

**NAME**

elf – format of Executable and Linking Format (ELF) files

**SYNOPSIS**

```
#include <elf.h>
```

**DESCRIPTION**

The header file `<elf.h>` defines the format of ELF executable binary files. Amongst these files are normal executable files, relocatable object files, core files, and shared objects.

An executable file using the ELF file format consists of an ELF header, followed by a program header table or a section header table, or both. The ELF header is always at offset zero of the file. The program header table and the section header table's offset in the file are defined in the ELF header. The two tables describe the rest of the particularities of the file.

This header file describes the above mentioned headers as C structures and also includes structures for dynamic sections, relocation sections and symbol tables.

**Basic types**

The following types are used for N-bit architectures (N=32,64, *ElfN* stands for *Elf32* or *Elf64*, *uintN\_t* stands for *uint32\_t* or *uint64\_t*):

<code>ElfN_Addr</code>	Unsigned program address, <code>uintN_t</code>
<code>ElfN_Off</code>	Unsigned file offset, <code>uintN_t</code>
<code>ElfN_Section</code>	Unsigned section index, <code>uint16_t</code>
<code>ElfN_Versym</code>	Unsigned version symbol information, <code>uint16_t</code>
<code>Elf_Byte</code>	unsigned char
<code>ElfN_Half</code>	<code>uint16_t</code>
<code>ElfN_Sword</code>	<code>int32_t</code>
<code>ElfN_Word</code>	<code>uint32_t</code>
<code>ElfN_Sxword</code>	<code>int64_t</code>
<code>ElfN_Xword</code>	<code>uint64_t</code>

(Note: the \*BSD terminology is a bit different. There, *Elf64\_Half* is twice as large as *Elf32\_Half*, and *Elf64\_Quarter* is used for `uint16_t`. In order to avoid confusion these types are replaced by explicit ones in the below.)

All data structures that the 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 on.

**ELF header (Ehdr)**

The ELF header is described by the type *Elf32\_Ehdr* or *Elf64\_Ehdr*:

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
```

```

        uint16_t      e_shstrndx;
    } ElfN_Ehdr;

```

The fields have the following meanings:

*e\_ident* This array of bytes specifies how to interpret the file, independent of the processor or the file's remaining contents. Within this array everything is named by macros, which start with the prefix **EI\_** and may contain values which start with the prefix **ELF**. The following macros are defined:

**EI\_MAG0**

The first byte of the magic number. It must be filled with **ELFMAG0**. (0: 0x7f)

**EI\_MAG1**

The second byte of the magic number. It must be filled with **ELFMAG1**. (1: 'E')

**EI\_MAG2**

The third byte of the magic number. It must be filled with **ELFMAG2**. (2: 'L')

**EI\_MAG3**

The fourth byte of the magic number. It must be filled with **ELFMAG3**. (3: 'F')

**EI\_CLASS**

The fifth byte identifies the architecture for this binary:

**ELFCLASSNONE**

This class is invalid.

**ELFCLASS32** This defines the 32-bit architecture. It supports machines with files and virtual address spaces up to 4 Gigabytes.

**ELFCLASS64** This defines the 64-bit architecture.

**EI\_DATA**

The sixth byte specifies the data encoding of the processor-specific data in the file. Currently, these encodings are supported:

**ELFDATANONE**

Unknown data format.

**ELFDATA2LSB**

Two's complement, little-endian.

**ELFDATA2MSB**

Two's complement, big-endian.

**EI\_VERSION**

The seventh byte is the version number of the ELF specification:

**EV\_NONE** Invalid version.

**EV\_CURRENT**

Current version.

**EI\_OSABI**

The eighth byte identifies the operating system and ABI to which the object is targeted. Some fields in other ELF structures have flags and values that have platform-specific meanings; the interpretation of those fields is determined by the value of this byte. For example:

**ELFOSABI\_NONE** Same as ELFOSABI\_SYSV

**ELFOSABI\_SYSV** UNIX System V ABI

**ELFOSABI\_HPUX** HP-UX ABI

**ELFOSABI\_NETBSD** NetBSD ABI

**ELFOSABI\_LINUX** Linux ABI

**ELFOSABI\_SOLARIS** Solaris ABI

**ELFOSABI\_IRIX** IRIX ABI

**ELFOSABI\_FREEBSD**

FreeBSD ABI

**ELFOSABI\_TRU64**

TRU64 UNIX ABI

**ELFOSABI\_ARM**

ARM architecture ABI

**ELFOSABI\_STANDALONE**

Stand-alone (embedded) ABI

**EI\_ABIVERSION**

The ninth byte 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\_OS-ABI** field. Applications conforming to this specification use the value 0.

**EI\_PAD** Start of padding. These bytes are reserved and set to zero. Programs which read them should ignore them. The value for **EI\_PAD** will change in the future if currently unused bytes are given meanings.

**EI\_NIDENT**The size of the *e\_ident* array.

*e\_type* This member of the structure identifies the object file type:

**ET\_NONE** An unknown type.  
**ET\_REL** A relocatable file.  
**ET\_EXEC** An executable file.  
**ET\_DYN** A shared object.  
**ET\_CORE** A core file.

*e\_machine* This member specifies the required architecture for an individual file. For example:

**EM\_NONE** An unknown machine  
**EM\_M32** AT&T WE 32100  
**EM\_SPARC** Sun Microsystems SPARC  
**EM\_386** Intel 80386  
**EM\_68K** Motorola 68000  
**EM\_88K** Motorola 88000  
**EM\_860** Intel 80860  
**EM\_MIPS** MIPS RS3000 (big-endian only)  
**EM\_PARISC** HP/PA  
**EM\_SPARC32PLUS**  
SPARC with enhanced instruction set  
**EM\_PPC** PowerPC  
**EM\_PPC64** PowerPC 64-bit  
**EM\_S390** IBM S/390  
**EM\_ARM** Advanced RISC Machines  
**EM\_SH** Renesas SuperH  
**EM\_SPARCV9** SPARC v9 64-bit  
**EM\_IA\_64** Intel Itanium  
**EM\_X86\_64** AMD x86-64  
**EM\_VAX** DEC Vax

*e\_version* This member identifies the file version:

**EV\_NONE** Invalid version  
**EV\_CURRENT** Current 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’. Currently, no flags have been defined.
- 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, *e\_phnum* holds the value zero.
- If the number of entries in the program header table is larger than or equal to **PN\_XNUM** (0xffff), this member holds **PN\_XNUM** (0xffff) and the real number of entries in the program header table is held in the *sh\_info* member of the initial entry in section header table. Otherwise, the *sh\_info* member of the initial entry contains the value zero.
- PN\_XNUM**  
This is defined as 0xffff, the largest number *e\_phnum* can have, specifying where the actual number of program headers is assigned.
- e\_shentsize* This member holds a sections 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 of zero.
- If the number of entries in the section header table is larger than or equal to **SHN\_LORESERVE** (0xff00), *e\_shnum* holds the value zero and the real number of entries in the section header table is held in the *sh\_size* member of the initial entry in section header table. Otherwise, the *sh\_size* member of the initial entry in the section header table 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**.
- If the index of section name string table section is larger than or equal to **SHN\_LORESERVE** (0xff00), this member holds **SHN\_XINDEX** (0xffff) and the real index of the section name string table section is held in the *sh\_link* member of the initial entry in section header table. Otherwise, the *sh\_link* member of the initial entry in section header table contains the value zero.

### Program header (Phdr)

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*. 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. The ELF program header is described by the type *Elf32\_Phdr* or *Elf64\_Phdr* depending on the architecture:

```
typedef struct {
    uint32_t    p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    uint32_t    p_filesz;
```

```

    uint32_t    p_memsz;
    uint32_t    p_flags;
    uint32_t    p_align;
} Elf32_Phdr;

typedef struct {
    uint32_t    p_type;
    uint32_t    p_flags;
    Elf64_Off   p_offset;
    Elf64_Addr  p_vaddr;
    Elf64_Addr  p_paddr;
    uint64_t    p_filesz;
    uint64_t    p_memsz;
    uint64_t    p_align;
} Elf64_Phdr;

```

The main difference between the 32-bit and the 64-bit program header lies in the location of the *p\_flags* member in the total struct.

*p\_type* This member of the structure indicates what kind of segment this array element describes or how to interpret the array element's information.

**PT\_NULL** The array element is unused and the other members' values are undefined. This lets the program header 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.

#### **PT\_INTERP**

The array element specifies the location and size of a null-terminated pathname to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects). However it may not occur more than once in a file. If it is present, it must precede any loadable segment entry.

**PT\_NOTE** The array element specifies the location of notes (ElfN\_Nhdr).

**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.

#### **PT\_LOPROC, PT\_HIPROC**

Values in the inclusive range [PT\_LOPROC, PT\_HIPROC] are reserved for processor-specific semantics.

#### **PT\_GNU\_STACK**

GNU extension which is used by the Linux kernel to control the state of the stack via the flags set in the *p\_flags* member.

- p\_offset* This member holds the offset from the beginning of the file at which the first byte of the segment resides.
- p\_vaddr* This member holds 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. Under BSD this member is not used and must be zero.
- p\_filesz* This member holds the number of bytes in the file image of the segment. It may be zero.
- p\_memsz* This member holds the number of bytes in the memory image of the segment. It may be zero.
- p\_flags* This member holds a bit mask of flags relevant to the segment:
- PF\_X** An executable segment.
  - PF\_W** A writable segment.
  - PF\_R** A readable segment.
- A text segment commonly has the flags **PF\_X** and **PF\_R**. A data segment commonly has **PF\_W** and **PF\_R**.
- p\_align* This member holds the value to which the segments are aligned in memory and in the file. Loadable process segments must have congruent values for *p\_vaddr* and *p\_offset*, modulo the page size. Values of zero and one mean no alignment is required. Otherwise, *p\_align* should be a positive, integral power of two, and *p\_vaddr* should equal *p\_offset*, modulo *p\_align*.

### Section header (Shdr)

A 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. The ELF header's *e\_shoff* member gives the byte offset from the beginning of the file to the section header table. *e\_shnum* holds the number of entries the section header table contains. *e\_shentsize* holds the size in bytes of each entry.

A section header table index is a subscript into this array. Some section header table indices are reserved: the initial entry and the indices between **SHN\_LORESERVE** and **SHN\_HIRESERVE**. The initial entry is used in ELF extensions for *e\_phnum*, *e\_shnum* and *e\_shstrndx*; in other cases, each field in the initial entry is set to zero. An object file does not have sections for these special indices:

#### **SHN\_UNDEF**

This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference.

#### **SHN\_LORESERVE**

This value specifies the lower bound of the range of reserved indices.

#### **SHN\_LOPROC, SHN\_HIPROC**

Values greater in the inclusive range [**SHN\_LOPROC**, **SHN\_HIPROC**] are reserved for processor-specific semantics.

#### **SHN\_ABS**

This value specifies the absolute value for the corresponding reference. For example, a symbol defined relative to section number **SHN\_ABS** has an absolute value and is 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 indices. The system reserves indices between **SHN\_LORESERVE** and **SHN\_HIRESERVE**, inclusive. The section header table does not contain entries for the reserved indices.

The section header has the following structure:

```
typedef struct {
    uint32_t    sh_name;
    uint32_t    sh_type;
```

```

uint32_t    sh_flags;
Elf32_Addr sh_addr;
Elf32_Off  sh_offset;
uint32_t    sh_size;
uint32_t    sh_link;
uint32_t    sh_info;
uint32_t    sh_addralign;
uint32_t    sh_entsize;
} Elf32_Shdr;

typedef struct {
uint32_t    sh_name;
uint32_t    sh_type;
uint64_t    sh_flags;
Elf64_Addr sh_addr;
Elf64_Off  sh_offset;
uint64_t    sh_size;
uint32_t    sh_link;
uint32_t    sh_info;
uint64_t    sh_addralign;
uint64_t    sh_entsize;
} Elf64_Shdr;

```

No real differences exist between the 32-bit and 64-bit section headers.

*sh\_name* This member specifies the name of the section. Its value is an index into the section header string table section, giving the location of a null-terminated string.

*sh\_type* This member categorizes the section's contents and semantics.

**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**

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

**SHT\_SYMTAB** This section holds a symbol table. 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. An object file can also contain a **SHT\_DYNSYM** section.

**SHT\_STRTAB** This section holds a string table. An object file may have multiple string table sections.

**SHT\_RELA** This section holds relocation entries with explicit addends, such as type *Elf32\_Rela* for the 32-bit class of object files. An object may have multiple relocation sections.

**SHT\_HASH** This section holds a symbol hash table. An object participating in dynamic linking must contain a symbol hash table. An object file may have only one hash table.

#### **SHT\_DYNAMIC**

This section holds information for dynamic linking. An object file may have only one dynamic section.

**SHT\_NOTE** This section holds notes (*ElfN\_Nhdr*).

**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** This section holds relocation offsets without explicit addends, such as type *Elf32\_Rel* for the 32-bit class of object files. An object file may have multiple relocation sections.
- SHT\_SHLIB** This section is reserved but has unspecified semantics.
- SHT\_DYNSYM** This section holds a minimal set of dynamic linking symbols. An object file can also contain a **SHT\_SYMTAB** section.
- SHT\_LOPROC, SHT\_HIPROC**  
Values in the inclusive range [**SHT\_LOPROC**, **SHT\_HIPROC**] are reserved for processor-specific semantics.
- SHT\_LOUSER** This value specifies the lower bound of the range of indices reserved for application programs.
- SHT\_HIUSER** This value specifies the upper bound of the range of indices 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.
- sh\_flags* Sections support one-bit flags that describe miscellaneous attributes. 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** This section contains data that should be writable during process execution.
- SHF\_ALLOC** This 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**  
This section contains executable machine instructions.
- SHF\_MASKPROC**  
All bits included in this mask are reserved for processor-specific semantics.
- sh\_addr* If this section appears in the memory image of a process, this member holds the address at which the section's first byte should reside. Otherwise, the member contains zero.
- sh\_offset* This member's value holds the byte offset from the beginning of the file to the first byte in the section. One section type, **SHT\_NOBITS**, occupies no space in the file, and its *sh\_offset* member locates the conceptual placement in the file.
- sh\_size* This member holds 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.
- sh\_info* This member holds extra information, whose interpretation depends on the section type.
- sh\_addralign*  
Some sections have address alignment constraints. 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 zero, modulo the value of *sh\_addralign*. Only zero and positive integral powers of two are allowed. The value 0 or 1 means that the section has no alignment constraints.
- sh\_entsize* Some sections hold a table of fixed-sized entries, such as a symbol table. For such a section, this member gives the size in bytes for each entry. This member contains zero if the section does not hold a table of fixed-size entries.

Various sections hold program and control information:



- .bss* This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. This section is of type **SHT\_NOBITS**. The attribute types are **SHF\_ALLOC** and **SHF\_WRITE**.
- .comment* This section holds version control information. This section is of type **SHT\_PROGBITS**. No attribute types are used.
- .ctors* This section holds initialized pointers to the C++ constructor functions. This section is of type **SHT\_PROGBITS**. The attribute types are **SHF\_ALLOC** and **SHF\_WRITE**.
- .data* This section holds initialized data that contribute to the program's memory image. This section is of type **SHT\_PROGBITS**. The attribute types are **SHF\_ALLOC** and **SHF\_WRITE**.
- .data1* This section holds initialized data that contribute to the program's memory image. This section is of type **SHT\_PROGBITS**. The attribute types are **SHF\_ALLOC** and **SHF\_WRITE**.
- .debug* This section holds information for symbolic debugging. The contents are unspecified. This section is of type **SHT\_PROGBITS**. No attribute types are used.
- .dtors* This section holds initialized pointers to the C++ destructor functions. This section is of type **SHT\_PROGBITS**. The attribute types are **SHF\_ALLOC** and **SHF\_WRITE**.
- .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. This section is of type **SHT\_DYNAMIC**. See the attributes above.
- .dynstr* This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. This section is of type **SHT\_STRTAB**. The attribute type used is **SHF\_ALLOC**.
- .dynsym* This section holds the dynamic linking symbol table. This section is of type **SHT\_DYNSYM**. The attribute used is **SHF\_ALLOC**.
- .fni* This section holds executable instructions that contribute to the process termination code. When a program exits normally the system arranges to execute the code in this section. This section is of type **SHT\_PROGBITS**. The attributes used are **SHF\_ALLOC** and **SHF\_EXECINSTR**.
- .gnu.version* This section holds the version symbol table, an array of *ElfN\_Half* elements. This section is of type **SHT\_GNU\_versym**. The attribute type used is **SHF\_ALLOC**.
- .gnu.version\_d* This section holds the version symbol definitions, a table of *ElfN\_Verdef* structures. This section is of type **SHT\_GNU\_verdef**. The attribute type used is **SHF\_ALLOC**.
- .gnu.version\_r* This section holds the version symbol needed elements, a table of *ElfN\_Verneed* structures. This section is of type **SHT\_GNU\_versym**. The attribute type used is **SHF\_ALLOC**.
- .got* This section holds the global offset table. This section is of type **SHT\_PROGBITS**. The attributes are processor-specific.
- .hash* This section holds a symbol hash table. This section is of type **SHT\_HASH**. The attribute used is **SHF\_ALLOC**.
- .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. This section is of type **SHT\_PROGBITS**. The attributes used are **SHF\_ALLOC** and **SHF\_EXECINSTR**.
- .interp* This section holds the pathname 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. This section is of type **SHT\_PROGBITS**.

- .line* This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code. The contents are unspecified. This section is of type **SHT\_PROGBITS**. No attribute types are used.
- .note* This section holds various notes. This section is of type **SHT\_NOTE**. No attribute types are used.
- .note.ABI-tag*  
This section is used to declare the expected run-time ABI of the ELF image. It may include the operating system name and its run-time versions. This section is of type **SHT\_NOTE**. The only attribute used is **SHF\_ALLOC**.
- .note.gnu.build-id*  
This section is used to hold an ID that uniquely identifies the contents of the ELF image. Different files with the same build ID should contain the same executable content. See the **--build-id** option to the GNU linker (**ld** (1)) for more details. This section is of type **SHT\_NOTE**. The only attribute used is **SHF\_ALLOC**.
- .note.GNU-stack*  
This section is used in Linux object files for declaring stack attributes. This section is of type **SHT\_PROGBITS**. The only attribute used is **SHF\_EXECINSTR**. This indicates to the GNU linker that the object file requires an executable stack.
- .note.openbsd.ident*  
OpenBSD native executables usually contain this section to identify themselves so the kernel can bypass any compatibility ELF binary emulation tests when loading the file.
- .plt* This section holds the procedure linkage table. This section is of type **SHT\_PROGBITS**. The attributes are processor-specific.
- .relNAME* This section holds relocation information as described below. If the file has a loadable segment that includes relocation, the section's attributes will include the **SHF\_ALLOC** bit. Otherwise, the bit will be off. By convention, "NAME" is supplied by the section to which the relocations apply. Thus a relocation section for **.text** normally would have the name **.rel.text**. This section is of type **SHT\_REL**.
- .relaNAME*  
This section holds relocation information as described below. If the file has a loadable segment that includes relocation, the section's attributes will include the **SHF\_ALLOC** bit. Otherwise, the bit will be off. By convention, "NAME" is supplied by the section to which the relocations apply. Thus a relocation section for **.text** normally would have the name **.rela.text**. This section is of type **SHT\_RELA**.
- .rodata* This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type **SHT\_PROGBITS**. The attribute used is **SHF\_ALLOC**.
- .rodata1* This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type **SHT\_PROGBITS**. The attribute used is **SHF\_ALLOC**.
- .shstrtab* This section holds section names. This section is of type **SHT\_STRTAB**. No attribute types are used.
- .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, the bit will be off. This section is of type **SHT\_STRTAB**.
- .symtab* This section holds a symbol table. If the file has a loadable segment that includes the symbol table, the section's attributes will include the **SHF\_ALLOC** bit. Otherwise, the bit will be off. This section is of type **SHT\_SYMTAB**.

*.text* This section holds the "text", or executable instructions, of a program. This section is of type **SHT\_PROGBITS**. The attributes used are **SHF\_ALLOC** and **SHF\_EXECINSTR**.

### String and symbol tables

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 byte ('\0'). Similarly, a string table's last byte is defined to hold a null byte, ensuring null termination for all strings.

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.

```
typedef struct {
    uint32_t      st_name;
    Elf32_Addr    st_value;
    uint32_t      st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t      st_shndx;
} Elf32_Sym;

typedef struct {
    uint32_t      st_name;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t      st_shndx;
    Elf64_Addr    st_value;
    uint64_t      st_size;
} Elf64_Sym;
```

The 32-bit and 64-bit versions have the same members, just in a different order.

*st\_name* This member holds an index into the object file's symbol string table, which holds character representations of the symbol names. If the value is nonzero, it represents a string table index that gives the symbol name. Otherwise, the symbol has no name.

*st\_value* This member gives the value of the associated symbol.

*st\_size* Many symbols have associated sizes. This member holds zero if the symbol has no size or an unknown size.

*st\_info* This member specifies the symbol's type and binding attributes:

#### **STT\_NOTYPE**

The symbol's type is not defined.

#### **STT\_OBJECT**

The symbol is associated with a data object.

**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** bindings.

**STT\_FILE** By convention, the symbol's name gives the name of the source file associated with the object file. A file symbol has **STB\_LOCAL** bindings, its section index is **SHN\_ABS**, and it precedes the other **STB\_LOCAL** symbols of the file, if it is present.

#### **STT\_LOPROC, STT\_HIPROC**

Values in the inclusive range [**STT\_LOPROC**, **STT\_HIPROC**] are reserved for processor-specific semantics.

**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 symbol.

**STB\_WEAK**

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

**STB\_LOPROC, STB\_HIPROC**

Values in the inclusive range [STB\_LOPROC, STB\_HIPROC] are reserved for processor-specific semantics.

There are macros for packing and unpacking the binding and type fields:

**ELF32\_ST\_BIND(*info*), ELF64\_ST\_BIND(*info*)**

Extract a binding from an *st\_info* value.

**ELF32\_ST\_TYPE(*info*), ELF64\_ST\_TYPE(*info*)**

Extract a type from an *st\_info* value.

**ELF32\_ST\_INFO(*bind*, *type*), ELF64\_ST\_INFO(*bind*, *type*)**

Convert a binding and a type into an *st\_info* value.

*st\_other* This member defines the symbol visibility.

**STV\_DEFAULT** Default symbol visibility rules. Global and weak symbols are available to other modules; references in the local module can be interposed by definitions in other modules.

**STV\_INTERNAL**

Processor-specific hidden class.

**STV\_HIDDEN** Symbol is unavailable to other modules; references in the local module always resolve to the local symbol (i.e., the symbol can't be interposed by definitions in other modules).

**STV\_PROTECTED**

Symbol is available to other modules, but references in the local module always resolve to the local symbol.

There are macros for extracting the visibility type:

**ELF32\_ST\_VISIBILITY(*other*) or ELF64\_ST\_VISIBILITY(*other*)**

*st\_shndx* Every symbol table entry is "defined" in relation to some section. This member holds the relevant section header table index.

**Relocation entries (Rel & Rela)**

Relocation is the process of connecting symbolic references with symbolic definitions. Relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Relocation entries are these data.

Relocation structures that do not need an addend:

```
typedef struct {
    Elf32_Addr r_offset;
    uint32_t   r_info;
} Elf32_Rel;

typedef struct {
```

```

    Elf64_Addr r_offset;
    uint64_t   r_info;
} Elf64_Rel;

```

Relocation structures that need an addend:

```

typedef struct {
    Elf32_Addr r_offset;
    uint32_t   r_info;
    int32_t    r_addend;
} Elf32_Rela;

typedef struct {
    Elf64_Addr r_offset;
    uint64_t   r_info;
    int64_t    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 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. 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 **ELF[32|64]\_R\_TYPE** or **ELF[32|64]\_R\_SYM**, respectively, to the entry's *r\_info* member.

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

### Dynamic tags (Dyn)

The *.dynamic* section contains a series of structures that hold relevant dynamic linking information. The *d\_tag* member controls the interpretation of *d\_un*.

```

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[];

```

*d\_tag* This member may have any of the following values:

**DT\_NULL** Marks end of dynamic section

**DT\_NEEDED**

String table offset to name of a needed library

<b>DT_PLTRELSZ</b>	Size in bytes of PLT relocation entries
<b>DT_PLTGOT</b>	Address of PLT and/or GOT
<b>DT_HASH</b>	Address of symbol hash table
<b>DT_STRTAB</b>	Address of string table
<b>DT_SYMTAB</b>	Address of symbol table
<b>DT_RELA</b>	Address of Rela relocation table
<b>DT_RELASZ</b>	Size in bytes of the Rela relocation table
<b>DT_RELAENT</b>	Size in bytes of a Rela relocation table entry
<b>DT_STRSZ</b>	Size in bytes of string table
<b>DT_SYMENT</b>	Size in bytes of a symbol table entry
<b>DT_INIT</b>	Address of the initialization function
<b>DT_FINI</b>	Address of the termination function
<b>DT_SONAME</b>	String table offset to name of shared object
<b>DT_RPATH</b>	String table offset to library search path (deprecated)
<b>DT_SYMBOLIC</b>	Alert linker to search this shared object before the executable for symbols
<b>DT_REL</b>	Address of Rel relocation table
<b>DT_RELSZ</b>	Size in bytes of Rel relocation table
<b>DT_RELENT</b>	Size in bytes of a Rel table entry
<b>DT_PLTREL</b>	Type of relocation entry to which the PLT refers (Rela or Rel)
<b>DT_DEBUG</b>	Undefined use for debugging
<b>DT_TEXTREL</b>	Absence of this entry indicates that no relocation entries should apply to a non-writable segment
<b>DT_JMPREL</b>	Address of relocation entries associated solely with the PLT
<b>DT_BIND_NOW</b>	Instruct dynamic linker to process all relocations before transferring control to the executable
<b>DT_RUNPATH</b>	String table offset to library search path
<b>DT_LOPROC, DT_HIPROC</b>	Values in the inclusive range [DT_LOPROC, DT_HIPROC] are reserved for processor-specific semantics

*d\_val* This member represents integer values with various interpretations.

*d\_ptr* This member represents program virtual addresses. When interpreting these addresses, the actual address should be computed based on the original file value and memory base address. Files do not contain relocation entries to fixup these addresses.

#### *\_\_DYNAMIC*

Array containing all the dynamic structures in the *.dynamic* section. This is automatically populated by the linker.

#### Notes (Nhdr)

ELF notes allow for appending arbitrary information for the system to use. They are largely used by core files (*e\_type* of **ET\_CORE**), but many projects define their own set of extensions. For example, the GNU tool chain uses ELF notes to pass information from the linker to the C library.

Note sections contain a series of notes (see the *struct* definitions below). Each note is followed by the name field (whose length is defined in *n\_namesz*) and then by the descriptor field (whose length is defined in *n\_descsz*) and whose starting address has a 4 byte alignment. Neither field is defined in the note struct due to their arbitrary lengths.

An example for parsing out two consecutive notes should clarify their layout in memory:

```
void *memory, *name, *desc;
Elf64_Nhdr *note, *next_note;

/* The buffer is pointing to the start of the section/segment */
note = memory;

/* If the name is defined, it follows the note */
name = note->n_namesz == 0 ? NULL : memory + sizeof(*note);

/* If the descriptor is defined, it follows the name
   (with alignment) */
desc = note->n_descsz == 0 ? NULL :
      memory + sizeof(*note) + ALIGN_UP(note->n_namesz, 4);

/* The next note follows both (with alignment) */
next_note = memory + sizeof(*note) +
            ALIGN_UP(note->n_namesz, 4) +
            ALIGN_UP(note->n_descsz, 4);
```

Keep in mind that the interpretation of *n\_type* depends on the namespace defined by the *n\_namesz* field. If the *n\_namesz* field is not set (e.g., is 0), then there are two sets of notes: one for core files and one for all other ELF types. If the namespace is unknown, then tools will usually fallback to these sets of notes as well.

```
typedef struct {
    Elf32_Word n_namesz;
    Elf32_Word n_descsz;
    Elf32_Word n_type;
} Elf32_Nhdr;

typedef struct {
    Elf64_Word n_namesz;
    Elf64_Word n_descsz;
    Elf64_Word n_type;
} Elf64_Nhdr;
```

- n\_namesz* The length of the name field in bytes. The contents will immediately follow this note in memory. The name is null terminated. For example, if the name is "GNU", then *n\_namesz* will be set to 4.
- n\_descsz* The length of the descriptor field in bytes. The contents will immediately follow the name field in memory.
- n\_type* Depending on the value of the name field, this member may have any of the following values:

**Core files (e\_type = ET\_CORE)**

Notes used by all core files. These are highly operating system or architecture specific and often require close coordination with kernels, C libraries, and debuggers. These are used when the namespace is the default (i.e., *n\_namesz* will be set to 0), or a fallback when the namespace is unknown.

<b>NT_PRSTATUS</b>	prstatus struct
<b>NT_FPREGSET</b>	fpregset struct
<b>NT_PRPSINFO</b>	prpsinfo struct
<b>NT_PRXREG</b>	prxregset struct
<b>NT_TASKSTRUCT</b>	task structure
<b>NT_PLATFORM</b>	String from sysinfo(SI_PLATFORM)
<b>NT_AUXV</b>	auxv array
<b>NT_GWINDOWS</b>	gwindows struct
<b>NT_ASRS</b>	asrset struct
<b>NT_PSTATUS</b>	pstatus struct
<b>NT_PSINFO</b>	psinfo struct
<b>NT_PRCRED</b>	prcred struct
<b>NT_UTSNAME</b>	utsname struct
<b>NT_LWPSTATUS</b>	lwpstatus struct
<b>NT_LWPSINFO</b>	lwpinfo struct
<b>NT_PRFPXREG</b>	fprxregset struct
<b>NT_SIGINFO</b>	siginfo_t (size might increase over time)
<b>NT_FILE</b>	Contains information about mapped files
<b>NT_PRXFPREG</b>	user_fxsr_struct
<b>NT_PPC_VMX</b>	PowerPC AltiVec/VMX registers
<b>NT_PPC_SPE</b>	PowerPC SPE/EVR registers
<b>NT_PPC_VSX</b>	PowerPC VSX registers
<b>NT_386_TLS</b>	i386 TLS slots (struct user_desc)
<b>NT_386_IOPERM</b>	x86 io permission bitmap (1=deny)
<b>NT_X86_XSTATE</b>	x86 extended state using xsave
<b>NT_S390_HIGH_GPRS</b>	s390 upper register halves
<b>NT_S390_TIMER</b>	s390 timer register
<b>NT_S390_TODCMP</b>	s390 time-of-day (TOD) clock comparator register
<b>NT_S390_TODPREG</b>	s390 time-of-day (TOD) programmable register
<b>NT_S390_CTRS</b>	s390 control registers
<b>NT_S390_PREFIX</b>	s390 prefix register
<b>NT_S390_LAST_BREAK</b>	s390 breaking event address
<b>NT_S390_SYSTEM_CALL</b>	s390 system call restart data
<b>NT_S390_TDB</b>	s390 transaction diagnostic block
<b>NT_ARM_VFP</b>	ARM VFP/NEON registers
<b>NT_ARM_TLS</b>	ARM TLS register
<b>NT_ARM_HW_BREAK</b>	ARM hardware breakpoint registers



**NT\_ARM\_HW\_WATCH**

ARM hardware watchpoint registers

**NT\_ARM\_SYSTEM\_CALL**

ARM system call number

**n\_name = GNU**

Extensions used by the GNU tool chain.

**NT\_GNU\_ABI\_TAG**

Operating system (OS) ABI information. The desc field will be 4 words:

- word 0: OS descriptor (**ELF\_NOTE\_OS\_LINUX**, **ELF\_NOTE\_OS\_GNU**, and so on)‘
- word 1: major version of the ABI
- word 2: minor version of the ABI
- word 3: subminor version of the ABI

**NT\_GNU\_HWCAP**

Synthetic hwcap information. The desc field begins with two words:

- word 0: number of entries
- word 1: bit mask of enabled entries

Then follow variable-length entries, one byte followed by a null-terminated hwcap name string. The byte gives the bit number to test if enabled,  $(1U \ll \text{bit}) \&$  bit mask.

**NT\_GNU\_BUILD\_ID**

Unique build ID as generated by the GNU **ld(1)** **--build-id** option. The desc consists of any nonzero number of bytes.

**NT\_GNU\_GOLD\_VERSION**

The desc contains the GNU Gold linker version used.

**Default/unknown namespace (e\_type != ET\_CORE)**

These are used when the namespace is the default (i.e., *n\_namesz* will be set to 0), or a fallback when the namespace is unknown.

**NT\_VERSION**

A version string of some sort.

**NT\_ARCH**

Architecture information.

**NOTES**

ELF first appeared in System V. The ELF format is an adopted standard.

The extensions for *e\_phnum*, *e\_shnum* and *e\_shstrndx* respectively are Linux extensions. Sun, BSD and AMD64 also support them; for further information, look under SEE ALSO.

**SEE ALSO**

**as(1)**, **elfedit(1)**, **gdb(1)**, **ld(1)**, **nm(1)**, **objdump(1)**, **patchelf(1)**, **readelf(1)**, **size(1)**, **strings(1)**, **strip(1)**, **execve(2)**, **dl\_iterate\_phdr(3)**, **core(5)**, **ld.so(8)**

Hewlett-Packard, *Elf-64 Object File Format*.

Santa Cruz Operation, *System V Application Binary Interface*.

UNIX System Laboratories, "Object Files", *Executable and Linking Format (ELF)*.

Sun Microsystems, *Linker and Libraries Guide*.

AMD64 ABI Draft, *System V Application Binary Interface AMD64 Architecture Processor Supplement*.

**COLOPHON**

This page is part of release 5.05 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <https://www.kernel.org/doc/man-pages/>.