NAME

elf - API for manipulating ELF objects

LIBRARY

ELF Access Library (libelf, -lelf)

SYNOPSIS

#include <libelf.h>

DESCRIPTION

The ELF Access Library (libelf, -lelf) provides functions that allow an application to read and manipulate ELF object files, and to read ar(1) archives. The library allows the manipulation of ELF objects in a byte ordering and word-size independent way, allowing an application to read and create ELF objects for 32 and 64 bit architectures and for little- and big-endian machines. The library is capable of processing ELF objects that use extended section numbering.

This manual page serves to provide an overview of the functionality in the ELF library. Further information may found in the manual pages for individual ELF(3) functions that comprise the library.

ELF Concepts

As described in elf(5), ELF files contain several data structures that are laid out in a specific way. ELF files begin with an "Executable Header", and may contain an optional "Program Header Table", and optional data in the form of ELF "sections". A "Section Header Table" describes the content of the data in these sections.

ELF objects have an associated "ELF class" which denotes the natural machine word size for the architecture the object is associated with. Objects for 32 bit architectures have an ELF class of ELFCLASS32. Objects for 64 bit architectures have an ELF class of ELFCLASS64.

ELF objects also have an associated "endianness" which denotes the endianness of the machine architecture associated with the object. This may be ELFDATA2LSB for little-endian architectures and ELFDATA2MSB for big-endian architectures.

ELF objects are also associated with an API version number. This version number determines the layout of the individual components of an ELF file and the semantics associated with these.

Data Representation And Translation

The ELF(3) library distinguishes between "native" representations of ELF data structures and their "file" representations.

An application would work with ELF data in its "native" representation, i.e., using the native byteorder and alignment mandated by the processor the application is running on. The "file" representation of the same data could use a different byte ordering and follow different constraints on object alignment than these native constraints.

Accordingly, the ELF(3) library offers translation facilities (elf32_xlatetof(3), elf32_xlatetom(3), elf64_xlatetof(3) and elf64_xlatetom(3)) to and from these representations. It also provides higher-level APIs (gelf_xlatetof(3), gelf_xlatetom(3)) that retrieve and store data from the ELF object in a class-agnostic manner.

Library Working Version

Conceptually, there are three version numbers associated with an application using the ELF library to manipulate ELF objects:

- The ELF version that the application was compiled against. This version determines the ABI expected by the application.
- The ELF version of the ELF object being manipulated by the application through the ELF library.
- The ELF version (or set of versions) supported by the ELF library itself.

In order to facilitate working with ELF objects of differing versions, the ELF library requires the application to call the **elf_version**() function before invoking many of its operations, in order to inform the library of the application’s desired working version.

In the current implementation, all three versions have to be EV_CURRENT.

Namespace use

The ELF library uses the following prefixes:

elf_ Used for class-independent functions.

elf32_ Used for functions working with 32 bit ELF objects.

elf64_ Used for functions working with 64 bit ELF objects.

Elf_ Used for class-independent data types.

ELF_C_ Used for command values used in a few functions. These symbols are defined as members of the *Elf_Cmd* enumeration.

ELF_E_ Used for error numbers.

ELF_F_ Used for flags.

ELF_K_ These constants define the kind of file associated with an ELF descriptor. See elf_kind(3). The symbols are defined by the *Elf_Kind* enumeration.

ELF_T_ These values are defined by the *Elf_Type* enumeration, and denote the types of ELF data structures that can be present in an ELF object.

In addition, the library uses symbols with prefixes _ELF and _libelf for its internal use.

Descriptors

Applications communicate with the library using descriptors. These are:

Elf An *Elf* descriptor represents an ELF object or an ar(1) archive. It is allocated using one of the **elf_begin**() or **elf_memory**() functions. An *Elf* descriptor can be used to read and write data to an ELF file. An *Elf* descriptor can be associated with zero or more *Elf_Scn* section descriptors.

Given an ELF descriptor, the application may retrieve the ELF object's class-dependent "Executable Header" structures using the **elf32_getehdr**() or **elf64_getehdr**() functions. A new Ehdr structure may be allocated using the **elf64_newehdr**() or **elf64_newehdr**() functions.

The "Program Header Table" associated with an ELF descriptor may be allocated using the **elf32_getphdr**() or **elf64_getphdr**() functions. A new program header table may be allocated or an existing table resized using the **elf32_newphdr**() or **elf64_newphdr**() functions.

The *Elf* structure is opaque and has no members visible to the application.

Elf_Data An *Elf_Data* data structure describes an individual chunk of a ELF file as represented in memory. It has the following application-visible members:

uint64_t d_align	The in-file alignment of the data buffer within its containing ELF section. This value must be non-zero and a power of two.
*void *d_buf*	A pointer to data in memory.
uint64_t d_off	The offset within the containing section where this descriptor's data would be placed. This field will be computed by the library unless the application requests full control of the ELF object's layout.
uint64_t d_size	The number of bytes of data in this descriptor.
Elf_Type d_type	The ELF type (see below) of the data in this descriptor.
unsigned int d_version	The operating version for the data in this buffer.

Elf_Data descriptors are usually used in conjunction with *Elf_Scn* descriptors.

Elf_Scn *Elf_Scn* descriptors represent sections in an ELF object. These descriptors are opaque and contain no application modifiable fields.

The *Elf_Scn* descriptor for a specific section in an ELF object can be retrieved using the **elf_getscn**() function. The sections contained in an ELF object can be traversed using the **elf_nextscn**() function. New sections are allocated using the **elf_newscn**() function.

The *Elf_Data* descriptors associated with a given section can be retrieved using the **elf_getdata**() function. New data descriptors can be added to a section descriptor using the **elf_newdata**() function. The untranslated "file" representation of data in a section can be retrieved using the **elf_rawdata**() function.

Supported Elf Types

The following ELF datatypes are supported by the library.

ELF_T_ADDR	Machine addresses.
ELF_T_BYTE	Byte data. The library will not attempt to translate byte data.
ELF_T_CAP	Software and hardware capability records.
ELF_T_DYN	Records used in a section of type SHT_DYNAMIC.
ELF_T_EHDR	ELF executable header.
ELF_T_GNUHASH	GNU-style hash tables.
ELF_T_HALF	16-bit unsigned words.
ELF_T_LWORD	64 bit unsigned words.
ELF_T_MOVE	ELF Move records.
ELF_T_NOTE	ELF Note structures.
ELF_T_OFF	File offsets.
ELF_T_PHDR	ELF program header table entries.
ELF_T_REL	ELF relocation entries.
ELF_T_RELA	ELF relocation entries with addends.
ELF_T_SHDR	ELF section header entries.
ELF_T_SWORD	Signed 32-bit words.
ELF_T_SXWORD	Signed 64-bit words.
ELF_T_SYMINFO	ELF symbol information.
ELF_T_SYM	ELF symbol table entries.
ELF_T_VDEF	Symbol version definition records.
ELF_T_VNEED	Symbol version requirement records.
ELF_T_WORD	Unsigned 32-bit words.

ELF_T_XWORD Unsigned 64-bit words.

The symbol ELF_T_NUM denotes the number of Elf types known to the library.

The following table shows the mapping between ELF section types defined in elf(5) and the types supported by the library.

Section Type	*Library Type*	*Description*
SHT_DYNAMIC	ELF_T_DYN	'.dynamic' section entries.
SHT_DYNSYM	ELF_T_SYM	Symbols for dynamic linking.
SHT_FINI_ARRAY	ELF_T_ADDR	Termination function pointers.
SHT_GNU_HASH	ELF_T_GNUHASH	GNU hash sections.
SHT_GNU_LIBLIST	ELF_T_WORD	List of libraries to be pre-linked.
SHT_GNU_verdef	ELF_T_VDEF	Symbol version definitions.
SHT_GNU_verneed	ELF_T_VNEED	Symbol versioning requirements.
SHT_GNU_versym	ELF_T_HALF	Version symbols.
SHT_GROUP	ELF_T_WORD	Section group marker.
SHT_HASH	ELF_T_HASH	Symbol hashes.
SHT_INIT_ARRAY	ELF_T_ADDR	Initialization function pointers.
SHT_NOBITS	ELF_T_BYTE	Empty sections. See elf(5).
SHT_NOTE	ELF_T_NOTE	ELF note records.
SHT_PREINIT_ARRAY	ELF_T_ADDR	Pre-initialization function pointers.
SHT_PROGBITS	ELF_T_BYTE	Machine code.
SHT_REL	ELF_T_REL	ELF relocation records.
SHT_RELA	ELF_T_RELA	Relocation records with addends.
SHT_STRTAB	ELF_T_BYTE	String tables.
SHT_SYMTAB	ELF_T_SYM	Symbol tables.
SHT_SYMTAB_SHNDX	ELF_T_WORD	Used with extended section numbering.
SHT_SUNW_dof	ELF_T_BYTE	Used by dtrace(1).
SHT_SUNW_move	ELF_T_MOVE	ELF move records.
SHT_SUNW_syminfo	ELF_T_SYMINFO	Additional symbol flags.
SHT_SUNW_verdef	ELF_T_VDEF	Same as SHT_GNU_verdef.
SHT_SUNW_verneed	ELF_T_VNEED	Same as SHT_GNU_verneed.
SHT_SUNW_versym	ELF_T_HALF	Same as SHT_GNU_versym.

Section types in the range [SHT_LOOS, SHT_HIUSER] are otherwise considered to be of type ELF_T_BYTE.

Functional Grouping

This section contains a brief overview of the available functionality in the ELF library. Each function

listed here is described further in its own manual page.

Archive Access

elf_getarsym()
: Retrieve the archive symbol table.

elf_getarhdr()
: Retrieve the archive header for an object.

elf_getbase()
: Retrieve the offset of a member inside an archive.

elf_next()
: Iterate through an ar(1) archive.

elf_rand()
: Random access inside an ar(1) archive.

Data Structures

elf_getdata()
: Retrieve translated data for an ELF section.

elf_getscn()
: Retrieve the section descriptor for a named section.

elf_ndxscn()
: Retrieve the index for a section.

elf_newdata()
: Add a new *Elf_Data* descriptor to an ELF section.

elf_newscn()
: Add a new section descriptor to an ELF descriptor.

elf_nextscn()
: Iterate through the sections in an ELF object.

elf_rawdata()
: Retrieve untranslated data for an ELF section.

elf_rawfile()
: Return a pointer to the untranslated file contents for an ELF object.

elf32_getehdr(), **elf64_getehdr**()
: Retrieve the Executable Header in an ELF object.

elf32_getphdr(), **elf64_getphdr**()
: Retrieve the Program Header Table in an ELF object.

elf32_getshdr(), **elf64_getshdr**()
: Retrieve the ELF section header associated with an *Elf_Scn* descriptor.

elf32_newehdr(), **elf64_newehdr**()
: Allocate an Executable Header in an ELF object.

elf32_newphdr(), **elf64_newphdr**()

Allocate or resize the Program Header Table in an ELF object.

Data Translation

elf32_xlatetof(), **elf64_xlatetof**()

Translate an ELF data structure from its native representation to its file representation.

elf32_xlatetom(), **elf64_xlatetom**()

Translate an ELF data structure from its file representation to a native representation.

Error Reporting

elf_errno()

Retrieve the current error.

elf_errmsg()

Retrieve a human readable description of the current error.

Initialization

elf_begin()

Opens an ar(1) archive or ELF object given a file descriptor.

elf_end()

Close an ELF descriptor and release all its resources.

elf_memory()

Opens an ar(1) archive or ELF object present in a memory arena.

elf_version()

Sets the operating version.

IO Control

elf_cntl() Manage the association between and ELF descriptor and its underlying file.

elf_flagdata() Mark an *Elf_Data* descriptor as dirty.

elf_flagehdr() Mark the ELF Executable Header in an ELF descriptor as dirty.

elf_flagphdr() Mark the ELF Program Header Table in an ELF descriptor as dirty.

elf_flagscn() Mark an *Elf_Scn* descriptor as dirty.

elf_flagshdr() Mark an ELF Section Header as dirty.

elf_setshstrndx()

Set the index of the section name string table for the ELF object.

elf_update() Recompute ELF object layout and optionally write the modified object back to the underlying file.

Queries

elf32_checksum(), **elf64_checkum**()

Compute checksum of an ELF object.

elf_getident() Retrieve the identification bytes for an ELF object.

elf_getphdrnum()
: Retrieve the number of program headers in an ELF object.

elf_getshdrnum()
: Retrieve the number of sections in an ELF object.

elf_getshdrstrndx()
: Retrieve the section index of the section name string table in an ELF object.

elf_hash() Compute the ELF hash value of a string.

elf_kind() Query the kind of object associated with an ELF descriptor.

elf32_fsize(), **elf64_fsize**()
: Return the size of the file representation of an ELF type.

Controlling ELF Object Layout

In the usual mode of operation, library will compute section offsets and alignments based on the contents of an ELF descriptor's sections without need for further intervention by the application.

However, if the application wishes to take complete charge of the layout of the ELF file, it may set the ELF_F_LAYOUT flag on an ELF descriptor using elf_flagelf(3), following which the library will use the data offsets and alignments specified by the application when laying out the file. Application control of file layout is described further in the elf_update(3) manual page.

Gaps in between sections will be filled with the fill character set by function **elf_fill**().

Error Handling

In case an error is encountered, these library functions set an internal error number and signal the presence of the error by returning an special return value. The application can check the current error number by calling elf_errno(3). A human readable description of the recorded error is available by calling elf_errmsg(3).

Memory Management Rules

The library keeps track of all *Elf_Scn* and *Elf_Data* descriptors associated with an ELF descriptor and recovers them when the descriptor is closed using elf_end(3). Thus the application must not call free(3) on data structures allocated by the ELF library.

Conversely the library will not free data that it has not allocated. As an example, an application may call elf_newdata(3) to allocate a new *Elf_Data* descriptor and can set the *d_off* member of the descriptor to point to a region of memory allocated using malloc(3). It is the applications responsibility to free this arena, though the library will reclaim the space used by the *Elf_Data* descriptor itself.

SEE ALSO

gelf(3), ar(5), elf(5)

HISTORY

The original **elf** API was developed for AT&T System V UNIX. The current implementation of the API appeared in FreeBSD 7.0.

AUTHORS

The ELF library was written by Joseph Koshy <*jkoshy@FreeBSD.org*>.