NAME elf_begin, elf_end, elf_memory, elf_next, elf_rand – process ELF object files

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

#include <libelf.h>

Elf *elf_begin(int *fildes*, **Elf_Cmd** *cmd*, **Elf** **ref*);

int elf_end(Elf **elf*);

Elf *elf_memory(char **image*, **size_t***sz*);

Elf_Cmd elf_next(Elf **elf*);

size_t elf_rand(Elf **elf*, **size_t** *offset*);

MT-LEVEL Unsafe

DESCRIPTION **elf_begin( )**, **elf_end( )**, **elf_memory( )**, **elf_next( )**, and **elf_rand( )** work together to process Executable and Linking Format (ELF) object files, either individually or as members of archives. After obtaining an ELF descriptor from **elf_begin( )** or **elf_memory( )**, the program may read an existing file, update an existing file, or create a new file. *fildes* is an open file descriptor that **elf_begin( )** uses for reading or writing. *elf* is an ELF descriptor previously returned from **elf_begin( )**. The initial file offset (see **lseek**(2)) is unconstrained, and the resulting file offset is undefined.

cmd may have the following values.

ELF_C_NULL When a program sets *cmd* to this value, **elf_begin( )** returns a null pointer, without opening a new descriptor. *ref* is ignored for this command. See the examples below for more information.

ELF_C_READ When a program wishes to examine the contents of an existing file, it should set *cmd* to this value. Depending on the value of *ref*, this command examines archive members or entire files. Three cases can occur.

First, if *ref* is a null pointer, **elf_begin( )** allocates a new ELF descriptor and prepares to process the entire file. If the file being read is an archive, **elf_begin( )** also prepares the resulting descriptor to examine the initial archive member on the next call to **elf_begin( )**, as if the program had used **elf_next( )** or **elf_rand( )** to ''move'' to the initial member.

Second, if *ref* is a non-null descriptor associated with an archive file, **elf_begin( )** lets a program obtain a separate ELF descriptor associated with an individual member. The program should have used **elf_next( )** or **elf_rand( )** to position *ref* appropriately (except for the initial member, which **elf_begin( )** prepares; see the example below). In this case, *fildes* should be the same file descriptor used for the parent archive.

Finally, if *ref* is a non-null ELF descriptor that is not an archive, **elf_begin( )** increments the number of activations for the descriptor and returns *ref*, without allocating a new descriptor and without changing the descriptor's read/write permissions. To terminate the descriptor for *ref*, the program must call **elf_end( )** once for each activation. See the examples below for more information.

ELF_C_RDWR　This command duplicates the actions of **ELF_C_READ** and additionally allows the program to update the file image (see **elf_update**(3E)). That is, using **ELF_C_READ** gives a read-only view of the file, while **ELF_C_RDWR** lets the program read *and* write the file. **ELF_C_RDWR** is not valid for archive members. If *ref* is non-null, it must have been created with the **ELF_C_RDWR** command.

ELF_C_WRITE　If the program wishes to ignore previous file contents, presumably to create a new file, it should set *cmd* to this value. *ref* is ignored for this command.

elf_begin( ) ''works'' on all files (including files with zero bytes), providing it can allocate memory for its internal structures and read any necessary information from the file. Programs reading object files thus may call **elf_kind**(3E) or **elf32_getehdr**(3E) to determine the file type (only object files have an ELF header). If the file is an archive with no more members to process, or an error occurs, **elf_begin( )** returns a null pointer. Otherwise, the return value is a non-null ELF descriptor.

Before the first call to **elf_begin( )**, a program must call **elf_version( )** to coordinate versions.

elf_end( ) is used to terminate an ELF descriptor, *elf*, and to deallocate data associated with the descriptor. Until the program terminates a descriptor, the data remain allocated. A null pointer is allowed as an argument, to simplify error handling. If the program wishes to write data associated with the ELF descriptor to the file, it must use **elf_update( )** before calling **elf_end( )**.

Calling **elf_end( )** removes one activation and returns the remaining activation count. The library does not terminate the descriptor until the activation count reaches zero. Consequently, a 0 return value indicates the ELF descriptor is no longer valid.

elf_memory( ) returns a pointer to an Elf descriptor, the Elf image has read operations enabled (**ELF_C_READ**). *image* is a pointer to an image of the Elf file mapped into memory, *sz* is the size of the Elf image. You may read and modify an Elf image that is mapped in with **elf_memory( )**, but you may not change the Elf image size.

elf_next( ) provides sequential access to the next archive member. That is, having an ELF descriptor, *elf*, associated with an archive member, **elf_next( )** prepares the containing archive to access the following member when the program calls **elf_begin( )**. After successfully positioning an archive for the next member, **elf_next( )** returns the value **ELF_C_READ**. Otherwise, the open file was not an archive, *elf* was null, or an error occurred, and the return value is **ELF_C_NULL**. In either case, the return value may be passed as an argument to **elf_begin( )**, specifying the appropriate action.

elf_rand( ) provides random archive processing, preparing *elf* to access an arbitrary archive member. *elf* must be a descriptor for the archive itself, not a member within the archive. *offset* gives the byte offset from the beginning of the archive to the archive header of the desired member. See **elf_getarsym**(3E) for more information about archive member offsets. When **elf_rand( )** works, it returns *offset*. Otherwise it returns 0, because an error occurred, *elf* was null, or the file was not an archive (no archive member can have a zero offset). A program may mix random and sequential archive processing.

SYSTEM SERVICES

When processing a file, the library decides when to read or write the file, depending on the program's requests. Normally, the library assumes the file descriptor remains usable for the life of the ELF descriptor. If, however, a program must process many files simultaneously and the underlying operating system limits the number of open files, the program can use **elf_cntl( )** to let it reuse file descriptors. After calling **elf_cntl( )** with appropriate arguments, the program may close the file descriptor without interfering with the library.

All data associated with an ELF descriptor remain allocated until **elf_end( )** terminates the descriptor's last activation. After the descriptors have been terminated, the storage is released; attempting to reference such data gives undefined behavior. Consequently, a program that deals with multiple input (or output) files must keep the ELF descriptors active until it finishes with them.

EXAMPLES

A prototype for reading a file appears
on the next page. If the file is a simple object file, the program executes the loop one time, receiving a null descriptor in the second iteration. In this case, both **elf** and **arf** will have the same value, the activation count will be two, and the program calls **elf_end( )** twice to terminate the descriptor.

If the file is an archive, the loop processes each archive member in turn, ignoring those that are not object files.

```
if (elf_version(EV_CURRENT) == EV_NONE)
{
        /* library out of date */
        /* recover from error */
}
cmd = ELF_C_READ;
arf = elf_begin(fildes, cmd, (Elf *)0);
while ((elf = elf_begin(fildes, cmd, arf)) != 0)
{
        if ((ehdr = elf32_getehdr(elf)) != 0)
        {
                /* process the file ... */
        }
        cmd = elf_next(elf);
        elf_end(elf);
}
elf_end(arf);
```

Alternatively, the next example illustrates random archive processing. After identifying the file as an archive, the program repeatedly processes archive members of interest. For clarity, this example omits error checking and ignores simple object files. Additionally, this fragment preserves the ELF descriptors for all archive members, because it does not call **elf_end( )** to terminate them.

```
elf_version(EV_CURRENT);
arf = elf_begin(fildes, ELF_C_READ, (Elf *)0);
if (elf_kind(arf) != ELF_K_AR)
{
        /* not an archive */
}
/* initial processing */
/* set offset = ... for desired member header */
while (elf_rand(arf, offset) == offset)
{
        if ((elf = elf_begin(fildes, ELF_C_READ, arf)) == 0)
                break;
        if ((ehdr = elf32_getehdr(elf)) != 0)
        {
                /* process archive member ... */
        }
        /* set offset = ... for desired member header */
}
```

An archive starts with a ''magic string'' that has **SARMAG** bytes; the initial archive member follows immediately. An application could thus provide the following function to rewind an archive (the function returns –1 for errors and 0 otherwise).

```
#include <ar.h>
#include <libelf.h>

int
rewindelf(Elf *elf)
{
    if (elf_rand(elf, (size_t)SARMAG) == SARMAG)
        return 0;
    return –1;
}
```

The following outline shows how one might create a new ELF file. This example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR|O_TRUNC|O_CREAT, 0666);
if ((elf = elf_begin(fildes, ELF_C_WRITE, (Elf *)0)) == 0)
        return;
ehdr = elf32_newehdr(elf);
phdr = elf32_newphdr(elf, count);
scn = elf_newscn(elf);
shdr = elf32_getshdr(scn);
data = elf_newdata(scn);
elf_update(elf, ELF_C_WRITE);
elf_end(elf);
```

Finally, the following outline shows how one might update an existing ELF file. Again, this example is simplified to show the overall flow.

```
elf_version(EV_CURRENT);
fildes = open("path/name", O_RDWR);
elf = elf_begin(fildes, ELF_C_RDWR, (Elf *)0);

/* add new or delete old information */
. . .

/* ensure that the memory image of the file is complete */
elf_update(elf, ELF_C_NULL);

elf_update(elf, ELF_C_WRITE);   /* update file */
elf_end(elf);
```

Notice that both file creation examples open the file with write *and* read permissions. On systems that support **mmap**, the library uses it to enhance performance, and **mmap** requires a readable file descriptor. Although the library can use a write-only file descriptor, the application will not obtain the performance advantages of **mmap**.

SEE ALSO

creat(2), **lseek**(2), **mmap**(2), **open**(2), **elf**(3E), **elf32_getehdr**(3E), **elf_cntl**(3E), **elf_getarhdr**(3E), **elf_getarsym**(3E), **elf_getbase**(3E), **elf_getdata**(3E), **elf_getscn**(3E), **elf_kind**(3E), **elf_rawfile**(3E), **elf_update**(3E), **elf_version**(3E), **ar**(4)