT_RELSZ	18	d_val	Mandatory	Optional
DT_RELENT	19	d_val	Mandatory	Optional
DT_PLTREL	20	d_val	Optional	Optional
DT_DEBUG	21	d_ptr	Optional	Ignored
DT_TEXTREL	22	Ignored	Optional	Optional
DT_JMPREL	23	d_ptr	Optional	Optional

Table 6-36 Dynamic Array Tags, d_tag *(2 of 2)*

Name	Value	d_un	Executable	Shared Object
DT_VERDEF	0x6ffffffc	d_ptr	Optional	Optional
DT_VERDEFNUM	0x6ffffffd	d_val	Optional	Optional
DT_VERNEED	0x6ffffffe	d_ptr	Optional	Optional
DT_VERNEEDNUM	0x6fffffff	d_val	Optional	Optional
DT_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 213 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 224. 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 225 later.

DT_FINI
This element holds the address of the termination function, discussed in "Initialization and Termination Functions" on page 225 later.

DT_SONAME
This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. See Section , "Shared Object Dependencies," on page 213 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 89. The offset is an index into the table recorded in the DT_STRTAB entry.

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

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

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

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

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

DT_DEBUG
This member is used for debugging.

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

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

DT_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 181 for 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 185 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.

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

Shared Object Dependencies

When the runtime linker creates the memory segments for an object file, the dependencies (recorded in DT_NEEDED entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the runtime linker builds a complete process image.

When resolving symbolic references, the runtime linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the DT_NEEDED entries (in order), then at the second level DT_NEEDED entries, and so on.

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

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

Global Offset Table (Processor-Specific)

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-

independence and shareability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries (see "Relocation" on page 167 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), `R_386_GLOB_DAT` (for x86), or `R_PPC_GLOB_DAT` (for PowerPC) 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, x86, and PowerPC 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.

SPARC: Procedure Linkage Table

As the global offset table converts position-independent address calculations to absolute locations, the procedure linkage table converts position-independent function calls to absolute locations. The link-editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. So, the link-editor puts the program transfer control to entries in the procedure linkage table.

On SPARC architectures, procedure linkage tables reside in private data. The runtime linker determines the destinations' absolute addresses and modifies the 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.

Code Example 6-1 *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 `jmp1` instruction, and then patching the `ba,a` to be `sethi`. If a reentrant function call happens between the two word updates, the `jmp1`

resides in the delay slot of the `ba,a` instruction, and cancels the delay instruction. So, the runtime linker gains control a second time. Although both invocations of the runtime linker modify the same procedure linkage table entry, their changes do not interfere with each other.

- The first `sethi` instruction of a procedure linkage table entry can fill the delay slot of the previous entry's `jmp1` instruction. Although the `sethi` changes the value of the `%g1` register, the previous contents can be safely discarded.
- After conversion, the last procedure linkage table entry (`.PLT102` above) needs a delay instruction for its `jmp1`. The required, trailing `nop` fills this delay slot.

The `LD_BIND_NOW` environment variable changes dynamic linking behavior. If its value is non-null, the runtime linker processes `R_SPARC_JMP_SLOT` relocation entries (procedure linkage table entries) before transferring control to the program. If `LD_BIND_NOW` is null, the runtime linker evaluates linkage table entries on the first execution of each table entry.

x86: Procedure Linkage Table

As for SPARC, the procedure linkage table redirects position-independent function calls to absolute locations. The link-editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. So, the link-editor has the program transfer control to entries in the procedure linkage table.

On x86 architectures, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The runtime linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The runtime linker thus redirects the entries without compromising the position-independence and shareability of the program's text. Executable files and shared object files have separate procedure linkage tables.

```
.PLT0:      pushl       got_plus_4
            jmp         *got_plus_8
            nop; nop
            nop; nop
.PLT1:      jmp         *name1_in_GOT
            pushl       $offset
            jmp         .PLT0@PC
.PLT2:      jmp         *name2_in_GOT
            pushl       $offset
            jmp         .PLT0@PC
            ...
```

```
.PLT0:      pushl       4(%ebx)
            jmp         *8(%ebx)
            nop; nop
            nop; nop
.PLT1:      jmp         *name1@GOT(%ebx)
            pushl       $offset
            jmp         .PLT0@PC
.PLT2:      jmp         *name2@GOT(%ebx)
            pushl       $offset
            jmp         .PLT0@PC
            ...
```

Code Example 6-2 *x86:* Procedure Linkage Table Example

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.

PowerPC: Procedure Linkage Table

As for SPARC and Intel, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. So, the link editor has the program transfer control to entries in the procedure linkage table.

For the PowerPC, the procedure linkage table (the `.plt` section) is not initialized in the executable or shared object file. Instead, the link editor simply reserves space for it and the dynamic linker initializes it and manages it according to its own, possibly implementation-dependent needs, subject to the following constraints:

- The first 18 words (72 bytes) of the procedure linkage table are reserved for use by the dynamic linker. There shall be no branches from the executable or shared object into these first 18 words.
- If the executable or shared object requires `N` procedure linkage table entries, the link editor reserves `3*N` words (`12*N` bytes) following the 18 reserved words. The first `2*N` of these words are the procedure linkage table entries themselves. The link-editor directs calls to bytes (`72 + (i-1)*8`), for `i` between `1` and `N` inclusive. The remaining `N` words (`4*N` bytes) are reserved for use by the dynamic linker.

As mentioned before, a relocation table is associated with the procedure linkage table. The `DT_JMPREL` entry in the `_DYNAMIC` array gives the location of the first relocation entry. The relocation table's entries parallel the procedure linkage table entries in a one-to-one correspondence. That is, relocation table entry 1 applies to procedure linkage table entry 1, and so on. The relocation type for each entry shall be `R_PPC_JMP_SLOT`, the relocation offset shall specify the address of the first byte of the associated procedure linkage table entry, and the symbol table index shall reference the appropriate symbol.

To illustrate procedure linkage tables, “Procedure Linkage Table Example.” on page 222 shows how the dynamic linker might initialize the procedure linkage table when loading the executable or shared object.

Code Example 6-3 *PowerPC*Procedure Linkage Table Example.

```
    .word   0   # tag word - no register saving
PLTresolve:
    addis   r12,r0,dynamic_linker@ha
    addi    r12,r12,dynamic_linker@l
    mtctr   r12
    addis   r12,r0,symtab_addr@ha
    addi    r12,r12,symtab_addr@l
    bctr
.PLTcall:
    addis   r11,r11,.PLTtable@ha
    lwz     r11,.PLTtable@lo(r11
    mtctr   r11)
    bctr
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
.PLT1:
    addi    r11,r0,4*0
    b       .PLTresolve
                . . .
.PLTi:
    addi    r11,4*i
    b       .PLTresolve
                . . .
.PLTN:
    addi    r11,4*(N-1)
    b       .PLTresolve

.PLTtable:
    <N word table begins here>
```

Following the steps below, the dynamic linker and the program cooperate to resolve symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the dynamic linker is not specified.

1. As shown above, all procedure linkage table entries initially transfer to `.PLTresolve`, allowing the dynamic linker to gain control at the first execution of each table entry. For illustration, assume the program calls *name*, which transfers control to the label `.PLTi`. The procedure linkage table entry loads into `r11` four times the index of the relocation entry for `.PLTi` and branches to `.PLTresolve`, which then calls into the dynamic linker with a pointer to the symbol table for the object in `r12`.

2. The dynamic linker finds relocation entry `i` corresponding to the index in `r11`. It will have type `R_PPC_JMP_SLOT`, its offset will specify the address of `.PLTi`, and its symbol table index will reference *name*.

3. Knowing this, the dynamic linker finds the symbol's "real" value. It then modifies the code at `.PLTi` in one of two ways. If the target symbol is reachable from `.PLTi` by a branch instruction, it overwrites the "`li r11,4*i`" instruction at `.PLTi` with a branch to the target. On the other hand, if the target symbol is not reachable from `.PLTi`, the dynamic linker loads the target address into word `.PLTtable+4*(i-1)` and overwrites the "`b .PLTresolve`" with a "`b .PLTcall`".

4. Subsequent executions of the procedure linkage table entry will transfer control directly to the function, either directly or by using `.PLTcall`, without invoking the dynamic linker.

For PLT indexes greater than or equal to `2^14`, only the even indexes shall be used and four words shall be allocated for each entry. If the above scheme is used, this allows four instructions for loading the index and branching to `.PLTresolve` or `.PLTcall`, instead of only two.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker resolves the function call binding at load time, before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_PPC_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a 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 6-13 on page 156). Labels appear below to help explain the hash table organization, but they are not part of the specification.

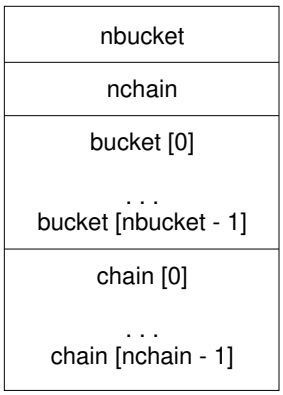


Figure 6-13 Symbol Hash Table

The `bucket` array contains `nbucket` entries, and the `chain` array contains `nchain` entries; indexes start at 0. Both `bucket` and `chain` hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`; so, symbol table indexes also select chain table entries.

A hashing function accepts a symbol name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value *x* for some name, `bucket [x%nbucket]` gives an index y into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[y]` gives the next symbol table entry with the same hash value.

One can follow the `chain` links until either the selected symbol table entry holds the desired name or the `chain` entry contains the value `STN_UNDEF`.

```
unsigned long
elf_Hash(const unsigned char *name)
{
    unsigned long h = 0, g;

    while (*name)
    {
        h = (h << 4) + *name++;
        if (g = h & 0xf0000000)
            h ^= g >> 24;
                h &= ~g;
    }
    return h;
```

Initialization and Termination Functions

After the runtime linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. These initialization functions are called in the reverse of the order at which they are encountered.

Similarly, shared objects may have termination functions, which are executed with the `atexit(3C)` mechanism after the base process begins its termination sequence. Refer to `atexit(3C)` for more information. These termination functions are called in the order they are encountered.

Shared objects designate their initialization and termination functions through the `DT_INIT` and `DT_FINI` entries in the dynamic structure, described in "Dynamic Section" above. Typically, the code for these functions resides in the *.init* and *.fini* sections, mentioned in "Sections" on page 148 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.

6