System V Application Binary Interface - DRAFT - 10 June 2013

Contents

Revision History

First draft published May 14, 1998.

Second draft published May 3, 1999.

- New values introduced for ELF header e_machine field.
- Revised language for EI_OSABI and EI_ABIVERSION fields of the ELF header e_ident array.
- New section flags SHF_MERGE and SHF_STRINGS added.
- New values added to a symbol table entry's st_other field to describe a symbol's visibility.
- New dynamic section tags DT_RUNPATH and DT_FLAGS added. Dynamic section tag DT_RPATH moved to level 2.
- New semantics for shared object path searching, including new ``Substitution Sequences''.

Third draft published May 12, 1999.

- A new symbol type, STT_COMMON, has been added.
- Added language restricting the types of objects that may contain symbols with the section index SHN_COMMON.
- Dynamic section entries DT_SYMBOLIC, DT_TEXTREL and DT_BIND_NOW have been moved to level 2. New DT_FLAGS values DF_SYMBOLIC, DF_TEXTREL and DF_BIND_NOW have been added as replacements.
- New rules for interpreting dynamic section tag encodings have been added.
- The OS and processor specific ranges for DT_FLAGS have been removed.
- The language motivating the use of DF_ORIGIN has been changed.

Fourth draft published July 6, 1999.

- New language has been added warning about the use of WEAK symbols in application programs.
- New rules have been defined for composition of consecutive relocation entries that reference the same location.
- Language has been added clarifying the order of execution for functions specified by initialization and termination arrays.

Fifth draft published July 21, 1999.

- New section types and section names added for init arrays, fini arrays and pre-init arrays.
- An object may now have both DT_INIT and DT_INIT_ARRAY entries (and

both DT_FINI and DT_FINI_ARRAY entries). The relative execution order is specified.
- The language describing the order of execution for termination functions has been revised.
- A new pre-initialization mechanism has been added.
- It is now up to the processor supplement for each processor to specify whether the dynamic linker must invoke the executable file's init and fini routines.

Sixth draft published September 14, 1999.

- Changed the numbering of some new section types previously added to account for type numbers already in use in particular vendor implementations.
- Increased the number of section flag bits available in the OS specific range.

Seventh draft published October 4, 1999.

- Changed the values used for some new section attribute flags to accommodate platforms already using previously assigned values.
- Added new section attribute flags SHF_INFO_LINK, SHF_LINK_ORDER and SHF_OS_NONCONFORMING
- Added rules for linkers when linking sections with unrecognized types or flags.

Eighth draft published March 30, 2000.

- Added the concept of section groups.
- Removed the macros for ELF32_ST_OTHER and ELF64_ST_OTHER.

Ninth draft published March 30, 2000.

- Added language clarifying the semantics of symbols marked as STV_PROTECTED.
- Added language clarifying the contents of the initialization and termination arrays.

Tenth draft published 22 June 2000.

- Added a sentence spelling out the behavior when resolving a symbol to a STV_PROTECTED definition from a shared object.
- Added support for more than 65,000 sections in the ELF header, and with SHT_SYMTAB_SHNDX sections, and in symbol tables.

Eleventh draft published 24 April 2001.

- Updated table of EM_* entries.
- Added GRP_MASKOS and GRP_MASKPROC. Changed section group

description in a few ways, clarifying some fuzzy points and rewriting the rules for symbols referencing into section groups.
- Changed the warning about using weak to be stronger.
- Reworded the EI_OSABI byte description to make its use clearer.
- Added the table of now generic EI_OSABI values.
- Added SHF_TLS, PT_TLS and its contents, DF_STATIC_TLS, STT_TLS, .tbss, and .tdata.
- Changed the rules for SHT_SYMTAB_SHNDX contents to require 0 when the corresponding st_shndx field is not SHN_XINDEX.

Twelfth draft published 26 March 2007.

- Updated table of EM_* entries.

Thirteenth draft published 03 November 2009.

- Updated table of EM_* entries.
- Added ELFOSABI_FENIXOS to the EI_OSABI values.
- Added ELFOSABI_GNU to the EI_OSABI values; aliased to ELFOSABI_LINUX.

Fourteenth draft published 10 June 2013.

- Added SHF_COMPRESSED to the Section Attribute Flags.
- Updated table of EM_* entries.

Recent Changes

- Clarified the description of SHT_SYMTAB_SHNDX; allow usage with any symbol table section.
- Added DT_SYMTAB_SHNDX to the Dynamic Array Tags.

Contents

Introduction

This chapter describes the object file format, called ELF (Executable and Linking Format). There are three main types of object files.

- A *relocatable file* holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable file* holds a program suitable for execution; the file specifies how exec(BA_OS) creates a program's process image.
- A *shared object file* holds code and data suitable for linking in two contexts. First, the link editor [see ld(BA_OS)] processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Created by the assembler and link editor, object files are binary representations of programs intended to be executed directly on a processor. Programs that require other abstract machines, such as shell scripts, are excluded.

After the introductory material, this chapter focuses on the file format and how it pertains to building programs. Chapter 5 also describes parts of the object file, concentrating on the information necessary to execute a program.

File Format

Object files participate in program linking (building a program) and program execution (running a program). For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of those activities. Figure 4-1 shows an object file's organization.

Figure 4-1: Object File Format

Linking View	Execution View
ELF Header	ELF Header
Program header table *optional*	Program header table *required*
Section 1	Segment 1
...	Segment 2
Section n	Segment 3

...	...
Section header table *required*	Section header table *optional*

An *ELF header* resides at the beginning and holds a ``road map'' describing the file's organization. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of special sections appear later in the chapter. Chapter 5 discusses *segments* and the program execution view of the file.

A *program header table* tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. 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 on. Files used during linking must have a section header table; other object files may or may not have one.

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 either 32-bit or 64-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.

Figure 4-2: 32-Bit Data Types

Name

Size

Alignment

Purpose

Elf32_Addr

4

4

Unsigned program address

Elf32_Off

4

4

Unsigned file offset

Elf32_Half

2

2

Unsigned medium integer

Elf32_Word

4

4

Unsigned integer

Elf32_Sword

4

4

Signed integer

unsigned char

1

1

Unsigned small integer

64-Bit Data Types

Name

Size

Alignment

Purpose

Elf64_Addr

8

8

Unsigned program address

Elf64_Off

8

8

Unsigned file offset

Elf64_Half

2

2

Unsigned medium integer

Elf64_Word

4

4

Unsigned integer

Elf64_Sword

4

4

Signed integer

Elf64_Xword

8

8

Unsigned long integer

Elf64_Sxword

8

8

Signed long integer

unsigned char

1

1

Unsigned small integer

All data structures that the object file format defines follow the ``natural'' size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 8-byte alignment for 8-byte objects, 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4 or 8, 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.

For portability reasons, 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 when and if extensions are defined.

Figure 4-3: ELF Header

```
#define EI_NIDENT 16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
```

} Elf64_Ehdr;

---

e_ident
: The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear below in ``ELF Identification''.

e_type
: This member identifies the object file type.

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

e_version
: This member identifies the object file version.

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*.

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.

  If the number of sections is greater than or equal to SHN_LORESERVE (0xff00), this member has the value zero and the actual number of section header table entries is contained in the sh_size field of the section header at index 0. (Otherwise, the sh_size member of the initial entry contains 0.)

e_shstrndx

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'' and ``String Table'' below for more information.

If the section name string table section index is greater than or equal to SHN_LORESERVE (0xff00), this member has the value SHN_XINDEX (0xffff) and the actual index of the section name string table section is contained in the sh_link field of the section header at index 0. (Otherwise, the sh_link member of the initial entry contains 0.)

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.

Figure 4-4: e_ident[] **Identification Indexes**

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_OSABI	7	Operating system/ABI identification
EI_ABIVERSION	8	ABI version
EI_PAD	9	Start of padding bytes
EI_NIDENT	16	Size of e_ident[]

These indexes access bytes that hold the following values.

EI_MAG0 to EI_MAG3

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

EI_CLASS

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

EI_DATA

Byte e_ident[EI_DATA] specifies the encoding of both the data structures used by object file container and data contained in object file sections. The following encodings are currently defined.

EI_VERSION

Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.

EI_OSABI

Byte e_ident[EI_OSABI] identifies the OS- or ABI-specific ELF extensions used by this file. Some fields in other ELF structures have flags and values that have operating system and/or ABI specific meanings; the interpretation of those fields is determined by the value of this byte. If the object file does not use any extensions, it is recommended that this byte be set to 0. If the value for this byte is 64 through 255, its meaning depends on the value of the e_machine header member. The ABI processor supplement for an architecture can define its own associated set of values for this byte in this range. If the processor supplement does not specify a set of values, one of the following values shall be used, where 0 can also be taken to mean *unspecified*.

Name	Value	Meaning
ELFOSABI_NONE	0	No extensions or unspecified
ELFOSABI_HPUX	1	Hewlett-Packard HP-UX
ELFOSABI_NETBSD	2	NetBSD
ELFOSABI_GNU	3	GNU
ELFOSABI_LINUX	3	Linux *historical - alias for ELFOSABI_GNU*
ELFOSABI_SOLARIS	6	Sun Solaris
ELFOSABI_AIX	7	AIX
ELFOSABI_IRIX	8	IRIX
ELFOSABI_FREEBSD	9	FreeBSD
ELFOSABI_TRU64	10	Compaq TRU64 UNIX
ELFOSABI_MODESTO	11	Novell Modesto
ELFOSABI_OPENBSD	12	Open BSD
ELFOSABI_OPENVMS	13	Open VMS
ELFOSABI_NSK	14	Hewlett-Packard Non-Stop Kernel
ELFOSABI_AROS	15	Amiga Research OS
		The FenixOS highly

ELFOSABI_FENIXOS	16	scalable multi-core OS
ELFOSABI_CLOUDABI	17	Nuxi CloudABI
ELFOSABI_OPENVOS	18	Stratus Technologies OpenVOS
	64-255	Architecture-specific value range

EI_ABIVERSION
: Byte e_ident[EI_ABIVERSION] identifies the version of the ABI to which the object is targeted. This field is used to distinguish among incompatible versions of an ABI. The interpretation of this version number is dependent on the ABI identified by the EI_OSABI field. If no values are specified for the EI_OSABI field by the processor supplement or no version values are specified for the ABI determined by a particular value of the EI_OSABI byte, the value 0 shall be used for the EI_ABIVERSION byte; it indicates *unspecified*.

EI_PAD
: This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and set to zero; programs that read object files should ignore them. 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. Class ELFCLASS32 files use objects that occupy 1, 2, and 4 bytes. Class ELFCLASS64 files use objects that occupy 1, 2, 4, and 8 bytes. Under the defined encodings, objects are represented as shown below.

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

Figure 4-5: Data Encoding ELFDATA2LSB, byte address zero on the left

01

0x01

02

01

0x0102

04

03

02

01

0x01020304

08

07

06

05

04

03

02

01

0x0102030405060708

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

Figure 4-6: Data Encoding ELFDATA2MSB, byte address zero on the left

01

0x01

01

02

0x0102

01

02

03

04

0x01020304

01

02

03

04

05

06

07

08

0x0102030405060708

Machine Information (Processor-Specific)

This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Sections

An object file's section header table lets one locate all the file's sections. The section header table is an array of Elf32_Shdr or Elf64_Shdr structures as described below. A section header table index is a subscript into this array. The ELF header's e_shoff member gives the byte offset from the beginning of the file to the section header table. e_shnum normally tells how many entries the section header table contains. e_shentsize gives the size in bytes of each entry.

If the number of sections is greater than or equal to SHN_LORESERVE (0xff00), e_shnum has the value SHN_UNDEF (0) and the actual number of section header table entries is contained in the sh_size field of the section header at index 0 (otherwise, the sh_size member of the initial entry contains 0).

Some section header table indexes are reserved in contexts where index size is restricted, for example, the st_shndx member of a symbol table entry and the e_shnum and e_shstrndx members of the ELF header. In such contexts, the reserved values do not represent actual sections in the object file. Also in such contexts, an escape value indicates that the actual section index is to be found elsewhere, in a larger field.

Figure 4-7: Special Section Indexes

Name	**Value**
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_LOOS	0xff20
SHN_HIOS	

0xff3f

SHN_ABS

0xfff1

SHN_COMMON

0xfff2

SHN_XINDEX

0xffff

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.

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

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

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

SHN_LOOS through SHN_HIOS
: Values in this inclusive range are reserved for operating system-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.

SHN_XINDEX
: This value is an escape value. It indicates that the actual section header index is too large to fit in the containing field and is to be found in another location (specific to the structure where it appears).

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. 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.

Figure 4-8: Section Header

```
typedef struct {
   Elf32_Word  sh_name;
   Elf32_Word  sh_type;
   Elf32_Word  sh_flags;
   Elf32_Addr  sh_addr;
   Elf32_Off   sh_offset;
   Elf32_Word  sh_size;
   Elf32_Word  sh_link;
   Elf32_Word  sh_info;
   Elf32_Word  sh_addralign;
   Elf32_Word  sh_entsize;
} Elf32_Shdr;

typedef struct {
   Elf64_Word  sh_name;
   Elf64_Word  sh_type;
   Elf64_Xword sh_flags;
   Elf64_Addr  sh_addr;
   Elf64_Off   sh_offset;
   Elf64_Xword sh_size;
   Elf64_Word  sh_link;
   Elf64_Word  sh_info;
   Elf64_Xword sh_addralign;
   Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

---

sh_name

This member specifies the name of the section. Its value is an index into the section header string table section [see ``String Table'' below], giving the location of a null-terminated string.

sh_type

This member categorizes the section's contents and semantics. Section types and their descriptions appear below.

sh_flags

Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions appear below.

sh_addr

If the section will 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's value gives the byte offset from the beginning of the file to the first byte in the section. One 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 non-zero 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. A table below describes the values.

sh_info

This member holds extra information, whose interpretation depends on the section type. A table below describes the values. If the sh_flags field for this section header includes the attribute SHF_INFO_LINK, then this member represents a section header table index.

sh_addralign

Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. 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.

Figure 4-9: Section Types,sh_type

Name

Value

SHT_NULL

0

SHT_PROGBITS

1

SHT_SYMTAB

2

SHT_STRTAB

3

SHT_RELA

4

SHT_HASH

5

SHT_DYNAMIC

6

SHT_NOTE

7

SHT_NOBITS

8

SHT_REL

9

SHT_SHLIB

10

SHT_DYNSYM

11

SHT_INIT_ARRAY

14

SHT_FINI_ARRAY

15

SHT_PREINIT_ARRAY

16

SHT_GROUP

17

SHT_SYMTAB_SHNDX

18

SHT_LOOS

0x60000000

SHT_HIOS

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 and SHT_DYNSYM

These sections hold a symbol table. Currently, an object file may have only one section of each type, but this restriction may be relaxed in the future. Typically, SHT_SYMTAB provides symbols for link editing, though it may also be used for dynamic linking. 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'' below for details.

SHT_STRTAB

The section holds a string table. An object file may have multiple string table sections. See ``String Table'' below 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 or type Elf64_Rela for the 64-bit class of object files. An object file may have multiple relocation sections. ``Relocation'' below for details.

SHT_HASH
: The section holds 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'' in the Chapter 5 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'' in Chapter 5 for details.

SHT_NOTE
: The section holds information that marks the file in some way. See ``Note Section'' in Chapter 5 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 or type Elf64_Rel for the 64-bit class of object files. An object file may have multiple relocation sections. See ``Relocation'' below for details.

SHT_SHLIB
: This section type is reserved but has unspecified semantics.

SHT_INIT_ARRAY
: This section contains an array of pointers to initialization functions, as described in ``Initialization and Termination Functions'' in Chapter 5. Each pointer in the array is taken as a parameterless procedure with a void return.

SHT_FINI_ARRAY
: This section contains an array of pointers to termination functions, as described in ``Initialization and Termination Functions'' in Chapter 5. Each pointer in the array is taken as a parameterless procedure with a void return.

SHT_PREINIT_ARRAY
: This section contains an array of pointers to functions that are invoked before all other initialization functions, as described in ``Initialization and Termination Functions'' in Chapter 5. Each pointer in the array is taken as a parameterless procedure with a void return.

SHT_GROUP
: This section defines a section group. A section group is a set of sections that are related and that must be treated specially by the linker (see below for further details). Sections of type SHT_GROUP may appear only in relocatable objects (objects with the ELF header e_type member set to ET_REL). The section header table entry for a group section must appear in the section header table before the entries for any of the sections that are members of the group.

SHT_SYMTAB_SHNDX
: This section is associated with a symbol table section and is required if any of the section header indexes referenced by that symbol table contain the escape value SHN_XINDEX. The section is an array of Elf32_Word values. Each value corresponds one to one with a symbol table entry and appear in the same order as those entries. The values represent the section header indexes against which the symbol table entries are defined. Only if the corresponding symbol table entry's st_shndx field contains the escape value SHN_XINDEX will the matching Elf32_Word hold the actual section header index; otherwise, the entry must be SHN_UNDEF (0).

SHT_LOOS through SHT_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

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

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

SHT_HIUSER
: This value specifies the upper bound of the range of indexes reserved for application programs. Section types between SHT_LOUSER and SHT_HIUSER may be used by the application, without conflicting with current or future system-defined section types.

Other section type values are reserved. As mentioned before, the section header for index 0 (SHN_UNDEF) exists, even though the index marks undefined section references. This entry holds the following.

Figure 4-10: 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 offset

sh_size

Unspecified

If non-zero, the actual number of section header entries

sh_link

Unspecified

If non-zero, the index of the section header string table section

sh_info

0

No auxiliary information

sh_addralign

0

No alignment

sh_entsize

0

No entries

A section header's sh_flags member holds 1-bit flags that describe the section's attributes. Defined values appear in the following table; other values are reserved.

Figure 4-11: Section Attribute Flags

Name

Value

SHF_WRITE

0x1

SHF_ALLOC

0x2

SHF_EXECINSTR

0x4

SHF_MERGE

0x10

SHF_STRINGS

0x20

SHF_INFO_LINK

0x40

SHF_LINK_ORDER

0x80

SHF_OS_NONCONFORMING

0x100

SHF_GROUP

0x200

SHF_TLS

0x400

SHF_COMPRESSED

0x800

SHF_MASKOS

0x0ff00000

SHF_MASKPROC

0xf0000000

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

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

SHF_ALLOC
: The section occupies memory during process execution. Some control

sections do not reside in the memory image of an object file; this attribute is off for those sections.

SHF_EXECINSTR
: The section contains executable machine instructions.

SHF_MERGE

The data in the section may be merged to eliminate duplication. Unless the SHF_STRINGS flag is also set, the data elements in the section are of a uniform size. The size of each element is specified in the section header's sh_entsize field. If the SHF_STRINGS flag is also set, the data elements consist of null-terminated character strings. The size of each character is specified in the section header's sh_entsize field.

Each element in the section is compared against other elements in sections with the same name, type and flags. Elements that would have identical values at program run-time may be merged. Relocations referencing elements of such sections must be resolved to the merged locations of the referenced values. Note that any relocatable values, including values that would result in run-time relocations, must be analyzed to determine whether the run-time values would actually be identical. An ABI-conforming object file may not depend on specific elements being merged, and an ABI-conforming link editor may choose not to merge specific elements.

SHF_STRINGS
: The data elements in the section consist of null-terminated character strings. The size of each character is specified in the section header's sh_entsize field.

SHF_INFO_LINK
: The sh_info field of this section header holds a section header table index.

SHF_LINK_ORDER
: This flag adds special ordering requirements for link editors. The requirements apply if the sh_link field of this section's header references another section (the linked-to section). If this section is combined with other sections in the output file, it must appear in the same relative order with respect to those sections, as the linked-to section appears with respect to sections the linked-to section is combined with.

A typical use of this flag is to build a table that references text or data sections in address order.

SHF_OS_NONCONFORMING
: This section requires special OS-specific processing (beyond the standard linking rules) to avoid incorrect behavior. If this section has either an sh_type value or contains sh_flags bits in the OS-specific ranges for those fields, and a link editor processing this section does not recognize those values, then the link editor should reject the object file containing this section with an error.

SHF_GROUP
: This section is a member (perhaps the only one) of a section group. The section must be referenced by a section of type SHT_GROUP. The SHF_GROUP flag may be set only for sections contained in relocatable objects (objects with the ELF header e_type member set to ET_REL). See below for further details.

SHF_TLS
: This section holds *Thread-Local Storage*, meaning that each separate execution flow has its own distinct instance of this data. Implementations need not support this flag.

SHF_COMPRESSED
: This flag identifies a section containing compressed data. SHF_COMPRESSED applies only to non-allocable sections, and cannot be used in conjunction with SHF_ALLOC. In addition, SHF_COMPRESSED cannot be applied to sections of type SHT_NOBITS.

  All relocations to a compressed section specifiy offsets to the uncompressed section data. It is therefore necessary to decompress the section data before relocations can be applied. Each compressed section specifies the algorithm independently. It is permissible for different sections in a given ELF object to employ different compression algorithms.

  Compressed sections begin with a compression header structure that identifies the compression algorithm.

Figure 4-12: Compression Header

```
typedef struct {
   Elf32_Word  ch_type;
   Elf32_Word  ch_size;
   Elf32_Word  ch_addralign;
} Elf32_Chdr;

typedef struct {
   Elf64_Word  ch_type;
   Elf64_Word  ch_reserved;
   Elf64_Xword ch_size;
   Elf64_Xword ch_addralign;
} Elf64_Chdr;
```

---

ch_type
: This member specifies the compression algorithm. Supported algorithms and their descriptions are listed in the ELF Compression Types table below.

ch_size
: This member provides the size in bytes of the uncompressed data. See sh_size.

ch_addralign
: Specifies the required alignment for the uncompressed data. See sh_addralign.

The sh_size and sh_addralign fields of the section header for a compressed section reflect the requirements of the compressed section. The ch_size and ch_addralign fields in the compression header provide the corresponding values for the uncompressed data, thereby supplying the values that sh_size and sh_addralign would have had if the section had not been compressed.

The layout and interpretation of the data that follows the compression header is specific to each algorithm, and is defined below for each value of ch_type. This area may contain algorithm specific parameters and alignment padding in addition to compressed data bytes.

A compression header's ch_type member specifies the compression algoritm employed, as shown in the following table.

Figure 4-13: ELF Compression Types, ch_type

Name

ELFCOMPRESS_ZLIB
: The section data is compressed with the ZLIB algoritm. The compressed ZLIB data bytes begin with the byte immediately following the compression header, and extend to the end of the section. Additional documentation for ZLIB may be found at http://zlib.net.

ELFCOMPRESS_LOOS - ELFCOMPRESS_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

ELFCOMPRESS_LOPROC - ELF_COMPRESS_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics.

SHF_MASKOS

All bits included in this mask are reserved for operating system-specific semantics.

SHF_MASKPROC

All bits included in this mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

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

Figure 4-14: sh_link and sh_info Interpretation

sh_type

sh_link

sh_info

SHT_DYNAMIC

The section header index of the string table used by entries in the section.

0

SHT_HASH

The section header index of the symbol table to which the hash table applies.

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_GROUP

The section header index of the associated symbol table.

The symbol table index of an entry in the associated symbol table. The name of the specified symbol table entry provides a signature for the section group.

SHT_SYMTAB_SHNDX

The section header index of the associated symbol table section.

0

Rules for Linking Unrecognized Sections

If a link editor encounters sections whose headers contain OS-specific values it does not recognize in the sh_type or sh_flags fields, the link editor should combine those sections as described below.

If the section's sh_flags bits include the attribute SHF_OS_NONCONFORMING, then the section requires special knowledge to be correctly processed, and the link editor should reject the object containing the section with an error.

Unrecognized sections that do not have the SHF_OS_NONCONFORMING attribute, are combined in a two-phase process. As the link editor combines sections using this process, it must honor the alignment constraints of the input sections (asserted by the sh_addralign field), padding between sections with zero bytes, if necessary, and producing a combination with the maximum alignment constraint of its component input sections.

1. In the first phase, input sections that match in name, type and attribute flags should be concatenated into single sections. The concatenation order should satisfy the requirements of any known input section attributes (e.g, SHF_MERGE and SHF_LINK_ORDER). When not otherwise constrained, sections should be emitted in input order.
2. In the second phase, sections should be assigned to segments or other units based on their attribute flags. Sections of each particular unrecognized type should be assigned to the same unit unless prevented by incompatible flags, and within a unit, sections of the same unrecognized type should be placed together if possible.

Non OS-specific processing (e.g. relocation) should be applied to unrecognized section types. An output section header table, if present, should contain entries for unknown sections. Any unrecognized section attribute flags should be removed.

It is recommended that link editors follow the same two-phase ordering approach described above when linking sections of known types. Padding between such sections may have values different from zero, where appropriate.

Section Groups

Some sections occur in interrelated groups. For example, an out-of-line definition of an inline function might require, in addition to the section containing its executable instructions, a read-only data section containing literals referenced, one or more debugging information sections and other informational sections. Furthermore, there may be internal references among these sections that would not make sense if one of the sections were removed or replaced by a duplicate from another object. Therefore, such groups must be included or omitted from the linked object as a unit. A section cannot be a member of more than one group.

A section of type SHT_GROUP defines such a grouping of sections. The name of a symbol from one of the containing object's symbol tables provides a signature for the section group. The section header of the SHT_GROUP section specifies the identifying symbol entry, as described above: the sh_link member contains the section header index of the symbol table section that contains the entry. The sh_info member contains the symbol table index of the identifying entry. The sh_flags member of the section header contains 0. The name of the section (sh_name) is not specified.

The referenced signature symbol is not restricted. Its containing symbol table section need not be a member of the group, for example.

The section data of a SHT_GROUP section is an array of Elf32_Word entries. The first entry is a flag word. The remaining entries are a sequence of section header indices.

The following flags are currently defined:

Figure 4-15: Section Group Flags

Name	Value
GRP_COMDAT	0x1
GRP_MASKOS	0x0ff00000
GRP_MASKPROC	0xf0000000

GRP_COMDAT
: This is a COMDAT group. It may duplicate another COMDAT group in another object file, where duplication is defined as having the same group signature. In such cases, only one of the duplicate groups may be retained by the linker, and the members of the remaining groups must be discarded.

GRP_MASKOS
: All bits included in this mask are reserved for operating system-specific semantics.

GRP_MASKPROC
: All bits included in this mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

The section header indices in the SHT_GROUP section identify the sections that make up the group. Each such section must have the SHF_GROUP flag set in its sh_flags section header member. If the linker decides to remove the section group, it must remove all members of the group.

This requirement is not intended to imply that special case behavior like removing debugging information requires removing the sections to which that information refers, even if they are part of the same group.

To facilitate removing a group without leaving dangling references and with only minimal processing of the symbol table, the following rules must be followed:

- A symbol table entry with STB_GLOBAL or STB_WEAK binding that is defined relative to one of a group's sections, and that is contained in a symbol table section that is not part of the group, must be converted to an undefined symbol (its section index must be changed to SHN_UNDEF) if the group members are discarded. References to this symbol table entry from outside the group are allowed.
- A symbol table entry with STB_LOCAL binding that is defined relative to one of a group's sections, and that is contained in a symbol table section that is not part of the group, must be discarded if the group members are discarded. References to this symbol table entry from outside the group are not allowed.
- An undefined symbol that is referenced only from one or more sections that are part of a particular group, and that is contained in a symbol table section that is not part of the group, is not removed when the group members are discarded. In other words, the undefined symbol is not removed even if no references to that symbol remain.
- There may not be non-symbol references to the sections comprising a group from outside the group, for example, use of a group member's section header index in an sh_link or sh_info member.

Special Sections

Various sections hold program and control information.

The following table shows sections that are used by the system and have the indicated types and attributes.

Figure 4-16: Special Sections

Name

Type

Attributes

.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

.debug

SHT_PROGBITS

none

.dynamic

SHT_DYNAMIC

see below

.dynstr

SHT_STRTAB

SHF_ALLOC

.dynsym

SHT_DYNSYM

SHF_ALLOC

.fini

SHT_PROGBITS

SHF_ALLOC+SHF_EXECINSTR

.fini_array

SHT_FINI_ARRAY

SHF_ALLOC+SHF_WRITE

.got

SHT_PROGBITS

see below

.hash

SHT_HASH

SHF_ALLOC

.init

SHT_PROGBITS

SHF_ALLOC+SHF_EXECINSTR

.init_array

SHT_INIT_ARRAY

SHF_ALLOC+SHF_WRITE

.interp

SHT_PROGBITS

see below

.line

SHT_PROGBITS

none

.note

SHT_NOTE

none

.plt

SHT_PROGBITS

see below

.preinit_array

SHT_PREINIT_ARRAY

SHF_ALLOC+SHF_WRITE

.rel*name*

SHT_REL

see below

.rela*name*

SHT_RELA

see below

.rodata

SHT_PROGBITS

SHF_ALLOC

.rodata1

SHT_PROGBITS

SHF_ALLOC

.shstrtab

SHT_STRTAB

none

.strtab

SHT_STRTAB

see below

.symtab

SHT_SYMTAB

see below

.symtab_shndx

SHT_SYMTAB_SHNDX

see below

.tbss

SHT_NOBITS

SHF_ALLOC+SHF_WRITE+SHF_TLS

.tdata

SHT_PROGBITS

SHF_ALLOC+SHF_WRITE+SHF_TLS

.tdata1

SHT_PROGBITS

SHF_ALLOC+SHF_WRITE+SHF_TLS

.text

SHT_PROGBITS

SHF_ALLOC+SHF_EXECINSTR

.bss
: This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS.

.comment
: This section holds version control information.

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

.debug
: This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix .debug are reserved for future use in the ABI.

.dynamic
: This section holds dynamic linking information. The section's attributes will include the SHF_ALLOC bit. Whether the SHF_WRITE bit is set is processor specific. See Chapter 5 for more information.

.dynstr
: This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See Chapter 5 for more information.

.dynsym
: This section holds the dynamic linking symbol table, as described in ``Symbol Table''. See Chapter 5 for more information.

.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.

.fini_array
: This section holds an array of function pointers that contributes to a single termination array for the executable or shared object containing the section.

.got
: This section holds the global offset table. See ``Coding Examples'' in Chapter 3, ``Special Sections'' in Chapter 4, and ``Global Offset Table'' in Chapter 5 of the processor supplement for more information.

.hash
: This section holds a symbol hash table. See ``Hash Table'' in Chapter 5 for more information.

.init
: This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs).

.init_array
: This section holds an array of function pointers that contributes to a single initialization array for the executable or shared object containing the section.

.interp

This section holds the path name of a program interpreter. 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. See Chapter 5 for more information.

.line

This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

.note

This section holds information in the format that ``Note Section''. in Chapter 5 describes.

.plt

This section holds the procedure linkage table. See ``Special Sections'' in Chapter 4 and ``Procedure Linkage Table'' in Chapter 5 of the processor supplement for more information.

.preinit_array

This section holds an array of function pointers that contributes to a single pre-initialization array for the executable or shared object containing the section.

.rel*name* and .rela*name*

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

.rodata and .rodata1

These sections hold read-only data that typically contribute to a non-writable segment in the process image. See ``Program Header'' in Chapter 5 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''. in this chapter 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.

.symtab_shndx

This section holds the special symbol table section index array, as described above. The section's attributes will include the SHF_ALLOC bit if the associated symbol table section does; otherwise that bit will be off.

.tbss

This section holds uninitialized *thread-local data* that contribute to the program's memory image. By definition, the system initializes the data with zeros when the data is instantiated for each new execution flow. The section occupies no file space, as indicated by the section type, SHT_NOBITS. Implementations need not support thread-local storage.

.tdata

This section holds initialized *thread-local data* that contributes to the program's memory image. A copy of its contents is instantiated by the system for each new execution flow. Implementations need not support thread-local storage.

.text

This section holds the ``text,'' or executable instructions, of a program.

Section names with a dot (.) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not shown in the previous list. 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 instance .FOO.psect is the psect section defined by the FOO architecture. Existing extensions are called by their historical names.

Pre-existing Extensions	
.sdata	.tdesc
.sbss	.lit4
.lit8	.reginfo
.gptab	.liblist
.conflict	

For information on processor-specific sections, see the ABI supplement for the desired processor.

Previous Contents Next

String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's sh_size member would contain zero. Non-zero 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 4-17: 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.

Name	Value
STN_UNDEF	0

A symbol table entry has the following format.

Figure 4-18: Symbol Table Entry

```
typedef struct {
   Elf32_Word  st_name;
   Elf32_Addr  st_value;
   Elf32_Word  st_size;
   unsigned char   st_info;
   unsigned char   st_other;
   Elf32_Half  st_shndx;
} Elf32_Sym;

typedef struct {
   Elf64_Word  st_name;
   unsigned char   st_info;
   unsigned char   st_other;
   Elf64_Half  st_shndx;
   Elf64_Addr  st_value;
   Elf64_Xword st_size;
} Elf64_Sym;
```

st_name
: 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 non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

External C symbols have the same names in C and 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 on; details appear below.

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 for both 32 and 64-bit objects.

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

#define ELF64_ST_BIND(i)   ((i)>>4)
#define ELF64_ST_TYPE(i)   ((i)&0xf)
#define ELF64_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
```

st_other

This member currently specifies a symbol's visibility. A list of the values and meanings appears below. The following code shows how to manipulate the values for both 32 and 64-bit objects. Other bits contain 0 and have no defined meaning.

```
#define ELF32_ST_VISIBILITY(o) ((o)&0x3)
#define ELF64_ST_VISIBILITY(o) ((o)&0x3)
```

st_shndx

Every symbol table entry is *defined* in relation to some section. This member holds the relevant section header table index. As the sh_link and sh_info interpretation table and the related text describe, some section indexes indicate special meanings.

If this member contains SHN_XINDEX, then the actual section header index is too large to fit in this field. The actual value is contained in the associated section of type SHT_SYMTAB_SHNDX.

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

Figure 4-19: Symbol Binding

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL
: Local 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_LOOS through STB_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

STB_LOPROC through STB_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

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 whose st_shndx field holds SHN_COMMON), the appearance of a weak symbol with the same name will not cause an error. The link editor honors the common definition and ignores the weak ones.
- When the link editor searches archive libraries [see ``Archive File'' in Chapter 7], it extracts archive members that contain definitions of undefined 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.

The behavior of weak symbols in areas not specified by this document is implementation defined. Weak symbols are intended primarily for use in system software. Applications using weak symbols are unreliable since changes in the runtime environment might cause the execution to fail.

In each symbol table, all symbols with STB_LOCAL binding precede the weak and global symbols. As ``Sections'', above 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.

Figure 4-20: Symbol Types

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	

4

STT_COMMON

5

STT_TLS

6

STT_LOOS

10

STT_HIOS

12

STT_LOPROC

13

STT_HIPROC

15

STT_NOTYPE
: The symbol's type is not specified.

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

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

STT_SECTION
: The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.

STT_FILE
: Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS, and it precedes the other STB_LOCAL symbols for the file, if it is present.

STT_COMMON
: The symbol labels an uninitialized common block. See below for details.

STT_TLS
: The symbol specifies a *Thread-Local Storage* entity. When defined, it gives the assigned offset for the symbol, not the actual address. Symbols of type STT_TLS can be referenced by only special thread-local storage relocations and thread-local storage relocations can only reference symbols with type STT_TLS. Implementation need not support thread-local storage.

STT_LOOS through STT_HIOS

Values in this inclusive range are reserved for operating system-specific semantics.

STT_LOPROC through STT_HIPROC

Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

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.

Symbols with type STT_COMMON label uninitialized common blocks. In relocatable objects, these symbols are not allocated and must have the special section index SHN_COMMON (see below). In shared objects and executables these symbols must be allocated to some section in the defining object.

In relocatable objects, symbols with type STT_COMMON are treated just as other symbols with index SHN_COMMON. If the link-editor allocates space for the SHN_COMMON symbol in an output section of the object it is producing, it must preserve the type of the output symbol as STT_COMMON.

When the dynamic linker encounters a reference to a symbol that resolves to a definition of type STT_COMMON, it may (but is not required to) change its symbol resolution rules as follows: instead of binding the reference to the first symbol found with the given name, the dynamic linker searches for the first symbol with that name with type other than STT_COMMON. If no such symbol is found, it looks for the STT_COMMON definition of that name that has the largest size.

A symbol's visibility, although it may be specified in a relocatable object, defines how that symbol may be accessed once it has become part of an executable or shared object.

Figure 4-21: Symbol Visibility

Name

Value

STV_DEFAULT

0

STV_INTERNAL

1

STV_HIDDEN

2

STV_PROTECTED

3

STV_DEFAULT

The visibility of symbols with the STV_DEFAULT attribute is as specified by the symbol's binding type. That is, global and weak symbols are visible outside of their defining *component* (executable file or shared object). Local symbols are *hidden*, as described below. Global and weak symbols are also *preemptable*, that is, they may by preempted by definitions of the same name in another component.

An implementation may restrict the set of global and weak symbols that are externally visible.

STV_PROTECTED

A symbol defined in the current component is *protected* if it is visible in other components but not preemptable, meaning that any reference to such a symbol from within the defining component must be resolved to the definition in that component, even if there is a definition in another component that would preempt by the default rules. A symbol with STB_LOCAL binding may not have STV_PROTECTED visibility. If a symbol definition with STV_PROTECTED visibility from a shared object is taken as resolving a reference from an executable or another shared object, the SHN_UNDEF symbol table entry created has STV_DEFAULT visibility.

The presence of the STV_PROTECTED flag on a symbol in a given load module does not affect the symbol resolution rules for references to that symbol from outside the containing load module.

STV_HIDDEN

A symbol defined in the current component is *hidden* if its name is not visible to other components. Such a symbol is necessarily protected. This attribute may be used to control the external interface of a component. Note that an object named by such a symbol may still be referenced from another component if its address is passed outside.

A hidden symbol contained in a relocatable object must be either removed or converted to STB_LOCAL binding by the link-editor when the relocatable object is included in an executable file or shared object.

STV_INTERNAL

The meaning of this visibility attribute may be defined by processor supplements to further constrain hidden symbols. A processor supplement's definition should be such that generic tools can safely treat internal symbols as hidden.

An internal symbol contained in a relocatable object must be either removed or converted to STB_LOCAL binding by the link-editor when the relocatable object is included in an executable file or shared object.

None of the visibility attributes affects resolution of symbols within an executable or shared object during link-editing -- such resolution is controlled by the binding type. Once the link-editor has chosen its resolution, these attributes impose two requirements, both based on the fact that references in the code being linked may have been optimized to take advantage of the attributes.

- First, all of the non-default visibility attributes, when applied to a symbol reference, imply that a definition to satisfy that reference must be provided within the current executable or shared object. If such a symbol reference has no definition within the component being linked, then the reference must have STB_WEAK binding and is resolved to zero.
- Second, if any reference to or definition of a name is a symbol with a non-default visibility attribute, the visibility attribute must be propagated to the resolving symbol in the linked object. If different visibility attributes are specified for distinct references to or definitions of a symbol, the most constraining visibility attribute must be propagated to the resolving symbol in the linked object. The attributes, ordered from least to most constraining, are: STV_PROTECTED, STV_HIDDEN and STV_INTERNAL.

If a symbol's value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, 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. 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. Symbols with section index SHN_COMMON may appear only in relocatable objects.

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

the actual definition.

SHN_XINDEX

This value is an escape value. It indicates that the symbol refers to a specific location within a section, but that the section header index for that section is too large to be represented directly in the symbol table entry. The actual section header index is found in the associated SHT_SYMTAB_SHNDX section. The entries in that section correspond one to one with the entries in the symbol table. Only those entries in SHT_SYMTAB_SHNDX that correspond to symbol table entries with SHN_XINDEX will hold valid section header indexes; all other entries will have value 0.

The symbol table entry for index 0 (STN_UNDEF) is reserved; it holds the following.

Figure 4-22: Symbol Table Entry:Index 0

Name	**Value**	**Note**
st_name	0	No name
st_value	0	Zero value
st_size	0	No size
st_info	0	No type, local binding
st_other	0	Default visibility
st_shndx	SHN_UNDEF	

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the st_value member.

- In relocatable files, st_value holds alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value holds a section offset for a defined symbol. st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable and shared object files, st_value holds a virtual address. To make these files' symbols more useful for the dynamic 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 allows 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. Relocatable files must have ``relocation entries'' which are necessary because they contain 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.

Figure 4-23: Relocation Entries

```
typedef struct {
   Elf32_Addr  r_offset;
   Elf32_Word  r_info;
} Elf32_Rel;

typedef struct {
   Elf32_Addr  r_offset;
   Elf32_Word  r_info;
   Elf32_Sword r_addend;
} Elf32_Rela;

typedef struct {
   Elf64_Addr  r_offset;
   Elf64_Xword r_info;
} Elf64_Rel;

typedef struct {
   Elf64_Addr  r_offset;
   Elf64_Xword r_info;
   Elf64_Sxword    r_addend;
} Elf64_Rela;
```

---

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

r_info

This member gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry would hold the symbol table index of the function being called. If the index is STN_UNDEF, the undefined symbol index, the relocation uses 0 as the ``symbol value''. Relocation types are processor-specific; descriptions of their behavior appear in the processor supplement. 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 ELF64_R_TYPE) or ELF32_R_SYM (or ELF64_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))

#define ELF64_R_SYM(i)    ((i)>>32)
#define ELF64_R_TYPE(i)   ((i)&0xffffffffL)
#define ELF64_R_INFO(s,t) (((s)<<32)+((t)&0xffffffffL))
```

---

r_addend

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

As specified previously, only Elf32_Rela and Elf64_Rela entries contain an explicit addend. Entries of type Elf32_Rel and Elf64_Rel store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Consequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

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'' above, 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. 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 dynamic 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.

The typical application of an ELF relocation is to determine the referenced symbol value, extract the addend (either from the field to be relocated or from the addend field contained in the relocation record, as appropriate for the type of relocation record), apply the expression implied by the relocation type to the symbol and addend, extract the desired part of the expression result, and place it in the field to be relocated.

If multiple *consecutive* relocation records are applied to the same relocation location (r_offset), they are *composed* instead of being applied independently, as described above. By *consecutive*, we mean that the relocation records are contiguous within a single relocation section. By *composed*, we mean that the standard application described above is modified as follows:

- In all but the last relocation operation of a composed sequence, the result of the relocation expression is retained, rather than having part extracted and placed in the relocated field. The result is retained at full pointer precision of the applicable ABI processor supplement.
- In all but the first relocation operation of a composed sequence, the addend used is the retained result of the previous relocation operation, rather than that implied by the relocation type.

Note that a consequence of the above rules is that the location specified by a relocation type is relevant for the first element of a composed sequence (and then only for relocation records that do not contain an explicit addend field) and for the last element, where the location determines where the relocated value will be placed. For all other relocation operands in a composed sequence, the location specified is ignored.

An ABI processor supplement may specify individual relocation types that always stop a composition sequence, or always start a new one.

Relocation Types (Processor-Specific)

This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Introduction

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. As section ''Virtual Address Space'' in Chapter 3 of the processor supplement describes, a process image has segments that hold its text, data, stack, and so on. This chapter's major sections discuss the following:

- Program Header. This section complements Chapter 4, describing object file structures that relate directly to program execution. The primary data structure, a program header table, locates segment images within the file and contains other information necessary to create the memory image for the program.
- Program Loading. Given an object file, the system must load it into memory for the program to run.
- Dynamic linking. After the system loads the program it must complete the process image by resolving symbolic references among the object files that compose the process.

The processor supplement defines a naming convention for ELF constants that have processor ranges specified. Names such as DT_, PT_, for processor specific extensions, incorporate the name of the processor: DT_M32_SPECIAL, for example. Pre-existing processor extensions not using this convention will be supported.

Pre-Existing Extensions

DT_JUMP_REL

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 ``Segment Contents'' describes below. 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'' in Chapter 4 for more information.

Figure 5-1: Program Header

```
typedef struct {
   Elf32_Word  p_type;
   Elf32_Off   p_offset;
   Elf32_Addr  p_vaddr;
   Elf32_Addr  p_paddr;
   Elf32_Word  p_filesz;
   Elf32_Word  p_memsz;
   Elf32_Word  p_flags;
   Elf32_Word  p_align;
} Elf32_Phdr;

typedef struct {
   Elf64_Word  p_type;
   Elf64_Word  p_flags;
   Elf64_Off   p_offset;
   Elf64_Addr  p_vaddr;
   Elf64_Addr  p_paddr;
   Elf64_Xword p_filesz;
   Elf64_Xword p_memsz;
   Elf64_Xword p_align;
} Elf64_Phdr;
```

p_type
: 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 appear below.

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 System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

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

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

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

p_align
: As ``Program Loading'' describes in this chapter of the processor supplement, 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.

Figure 5-2: 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_TLS

7

PT_LOOS

0x60000000

PT_HIOS

0x6fffffff

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 have 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'' below 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'' below for more information.

PT_NOTE
: The array element specifies the location and size of auxiliary information. See ``Note Section'' below for more information.

PT_SHLIB
: This segment type is reserved but has unspecified semantics. Programs that contain an array element of this type do not conform to the ABI.

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'' below for more information.

PT_TLS
: The array element specifies the *Thread-Local Storage* template. Implementations need not support this program table entry. See ``Thread-Local Storage'' below for more information.

PT_LOOS through PT_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics.

PT_LOPROC through PT_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Unless specifically required elsewhere, all program header segment types are optional. A file's program header table may contain only those elements relevant to its contents.

Base Address

As ``Program Loading'' in this chapter of the processor supplement describes, the virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image. Executable files typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. On some platforms, while the system chooses virtual addresses for individual processes, it maintains the *relative* position of one segment to another within any one shared object. Because position-independent code on those platforms uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The differences between the virtual address of any segment in memory and the corresponding virtual address in the file is thus a single constant value for any one executable or shared object in a given process. This difference is the *base address*. One use of the base address is to relocate the memory image of the file during dynamic linking.

An executable or shared object file's base address (on platforms that support the concept) is calculated during execution from three values: the virtual memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. To compute the base address, one determines the memory address associated with the lowest p_vaddr value for a PT_LOAD segment. This address is truncated to the nearest multiple of the maximum page size. The corresponding p_vaddr value itself is also truncated to the nearest multiple of the maximum page size. The base address is the difference between the truncated memory address and the truncated p_vaddr value.

See this chapter in the processor supplement for more information and examples. ``Operating System Interface'' of Chapter 3 in the processor supplement contains more information about the virtual address space and page size.

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.

Figure 5-3: Segment Flag Bits, p_flags

Name	**Value**	**Meaning**
PF_X	0x1	Execute
PF_W	0x2	Write
PF_R	0x4	Read

PF_MASKOS

0x0ff00000

Unspecified

PF_MASKPROC

0xf0000000

Unspecified

All bits included in the PF_MASKOS mask are reserved for operating system-specific semantics.

All bits included in the PF_MASKPROC mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

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 table shows both the exact flag interpretation and the allowable flag interpretation. ABI-conforming systems may provide either.

Figure 5-4: 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. See the processor supplement for details.

Text segments contain read-only instructions and data, typically including the following sections described in Chapter 4. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

Figure 5-5: Text Segment

.text
.rodata
.hash
.dynsym
.dynstr
.plt
.rel.got

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

Figure 5-6: Data Segment

.data
.dynamic
.got
.bss

A PT_DYNAMIC program header element points at the .dynamic section, explained in ``Dynamic Section'' below. The .got and .plt sections also hold information related to position-independent code and dynamic linking. Although the .plt appears in a text segment in the previous table, it may reside in a text or a data segment, depending on the processor. See ``Global Offset Table'' and ``Procedure Linkage Table'' in this section of the processor supplement for details.

As ``Sections'' in Chapter 4 describes, the .bss section has the type SHT_NOBITS. Although it occupies no space in the file, it contributes to the segment's memory image. Normally, these uninitialized data reside at the end of the segment, thereby making p_memsz larger than p_filesz in the associated program header element.

Note Section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, etc. 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 a variable amount of entries. In 64-bit objects (files with e_ident[EI_CLASS] equal to ELFCLASS64), each entry is an array of 8-byte words in the format of the target processor. In 32-bit objects (files with e_ident[EI_CLASS] equal to ELFCLASS32), each entry is an array of 4-byte words in the format of the target processor. Labels appear below to help explain note information organization, but they are not part of the specification.

Figure 5-7: Note Information

namesz
descsz
type
name
. . .
desc
. . .

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 8 or 4-byte alignment for the descriptor (depending on whether the file is a 64-bit or 32-bit object). Such padding is not included in namesz.

descsz and desc
: The first descsz bytes in desc hold the note descriptor. The ABI places no constraints on a descriptor's contents. If no descriptor is present, descsz contains 0. Padding is present, if necessary, to ensure 8 or 4-byte alignment for the next note entry (depending on whether the file is a 64-bit or 32-bit object). 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 recognize a descriptor. Types currently must be non-negative. The ABI does not define what descriptors mean.

To illustrate, the following note segment holds two entries.

Figure 5-8: Example Note Segment

namesz
descsz
type
name

namesz
descsz
type
name

desc

+0

+1

+2

+3

7

0

1

x

y

z

c

o

\0

pad

7

8

3

x

y

z

c

o

\0

pad

word 0

word 1

No descriptor

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.

Note information is optional. The presence of note information does not affect a program's ABI conformance, provided the information does not affect the program's execution behavior. Otherwise, the program does not conform to the ABI and has undefined behavior.

Thread-Local Storage

To permit association of separate copies of data allocated at compile-time with individual threads of execution, thread-local storage sections can be used to specify the size and initial contents of such data. Implementations need not support thread-local storage. A PT_TLS program entry has the following members:

Member

Value

p_offset

File offset of the TLS initialization image

p_vaddr

Virtual memory address of the TLS initialization image

p_paddr

reserved

p_filesz

Size of the TLS initialization image

p_memsz

Total size of the TLS template

p_flags

PF_R

p_align

Alignment of the TLS template

The *TLS template* is formed from the combination of all sections with the flag SHF_TLS. The portion of the TLS template that holds initialized data is the *TLS initialization image*. (The remaining portion of the TLS template is one or more sections of type SHT_NOBITS.)

Previous Contents Next

Program Loading (Processor-Specific)

This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Dynamic Linking

Program Interpreter

An executable file that participates in dynamic linking shall have one PT_INTERP program header element. During exec(BA_OS), 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 the original executable file's segment images, 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.

As ``Process Initialization'' in Chapter 3 of the processor supplement mentions, the interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by mmap(KE_OS) and related services [See ``Virtual Address Space'' in Chapter 3 of the processor supplement]. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.
- An executable file may be loaded at fixed addresses; if so, 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.

Dynamic 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 dynamic linker as the program interpreter.

The locations of the system provided dynamic linkers are processor specific.

Exec(BA_OS) and the dynamic 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 dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from exec(BA_OS).

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown above in ``Program Header'', this data resides in loadable segments, making them available during execution. (Once again, recall the exact segment contents are processor-specific. See the processor supplement for complete information).

- 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. Chapter 3 discusses how programs use the global offset table for position-independent code. Sections below explain how the dynamic linker uses and changes the tables to create memory images for object files.

Because every ABI-conforming program imports the basic system services from a shared object library [See ``System Library'' in Chapter 6], the dynamic linker participates in every ABI-conforming program execution.

As `Program Loading'' explains in the processor supplement, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment [see exec(BA_OS)] contains a variable named LD_BIND_NOW with a non-null value, the dynamic linker processes all relocations before transferring control to the program. For example, all the following environment entries would specify this behavior.

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

Otherwise, LD_BIND_NOW either does not occur in the environment or has a null value. The dynamic linker is permitted to evaluate procedure linkage table entries lazily, thus avoiding symbol resolution and relocation overhead for functions that are not called. See ``Procedure Linkage Table'' in this chapter of the processor supplement 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.

Figure 5-9: Dynamic Structure

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

extern Elf32_Dyn    _DYNAMIC[];

typedef struct {
   Elf64_Sxword    d_tag;
   union {
      Elf64_Xword d_val;
      Elf64_Addr  d_ptr;
   } d_un;
} Elf64_Dyn;

extern Elf64_Dyn    _DYNAMIC[];
```

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

d_val
: These objects represent integer values with various interpretations.

d_ptr
: These objects represent program virtual addresses. 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 dynamic linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do *not* contain relocation entries to ``correct'' addresses in the dynamic structure.

To make it simpler for tools to interpret the contents of dynamic section entries, the value of each tag, except for those in two special compatibility ranges, will determine the interpretation of the d_un union. A tag whose value is an even number indicates a dynamic section entry that uses d_ptr. A tag whose value is an odd number indicates a dynamic section entry that uses d_val or that uses neither d_ptr nor d_val. Tags whose values are less than the special value DT_ENCODING and tags whose values fall between DT_HIOS and DT_LOPROC do not follow these rules.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked ``mandatory'', the dynamic linking array for an ABI-conforming file must have an entry of that type. Likewise, ``optional'' means an entry for the tag may appear but is not required.

Figure 5-10: Dynamic Array Tags, d_tag

Name	**Value**	d_un	**Executable**	**Shared Object**
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional

DT_PLTGOT

3

d_ptr

optional

optional

DT_HASH

4

d_ptr

mandatory

mandatory

DT_STRTAB

5

d_ptr

mandatory

mandatory

DT_SYMTAB

6

d_ptr

mandatory

mandatory

DT_RELA

7

d_ptr

mandatory

optional

DT_RELASZ

8

d_val

mandatory

optional

DT_RELAENT

9

d_val

mandatory

optional

DT_STRSZ

10

d_val

mandatory

mandatory

DT_SYMENT

11

d_val

mandatory

mandatory

DT_INIT

12

d_ptr

optional

optional

DT_FINI

13

d_ptr

optional

optional

DT_SONAME

14

d_val

ignored

optional

DT_RPATH*

15

d_val

optional

ignored

DT_SYMBOLIC*

16

ignored

ignored

optional

DT_REL

17

d_ptr

mandatory

optional

DT_RELSZ

18

d_val

mandatory

optional

DT_RELENT

19

d_val

mandatory

optional

DT_PLTREL

20

d_val

optional

optional

DT_DEBUG

21

d_ptr

optional

ignored

DT_TEXTREL*

22

ignored

optional

optional

DT_JMPREL

23

d_ptr

optional

optional

DT_BIND_NOW*

24

ignored

optional

optional

DT_INIT_ARRAY

25

d_ptr

optional

optional

DT_FINI_ARRAY

26

d_ptr

optional

optional

DT_INIT_ARRAYSZ

27

d_val

optional

optional

DT_FINI_ARRAYSZ

28

d_val

optional

optional

DT_RUNPATH

29

d_val

optional

optional

DT_FLAGS

30

d_val

optional

optional

DT_ENCODING

32

unspecified

unspecified

unspecified

DT_PREINIT_ARRAY

32

d_ptr

optional

ignored

DT_PREINIT_ARRAYSZ

33

d_val

optional

ignored

DT_SYMTAB_SHNDX

34

d_ptr

optional

optional

DT_LOOS

0x6000000D

unspecified

unspecified

unspecified

DT_HIOS

0x6ffff000

unspecified

unspecified

unspecified

DT_LOPROC

0x70000000

unspecified

unspecified

unspecified

DT_HIPROC

0x7fffffff

unspecified

unspecified

unspecified

* Signifies an entry that is at level 2.

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 library. The offset is an index into the table recorded in the DT_STRTAB code. See ``Shared Object Dependencies'' 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. See this section in the processor supplement for details.

DT_HASH
: This element holds the address of the symbol hash table, described in ``Hash Table''. This hash table refers to the symbol table referenced by the DT_SYMTAB element.

DT_STRTAB
: This element holds the address of the string table, described in Chapter 4. Symbol names, library names, and other strings 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 and Elf64_Sym entries for the 64-bit class of files.

DT_RELA
: This element holds the address of a relocation table, described in Chapter 4. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class or Elf64_Rela for the 64-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 dynamic linker sees a single table. When the dynamic 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'' below.

DT_FINI

This element holds the address of the termination function, discussed in ``Initialization and Termination Functions'' below.

DT_SONAME

This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. See ``Shared Object Dependencies'' below 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 Object Dependencies''. The offset is an index into the table recorded in the DT_STRTAB entry. This entry is at level 2. Its use has been superseded by DT_RUNPATH.

DT_SYMBOLIC

This element's presence in a shared object library alters the dynamic linker's symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual. This entry is at level 2. Its use has been superseded by the DF_SYMBOLIC flag.

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 or Elf64_Rel for the 64-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. Its contents are not specified for the ABI; programs that access this entry are not ABI-conforming.

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 dynamic linker can prepare accordingly. This entry is at level 2. Its use has been superseded by the DF_TEXTREL flag.

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 dynamic 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_BIND_NOW
: If present in a shared object or executable, this entry instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via dlopen(BA_LIB). This entry is at level 2. Its use has been superseded by the DF_BIND_NOW flag.

DT_INIT_ARRAY
: This element holds the address of the array of pointers to initialization functions, discussed in ``Initialization and Termination Functions'' below.

DT_FINI_ARRAY
: This element holds the address of the array of pointers to termination functions, discussed in ``Initialization and Termination Functions'' below.

DT_INIT_ARRAYSZ
: This element holds the size in bytes of the array of initialization functions pointed to by the DT_INIT_ARRAY entry. If an object has a DT_INIT_ARRAY entry, it must also have a DT_INIT_ARRAYSZ entry.

DT_FINI_ARRAYSZ
: This element holds the size in bytes of the array of termination functions pointed to by the DT_FINI_ARRAY entry. If an object has a DT_FINI_ARRAY entry, it must also have a DT_FINI_ARRAYSZ entry.

DT_RUNPATH
: This element holds the string table offset of a null-terminated library search path string discussed in ``Shared Object Dependencies''. The offset is an index into the table recorded in the DT_STRTAB entry.

DT_FLAGS
: This element holds flag values specific to the object being loaded. Each flag value will have the name DF_*flag_name*. Defined values and their meanings are described below. All other values are reserved.

DT_PREINIT_ARRAY
: This element holds the address of the array of pointers to pre-initialization functions, discussed in ``Initialization and Termination Functions'' below. The DT_PREINIT_ARRAY table is processed only in an

executable file; it is ignored if contained in a shared object.

DT_PREINIT_ARRAYSZ
: This element holds the size in bytes of the array of pre-initialization functions pointed to by the DT_PREINIT_ARRAY entry. If an object has a DT_PREINIT_ARRAY entry, it must also have a DT_PREINIT_ARRAYSZ entry. As with DT_PREINIT_ARRAY, this entry is ignored if it appears in a shared object.

DT_SYMTAB_SHNDX
: This element holds the address of the SHT_SYMTAB_SHNDX section associated with the dynamic symbol table referenced by the DT_SYMTAB element.

DT_ENCODING
: Values greater than or equal to DT_ENCODING and less than DT_LOOS follow the rules for the interpretation of the d_un union described above.

DT_LOOS through DT_HIOS
: Values in this inclusive range are reserved for operating system-specific semantics. All such values follow the rules for the interpretation of the d_un union described above.

DT_LOPROC through DT_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them. All such values follow the rules for the interpretation of the d_un union described above.

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.

Figure 5-11: DT_FLAGS values

Name	Value
DF_ORIGIN	0x1
DF_SYMBOLIC	0x2
DF_TEXTREL	0x4
DF_BIND_NOW	0x8
DF_STATIC_TLS	

0x10

DF_ORIGIN
: This flag signifies that the object being loaded may make reference to the $ORIGIN substitution string (see ``Substitution Sequences''). The dynamic linker must determine the pathname of the object containing this entry when the object is loaded.

DF_SYMBOLIC
: If this flag is set in a shared object library, the dynamic linker's symbol resolution algorithm for references within the library is changed. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.

DF_TEXTREL
: If this flag is not set, 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 flag is set, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

DF_BIND_NOW
: If set in a shared object or executable, this flag instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via dlopen(BA_LIB).

DF_STATIC_TLS
: If set in a shared object or executable, this flag instructs the dynamic linker to reject attempts to load this file dynamically. It indicates that the shared object or executable contains code using a *static thread-local storage* scheme. Implementations need not support any form of thread-local storage.

Shared Object Dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution.

When the dynamic 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 dynamic linker builds a complete process image. When resolving symbolic references, the dynamic 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), and then at the second level DT_NEEDED entries, and so on. Shared object files must be readable by the process; other permissions are not required.

Even when a shared object is referenced multiple times in the dependency list, the dynamic 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. For example, if the link editor builds an executable file using one shared object with a DT_SONAME entry of lib1 and another shared object library with the path name /usr/lib/lib2, the executable file will contain lib1 and /usr/lib/lib2 in its dependency list.

If a shared object name has one or more slash (/) characters anywhere in the name, such as /usr/lib/lib2 or directory/file, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as lib1, three facilities specify shared object path searching.

- The dynamic array tag DT_RUNPATH gives a string that holds a list of directories, separated by colons (:). For example, the string /home/dir/lib:/home/dir2/lib: tells the dynamic linker to search first the directory /home/dir/lib, then /home/dir2/lib, and then the current directory to find dependencies.

  The set of directories specified by a given DT_RUNPATH entry is used to find only the immediate dependencies of the executable or shared object containing the DT_RUNPATH entry. That is, it is used only for those dependencies contained in the DT_NEEDED entries of the dynamic structure containing the DT_RUNPATH entry, itself. One object's DT_RUNPATH entry does not affect the search for any other object's dependencies.

- A variable called LD_LIBRARY_PATH in the process environment [see exec(BA_OS)] may hold a list of directories as above, optionally followed by a semicolon (;) and another directory list. The following values would be equivalent to the previous example:

  - LD_LIBRARY_PATH=/home/dir/usr/lib:/home/dir2/usr/lib:
  - LD_LIBRARY_PATH=/home/dir/usr/lib;/home/dir2/usr/lib:
  - LD_LIBRARY_PATH=/home/dir/usr/lib:/home/dir2/usr/lib:;

  Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the semantics described previously.

  All LD_LIBRARY_PATH directories are searched before those from DT_RUNPATH.

- Finally, if the other two groups of directories fail to locate the desired library, the dynamic linker searches the default directories, /usr/lib or such other directories as may be specified by the ABI supplement for a given processor.

When the dynamic linker is searching for shared objects, it is not a fatal error if an ELF file with the wrong attributes is encountered in the search. Instead, the dynamic linker shall exhaust the search of all paths before determining that a matching object could not be found. For this determination, the relevant attributes are contained in the following ELF header fields: e_ident[EI_DATA], e_ident[EI_CLASS], e_ident[EI_OSABI], e_ident[EI_ABIVERSION], e_machine, e_type, e_flags and e_version.

For security, the dynamic linker ignores LD_LIBRARY_PATH for set-user and set-group ID programs. It does, however, search DT_RUNPATH directories and the default directories. The same restriction may be applied to processes that have more than minimal privileges on systems with installed extended security mechanisms.

A fourth search facility, the dynamic array tag DT_RPATH, has been moved to level 2 in the ABI. It provides a colon-separated list of directories to search. Directories specified by DT_RPATH are searched before directories specified by LD_LIBRARY_PATH.

If both DT_RPATH and DT_RUNPATH entries appear in a single object's dynamic array, the dynamic linker processes only the DT_RUNPATH entry.

Substitution Sequences

Within a string provided by dynamic array entries with the DT_NEEDED or DT_RUNPATH tags and in pathnames passed as parameters to the dlopen() routine, a dollar sign ($) introduces a substitution sequence. This sequence consists of the dollar sign immediately followed by either the longest *name* sequence or a name contained within left and right braces ({) and (}). A name is a sequence of bytes that start with either a letter or an underscore followed by zero or more letters, digits or underscores. If a dollar sign is not immediately followed by a name or a brace-enclosed name, the behavior of the dynamic linker is unspecified.

If the name is ``ORIGIN'', then the substitution sequence is replaced by the dynamic linker with the absolute pathname of the directory in which the object containing the substitution sequence originated. Moreover, the pathname will contain no symbolic links or use of ``.'' or ``..'' components. Otherwise (when the name is not ``ORIGIN'') the behavior of the dynamic linker is unspecified.

When the dynamic linker loads an object that uses $ORIGIN, it must calculate the pathname of the directory containing the object. Because this calculation can be computationally expensive, implementations may want to avoid the calculation for objects that do not use $ORIGIN. If an object calls dlopen() with a string containing $ORIGIN and does not use $ORIGIN in one if its dynamic array entries, the dynamic linker may not have calculated the pathname for the object until the dlopen() actually occurs. Since the application may have changed its current working directory before the dlopen() call, the calculation may not yield the correct result. To avoid this possibility, an object may signal its intention to reference $ORIGIN by setting the DF_ORIGIN flag. An implementation may reject an attempt to use $ORIGIN within a dlopen() call from an object that did not set the DF_ORIGIN flag and did not use $ORIGIN within its dynamic array.

For security, the dynamic linker does not allow use of $ORIGIN substitution sequences for set-user and set-group ID programs. For such sequences that appear within strings specified by DT_RUNPATH dynamic array entries, the specific search path containing the $ORIGIN sequence is ignored (though other search paths in the same string are processed). $ORIGIN sequences within a DT_NEEDED entry or path passed as a parameter to dlopen() are treated as errors. The same restrictions may be applied to processes that have more than minimal privileges on systems with installed extended security mechanisms.

Global Offset Table

This section requires processor-specific information. The *System V Application Binary Interface* supplement for the desired processor describes the details.

Procedure Linkage Table

This section requires processor-specific information. The *System V Application Binary Interface* supplement for the desired processor describes the details.

Hash Table

A hash table of Elf32_Word objects supports symbol table access. The same table layout is used for both the 32-bit and 64-bit file class. Labels appear below to help explain the hash table organization, but they are not part of the specification.

Figure 5-12: Symbol Hash Table

nbucket
nchain
bucket[0]
. . .
bucket[nbucket-1]
chain[0]
. . .
chain[nchain-1]

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 (shown below) 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.

Figure 5-13: Hashing Function

```
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 dynamic linker has built the process image and performed the relocations, each shared object and the executable file get the opportunity to execute some initialization functions. All shared object initializations happen before the executable file gains control.

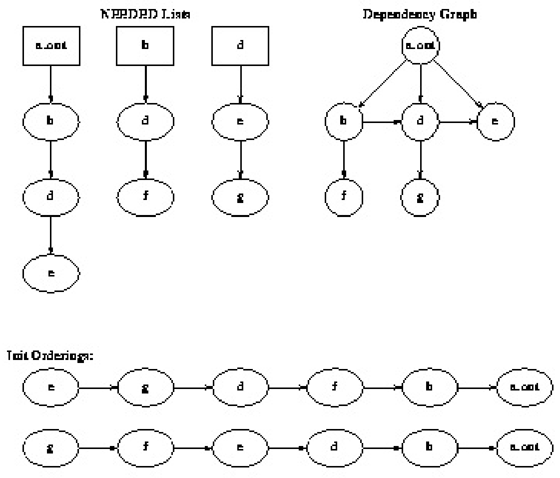
Before the initialization functions for any object A is called, the initialization functions for any other objects that object A depends on are called. For these purposes, an object A depends on another object B, if B appears in A's list of needed objects (recorded in the DT_NEEDED entries of the dynamic structure). The order of initialization for circular dependencies is undefined.

The initialization of objects occurs by recursing through the needed entries of each object. The initialization functions for an object are invoked after the needed entries for that object have been processed. The order of processing among the entries of a particular list of needed objects is unspecified.

Each processor supplement may optionally further restrict the algorithm used to determine the order of initialization. Any such restriction, however, may not conflict with the rules described by this specification.

The following example illustrates two of the possible correct orderings which can be generated for the example NEEDED lists. In this example the *a.out* is dependent on b, d, and e. b is dependent on d and f, while d is dependent on e and g. From this information a dependency graph can be drawn. The above algorithm on initialization will then allow the following specified initialization orderings among others.

Figure 5-14: Initialization Ordering Example



Similarly, shared objects and executable files may have termination functions, which are executed with the atexit(BA_OS) mechanism after the base process begins its termination sequence. The termination functions for any object A must be called before the termination functions for any other objects that object A depends on. For these purposes, an object A depends on another object B, if B appears in A's list of needed objects (recorded in the DT_NEEDED entries of the dynamic structure). The order of termination for circular dependencies is undefined.

Finally, an executable file may have pre-initialization functions. These functions are executed after the dynamic linker has built the process image and performed relocations but before any shared object initialization functions. Pre-initialization functions are not permitted in shared objects.

Complete initialization of system libraries may not have occurred when pre-initializations are executed, so some features of the system may not be available to pre-initialization code. In general, use of pre-initialization code can be considered portable only if it has no dependencies on system libraries.

The dynamic linker ensures that it will not execute any initialization, pre-initialization, or termination functions more than once.

Shared objects designate their initialization and termination code in one of two ways. First, they may specify the address of a function to execute via the DT_INIT and DT_FINI entries in the dynamic structure, described in ``Dynamic Section'' above.

Note that the address of a function need not be the same as a pointer to a function as defined by the processor supplement.

Shared objects may also (or instead) specify the address and size of an array of function pointers. Each element of this array is a pointer to a function to be executed by the dynamic linker. Each array element is the size of a pointer in the programming model followed by the object containing the array. The address of the array of initialization function pointers is specified by the DT_INIT_ARRAY entry in the dynamic structure. Similarly, the address of the array of pre-initialization functions is specified by DT_PREINIT_ARRAY and the address of the array of termination functions is specified by DT_FINI_ARRAY. The size of each array is specified by the DT_INIT_ARRAYSZ, DT_PREINIT_ARRAYSZ, and DT_FINI_ARRAYSZ entries.

The addresses contained in the initialization and termination arrays are function pointers as defined by the processor supplement for each processor. On some architectures, a function pointer may not contain the actual address of the function.

The functions pointed to in the arrays specified by DT_INIT_ARRAY and by DT_PREINIT_ARRAY are executed by the dynamic linker in the same order in which their addresses appear in the array; those specified by DT_FINI_ARRAY are executed in reverse order.

If an object contains both DT_INIT and DT_INIT_ARRAY entries, the function referenced by the DT_INIT entry is processed before those referenced by the DT_INIT_ARRAY entry for that object. If an object contains both DT_FINI and DT_FINI_ARRAY entries, the functions referenced by the DT_FINI_ARRAY entry are processed before the one referenced by the DT_FINI entry for that object.

Although the atexit(BA_OS) 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 [see exit(BA_OS)] or if the process dies because it received a signal that it neither caught nor ignored.

The processor supplement for each processor specifies whether the dynamic linker is responsible for calling the executable file's initialization function or registering the executable file's termination function with atexit(BA_OS). Termination functions specified by users via the atexit(BA_OS) mechanism must be executed before any termination functions of shared objects.
