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 libraries.

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.

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*):

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

(Note: The *BSD terminology is a bit different. There *Elf64_Half* is twice as large as *Elf32_Half*, and *Elf64Quarter* 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, etc.

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 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 version number of the ELF specification:

EV_NONE Invalid version.

EV_CURRENT
Current version.

EI_OSABI This 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. E.g.:

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
This 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_OSABI** 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_BRAND Start of architecture identification.

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. E.g.:

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.

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.

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**.

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.

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

SHN_LOPROC
Values greater than or equal to **SHN_HIPROC** are reserved for processor-specific semantics.

SHN_HIPROC
Values less than or equal to **SHN_LOPROC** 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 indices 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 indices.

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 Phdr struct tells 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 and size for auxiliary information.

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

PT_PHDR The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may only occur 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
Values greater than or equal to **PT_HIPROC** are reserved for processor-specific semantics.

PT_HIPROC
Values less than or equal to **PT_LOPROC** 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 bitmask 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_X**, **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*.

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. 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
Values greater than or equal to **SHN_HIPROC** are reserved for processor-specific semantics.

SHN_HIPROC
Values less than or equal to **SHN_LOPROC** 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 information that marks the file in some way.

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 This value up to and including **SHT_HIPROC** is reserved for processor-specific semantics.

SHT_HIPROC This value down to and including **SHT_LOPROC** is 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. Values of zero or one mean 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**.

.fini 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 information in the "Note Section" format described below. This section is of type **SHT_NOTE**. No attribute types are used. OpenBSD native executables usually contain a *.note.openbsd.ident* section to identify themselves, for the kernel to bypass any compatibility ELF binary emulation tests when loading the file.

.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.

.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 non-writable 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 non-writable 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 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 non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table 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
This value up to and including **STT_HIPROC** is reserved for processor-specific semantics.

STT_HIPROC
This value down to and including **STT_LOPROC** is 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
This value up to and including **STB_HIPROC** is reserved for processor-specific semantics.

STB_HIPROC
This value down to and including **STB_LOPROC** is reserved for processor-specific semantics.

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

ELF32_ST_BIND(info) or **ELF64_ST_BIND**(info) extract a binding from an *st_info* value.

ELF32_ST_TYPE(info) or **ELF64_ST_TYPE**(info)
extract a type from an *st_info* value.

ELF32_ST_INFO(bind, type) or **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.
STV_INTERNAL
Processor-specific hidden class.
STV_HIDDEN Symbol is unavailable in other modules.
STV_PROTECTED
Not preemptible, not exported.

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 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.

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

DT_PLTRELSZ
Size in bytes of PLT relocs

DT_PLTGOT
Address of PLT and/or GOT

DT_HASH Address of symbol hash table

DT_STRTAB
Address of string table

DT_SYMTAB
Address of symbol table

DT_RELA Address of Rela relocs table

DT_RELASZ
Size in bytes of Rela table

DT_RELAENT
Size in bytes of a Rela table entry

DT_STRSZ Size in bytes of string table

DT_SYMENT
Size in bytes of a symbol table entry

DT_INIT Address of the initialization function

DT_FINI Address of the termination function

DT_SONAME
String table offset to name of shared object

DT_RPATH String table offset to library search path (deprecated)

DT_SYMBOLIC
Alert linker to search this shared object before the executable for symbols

DT_REL Address of Rel relocs table

DT_RELSZ Size in bytes of Rel table

DT_RELENT
Size in bytes of a Rel table entry

DT_PLTREL
Type of reloc the PLT refers (Rela or Rel)

DT_DEBUG Undefined use for debugging

DT_TEXTREL
Absence of this indicates no relocs should apply to a non-writable segment

DT_JMPREL
Address of reloc entries solely for the PLT

DT_BIND_NOW
Instruct dynamic linker to process all relocs before transferring control to the executable

DT_RUNPATH
String table offset to library search path

DT_LOPROC
Start of processor-specific semantics

DT_HIPROC
End of 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

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

SEE ALSO

as(1), **gdb**(1), **ld**(1), **objdump**(1), **execve**(2), **core**(5)

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)*.

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.