13 Object Files

Introduction

This chapter describes the executable and linking format (ELF) of the object files produced by the C compilation system. The first section, "Program Linking," focuses on how the format pertains to building programs. The second section, "Program Execution," focuses on how the format pertains to loading programs. For background, see the "Link Editing" section in Chapter 2.

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()` creates a program's process image.
- A *shared object file* holds code and data suitable for linking in two contexts. First, the link editor 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.

Programs manipulate object files with the functions contained in the ELF access library, `libelf`. Subsection 3E of the *Programmer's Reference Manual* describes its contents.

File Format

As indicated, object files participate in program linking and program execution. For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities. The figure below shows an object file's organization.

Figure 13-1: Object File Format

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

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 in the first part of this chapter. The second part of this chapter discusses *segments* and the program execution view of the file.

A *program header table,* if present, tells the system how to create a process image. Files used to build a process image (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 forth. 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 13-2: 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, 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.

Program Linking

This section describes the object file information and system actions that create static program representations from relocatable files and shared objects.

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 13-3: 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

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 "ELF Header Flags" 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 "Section Header" 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 13-4: 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.

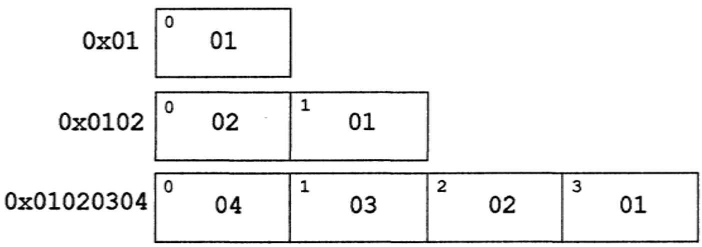
EI_VERSION — Byte e_ident[EI_VERSION] specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.

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.

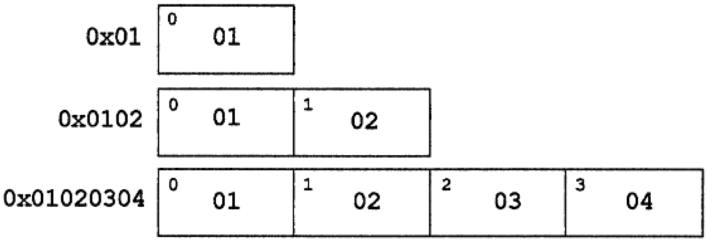
Encoding ELFDATA2LSB specifies 2's complement values, with the least significant byte occupying the lowest address.

Figure 13-5: Data Encoding ELFDATA2LSB



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

Figure 13-6: Data Encoding ELFDATA2MSB



ELF Header Flags (3B2 Computer-Specific)

For file identification in `e_ident`, the WE 32100 requires the following values.

Figure 13-7: WE 32100 Identification, e_ident

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS32`
`e_ident[EI_DATA]`	`ELFDATA2MSB`

Processor identification resides in the ELF header's `e_machine` member and must have the value `1`, defined as the name `EM_M32`.

The ELF header's `e_flags` member holds bit flags associated with the file.

Figure 13-8: Processor-Specific Flags, e_flags

Name	Value
EF_M32_MAU	0x1

`EF_M32_MAU` If this bit is asserted, the program in the file must execute on a machine with a Math Acceleration Unit. Otherwise, the program will execute on a machine with or without a MAU.

ELF Header Flags (6386 Computer-Specific)

For file identification in `e_ident`, the 6386 computer requires the following values.

Figure 13-9: 6386 Computer 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 `3`, defined as the name `EM_386`.

The ELF header's `e_flags` member holds bit flags associated with the file. The 6386 computer defines no flags; so this member contains zero.

Section Header

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 13-10: 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 13-11: 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 13-12: 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

Figure 13-12: Section Types, sh_type (continued)

Name	Value
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. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See "Hash Table" in the second part of this chapter 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 the second part of this chapter for details.

SHT_NOTE — The section holds information that marks the file in some way. See "Note Section" in the second part of this chapter 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.

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.