NAME

elf_getdata, elf_newdata, elf_rawdata – get section data

SYNOPSIS

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

#include <libelf.h>

Elf_Data ∗**elf_getdata(Elf_Scn** ∗*scn*, **Elf_Data** ∗*data*);

Elf_Data ∗**elf_newdata(Elf_Scn** ∗*scn*);

Elf_Data ∗**elf_rawdata(Elf_Scn** ∗*scn*, **Elf_Data** ∗*data*);

MT-LEVEL

Unsafe

DESCRIPTION

These functions access and manipulate the data associated with a section descriptor, *scn*. When reading an existing file, a section will have a single data buffer associated with it. A program may build a new section in pieces, however, composing the new data from multiple data buffers. For this reason, the data for a section should be viewed as a list of buffers, each of which is available through a data descriptor.

elf_getdata( ) lets a program step through a section's data list. If the incoming data descriptor, *data*, is null, the function returns the first buffer associated with the section. Otherwise, *data* should be a data descriptor associated with *scn*, and the function gives the program access to the next data element for the section. If *scn* is null or an error occurs, **elf_getdata( )** returns a null pointer.

elf_getdata( ) translates the data from file representations into memory representations (see **elf32_xlatetof**(3E)) and presents objects with memory data types to the program, based on the file's *class* (see **elf**(3E)). The working library version (see **elf_version**(3E)) specifies what version of the memory structures the program wishes **elf_getdata( )** to present.

elf_newdata( ) creates a new data descriptor for a section, appending it to any data elements already associated with the section. As described below, the new data descriptor appears empty, indicating the element holds no data. For convenience, the descriptor's type (**d_type** below) is set to **ELF_T_BYTE**, and the version (**d_version** below) is set to the working version. The program is responsible for setting (or changing) the descriptor members as needed. This function implicitly sets the **ELF_F_DIRTY** bit for the section's data (see **elf_flagdata**(3E)). If *scn* is null or an error occurs, **elf_newdata( )** returns a null pointer.

elf_rawdata( ) differs from **elf_getdata( )** by returning only uninterpreted bytes, regardless of the section type. This function typically should be used only to retrieve a section image from a file being read, and then only when a program must avoid the automatic data translation described below. Moreover, a program may not close or disable (see **elf_cntl**(3E)) the file descriptor associated with *elf* before the initial raw operation, because **elf_rawdata( )** might read the data from the file to ensure it doesn't interfere with **elf_getdata( )**. See **elf_rawfile**(3E) for a related facility that applies to the entire file. When **elf_getdata( )** provides the right translation, its use is recommended over **elf_rawdata( )**. If *scn* is null or an error occurs, **elf_rawdata( )** returns a null pointer.

The **Elf_Data** structure includes the following members:

```
void        *d_buf;
Elf_Type    d_type;
size_t      d_size;
off_t       d_off;
size_t      d_align;
unsigned    d_version;
```

These members are available for direct manipulation by the program. Descriptions appear below.

d_buf A pointer to the data buffer resides here. A data element with no data has a null pointer.

d_type This member's value specifies the type of the data to which **d_buf** points. A section's type determines how to interpret the section contents, as summarized below.

d_size This member holds the total size, in bytes, of the memory occupied by the data. This may differ from the size as represented in the file. The size will be zero if no data exist. (See the discussion of **SHT_NOBITS** below for more information.)

d_off This member gives the offset, within the section, at which the buffer resides. This offset is relative to the file's section, not the memory object's.

d_align This member holds the buffer's required alignment, from the beginning of the section. That is, **d_off** will be a multiple of this member's value. For example, if this member's value is four, the beginning of the buffer will be four-byte aligned within the section. Moreover, the entire section will be aligned to the maximum of its constituents, thus ensuring appropriate alignment for a buffer within the section and within the file.

d_version This member holds the version number of the objects in the buffer. When the library originally read the data from the object file, it used the working version to control the translation to memory objects.

DATA ALIGNMENT

As mentioned above, data buffers within a section have explicit alignment constraints. Consequently, adjacent buffers sometimes will not abut, causing ''holes'' within a section. Programs that create output files have two ways of dealing with these holes.

First, the program can use **elf_fill( )** to tell the library how to set the intervening bytes. When the library must generate gaps in the file, it uses the fill byte to initialize the data there. The library's initial fill value is zero, and **elf_fill( )** lets the application change that.

Second, the application can generate its own data buffers to occupy the gaps, filling the gaps with values appropriate for the section being created. A program might even use different fill values for different sections. For example, it could set text sections' bytes to no-operation instructions, while filling data section holes with zero. Using this technique, the library finds no holes to fill, because the application eliminated them.

SECTION AND MEMORY TYPES

elf_getdata( ) interprets sections' data according to the section type, as noted in the section header available through **elf32_getshdr( )**. The following table shows the section types and how the library represents them with memory data types for the 32-bit file class. Other classes would have similar tables. By implication, the memory data types control translation by **elf32_xlatetof**(3E).

Section Type	**Elf_Type**	32-Bit Type
SHT_DYNAMIC	**ELF_T_DYN**	**Elf32_Dyn**
SHT_DYNSYM	**ELF_T_SYM**	**Elf32_Sym**
SHT_HASH	**ELF_T_WORD**	**Elf32_Word**
SHT_NOBITS	**ELF_T_BYTE**	**unsigned char**
SHT_NOTE	**ELF_T_BYTE**	**unsigned char**
SHT_NULL	*none*	*none*
SHT_PROGBITS	**ELF_T_BYTE**	**unsigned char**
SHT_REL	**ELF_T_REL**	**Elf32_Rel**
SHT_RELA	**ELF_T_RELA**	**Elf32_Rela**
SHT_STRTAB	**ELF_T_BYTE**	**unsigned char**
SHT_SYMTAB	**ELF_T_SYM**	**Elf32_Sym**
SHT_SUNW_verdef	**ELF_T_VDEF**	**Elf32_Verdef**
SHT_SUNW_verneed		**ELF_T_VNEEDElf32_Verneed**
SHT_SUNW_versym	**ELF_T_HALF**	**Elf32_Versym**
other	**ELF_T_BYTE**	**unsigned char**

elf_rawdata( ) creates a buffer with type **ELF_T_BYTE**.

As mentioned above, the program's working version controls what structures the library creates for the application. The library similarly interprets section types according to the versions. If a section type belongs to a version newer than the application's working version, the library does not translate the section data. Because the application cannot know the data format in this case, the library presents an untranslated buffer of type **ELF_T_BYTE**, just as it would for an unrecognized section type.

A section with a special type, **SHT_NOBITS**, occupies no space in an object file, even when the section header indicates a non-zero size. **elf_getdata( )** and **elf_rawdata( )** work on such a section, setting the *data* structure to have a null buffer pointer and the type indicated above. Although no data are present, the **d_size** value is set to the size from the section header. When a program is creating a new section of type **SHT_NOBITS**, it should use **elf_newdata( )** to add data buffers to the section. These empty data buffers should have the **d_size** members set to the desired size and the **d_buf** members set to null.

EXAMPLES

The following fragment obtains the string table that holds section names (ignoring error checking). See **elf_strptr**(3E) for a variation of string table handling.

```
ehdr = elf32_getehdr(elf);
scn = elf_getscn(elf, (size_t)ehdr→e_shstrndx);
shdr = elf32_getshdr(scn);
if (shdr→sh_type != SHT_STRTAB)
{
        /∗ not a string table ∗/
}
data = 0;
if ((data = elf_getdata(scn, data)) == 0 | | data→d_size == 0)
{
        /∗ error or no data ∗/
}
```

The **e_shstrndx** member in an ELF header holds the section table index of the string table. The program gets a section descriptor for that section, verifies it is a string table, and then retrieves the data. When this fragment finishes, **data→d_buf** points at the first byte of the string table, and **data→d_size** holds the string table's size in bytes.

SEE ALSO **elf**(3E), **elf32_getehdr**(3E), **elf32_getshdr**(3E), **elf32_xlatetof**(3E), **elf_cntl**(3E), **elf_fill**(3E), **elf_flagdata**(3E), **elf_getscn**(3E), **elf_rawfile**(3E), **elf_strptr**(3E) **elf_version**(3E)