NAME

elf - format of ELF executable binary 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.

Applications which wish to process ELF binary files for their native architecture only should include <*elf.h*> in their source code. These applications should need to refer to all the types and structures by their generic names "Elf_xxx" and to the macros by "ELF_xxx". Applications written this way can be compiled on any architecture, regardless whether the host is 32-bit or 64-bit.

Should an application need to process ELF files of an unknown architecture then the application needs to include both <*sys/elf32.h*> and <*sys/elf64.h*> instead of <*elf.h*>. Furthermore, all types and structures need to be identified by either "Elf32_xxx" or "Elf64_xxx". The macros need to be identified by "ELF32_xxx" or "ELF64_xxx".

Whatever the system's architecture is, it will always include <*sys/elf_common.h*> as well as <*sys/elf_generic.h*>.

These header files describe the above mentioned headers as C structures and also include structures for dynamic sections, relocation sections and symbol tables.

The following types are being used for 32-bit architectures:

Elf32_Addr	Unsigned 32-bit program address
Elf32_Half	Unsigned 16-bit field
Elf32_Lword	Unsigned 64-bit field
Elf32_Off	Unsigned 32-bit file offset
Elf32_Sword	Signed 32-bit field or integer
Elf32_Word	Unsigned 32-bit field or integer

For 64-bit architectures we have the following types:

Elf64_Addr	Unsigned 64-bit program address
Elf64_Half	Unsigned 16-bit field
Elf64_Lword	Unsigned 64-bit field
Elf64_Off	Unsigned 64-bit file offset
Elf64_Sword	Signed 32-bit field
Elf64_Sxword	Signed 64-bit field or integer
Elf64_Word	Unsigned 32-bit field
Elf64_Xword	Unsigned 64-bit field or integer

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:

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

typedef struct {
                unsigned char   e_ident[EI_NIDENT];
                Elf64_Half      e_type;
                Elf64_Half      e_machine;
                Elf64_Word      e_version;
                Elf64_Addr      e_entry;
```

```
        Elf64_Off      e_phoff;
        Elf64_Off      e_shoff;
        Elf64_Word     e_flags;
        Elf64_Half     e_ehsize;
        Elf64_Half     e_phentsize;
        Elf64_Half     e_phnum;
        Elf64_Half     e_shentsize;
        Elf64_Half     e_shnum;
        Elf64_Half     e_shstrndx;
} Elf64_Ehdr;
```

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**.

EI_MAG1
: The second byte of the magic number. It must be filled with **ELFMAG1**.

EI_MAG2
: The third byte of the magic number. It must be filled with **ELFMAG2**.

EI_MAG3
: The fourth byte of the magic number. It must be filled with **ELFMAG3**.

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. The following values are currently defined:

ELFOSABI_SYSV	UNIX System V ABI.
ELFOSABI_HPUX	HP-UX operating system ABI.
ELFOSABI_NETBSD	NetBSD operating system ABI.
ELFOSABI_LINUX	GNU/Linux operating system ABI.
ELFOSABI_HURD	GNU/Hurd operating system ABI.
ELFOSABI_86OPEN	86Open Common IA32 ABI.
ELFOSABI_SOLARIS	Solaris operating system ABI.
ELFOSABI_MONTEREY	Monterey project ABI.
ELFOSABI_IRIX	IRIX operating system ABI.
ELFOSABI_FREEBSD	FreeBSD operating system ABI.
ELFOSABI_TRU64	TRU64 UNIX operating system ABI.
ELFOSABI_ARM	ARM architecture ABI.
ELFOSABI_STANDALONE	Standalone (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:

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_486	Intel 80486.
EM_860	Intel 80860.
EM_MIPS	MIPS RS3000 (big-endian only).
EM_MIPS_RS4_BE	MIPS RS4000 (big-endian only).
EM_SPARC64	SPARC v9 64-bit unofficial.
EM_PARISC	HPPA.
EM_PPC	PowerPC.
EM_ALPHA	Compaq [DEC] Alpha.

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. If the file is using extended program header numbering, then the **e_phnum** member will contain the value PN_XNUM and the actual number of program header table entries will be stored in the **sh_info** member of the section header at index SHN_UNDEF. The product of **e_phentsize** and the number of program header table entries gives the program header 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. If the file is using extended section numbering, then the **e_shnum** member will be zero and the actual section number will be stored in the **sh_size** member of the section header at index SHN_UNDEF. If a file has no section header table, both the **e_shnum** and the **e_shoff** fields of the ELF header will be zero. The product of **e_shentsize** and the number of sections in the file gives the section header table's size in bytes.

e_shstrndx This member holds the section header table index of the entry associated with the section name string table. If extended section numbering is being used, this field will hold the value **SHN_XINDEX**, and the actual section header table index will be present in the **sh_link** field of the section header entry at index SHN_UNDEF. If the file has no section name string table, this member holds the value **SHN_UNDEF**.

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. As with the Elf executable header, the program header also has different versions depending on the architecture:

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

typedef struct {
        Elf64_Word      p_type;
        Elf64_Word      p_flags;
```

```
        Elf64_Off      p_offset;
        Elf64_Addr     p_vaddr;
        Elf64_Addr     p_paddr;
        Elf64_Xword    p_filesz;
        Elf64_Xword    p_memsz;
        Elf64_Xword    p_align;
    } Elf64_Phdr;
```

The main difference between the 32-bit and the 64-bit program header lies only in the location of a **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 path name 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 This value up to and including **PT_HIPROC** are reserved for processor-specific semantics.

PT_HIPROC This value down to and including **PT_LOPROC** are reserved for processor-specific semantics.

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 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**.

An 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. 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 This value up to and including **SHN_HIPROC** are reserved for processor-specific semantics.

SHN_HIPROC This value down to and including **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. 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 {
        Elf32_Word    sh_name;
        Elf32_Word    sh_type;
        Elf32_Word    sh_flags;
        Elf32_Addr    sh_addr;
        Elf32_Off     sh_offset;
        Elf32_Word    sh_size;
        Elf32_Word    sh_link;
        Elf32_Word    sh_info;
        Elf32_Word    sh_addralign;
        Elf32_Word    sh_entsize;
} Elf32_Shdr;

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

```
        Elf64_Xword     sh_entsize;
} Elf64_Shdr;
```

sh_name This member specifies the name of the section. Its value is an index into the section header string table section, 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 The 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 **SHN_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. All 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 **SHN_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 **SHN_SYMTAB** section.

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

SHT_HIPROC This value down to and including **SHT_LOPROC** 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 The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file. This attribute is off for those sections.

SHF_EXECINSTR The section contains executable machine instructions.

SHF_MASKPROC

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

SHF_COMPRESSED

The section data is compressed.

sh_addr If the section will appear 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 (Block Started by Symbol) 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 attributes 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.

.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.

.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**.

.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.

.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**.

.jcr This section holds information about Java classes that must be registered.

.eh_frame
This section holds information used for C++ exception-handling.

A section with the SHF_COMPRESSED flag set contains a compressed copy of the section data. Compressed section data begins with an *Elf64_Chdr* or *Elf32_Chdr structure* which encodes the compression algorithm and some characteristics of the uncompressed data.

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

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

ch_type The compression algorithm used. A value of ELFCOMPRESS_ZLIB indicates that the data is compressed using zlib(3). A value of ELFCOMPRESS_ZSTD indicates that the data is compressed using Zstandard.

ch_size The size, in bytes, of the uncompressed section data. This corresponds to the **sh_size** field of a section header containing uncompressed data.

ch_addralign The address alignment of the uncompressed section data. This corresponds to the **sh_addralign** field of a section header containing uncompressed data.

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Similarly, a string table's last byte is defined to hold a null character, 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 {
        Elf32_Word     st_name;
        Elf32_Addr     st_value;
        Elf32_Word     st_size;
        unsigned char  st_info;
        unsigned char  st_other;
        Elf32_Half     st_shndx;
```

```
} Elf32_Sym;

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

st_name This member holds an index into the object file's symbol string table, which holds 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** are reserved for processor-specific semantics.

STT_HIPROC This value down to and including **STT_LOPROC** 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 file 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** are reserved for processor-specific semantics.

STB_HIPROC This value down to and including **STB_LOPROC** are 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.

ELF64_ST_TYPE(*info*) or **ELF32_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 currently holds zero and has no defined meaning.

st_shndx

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

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' program image. Relocation entries are these data.

Relocation structures that do not need an addend:

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

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

Relocation structures that need an addend:

```
typedef struct {
```

```
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;

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

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

Note Section

ELF note sections consist of entries with the following format:

Field	**Size**	**Description**
namesz	32 bits	Size of name
descsz	32 bits	Size of desc
type	32 bits	OS-dependent note type
name	*namesz*	Null-terminated originator name
desc	*descsz*	OS-dependent note data

The *name* and *desc* fields are padded to ensure 4-byte alignemnt. *namesz* and *descsz* specify the unpadded length.

FreeBSD defines the following ELF note types (with corresponding interpretation of *desc*):

NT_FREEBSD_ABI_TAG (Value: 1)

Indicates the OS ABI version in a form of a 32-bit integer containing expected ABI version (i.e., __FreeBSD_version).

NT_FREEBSD_NOINIT_TAG (Value: 2)
Indicates that the C startup does not call initialization routines, and thus rtld(1) must do so. *desc* is ignored.

NT_FREEBSD_ARCH_TAG (Value: 3)
Contains the MACHINE_ARCH that the executable was built for.

NT_FREEBSD_FEATURE_CTL (Value: 4)
Contains a bitmask of mitigations and features to enable:

NT_FREEBSD_FCTL_ASLR_DISABLE (Value: 0x01)
Request that address randomization (ASLR) not be performed. See security(7).

NT_FREEBSD_FCTL_PROTMAX_DISABLE (Value: 0x02)
Request that mmap(2) calls not set PROT_MAX to the initial value of the *prot* argument.

NT_FREEBSD_FCTL_STKGAP_DISABLE (Value: 0x04)
Disable stack gap.

NT_FREEBSD_FCTL_WXNEEDED (Value: 0x08)
Indicate that the binary requires mappings that are simultaneously writeable and executable.

SEE ALSO

as(1), gdb(1) (*ports/devel/gdb*), ld(1), objdump(1), readelf(1), execve(2), zlib(3), ar(5), 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)*.

HISTORY

The ELF header files made their appearance in FreeBSD 2.2.6. ELF in itself first appeared in AT&T System V UNIX. The ELF format is an adopted standard.

AUTHORS

This manual page was written by Jeroen Ruigrok van der Werven <*asmodai@FreeBSD.org*> with inspiration from BSDi's BSD/OS **elf** manpage.