NAME elf – object file access library

SYNOPSIS **cc** [*flag* . . .] *file* . . . **–lelf** [*library* . . .]

#include <libelf.h>

MT-LEVEL Unsafe

DESCRIPTION Functions in the ELF access library let a program manipulate ELF (Executable and Linking Format) object files, archive files, and archive members. The header provides type and function declarations for all library services.

Programs communicate with many of the higher-level routines using an *ELF descriptor*. That is, when the program starts working with a file, **elf_begin**(3E) creates an ELF descriptor through which the program manipulates the structures and information in the file. These ELF descriptors can be used both to read and to write files. After the program establishes an ELF descriptor for a file, it may then obtain *section descriptors* to manipulate the sections of the file (see **elf_getscn**(3E)). Sections hold the bulk of an object file's real information, such as text, data, the symbol table, and so on. A section descriptor ''belongs'' to a particular ELF descriptor, just as a section belongs to a file. Finally, *data descriptors* are available through section descriptors, allowing the program to manipulate the information associated with a section. A data descriptor ''belongs'' to a section descriptor.

Descriptors provide private handles to a file and its pieces. In other words, a data descriptor is associated with one section descriptor, which is associated with one ELF descriptor, which is associated with one file. Although descriptors are private, they give access to data that may be shared. Consider programs that combine input files, using incoming data to create or update another file. Such a program might get data descriptors for an input and an output section. It then could update the output descriptor to reuse the input descriptor's data. That is, the descriptors are distinct, but they could share the associated data bytes. This sharing avoids the space overhead for duplicate buffers and the performance overhead for copying data unnecessarily.

FILE CLASSES ELF provides a framework in which to define a family of object files, supporting multiple processors and architectures. An important distinction among object files is the *class*, or capacity, of the file. The 32-bit class supports architectures in which a 32-bit object can represent addresses, file sizes, etc., as in the following.

Name	Purpose
Elf32_Addr	Unsigned address
Elf32_Half	Unsigned medium integer
Elf32_Off	Unsigned file offset
Elf32_Sword	Signed large integer
Elf32_Word	Unsigned large integer
unsigned char	Unsigned small integer

Other classes will be defined as necessary, to support larger (or smaller) machines. Some library services deal only with data objects for a specific class, while others are class-independent. To make this distinction clear, library function names reflect their status, as described below.

DATA REPRESENTATION

Conceptually, two parallel sets of objects support cross compilation environments. One set corresponds to file contents, while the other set corresponds to the native memory image of the program manipulating the file. Type definitions supplied by the headers work on the native machine, which may have different data encodings (size, byte order, etc.) than the target machine. Although native memory objects should be at least as big as the file objects (to avoid information loss), they may be bigger if that is more natural for the host machine.

Translation facilities exist to convert between file and memory representations. Some library routines convert data automatically, while others leave conversion as the program's responsibility. Either way, programs that create object files must write file-typed objects to those files; programs that read object files must take a similar view. See **elf32_xlatetof**(3E) and **elf32_fsize**(3E) for more information.

Programs may translate data explicitly, taking full control over the object file layout and semantics. If the program prefers not to have and exercise complete control, the library provides a higher-level interface that hides many object file details. **elf_begin( )** and related functions let a program deal with the native memory types, converting between memory objects and their file equivalents automatically when reading or writing an object file.

ELF VERSIONS

Object file versions allow ELF to adapt to new requirements. *Three independent versions* can be important to a program. First, an application program knows about a particular version by virtue of being compiled with certain headers. Second, the access library similarly is compiled with header files that control what versions it understands. Third, an ELF object file holds a value identifying its version, determined by the ELF version known by the file's creator. Ideally, all three versions would be the same, but they may differ.

> If a program's version is newer than the access library, the program might use information unknown to the library. Translation routines might not work properly, leading to undefined behavior. This condition merits installing a new library.
>
> The library's version might be newer than the program's and the file's. The library understands old versions, thus avoiding compatibility problems in this case.
>
> Finally, a file's version might be newer than either the program or the library understands. The program might or might not be able to process the file properly, depending on whether the file has extra information and whether that information can be safely ignored. Again, the safe alternative is to install a new library that understands the file's version.

To accommodate these differences, a program must use **elf_version**(3E) to pass its version to the library, thus establishing the *working version* for the process. Using this, the library accepts data from and presents data to the program in the proper representations. When the library reads object files, it uses each file's version to interpret the data. When writing files or converting memory types to the file equivalents, the library uses the program's working version for the file data.

SYSTEM SERVICES

As mentioned above, **elf_begin( )** and related routines provide a higher-level interface to ELF files, performing input and output on behalf of the application program. These routines assume a program can hold entire files in memory, without explicitly using temporary files. When reading a file, the library routines bring the data into memory and perform subsequent operations on the memory copy. Programs that wish to read or write large object files with this model must execute on a machine with a large process virtual address space. If the underlying operating system limits the number of open files, a program can use **elf_cntl**(3E) to retrieve all necessary data from the file, allowing the program to close the file descriptor and reuse it.

Although the **elf_begin( )** interfaces are convenient and efficient for many programs, they might be inappropriate for some. In those cases, an application may invoke the **elf32_xlatetom**(3E) or **elf32_xlatetof**(3E) data translation routines directly. These routines perform no input or output, leaving that as the application's responsibility. By assuming a larger share of the job, an application controls its input and output model.

LIBRARY NAMES

Names associated with the library take several forms.

elf_*name* These class-independent names perform some service, *name*, for the program.

elf32_*name* Service names with an embedded class, **32** here, indicate they work only for the designated class of files.

Elf_*Type* Data types can be class-independent as well, distinguished by *Type*.

Elf32_*Type* Class-dependent data types have an embedded class name, **32** here.

ELF_C_*CMD* Several functions take commands that control their actions. These values are members of the **Elf_Cmd** enumeration; they range from zero through **ELF_C_NUM**−1**.**

ELF_F_*FLAG* Several functions take flags that control library status and/or actions. Flags are bits that may be combined.

ELF32_FSZ_*TYPE* These constants give the file sizes in bytes of the basic ELF types for the 32-bit class of files. See **elf32_fsize( )** for more information.

ELF_K_*KIND* The function **elf_kind( )** identifies the *KIND* of file associated with an ELF descriptor. These values are members of the **Elf_Kind** enumeration; they range from zero through **ELF_K_NUM**−1**.**

ELF_T_*TYPE* When a service function, such as **elf32_xlatetom( )** or **elf32_xlatetof( )**, deals with multiple types, names of this form specify the desired *TYPE*. Thus, for example, **ELF_T_EHDR** is directly related to **Elf32_Ehdr**. These values are members of the **Elf_Type** enumeration; they range from zero through **ELF_T_NUM**–1**.**

EXAMPLES

The basic interpretation of an ELF file consists of:

- opening an ELF object file
- obtaining an ELF descriptor
- analyzing the file using the descriptor.

The following example opens the file, obtains the ELF descriptor, and prints out the names of each section in the file.

```
#include        <fcntl.h>
#include        <stdio.h>
#include        <libelf.h>
#include        <stdlib.h>
#include        <string.h>
static void failure(void);

void
main(int argc, char ** argv)
{
        Elf32_Shdr *    shdr;
        Elf32_Ehdr *    ehdr;
        Elf *           elf;
        Elf_Scn *       scn;
        Elf_Data *      data;
        int             fd;
        unsigned int    cnt;

                /* Open the input file */
        if ((fd = open(argv[1], O_RDONLY)) == -1)
                exit(1);

                /* Obtain the ELF descriptor */
        (void) elf_version(EV_CURRENT);
        if ((elf = elf_begin(fd, ELF_C_READ, NULL)) == NULL)
                failure();

                /* Obtain the .shstrtab data buffer */
        if (((ehdr = elf32_getehdr(elf)) == NULL) ||
           ((scn = elf_getscn(elf, ehdr->e_shstrndx)) == NULL) ||
           ((data = elf_getdata(scn, NULL)) == NULL))
                failure();
```

```
                /* Traverse input filename, printing each section */
        for (cnt = 1, scn = NULL; scn = elf_nextscn(elf, scn); cnt++) {
                if ((shdr = elf32_getshdr(scn)) == NULL)
                        failure();

        (void) printf("[%d]%s0, cnt,
                (char *)data->d_buf + shdr->sh_name);
        }
}               /* end main */

static void
failure()
{
        (void) fprintf(stderr, "%s0, elf_errmsg(elf_errno()));
        exit(1);
}
```

Below is sample output from compiling and executing the above code, which prints the names of the sections using itself as the input file

```
        % cc -o elfprint example.c -lelf
        % elfprint elfprint
        [1]     .interp
        [2]     .hash
        [3]     .dynsym
        [4]     .dynstr
        [5]     .rela.ex_shared
        [6]     .rela.bss
        [7]     .rela.plt
        [8]     .text
        [9]     .init
        [10]    .fini
        [11]    .exception_ranges
        . . .
```

SEE ALSO

elf32_fsize(3E), **elf32_getshdr**(3E), **elf32_xlatetof**(3E), **elf_begin**(3E), **elf_cntl**(3E), **elf_errmsg**(3E), **elf_fill**(3E), **elf_getarhdr**(3E), **elf_getarsym**(3E), **elf_getbase**(3E), **elf_getdata**(3E), **elf_getident**(3E), **elf_getscn**(3E), **elf_hash**(3E), **elf_kind**(3E), **elf_memory**(3E), **elf_rawfile**(3E), **elf_strptr**(3E), **elf_update**(3E), **elf_version**(3E), **ar**(4)

ANSI C Programmer's Guide

SPARC only

a.out(4)

NOTES

Information in the ELF headers is separated into common parts and processor-specific parts. A program can make a processor's information available by including the appropriate header: **<sys/elf_*NAME*.h>** where *NAME* matches the processor name as used in the ELF file header.

Name	Processor
M32	AT&T WE 32100
SPARC	SPARC
386	Intel 80386
486	Intel 80486
860	Intel 80860
68K	Motorola 68000
88K	Motorola 88000

Other processors will be added to the table as necessary.

To illustrate, a program could use the following code to ''see'' the processor-specific information for the SPARC.

```
#include <libelf.h>
#include <sys/elf_SPARC.h>
```

Without the **<sys/elf_SPARC.h>** definition, only the common ELF information would be visible.

A program could use the following code to ''see'' the processor-specific information for the Intel 80386:

```
#include <libelf.h>
#include <sys/elf_386.h>
```

Without the **<sys/elf_386.h>** definition, only the common ELF information would be visible.

Although reading the objects is rather straightforward, writing/updating them can corrupt the shared offsets among sections. Upon creation, relationships are established among the sections that must be maintained even if the object's size is changed.