I

Executable and Linkable Format (ELF)

Contents

Figures and Tables

Preface

ELF: Executable and Linking Format

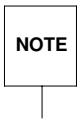
The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). The Tool Interface Standards committee (TIS) has selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems.

The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments. This should reduce the number of different interface implementations, thereby reducing the need for recoding and recompiling code.

About This Document

This document is intended for developers who are creating object or executable files on various 32-bit environment operating systems. It is divided into the following three parts:

- Part 1, ''Object Files'' describes the ELF object file format for the three main types of object files.
- Part 2, ''Program Loading and Dynamic Linking'' describes the object file information and system actions that create running programs.
- Part 3, ''C Library'' lists the symbols contained in `libsys`, the standard ANSI C and `libc` routines, and the global data symbols required by the `libc` routines.



References to X86 architecture have been changed to Intel Architecture.

1 OBJECT FILES

Introduction

Part 1 describes the iABI 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`(SD_CMD)] may process it 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.

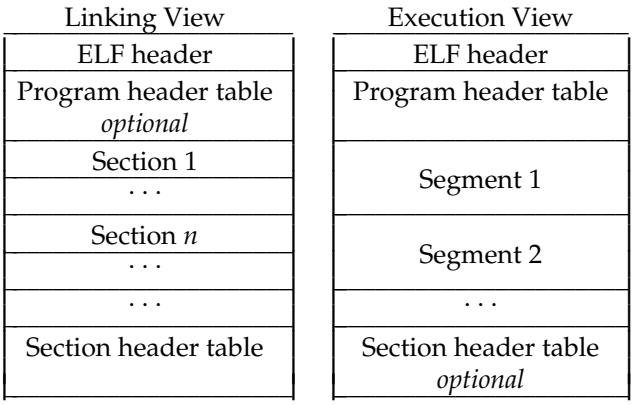
Created by the assembler and link editor, object files are binary representations of programs intended to execute directly on a processor. Programs that require other abstract machines, such as shell scripts, are excluded.

After the introductory material, Part 1 focuses on the file format and how it pertains to building programs. Part 2 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 these activities. Figure 1-1 shows an object file's organization.

Figure 1-1: Object File Format



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 Part 1. Part 2 discusses *segments* and the program execution view of the file.

A *program header table*, if present, tells the system how to create a process image. Files used to build a process image (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, etc. Files used during linking must have a section header table; other object files may or may not have one.

NOTE

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

Data Representation

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

Figure 1-2: 32-Bit Data Types

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

All data structures that the object file format defines follow the ''natural'' size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, etc. 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 1-3: 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;
```

`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.

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

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

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

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

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

`e_version` This member identifies the object file version.

Name	Value	Meaning
`EV_NONE`	`0`	Invalid version
`EV_CURRENT`	`1`	Current version

The value `1` signifies the original file format; extensions will create new versions with higher numbers. The value of `EV_CURRENT`, though given as `1` above, will change as necessary to reflect the current version number.

`e_entry` This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

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

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

`e_flags` This member holds processor-specific flags associated with the file. Flag names take the form `EF_`*machine_flag*. See ''Machine Information'' for flag definitions.

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

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

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

`e_shentsize` This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

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

`e_shstrndx` This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`. See ''Sections'' and ''String Table'' below for more information.

ELF Identification

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

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

Figure 1-4: `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_PAD`	7	Start of padding bytes
`EI_NIDENT`	16	Size of `e_ident[]`

These indexes access bytes that hold the following values.

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

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

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

Name	Value	Meaning
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class ELFCLASS32 supports machines with files and virtual address spaces up to 4 gigabytes; it uses the basic types defined above.

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

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

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

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

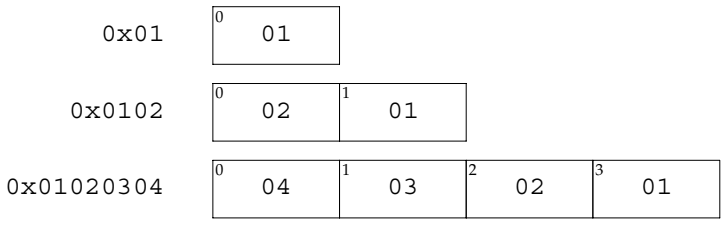
EI_VERSION Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.

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

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

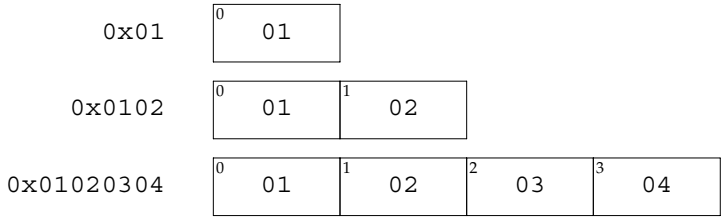
Encoding ELFDATA2LSB specifies 2's complement values, with the least significant byte occupying the lowest address.

Figure 1-5: Data Encoding ELFDATA2LSB



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

Figure 1-6: Data Encoding ELFDATA2MSB



Machine Information

For file identification in e_ident, the 32-bit Intel Architecture requires the following values.

Figure 1-7: 32-bit Intel Architecture Identification, e_ident

Position	Value
e_ident[EI_CLASS]	ELFCLASS32
e_ident[EI_DATA]	ELFDATA2LSB

Processor identification resides in the ELF header's e_machine member and must have the value EM_386.

The ELF header's e_flags member holds bit flags associated with the file. The 32-bit Intel Architecture defines no flags; so this member contains zero.

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

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

Figure 1-8: Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff

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

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

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

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

`SHN_ABS` This value specifies absolute values for the corresponding reference. For example, symbols defined relative to section number `SHN_ABS` have absolute values and are not affected by relocation.

`SHN_COMMON` Symbols defined relative to this section are common symbols, such as FORTRAN `COMMON` or unallocated C external variables.

`SHN_HIRESERVE` This value specifies the upper bound of the range of reserved indexes. The system reserves indexes between `SHN_LORESERVE` and `SHN_HIRESERVE`, inclusive; the values do not reference the section header table. That is, the section header table does *not* contain entries for the reserved indexes.

Sections contain all information in an object file, except the ELF header, the program header table, and the section header table. Moreover, object files' sections satisfy several conditions.

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may have inactive space. The various headers and the sections might not ''cover'' every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure.

Figure 1-9: 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;
```

`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.

`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. That is, the value of `sh_addr` must be congruent to 0, modulo the value of `sh_addralign`. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

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

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

Figure 1-10: Section Types, `sh_type`

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

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

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

`SHT_SYMTAB` and `SHT_DYNSYM`
These sections hold a symbol table. 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. An object file may have multiple relocation sections. See ''Relocation'' below for details.

SHT_HASH The section holds a symbol hash table. All objects participating in dynamic linking must contain a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See ''Hash Table'' in Part 2 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 Part 2 for details.

SHT_NOTE The section holds information that marks the file in some way. See ''Note Section'' in Part 2 for details.

SHT_NOBITS A section of this type occupies no space in the file but otherwise resembles SHT_PROGBITS. Although this section contains no bytes, the sh_offset member contains the conceptual file offset.

SHT_REL The section holds relocation entries without explicit addends, such as type Elf32_Rel for the 32-bit class of object files. An object file may have multiple relocation sections. See ''Relocation'' below for details.

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

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

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

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

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

Figure 1-11: Section Header Table Entry: Index 0

Name	Value	Note
sh_name	0	No name
sh_type	SHT_NULL	Inactive
sh_flags	0	No flags
sh_addr	0	No address
sh_offset	0	No file offset
sh_size	0	No size

Figure 1-11: Section Header Table Entry: Index 0 (continued)

sh_link	SHN_UNDEF	No link information
sh_info	0	No auxiliary information
sh_addralign	0	No alignment
sh_entsize	0	No entries

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

Figure 1-12: Section Attribute Flags, sh_flags

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

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

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

SHF_ALLOC The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.

SHF_EXECINSTR The section contains executable machine instructions.

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

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

Figure 1-13: `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).
other	SHN_UNDEF	0

Special Sections

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

Figure 1-14: 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
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	see below
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.plt	SHT_PROGBITS	see below
.rel*name*	SHT_REL	see below

Figure 1-14: Special Sections (continued)

.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
.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.

.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 Part 2 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 Part 2 for more information.

.dynsym This section holds the dynamic linking symbol table, as ''Symbol Table'' describes. See Part 2 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.

.got This section holds the global offset table. See ''Special Sections'' in Part 1 and ''Global Offset Table'' in Part 2 for more information.

.hash This section holds a symbol hash table. See ''Hash Table'' in Part 2 for more information.

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

.interp This section holds the path name of a program interpreter. If the file has a loadable segment that includes the section, the section's attributes will include the SHF_ALLOC bit; otherwise, that bit will be off. See Part 2 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 Part 2 describes.

`.plt` This section holds the procedure linkage table. See ''Special Sections'' in Part 1 and ''Procedure Linkage Table'' in Part 2 for more information.

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

`.rodata` and `.rodata1`
These sections hold read-only data that typically contribute to a non-writable segment in the process image. See ''Program Header'' in Part 2 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 section describes. If the file has a loadable segment that includes the symbol table, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.

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

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

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for `e_machine`. For 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`	

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 1-15: String Table Indexes

Index	String
0	*none*
1	`name.`
7	`Variable`
11	`able`
16	`able`
24	*null string*

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

Symbol Table

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

Name	Value
STN_UNDEF	0

A symbol table entry has the following format.

Figure 1-16: 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;
```

`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.

NOTE 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, etc.; 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.

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

st_other This member currently holds 0 and has no defined meaning.

st_shndx Every symbol table entry is ''defined'' in relation to some section; this member holds the relevant section header table index. As Figure 1-7 and the related text describe, some section indexes indicate special meanings.

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

Figure 1-17: Symbol Binding, ELF32_ST_BIND

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

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

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

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

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

Global and weak symbols differ in two major ways.

- 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 (i.e., 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, 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.

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 1-18: Symbol Types, `ELF32_ST_TYPE`

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

`STT_NOTYPE` The symbol's type is not specified.

`STT_OBJECT` The symbol is associated with a data object, such as a variable, an array, etc.

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

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

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

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

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

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

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

`SHN_COMMON` The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's `sh_addralign` member. That is, the link editor will allocate the storage for the symbol at an address that is a multiple of `st_value`. The symbol's size tells how many bytes are required.

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

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

Figure 1-19: Symbol Table Entry: Index 0

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

Symbol Values

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

- In relocatable files, st_value holds alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value holds a section offset for a defined symbol. That is, st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable and shared object files, st_value holds a virtual address. To make these files' symbols more useful for the 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 allow efficient access by the appropriate programs.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. *Relocation entries* are these data.

Figure 1-20: 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;
```

`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. When the text refers to a relocation entry's relocation type or symbol table index, it means the result of applying `ELF32_R_TYPE` or `ELF32_R_SYM`, respectively, to the entry's `r_info` member.

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

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

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

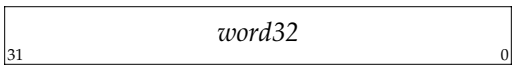
- In relocatable files, `r_offset` holds a section offset. That is, the relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, `r_offset` holds a virtual address. To make these files' relocation entries more useful for the 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.

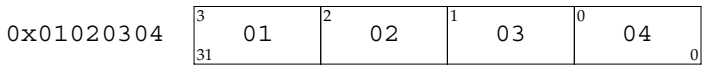
Relocation Types

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners).

Figure 1-21: Relocatable Fields



word32 This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the 32-bit Intel Architecture.



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

`A` This means the addend used to compute the value of the relocatable field.

`B` This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different.

G — This means the offset into the global offset table at which the address of the relocation entry's symbol will reside during execution. See ''Global Offset Table'' in Part 2 for more information.

GOT — This means the address of the global offset table. See ''Global Offset Table'' in Part 2 for more information.

L — This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See ''Procedure Linkage Table'' in Part 2 for more information.

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

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

A relocation entry's `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The SYSTEM V architecture uses only `Elf32_Rel` relocation entries, the field to be relocated holds the addend. In all cases, the addend and the computed result use the same byte order.

Figure 1-22: Relocation Types

Name	Value	Field	Calculation
R_386_NONE	0	none	none
R_386_32	1	*word32*	S + A
R_386_PC32	2	*word32*	S + A - P
R_386_GOT32	3	*word32*	G + A - P
R_386_PLT32	4	*word32*	L + A - P
R_386_COPY	5	none	none
R_386_GLOB_DAT	6	*word32*	S
R_386_JMP_SLOT	7	*word32*	S
R_386_RELATIVE	8	*word32*	B + A
R_386_GOTOFF	9	*word32*	S + A - GOT
R_386_GOTPC	10	*word32*	GOT + A - P

Some relocation types have semantics beyond simple calculation.

R_386_GOT32 — This relocation type computes the distance from the base of the global offset table to the symbol's global offset table entry. It additionally instructs the link editor to build a global offset table.

R_386_PLT32 — This relocation type computes the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_386_COPY — The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_386_GLOB_DAT	This relocation type is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.
R_3862_JMP_SLOT	The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a procedure linkage table entry. The dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address [see ''Procedure Linkage Table'' in Part 2].
R_386_RELATIVE	The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.
R_386_GOTOFF	This relocation type computes the difference between a symbol's value and the address of the global offset table. It additionally instructs the link editor to build the global offset table.
R_386_GOTPC	This relocation type resembles R_386_PC32, except it uses the address of the global offset table in its calculation. The symbol referenced in this relocation normally is _GLOBAL_OFFSET_TABLE_, which additionally instructs the link editor to build the global offset table.

Index

Index

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z