`libelf` by Example

Joseph Koshy

January 12, 2010

Contents

Preface

This tutorial introduces the `libelf` library being developed at the ElfToolChain project on SourceForge.Net. It shows how this library can be used to create tools that can manipulate ELF objects for native and non-native architectures.

The ELF(3)/GELF(3) APIs are discussed, as is handling of ar(1) archives. The ELF format is discussed to the extent needed to understand the use of the ELF(3) library.

Knowledge of the C programming language is a pre-requisite.

Legal Notice

Disclaimer

Chapter 1

Introduction

ELF stands for Extensible Linking Format. It is a format for use by compilers, linkers, loaders and other tools that manipulate object code.

The ELF specification was released to the public in 1990 as an "open standard" by a group of vendors. As a result of its ready availability it has been widely adopted by industry and the open-source community. The ELF standard supports 32- and 64-bit architectures of both big and little-endian kinds, and supports features like cross-compilation and dynamic shared libraries. ELF also supports the special compilation needs of the C++ language. Among the current set of open-source operating systems, FreeBSD switched to using ELF as its object format in FreeBSD 3.0 (October 1998).

The `libelf` library provides an API set (ELF(3) and GELF(3)) for application writers to read and write ELF objects with. The library eases the task of writing cross-tools that can run on one machine architecture and manipulate ELF objects for another.

Rationale for this tutorial

The ELF(3) and GELF(3) API set is large, with over 80 callable functions. So the task of getting started with the library can appear daunting at first glance. This tutorial has been written to provide a gentle introduction to the API set.

Target Audience

This tutorial would be of interest to developers wanting to create ELF processing tools using the `libelf` library.

1.1 Tutorial Overview

The tutorial covers the following:

- The basics of the ELF format (as much as is needed to understand how to use the API set); how the ELF format structures the contents of executables, relocatables and shared objects.

- How to get started building applications that use the `libelf` library.
- The basic abstractions offered by the ELF(3) and GELF(3) APIs—how the ELF library abstracts out the ELF class and endianness of ELF objects and allows an application to work with native forms of these objects, while the library translates to and from the desired target representation behind the scenes.
- How to use the APIs in the library to look inside an ELF object and examine its executable header, program header table and its component sections.
- How to create a new ELF object using the ELF library.
- An introduction to the class-independent GELF(3) interfaces, and when and where to use them instead of the class-dependent functions in the ELF(3) API set.
- How to process **ar** archives using the facilities provided by the library.

1.2 Tutorial Structure

One of the goals of this tutorial is to illustrate how to write programs using `libelf`. So we will jump into writing code at the earliest opportunity. As we progress through the examples, we introduce the concepts necessary to understand what is happening "behind the scenes."

Chapter 2 on page 11 covers the basics involved in getting started with the ELF(3) library—how to compile and link an application that uses `libelf`. We look at the way a working ELF version number is established by an application, how a handle to ELF objects are obtained, and how error messages from the ELF library are reported. The functions used in this section include `elf_begin`, `elf_end`, `elf_errmsg`, `elf_errno`, `elf_kind` and `elf_version`.

Chapter 3 on page 15 shows how an application can look inside an ELF object and understand its basic structure. Along the way we will examine the way the ELF objects are laid out. Other key concepts covered are the notions of "file representation" and "memory representation" of ELF data types. New APIs covered include `elf_getident`, `elf_getphdrnum`, `elf_getshdrnum`, `elf_getshdrstrndx`, `gelf_getehdr` and `gelf_getclass`.

Chapter 4 on page 25 describes the ELF program header table and shows how an application can retrieve this table from an ELF object. This chapter introduces the `gelf_getphdr` function.

Chapter 5 on page 33 then looks at how data is stored in ELF sections. A program that looks at ELF sections is examined. The `Elf_Scn` and `Elf_Data` data types used by the library are introduced. The functions covered in this chapter include `elf_getscn`, `elf_getdata`, `elf_nextscn`, `elf_strptr`, and `gelf_-getshdr`.

Chapter 6 on page 43 looks at how we create ELF objects. We cover the rules in ordering of the individual API calls when creating ELF objects. We look at the library's object layout rules and how an application can choose to override these. The APIs covered include `elf_fill`, `elf32_getshdr`, `elf32_new-`

`ehdr`, `elf32_newphdr`, `elf_flagphdr`, `elf_ndxscn`, `elf_newdata`, `elf_newscn`, and `elf_update`.

The `libelf` library also assists applications that need to read **ar** archives. Chapter 7 on page 51 covers how to use the ELF(3) library to handle **ar** archives. This chapter covers the use of the `elf_getarhdr`, `elf_getarsym`, `elf_next` and `elf_rand` functions.

Chapter 8 on page 57 ends the tutorial with suggestions for further reading.

Chapter 2

Getting Started

Let us dive in and get a taste of programming with `libelf`.

2.1 Example: Getting started with `libelf`

Our first program (Program 1, listing 2.1) will open a filename presented to it on its command line and retrieve the file type as recognized by the ELF library.

This example is covers the basics involved in using `libelf`; how to compile a program using `libelf`, how to initialize the library, how to report errors, and how to wind up.

Listing 2.1: Program 1

```
#include <err.h>
#include <fcntl.h>
#include <libelf.h> [1]
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>

int
main(int argc, char **argv)
{
    int fd;
    Elf *e; [2]
    char *k;
    Elf_Kind ek; [3]

    if (argc != 2)
        errx(EX_USAGE, "usage:␣%s␣file-name", argv[0]);


    if (elf_version(EV_CURRENT) == EV_NONE) [4]
        errx(EX_SOFTWARE, "ELF␣library␣initialization␣"
            "failed:␣%s", elf_errmsg(-1));
    if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
        err(EX_NOINPUT, "open␣\%s\"␣failed", argv[1]);

    if ((e = elf_begin(fd, ELF_C_READ 5 , NULL)) == NULL)
        errx(EX_SOFTWARE, "elf_begin()␣failed:␣%s.",
            elf_errmsg(-1)); 6

    ek = elf_kind(e); 7

    switch (ek) {
    case ELF_K_AR:
        k = "ar(1)␣archive";
        break;
    case ELF_K_ELF:
        k = "elf␣object";
        break;
    case ELF_K_NONE:
        k = "data";
        break;
    default:
        k = "unrecognized";
    }

    (void) printf("%s:␣%s\n", argv[1], k);

    (void) elf_end(e); 8
    (void) close(fd);

    exit(EX_OK);
}
```

1 The functions and dataypes that make up the ELF(3) API are declared in the header `libelf.h`. This file must be included in every application that desires to use the `libelf` library.

2 The ELF(3) library uses an opaque type `Elf` as a handle for the ELF object being processed.

4 Before the functions in the library can be invoked, an application must indicate to the library the version of the ELF specification it is expecting to use. This is done by the call to `elf_version`.

A call to `elf_version` is mandatory before other functions in the ELF library can be invoked.

There are multiple version numbers that come into play when an application is manipulating an ELF object.

- First, there is the version of the ELF specification (v_1) that the application understands.

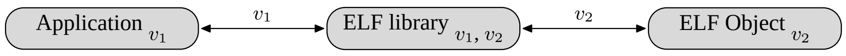


Figure 2.1: ELF versions

- Second, we have the ELF version associated with the ELF object being processed (v_2).
- Third, we have the versions recognized by the `libelf` library: v_1 and v_2. The library may know how to translate between versions v_1 and v_2.

In figure 2.1 the application expects to work with ELF specification version (v_1). The ELF object file conforms to ELF specification version (v_2). The library understands both version v_1 and v_2 of ELF semantics and so is able to mediate between the application and the ELF object.

In practice, the ELF version has not changed since inception, so the current version (`EV_CURRENT`) is 1.

5 The `elf_begin` function takes an open file descriptor and converts it an `Elf` handle according to the command specified.

The second parameter to `elf_begin` can be one of '`ELF_C_READ`' for opening an ELF object for reading, '`ELF_C_WRITE`' for creating a new ELF object, or '`ELF_C_RDWR`' for opening an ELF object for updates. The mode with which file descriptor `fd` was opened with must be consistent with the this parameter.

The third parameter to `elf_begin` is only used when processing **ar** archives. We will look at **ar** archive processing in chapter 7 on page 51.

6 When the ELF library encounters an error, it records an error number in an internal location. This error number may be retrieved using the `elf_errno` function.

The `elf_errmsg` function returns a human readable string describing the error number passed in. As a programming convenience, a value of -1 denotes the current error number.

3 **7** The ELF library can operate on **ar** archives and ELF objects. The function `elf_kind` returns the kind of object associated with an `Elf` handle. The return value of the `elf_kind` function is one of the values defined by the `Elf_Kind` enumeration in `libelf.h`.

8 When you are done with a handle, it is good practice to release its resources using the `elf_end` function.

Now it is time to get something running.

Save the listing in listing 2.1 on page 11 to file `prog1.c` and then compile and run it as shown in listing 2.2 on the next page.

Listing 2.2: Compiling and running prog1

```
% cc -o prog1 prog1.c -lelf 1
% ./prog1 prog1 2
prog1: elf object
% ./prog1 /usr/lib/libc.a 3
/usr/lib/libc.a: ar(1) archive
```

$\boxed{1}$ The `-lelf` option to the **cc** comand informs it to link **prog1** against the `libelf` library.

$\boxed{2}$ We invoke **prog1** on itself, and it recognizes its own executable as ELF object. All is well.

$\boxed{3}$ Here we see that **prog1** recognizes an **ar** archive correctly.

Congratulations! You have created your first ELF handling program using `libelf`.

In the next chapter we will look deeper into the ELF format and learn how to pick an ELF object apart into its component pieces.

Chapter 3

Peering Inside an ELF Object

Next, we will look inside an ELF object. We will look at how an ELF object is laid out and we will introduce its major parts, namely the ELF executable header, the ELF program header table and ELF sections. Along the way we will look at the way `libelf` handles non-native objects.

3.1 The Layout of an ELF file

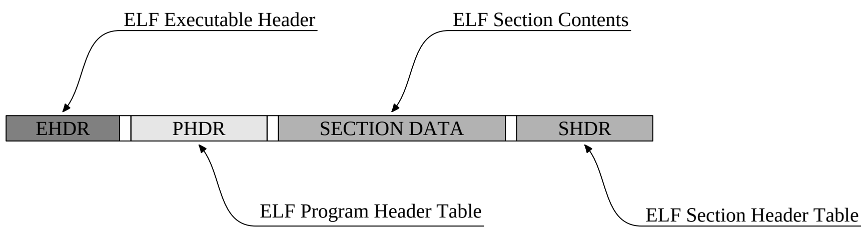
As an object format, ELF supports multiple kinds of objects:

- Compilers generate *relocatable objects* that contain fragments of machine code along with the "glue" information needed when combining multiple such objects to form a final executable.
- *Executables* are programs that are in a form that an operating system can launch in a process. The process of forming executables from collections of relocatable objects is called *linking*.
- *Dynamically loadable objects* are those that can be loaded by an executable after it has started executing. Dynamically loadable *shared libraries* are examples of such objects.

An ELF object consists of a mandatory header named the *ELF executable header*, followed by optional content in the form of ELF *program header table* and zero or more *ELF sections* (see figure 3.1 on the following page).

- The ELF *executable header* defines the structure of the rest of the file. This header is *always* present in a valid ELF file. It describes the class of the file (whether 32 bit or 64 bit), the type (whether a relocatable, executable or shared object), and the byte ordering used (little endian or big endian). It also describes the overall layout of the ELF object. The ELF header is described below.
- An optional ELF *program header table* is present in executable objects and contains information used by at program load time. The program header table is described in chapter 4 on page 25.

Figure 3.1: The layout of a typical ELF File



- The contents of a relocatable ELF object are contained in *ELF sections*. These sections are described by entries in an *ELF section header table*. This table has one entry per section present in the file. Chapter 5 on page 33 describes ELF sections and the section header table in further detail.

Every ELF object is associated with three parameters:

- Its *class* denotes whether it is a 32 bit ELF object (`ELFCLASS32`) or a 64 bit (`ELFCLASS64`) one.
- Its *endianness* denotes whether it is using little-endian (`ELFDATA2LSB`) or big-endian addressing (`ELFDATA2MSB`).
- Finally, each ELF object is associated with a *version* number as discussed in chapter 2 on page 11.

These parameters are stored in the ELF executable header. Let us now take a closer look at the ELF executable header.

The ELF Executable Header

Table 3.1 on the facing page describes the layout of an ELF executable header using a "C-like" notation.

1 The first 16 bytes (the `e_ident` array) contain values that determine the ELF class, version and endianness of the rest of the file. See figure 3.2.

Figure 3.2: The `e_ident` array

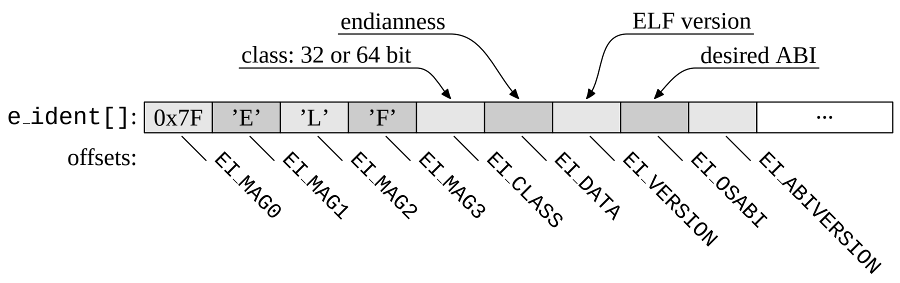
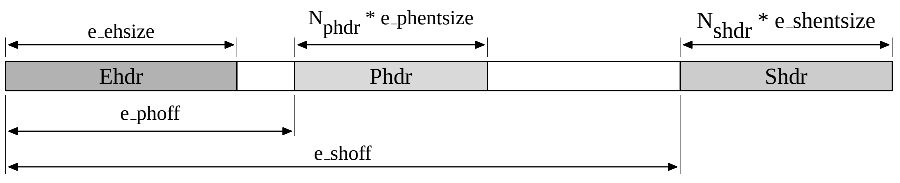


Table 3.1: The ELF Executable Header

	32 bit Executable Header	**64 bit Executable Header**
	`typedef struct {`	`typedef struct {`
$\boxed{1}$	`unsigned char e_ident[16];`	`unsigned char e_ident[16];`
$\boxed{2}$	`uint16_t e_type;`	`uint16_t e_type;`
$\boxed{3}$	`uint16_t e_machine;`	`uint16_t e_machine;`
	`uint32_t e_version;`	`uint32_t e_version;`
	`uint32_t e_entry;`	`uint32_t e_entry;`
$\boxed{4}$	`uint32_t e_phoff;`	`uint64_t e_phoff;`
$\boxed{5}$	`uint32_t e_shoff;`	`uint64_t e_shoff;`
	`uint32_t e_flags;`	`uint32_t e_flags;`
	`uint16_t e_ehsize;`	`uint16_t e_ehsize;`
	`uint16_t e_phentsize;`	`uint16_t e_phentsize;`
$\boxed{6}$	`uint16_t e_phnum;`	`uint16_t e_phnum;`
$\boxed{7}$	`uint16_t e_shnum;`	`uint16_t e_shnum;`
$\boxed{8}$	`uint16_t e_shstrndx;`	`uint16_t e_shstrndx;`
	`} Elf32_Ehdr;`	`} Elf64_Ehdr;`

Figure 3.3: The ELF Executable Header and Object Layout



The first 4 bytes of an ELF object are always 0x7F, 'E', 'L' and 'F'. The next three bytes specify the class of the ELF object (`ELFCLASS32` or `ELFCLASS64`), its data ordering (`ELFDATA2LSB` or `ELFDATA2MSB`) and the ELF version the object conforms to. With this information on hand, the `libelf` library can then interpret the rest of the ELF executable header correctly.

$\boxed{2}$ The `e_type` member determines the type of the ELF object. For example, it would contain a '1' (`ET_REL`) in a relocatable or '3' (`ET_DYN`) in a shared object.

$\boxed{3}$ The `e_machine` member describes the machine architecture this ELF object is for. Example values are '3' (`EM_386`) for the Intel® i386^{TM} architecture and '20' (`EM_PPC`) for the 32-bit PowerPC^{TM} architecture.

$\boxed{4}$ $\boxed{5}$ The ELF executable header also describes the layout of the rest of the ELF object (Figure 3.3). The `e_phoff` and `e_shoff` fields contain the file offsets where the ELF program header table and ELF section header table reside. These fields are zero if the file does not have a program header table or section header table respectively. The sizes of these components are determined by the e_phentsize and e_shentsize members respectively in conjunction with the number of entries in these tables.

The ELF executable header describes its own size (in bytes) in field e_ehsize.

$\boxed{6}$ $\boxed{7}$ The e_phnum and e_shnum fields usually contain the number of ELF program header table entries and section header table entries. Note that these fields are only 2 bytes wide, so if an ELF object has a large number of sections or program header table entries, then a scheme known as *Extended Numbering* (section 3.1 on the next page) is used to encode the actual number of sections or program header table entries. When extended numbering is in use these fields will contain "magic numbers" instead of actual counts.

$\boxed{8}$ If the ELF object contains sections, then we need a way to get at the names of sections. Section names are stored in a string table. The e_shstrndx stores the section index of this string table (see 3.1 on the facing page) so that processing tools know which string table to use for retrieving the names of sections. We will cover ELF string tables in more detail in section 5.1.1 on page 37.

The fields e_entry and e_flags are used for executables and are placed in the executable header for easy access at program load time. We will not look at them further in this tutorial.

ELF Class- and Endianness- Independent Processing

Now let us look at the way the libelf API set abstracts out ELF class and endianness for us.

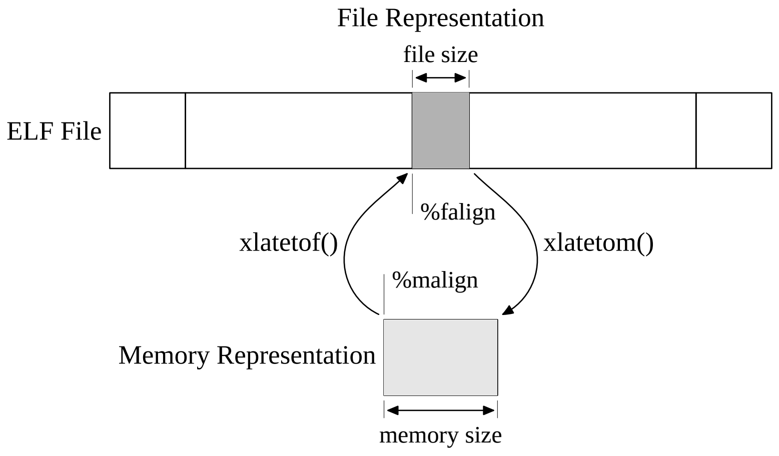
Imagine that you are writing an ELF processing application that is going to support processing of non-native binaries (say for a machine with a different native endianness and word size). It should be evident that ELF data structures would have two distinct representations: an *in-memory representation* that follows the rules for the machine architecture that the application running on, and an *in-file representation* that corresponds to the target architecture for the ELF object.

The application would like to manipulate data in its native memory representation. This memory representation would conform to the native endianness of the host's CPU and would conform to the address alignment and structure padding requirements set by the host's machine architecture.

When this data is written into the target object it may need to be formatted differently. For example, it could be packed differently compared to the "native" memory representation and may have to be laid out according a different set of rules for alignment. The endianness of the data in-file could be different from that of the in-memory representation.

Figure 3.4 on the facing page depicts the relationship between the file and memory representation of an ELF data structure. As shown in the figure, the

Figure 3.4: File and Memory Representations



size of an ELF data structure in the file could be different from its size in memory. The alignment restrictions (`%falign` and `%malign` in the figure) could be different. The byte ordering of the data could be different too.

The ELF(3) and GELF(3) API set can handle the conversion of ELF data structures to and from their file and memory representations automatically. For example, when we read in the ELF executable header in program 3.1 on the next page below, the `libelf` library will automatically do the necessary byteswapping and alignment adjustments for us.

For applications that desire finer-grain control over the conversion process, the `elf`*NN*`_xlatetof` and `elf`*NN*`_xlatetom` functions are available. These functions will translate data buffers containing ELF data structures between their memory and file represention.

Extended numbering

The `e_shnum`, `e_phnum` and `e_shstrndx` fields of the ELF executable header are only 2 bytes long and are not physically capable of representing numbers larger than 65535. For ELF objects with a large number of sections, we need a different way of encoding section numbers.

ELF objects with such a large number of sections can arise due to the way GCC copes with C++ templates. When compiling C++ code which uses templates, GCC generates many sections with names following the pattern ".gnu.linkonce.*name*". While each compiled ELF relocatable object will now contain replicated data, the linker is expected to treat such sections specially at the final link stage, discarding all but one of each section.

When extended numbering is in use:

- The `e_shnum` field of the ELF executable header is always zero and the true number of sections is stored in the `sh_size` field of the section header table entry at index 0.
- The true index of the section name string table is stored in field `sh_link`

field of the zeroth entry of the section header table, while the e_shstrndx field of the executable header set to SHN_XINDEX (0xFFFF).

- For extended program header table numbering the scheme is similar, with the e_phnum field of the executable header holding the value PN_XNUM (0xFFFF) and the sh_link field of the zeroth section header table holding the actual number of program header table entries.

An application may use the functions elf_getphdrnum, elf_getshdrnum and elf_getshdrstrndx to retrieve the correct value of these fields when extended numbering is in use.

3.2 Example: Reading an ELF executable header

We will now look at a small program that will print out the ELF executable header in an ELF object. For this example we will introduce the GELF(3) API set.

The ELF(3) API is defined in terms of ELF class-dependent types (Elf32_-Ehdr, Elf64_Shdr, etc.) and consequently has many operations that have both 32- and 64- bit variants. So, in order to retrieve an ELF executable header from a 32 bit ELF object we would need to use the function elf32_getehdr, which would return a pointer to an Elf32_Ehdr structure. For a 64-bit ELF object, the function we would need to use would be elf64_getehdr, which would return a pointer to an Elf64_Ehdr structure. This duplication is awkward when you want to write applications that can transparently process either class of ELF objects.

The GELF(3) APIs provide an ELF class independent way of writing ELF applications. These functions are defined in terms of "generic" types that are large enough to hold the values of their corresponding 32- and 64- bit ELF types. Further, the GELF(3) APIs always work on *copies* of ELF data structures thus bypassing the problem of 32- and 64- bit ELF data structures having incompatible memory layouts. You can freely mix calls to GELF(3) and ELF(3) functions.

The downside of using the GELF(3) APIs is the extra copying and conversion of data that occurs. This overhead is usually not significant to most applications.

Listing 3.1: Program 2

```
/*
 * Print the ELF Executable Header from an ELF object.
 */

#include <err.h>
#include <fcntl.h>
#include <gelf.h> [1]
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>
#include <vis.h>
```

```
int
main(int argc, char **argv)
{
    int i, fd;
    Elf *e;
    char *id, bytes[5];
    size_t n;
    GElf_Ehdr ehdr; [2]

    if (argc != 2)
        errx(EX_USAGE, "usage:␣%s␣file-name", argv[0]);

    if (elf_version(EV_CURRENT) == EV_NONE)
        errx(EX_SOFTWARE, "ELF␣library␣initialization␣"
            "failed:␣%s", elf_errmsg(-1));

    if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
        err(EX_NOINPUT, "open␣\"%s\"␣failed", argv[1]);

    if ((e = elf_begin(fd, ELF_C_READ, NULL)) == NULL)
        errx(EX_SOFTWARE, "elf_begin()␣failed:␣%s.",
            elf_errmsg(-1));

    if (elf_kind(e) != ELF_K_ELF)
        errx(EX_DATAERR, "\"%s\"␣is␣not␣an␣ELF␣object.",
            argv[1]);

    if (gelf_getehdr(e, &ehdr) == NULL) [3]
        errx(EX_SOFTWARE, "getehdr()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((i = gelf_getclass(e)) == ELFCLASSNONE) [4]
        errx(EX_SOFTWARE, "getclass()␣failed:␣%s.",
            elf_errmsg(-1));

    (void) printf("%s:␣%d-bit␣ELF␣object\n", argv[1],
        i == ELFCLASS32 ? 32 : 64);

    if ((id = elf_getident(e, NULL)) == NULL) [5]
        errx(EX_SOFTWARE, "getident()␣failed:␣%s.",
            elf_errmsg(-1));

    (void) printf("%3s␣e_ident[0..%1d]␣%7s", "␣",
        EI_ABIVERSION, "␣");

    for (i = 0; i <= EI_ABIVERSION; i++) {
        (void) vis(bytes, id[i], VIS_WHITE, 0);
        (void) printf("␣['%s'␣%X]", bytes, id[i]);
    }
    (void) printf("\n");

#define         PRINT_FMT         "␣␣␣␣%-20s␣0x%jx\n"
#define         PRINT_FIELD(N) do { \
        (void) printf(PRINT_FMT, #N, (uintmax_t) ehdr.N); \
    } while (0)

    PRINT_FIELD(e_type); 6
    PRINT_FIELD(e_machine);
    PRINT_FIELD(e_version);
    PRINT_FIELD(e_entry);
    PRINT_FIELD(e_phoff);
    PRINT_FIELD(e_shoff);
    PRINT_FIELD(e_flags);
    PRINT_FIELD(e_ehsize);
    PRINT_FIELD(e_phentsize);
    PRINT_FIELD(e_shentsize);

    if (elf_getshdrnum(e, &n) != 0) 7
        errx(EX_SOFTWARE, "getshdrnum()␣failed:␣%s.",
            elf_errmsg(-1));
    (void) printf(PRINT_FMT, "(shnum)", (uintmax_t) n);

    if (elf_getshdrstrndx(e, &n) != 0) 8
        errx(EX_SOFTWARE, "getshdrstrndx()␣failed:␣%s.",
            elf_errmsg(-1));
    (void) printf(PRINT_FMT, "(shstrndx)", (uintmax_t) n);

    if (elf_getphdrnum(e, &n) != 0) 9
        errx(EX_SOFTWARE, "getphdrnum()␣failed:␣%s.",
            elf_errmsg(-1));
    (void) printf(PRINT_FMT, "(phnum)", (uintmax_t) n);

    (void) elf_end(e);
    (void) close(fd);
    exit(EX_OK);
}
```

1 Programs using the GELF(3) API set need to include `gelf.h`.

2 The GELF(3) functions always operate on a local copies of data structures. The `GElf_Ehdr` type has fields that are large enough to contain values for a 64 bit ELF executable header.

3 We retrieve the ELF executable header using function `gelf_getehdr`. This function will translate the ELF executable header in the ELF object being read to the appropriate in-memory representation for type `GElf_Ehdr`. For example, if a 32-bit ELF object is being examined, then the values in its executable header would be appropriately converted (expanded and/or byteswapped) by this function.

$\boxed{4}$ The `gelf_getclass` function retrieves the ELF class of the object being examined.

$\boxed{5}$ Here we show the use of the `elf_getident` function to retrieve the contents of the `e_ident[]` array from the underlying file. These bytes would also be present in the `e_ident` member of the `ehdr` structure.

We print the first few bytes of the `e_ident` field of the ELF executable header.

$\boxed{6}$ Following the `e_ident` bytes, we print the values of some of the fields of the ELF executable header structure.

$\boxed{7}$ $\boxed{8}$ $\boxed{9}$ The functions `elf_getphdrnum`, `elf_getshdrnum` and `elf_get-shdrstrndx` described in section 3.1 on page 19 should be used to retrieve the count of program header table entries, the number of sections, and the section name string table index respectively. Using these functions insulates your application from the quirks of extended numbering.

Save the program in listing 3.1 on page 20 to file `prog2.c` and then compile and run it as shown in listing 3.2.

Listing 3.2: Compiling and Running prog2

```
% cc -o prog2 prog2.c -lelf [1]
% ./prog2 prog2 [2]
prog2: 64-bit ELF object
    e_ident[0..8]     ['\^?' 7F] ['E' 45] ['L' 4C] ['F' 46] \
    ['\^B' 2] ['\^A' 1] ['\^A' 1] ['\^I' 9] ['\^@' 0]
    e_type            0x2
    e_machine         0x3e
    e_version         0x1
    e_entry           0x400a10
    e_phoff           0x40
    e_shoff           0x16f8
    e_flags           0x0
    e_ehsize          0x40
    e_phentsize       0x38
    e_shentsize       0x40
    (shnum)           0x18
    (shstrndx)        0x15
    (phnum)           0x5
```

$\boxed{1}$ The process for compiling and linking a GELF(3) using application is the same as that for ELF(3) programs.

$\boxed{2}$ We run our program on itself. This listing in this tutorial was generated on an AMD64^{TM} machine running FreeBSD.

You should now run **prog2** on other object files that you have lying around. Try it on a few non-native ELF object files too.

Chapter 4

Examining the Program Header Table

Before a program on disk can be executed by a processor it needs to brought into main memory. This process is conventionally called “loading”.

When loading an ELF object into memory, the operating system views it as comprising of “segments”. Each such segment is a contiguous region of data inside the ELF object that is associated with a particular protection characteristic (for example, read-only or read-write) and that gets placed at a specific virtual memory address.

For example, the FreeBSD^{TM} operating system expects executables to have an “executable” segment containing code, and a “data” segment containing statically initialized data.The executable segment would be mapped in with read and execute permissions and could be shared across multiple processes using the same ELF executable. The data segment would be mapped in with read and write permissions and would be made private to each process. For dynamically linked executables, the basic idea of grouping related parts of an ELF object into contiguous “segments” still holds, though there may be multiple segments of each type per process.

4.1 The ELF Program Header Table

The ELF *program header table* describes the segments present in an ELF file. The location of the program header table is described by the `e_phoff` field of the ELF executable header (see section 3.1 on page 16). The program header table is a contiguous array of program header table entries, one entry per segment.

Figure 4.1 on the following page shows graphically how the fields of a program header table entry specify the segment’s placement in file and in memory.

The structure of each program header table entry is shown in table 4.1 on the next page.

Figure 4.1: ELF Segment Placement

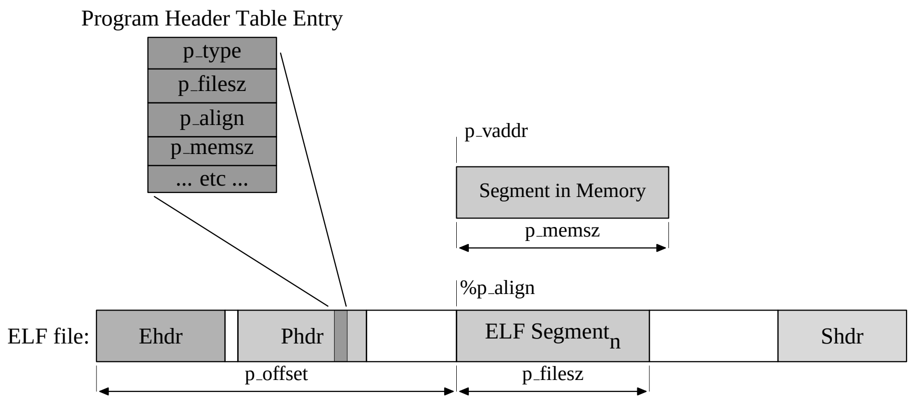


Table 4.1: ELF Program Header Table Entries

	32 bit PHDR Table Entry	**64 bit PHDR Table Entry**
	`typedef struct {`	`typedef struct {`
1	`Elf32_Word    p_type;`	`Elf64_Word    p_type;`
2	`Elf32_Off     p_offset;`	`Elf64_Word    p_flags;`
3	`Elf32_Addr    p_vaddr;`	`Elf64_Off     p_offset;`
4	`Elf32_Addr    p_paddr;`	`Elf64_Addr    p_vaddr;`
5	`Elf32_Word    p_filesz;`	`Elf64_Addr    p_paddr;`
6	`Elf32_Word    p_memsz;`	`Elf64_Xword   p_filesz;`
7	`Elf32_Word    p_flags;`	`Elf64_Xword   p_memsz;`
8	`Elf32_Word    p_align;`	`Elf64_Xword   p_align;`
	`} Elf32_Phdr;`	`} Elf64_Phdr;`

1 The type of the program header table entry is encoded using this field. It holds one of the `PT_*` constants defined in the system headers.

Examples include:

- A segment of type `PT_LOAD` is loaded into memory.
- A segment of type `PT_NOTE` contains auxiliary information. For example, core filesuse `PT_NOTE` sections to record the name of the process that dumped core.
- A `PT_PHDR` segment describes the program header table itself.

The ELF specification reserves type values from 0x60000000 (`PT_LOOS`) to 0x6FFFFFFF (`PT_HIOS`) for OS-private information. Values from 0x7000-0000 (`PT_LOPROC`) to 0x7FFFFFFF (`PT_HIPROC`) are similarly reserved for processor-specific information.

$\boxed{2}$ The `p_offset` field holds the file offset in the ELF object to the start of the segment being described by this table entry.

$\boxed{3}$ The virtual address this segment should be loaded at.

$\boxed{4}$ The physical address this segment should be loaded at. This field does not apply for userland objects.

$\boxed{5}$ The number of bytes the segment takes up in the file. This number is zero for segments that do not have data associated with them in the file.

$\boxed{6}$ The number of bytes the segment takes up in memory.

$\boxed{7}$ Additional flags that specify segment properties. For example, flag `PF_X` specifies that the segment in question should be made executable and flag `PF_W` denotes that the segment should be writable.

$\boxed{8}$ The alignment requirements of the segment both in memory and in the file. This field holds a value that is a power of two.

Note: The careful reader will note that the 32- and 64- bit `Elf_Phdr` structures are laid out differently in memory. These differences are handled for you by the functions in the `libelf` library.

4.2 Example: Reading a Program Header Table

We will now look at a program that will print out the program header table associated with an ELF object. We will continue to use the GELF(3) API set for this example. The ELF(3) API set also offers two ELF class-dependent APIs that retrieve the program header table from an ELF object: `elf32_getphdr` and `elf64_getphdr`, but these require us to know the ELF class of the object being handled.

Listing 4.1: Program 3

```
/*
 * Print the ELF Program Header Table in an ELF object.
 */

#include <err.h>
#include <fcntl.h>
#include <gelf.h> [1]
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>
#include <vis.h>

void
```

```
print_ptype(size_t pt) 7
{
    char *s;

#define C(V) case PT_##V: s = #V; break
    switch (pt) {
        C(NULL);          C(LOAD);          C(DYNAMIC);
        C(INTERP);        C(NOTE);          C(SHLIB);
        C(PHDR);          C(TLS);           C(SUNW_UNWIND);
        C(SUNWBSS);       C(SUNWSTACK);     C(SUNWDTRACE);
        C(SUNWCAP);
    default:
        s = "unknown";
        break;
    }
    (void) printf("␣\"%s\"", s);
#undef  C
}

int
main(int argc, char **argv)
{
    int i, fd;
    Elf *e;
    char *id, bytes[5];
    size_t n;
    GElf_Phdr phdr; 2

    if (argc != 2)
        errx(EX_USAGE, "usage:␣%s␣file-name", argv[0]);

    if (elf_version(EV_CURRENT) == EV_NONE)
        errx(EX_SOFTWARE, "ELF␣library␣initialization␣"
            "failed:␣%s", elf_errmsg(-1));

    if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
        err(EX_NOINPUT, "open␣\"%s\"␣failed", argv[1]);

    if ((e = elf_begin(fd, ELF_C_READ, NULL)) == NULL)
        errx(EX_SOFTWARE, "elf_begin()␣failed:␣%s.",
            elf_errmsg(-1));

    if (elf_kind(e) != ELF_K_ELF)
        errx(EX_DATAERR, "\"%s\"␣is␣not␣an␣ELF␣object.",
            argv[1]);

    if (elf_getphdrnum(e, &n) != 0) 3
        errx(EX_DATAERR, "elf_getphdrnum()␣failed:␣%s.",
            elf_errmsg(-1));

    for (i = 0; i < n; i++) { 4
```

```
        if (gelf_getphdr(e, i, &phdr) != &phdr) 5
            errx(EX_SOFTWARE, "getphdr()␣failed:␣%s.",
                elf_errmsg(-1));

        (void) printf("PHDR␣%d:\n");
#define        PRINT_FMT        "␣␣␣␣%-20s␣0x%jx"
#define        PRINT_FIELD(N) do { \
        (void) printf(PRINT_FMT, #N, (uintmax_t) phdr.N); \
    } while (0)
#define         NL() do { (void) printf("\n"); } while (0)
        PRINT_FIELD(p_type); 6
        print_ptype(phdr.p_type);       NL();
        PRINT_FIELD(p_offset);          NL();
        PRINT_FIELD(p_vaddr);           NL();
        PRINT_FIELD(p_paddr);           NL();
        PRINT_FIELD(p_filesz);          NL();
        PRINT_FIELD(p_memsz);           NL();
        PRINT_FIELD(p_flags);
        (void) printf("␣[");
        if (phdr.p_flags & PF_X)
            (void) printf("␣execute");
        if (phdr.p_flags & PF_R)
            (void) printf("␣read");
        if (phdr.p_flags & PF_W)
            (void) printf("␣write");
        printf("␣]");                   NL();
        PRINT_FIELD(p_align);           NL();
    }

    (void) elf_end(e);
    (void) close(fd);
    exit(EX_OK);
}
```

1 We need to include `gelf.h` in order to use the GELF(3) APIs.

2 The `GElf_Phdr` type has fields that are large enough to contain the values in an `Elf32_Phdr` type and an `Elf64_Phdr` type.

3 We retrieve the number of program header table entries using the function `elf_getphdrnum`. Note that the program header table is optional; for example, an ELF relocatable object will not have a program header table.

4 5 We iterate over all valid indices for the object's program header table, retrieving the table entry at each index using the `gelf_getphdr` function.

6 7 We then print out the contents of the entry so retrieved. We use a helper function `print_ptype` to convert the `p_type` member to a readable string.

Save the program in listing 4.1 on page 27 to file `prog3.c` and then compile and run it as shown in listing 4.2.

Listing 4.2: Compiling and Running prog3

```
% cc -o prog3 prog3.c -lelf 1
% ./prog3 prog3 2
PHDR 0:
    p_type               0x6 "PHDR" 3
    p_offset             0x34
    p_vaddr              0x8048034
    p_paddr              0x8048034
    p_filesz             0xc0
    p_memsz              0xc0
    p_flags              0x5 [ execute read ]
    p_align              0x4
PHDR 1:
    p_type               0x3 "INTERP" 4
    p_offset             0xf4
    p_vaddr              0x80480f4
    p_paddr              0x80480f4
    p_filesz             0x15
    p_memsz              0x15
    p_flags              0x4 [ read ]
    p_align              0x1
PHDR 2:
    p_type               0x1 "LOAD" 5
    p_offset             0x0
    p_vaddr              0x8048000
    p_paddr              0x8048000
    p_filesz             0xe67
    p_memsz              0xe67
    p_flags              0x5 [ execute read ]
    p_align              0x1000
PHDR 3:
    p_type               0x1 "LOAD" 6
    p_offset             0xe68
    p_vaddr              0x8049e68
    p_paddr              0x8049e68
    p_filesz             0x11c
    p_memsz              0x13c
    p_flags              0x6 [ read write ]
    p_align              0x1000
PHDR 4:
    p_type               0x2 "DYNAMIC"
    p_offset             0xe78
    p_vaddr              0x8049e78
    p_paddr              0x8049e78
    p_filesz             0xb8
    p_memsz              0xb8
    p_flags              0x6 [ read write ]
    p_align             0x4
PHDR 5:
    p_type              0x4 "NOTE"
    p_offset            0x10c
    p_vaddr             0x804810c
    p_paddr             0x804810c
    p_filesz            0x18
    p_memsz             0x18
    p_flags             0x4 [ read ]
    p_align             0x4
```

1 Compile and link the program in the standard way.

2 We make our program examine its own program header table. This listing was generated on an i386^{TM} machine running FreeBSD^{TM}.

3 The very first entry in this program header table describes the program header table itself.

4 An entry of type `PT_INTERP` is used to point the kernel to the “interpreter” associated with this ELF object. This is usually a runtime loader, such as `/libexec/ld-elf.so.1`.

5 6 This object has two loadable segments: one with execute and read permissions and one with read and write permissions. Both these segments require page alignment.

You should now run **prog3** on other object files.

- Try a relocatable object file created by a **cc -c** invocation. Does it have an program header table?
- Try **prog3** on shared libraries. What do their program header tables look like?
- Can you locate ELF objects on your system that have `PT_TLS` header entries?

Chapter 5

Looking at Sections

In the previous chapter we looked at the way an executable ELF objects are viewed by the operating system. In this section we will look at the features of the ELF format that are used by compilers and linkers.

For linking, data in an ELF object is grouped into *sections*. Each ELF section represents one kind of data. For example, a section could contain a table of strings used for program symbols, another could contain debug information, and another could contain machine code. Non-empty sections do not overlap in the file.

ELF sections are described by entries in an *ELF section header table*. This table is usually placed at the very end of the ELF object (see figure 3.1 on page 16). Table 5.1 describes the elements of section header table entry and figure 5.1 on page 35 shows graphically how the fields of an ELF section header specify the section's placement.

Table 5.1: ELF Section Header Table Entries

	32 bit SHDR Table Entry	**64 bit SHDR Table Entry**
	`typedef struct {`	`typedef struct {`
$\boxed{1}$	`Elf32_Word   sh_name;`	`Elf64_Word   sh_name;`
$\boxed{2}$	`Elf32_Word   sh_type;`	`Elf64_Word   sh_type;`
$\boxed{3}$	`Elf32_Xword  sh_flags;`	`Elf64_Xword  sh_flags;`
	`Elf32_Addr   sh_addr;`	`Elf64_Addr   sh_addr;`
	`Elf32_Off    sh_offset;`	`Elf64_Off    sh_offset;`
$\boxed{4}$	`Elf32_Xword  sh_size;`	`Elf64_Xword  sh_size;`
$\boxed{5}$	`Elf32_Word   sh_link;`	`Elf64_Word   sh_link;`
$\boxed{6}$	`Elf32_Word   sh_info;`	`Elf64_Word   sh_info;`
$\boxed{7}$	`Elf32_Word   sh_addralign;`	`Elf64_Word   sh_addralign;`
$\boxed{8}$	`Elf32_Word   sh_entsize;`	`Elf64_Word   sh_entsize;`
	`} Elf32_Shdr;`	`} Elf64_Shdr;`

$\boxed{1}$ The `sh_name` field is used to encode a section's name. As section names are variable length strings, they are not kept in the section header table entry itself.Instead, all section names are collected into an object-wide string table holding section names and the `sh_name` field of each section header stores an *index* into the string table. The ELF executable header has an `e_shstrndx` member that points to the section index of this string table. ELF string tables, and the way to read them programmatically are described in section 5.1.1 on page 37.

$\boxed{2}$ The `sh_type` field specifies the section type. Section types are defined by the `SHT_*` constants defined in the system's ELF headers. For example, a section of type `SHT_PROGBITS` is defined to contain executable code, while a section type `SHT_SYMTAB` denotes a section containing a symbol table.

The ELF specification reserves values in the range 0x60000000 to 0x6FFF-FFFF to denote OS-specific section types, and values in the range 0x7000-0000 to 0x7FFFFFFF for processor-specific section types. In addition, applications have been given the range 0x80000000 to 0xFFFFFFFF for their own use.

$\boxed{3}$ Section flags indicate whether a section has specific properties, e.g., whether it contains writable data or instructions, or whether it has special link ordering requirements. Flag values from 0x00100000 to 0x08000000 (8 flags) are reserved for OS-specific uses. Flags values from 0x10000000 to 0x80000000 (4 flags) are reserved for processor specific uses.

$\boxed{4}$ The `sh_size` member specifies the size of the section in bytes.

$\boxed{5}$ $\boxed{6}$ The `sh_link` and `sh_info` fields contain additional additional section specific information. These fields are described in the elf(5) manual page.

$\boxed{7}$ For sections that have specific alignment requirements, the `sh_addralign` member holds the required alignment. Its value is a power of two.

$\boxed{8}$ For sections that contain arrays of fixed-size elements, the `sh_entsize` member specifies the size of each element.

There are a couple of other quirks associated with ELF sections. Valid section indices range from `SHN_UNDEF` (0) upto but not including `SHN_LORESERVE` (0xFF00). Section indices between 0xFF00 and 0xFFFF are used to denote special sections (like FORTRAN COMMON blocks). Thus if an ELF file has more than 65279 (0xFEFF) sections, then it needs to use extended section numbering (see section 3.1 on page 19).

The section header table entry at index '0' (`SHN_UNDEF`) is treated specially: it is always of type `SHT_NULL`. It has its members set to zero except when extended numbering is in use, see section 3.1 on page 19.

5.1 ELF section handling with `libelf`

You can conveniently retrieve the contents of sections and section headers using the APIs in the ELF(3) library. Function `elf_getscn` will retrieve section

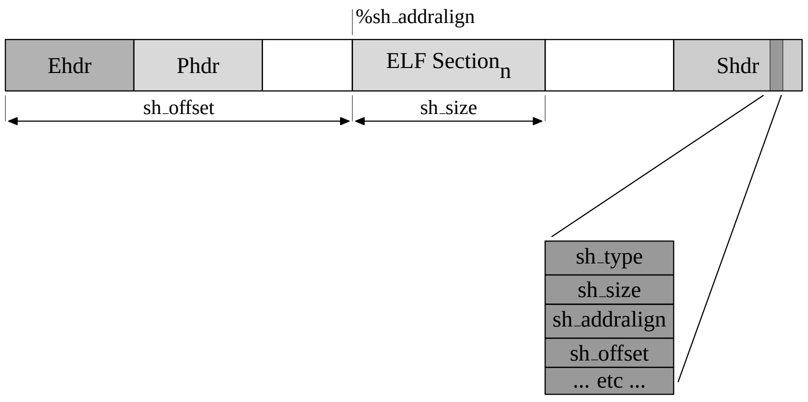


Figure 5.1: Section layout

information for a requested section number.Iteration through the sections of an ELF file is possible using function `elf_nextscn`.These routines will take care of translating between in-file and in-memory representations, thus simplifying your application.

In the ELF(3) API set, ELF sections are managed using `Elf_Scn` descriptors. There is one `Elf_Scn` descriptor per ELF section in the ELF object. Functions `elf_getscn` and `elf_nextscn` retrieve pointers to `Elf_Scn` descriptors for pre-existing sections in the ELF object. (Chapter 6 on page 43 covers the use of function `elf_newscn` for allocating new sections)..

Given an `Elf_Scn` descriptor, functions `elf32_getshdr` and `elf64_getshdr` retrieve its associated section header table entry. The GELF(3) API set offers an equivalent ELF-class independent function `gelf_getshdr`.

Each `Elf_Scn` descriptor can be associated with zero or more `Elf_Data` descriptors. `Elf_Data` descriptors describe regions of application memory that contain the actual data in the ELF section. `Elf_Data` descriptors for a given `Elf_Scn` descriptor are retrieved using the `elf_getdata` function.

Figure 5.2 on the next page shows graphically how an `Elf_Scn` descriptor could conceptually cover the content of a section with `Elf_Data` descriptors.

Figure 5.3 on the following page depicts how an `Elf_Data` structure describes a chunk of application memory. Note that the figure reflects the fact that the in-memory representation of data could have a different size and endianness than its in-file representation.

Figure 5.1 shows the C definition of the `Elf_Scn` and `Elf_Data` descriptors.

Listing 5.1: Definition of Elf_Data and Elf_Scn

```
typedef struct _Elf_Scn Elf_Scn;        1
typedef struct _Elf_Data {
        /*
         * 'Public' members that are part of the ELF(3) API.
         */
        uint64_t        d_align;        2
        void            *d_buf;         3
```

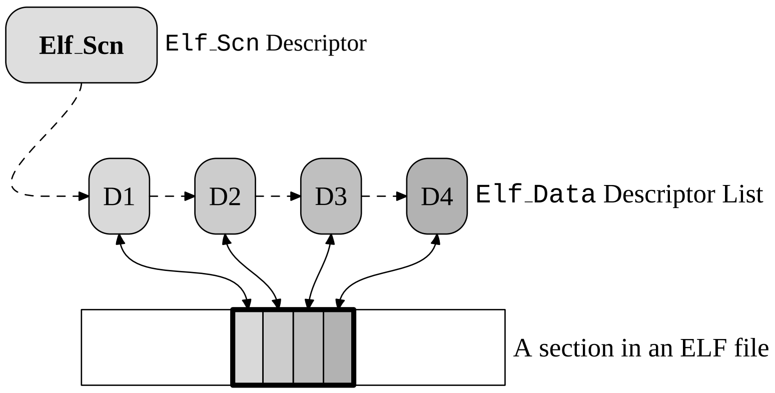


Figure 5.2: Managing data in an Elf Section

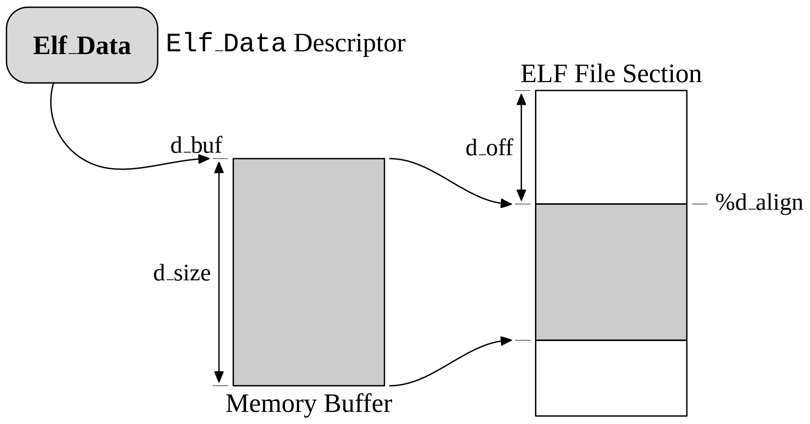


Figure 5.3: `Elf_Data` descriptors

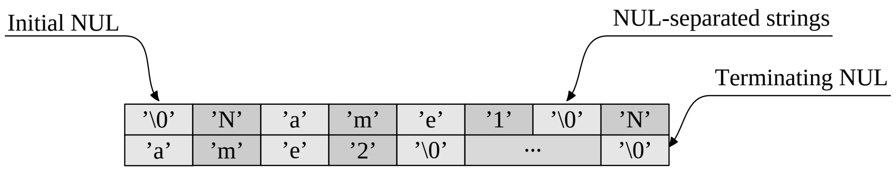

Initial NUL
NUL-separated strings
Terminating NUL
'\0' 'N' 'a' 'm' 'e' '1' '\0' 'N'
'a' 'm' 'e' '2' '\0' ··· '\0'

Figure 5.4: String Table Layout

```
        uint64_t        d_off;         4
        uint64_t        d_size;        5
        Elf_Type        d_type;        6
        unsigned int    d_version;     7
        /* ... other library-private fields ... */
} Elf_Data;
```

1 The `Elf_Scn` type is opaque to the application.

2 The `d_align` member specifies alignment of data referenced in the `Elf_Data` with respect to its containing section.

3 The `d_buf` member points to a contiguous region of memory holding data.

4 The `d_off` member contains the file offset *from the start of the section* of the data in this buffer. This field is usually managed by the library, but is under application control if the application has requested full control of the ELF file's layout (see chapter 6 on page 43).

5 The `d_size` member contains the size of the memory buffer.

6 The `d_type` member specifies the ELF type of the data contained in the data buffer. Legal values for this member are precisely those defined by the `Elf_Type` enumeration in `libelf.h`.

7 The `d_version` member specifies the working version for the data in this descriptor. It must be one of the values supported by the `libelf` library.

Before we look at an example program we need to understand how string tables are implemented by `libelf`.

5.1.1 String Tables

String tables hold variable length strings, allowing other structures in an ELF object to refer to strings using offsets into the string table. Sections containing string tables have type `SHT_STRTAB`.

Figure 5.4 shows the layout of a string table graphically:

- The initial byte of a string table is `NUL` (a '\0'). This allows an string offset value of zero to denote the NULL string.
- Subsequent strings are separated by `NUL` bytes.
- The final byte in the section is again a `NUL` so as to terminate the last string in the string table.

An ELF file can have multiple string tables; for example, section names could be kept in one string table and symbol names in another.

Given the section index of a section containing a string table, applications would use the `elf_strptr` function to convert a string offset to `char *` pointer usable by C code.

5.2 Example: Listing section names

Let us now write a program that would retrieve and print the names of the sections present in an ELF object. This example will show you how to use:

- Functions `elf_nextscn` and `elf_getscn` to retrieve `Elf_Scn` descriptors.
- Function `gelf_getshdr` to retrieve a section header table entry corresponding to a section descriptor.
- Function `elf_strptr` to convert section name indices to NUL-terminated strings.
- Function `elf_getdata` to retrieve translated data associated with a section.

Listing 5.2: Program 4

```
/*
 * Print the names of ELF sections.
 */

#include <err.h>
#include <fcntl.h>
#include <gelf.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>
#include <vis.h>

int
main(int argc, char **argv)
{
    int fd;
    Elf *e;
    char *name, *p, pc[4*sizeof(char)];
    Elf_Scn *scn;
    Elf_Data *data;
```

```
GElf_Shdr shdr;
size_t n, shstrndx, sz;

if (argc != 2)
    errx(EX_USAGE, "usage:␣%s␣file-name", argv[0]);

if (elf_version(EV_CURRENT) == EV_NONE)
    errx(EX_SOFTWARE, "ELF␣library␣initialization␣"
        "failed:␣%s", elf_errmsg(-1));

if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
    err(EX_NOINPUT, "open␣\%s\"␣failed", argv[1]);

if ((e = elf_begin(fd, ELF_C_READ, NULL)) == NULL)
    errx(EX_SOFTWARE, "elf_begin()␣failed:␣%s.",
        elf_errmsg(-1));

if (elf_kind(e) != ELF_K_ELF)
    errx(EX_DATAERR, "%s␣is␣not␣an␣ELF␣object.",
        argv[1]);

if (elf_getshdrstrndx(e, &shstrndx) != 0) [1]
    errx(EX_SOFTWARE, "elf_getshdrstrndx()␣failed:␣%s.",
        elf_errmsg(-1));

scn = NULL; [2]
while ((scn = elf_nextscn(e, scn)) != NULL) { [3]
    if (gelf_getshdr(scn, &shdr) != &shdr) [4]
        errx(EX_SOFTWARE, "getshdr()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((name = elf_strptr(e, shstrndx, shdr.sh_name))
        == NULL) [5]
        errx(EX_SOFTWARE, "elf_strptr()␣failed:␣%s.",
            elf_errmsg(-1));

    (void) printf("Section␣%-4.4jd␣%s\n", (uintmax_t)
        elf_ndxscn(scn), name);
}

if ((scn = elf_getscn(e, shstrndx)) == NULL) [6]
    errx(EX_SOFTWARE, "getscn()␣failed:␣%s.",
        elf_errmsg(-1));

if (gelf_getshdr(scn, &shdr) != &shdr)
    errx(EX_SOFTWARE, "getshdr(shstrndx)␣failed:␣%s.",
        elf_errmsg(-1));

(void) printf(".shstrab:␣size=%jd\n", (uintmax_t)
    shdr.sh_size);
    data = NULL; n = 0;
    while (n < shdr.sh_size &&

            (data = elf_getdata(scn, data)) != NULL) { 7
        p = (char *) data->d_buf;
        while (p < (char *) data->d_buf + data->d_size) {
            if (vis(pc, *p, VIS_WHITE, 0))
                printf("%s", pc);
            n++; p++;
            (void) putchar((n % 16) ? '␣' : '\n');
        }
    }
    (void) putchar('\n');

    (void) elf_end(e);
    (void) close(fd);
    exit(EX_OK);
}
```

1 We retrieve the section index of the ELF section containing the string table of section names using function **elf_getshdrstrndx**. The use of **elf_getshdrstrndx** allows our program to work correctly when the object being examined has a very large number of sections.

2 Function **elf_nextscn** has the useful property that it returns the pointer to section number '1' if a NULL section pointer is passed in. Recall that section number '0' is always of type SHT_NULL and is not interesting to applications.

3 We loop over all sections in the ELF object. Function **elf_nextscn** will return NULL at the end, which is a convenient way to exit the processing loop.

4 Given a Elf_Scn pointer, we retrieve the associated section header using function **gelf_getshdr**. The **sh_name** member of this structure holds the required offset into the section name string table.indexsections!header table entry!retrieval of

5 We convert the string offset in member **sh_name** to a **char *** pointer using function **elf_strptr**. This value is then printed using **printf**.

6 We retrieve the section descriptor associate with the string table holding section names. Variable **shstrndx** was retrieved by a prior call to function **elf_getshdrstrndx**.

7 We cycle through the **Elf_Data** descriptors associated with the section in question, printing the characters in each data buffer.

Save the program in listing 5.2 on page 38 to file prog4.c and then compile and run it as shown in listing 5.3 on the next page.

Listing 5.3: Compiling and Running prog4

```
% cc -o prog4 prog4.c -lelf 1
% ./prog4 prog4 2
Section 0001 .interp
Section 0002 .note.ABI-tag
Section 0003 .hash
Section 0004 .dynsym
Section 0005 .dynstr
Section 0006 .rela.plt
Section 0007 .init
Section 0008 .plt
Section 0009 .text
Section 0010 .fini
Section 0011 .rodata
Section 0012 .data
Section 0013 .eh_frame
Section 0014 .dynamic
Section 0015 .ctors
Section 0016 .dtors
Section 0017 .jcr
Section 0018 .got
Section 0019 .bss
Section 0020 .comment
Section 0021 .shstrtab 3
Section 0022 .symtab
Section 0023 .strtab
.shstrab: size=287 4
\^@ . s y m t a b \^@ . s t r t a b
\^@ . s h s t r t a b \^@ . i n t e
r p \^@ . h a s h \^@ . d y n s y m
...etc...
```

$\boxed{1}$ Compile and link the program in the standard way.

$\boxed{2}$ We make our program print the names of its own sections.

$\boxed{3}$ One of the sections contains the string table used for sections names themselves. This section is called `.shstrtab` by convention.

$\boxed{4}$ This is the content of the string table holding section names.

Chapter 6

Creating new ELF objects

We will now look at how ELF objects can be created (and modified, see section 6.2.4 on page 49) using the `libelf` library.

Broadly speaking, the steps involved in creating an ELF file with `libelf` are:

1. An ELF descriptor needs to be allocated with a call to `elf_begin`, passing in the parameter `ELF_C_WRITE`.

2. You would then allocate an ELF executable header using one of the `elf32_newehdr`, `elf64_newehdr` or `gelf_newehdr` functions. Note that this is a mandatory step since an ELF executable header is always present in an ELF object. The ELF "class", of the object, i.e., whether the object is a 32-bit or 64-bit one, is fixed at this time.

3. An ELF program header table is optional and can be allocated using one of functions `elf32_newphdr`, `elf64_newphdr` or `gelf_newphdr`. The program header table can be allocated anytime after the executable header has been allocated.

4. Sections may be added to an ELF object using function `elf_newscn`. `Elf_Data` descriptors associated with an ELF section can be added to a section descriptor using function `elf_newdata`. ELF sections can be allocated anytime after the object's executable header has been allocated.

5. If you are creating an ELF object for a non-native architecture, you can change the byte ordering of the object by changing the byte order byte at offset `EI_DATA` in the ELF header.

6. Once your data is in place, you then ask the `libelf` library to write out the final ELF object using function `elf_update`.

7. Finally, you close the ELF descriptor allocated using function `elf_end`.

6.1 Example: Creating an ELF object

In listing 6.1 on the next page we will look at a program that creates a simple ELF object with a program header table, one ELF section containing translatable data and one ELF section containing a section name string table. We will mark the ELF of the object as using a 32-bit, MSB-first data ordering.

Listing 6.1: Program 5

```
/*
 * Create an ELF object.
 */

#include <err.h>
#include <fcntl.h>
#include <libelf.h>  1
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>


uint32_t hash_words[] = {  2
    0x01234567,
    0x89abcdef,
    0xdeadc0de
};


char string_table[] = {  3
    /* Offset 0 */  '\0',
    /* Offset 1 */  '.', 'f', 'o', 'o', '\0',
    /* Offset 6 */  '.', 's', 'h', 's', 't',
                    'r', 't', 'a', 'b', '\0'
};

int
main(int argc, char **argv)
{
    int fd;
    Elf *e;
    Elf_Scn *scn;
    Elf_Data *data;
    Elf32_Ehdr *ehdr;
    Elf32_Phdr *phdr;
    Elf32_Shdr *shdr;

    if (argc != 2)
        errx(EX_USAGE, "usage:␣%s␣file-name", argv[0]);

    if (elf_version(EV_CURRENT) == EV_NONE)
        errx(EX_SOFTWARE, "ELF␣library␣initialization␣"
            "failed:␣%s", elf_errmsg(-1));


    if ((fd = open(argv[1], O_WRONLY|O_CREAT, 0777)) < 0)  4
        err(EX_OSERR, "open␣\%s\"␣failed", argv[1]);


    if ((e = elf_begin(fd, ELF_C_WRITE, NULL)) == NULL)  5
```

```
        errx(EX_SOFTWARE, "elf_begin()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((ehdr = elf32_newehdr(e)) == NULL) 6
        errx(EX_SOFTWARE, "elf32_newehdr()␣failed:␣%s.",
            elf_errmsg(-1));

    ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
    ehdr->e_machine = EM_PPC; /* 32-bit PowerPC object */
    ehdr->e_type = ET_EXEC;

    if ((phdr = elf32_newphdr(e, 1)) == NULL) 7
        errx(EX_SOFTWARE, "elf32_newphdr()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((scn = elf_newscn(e)) == NULL) 8
        errx(EX_SOFTWARE, "elf_newscn()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((data = elf_newdata(scn)) == NULL)
        errx(EX_SOFTWARE, "elf_newdata()␣failed:␣%s.",
            elf_errmsg(-1));

    data->d_align = 4;
    data->d_off   = 0LL;
    data->d_buf   = hash_words;
    data->d_type = ELF_T_WORD;
    data->d_size = sizeof(hash_words);
    data->d_version = EV_CURRENT;

    if ((shdr = elf32_getshdr(scn)) == NULL)
        errx(EX_SOFTWARE, "elf32_getshdr()␣failed:␣%s.",
            elf_errmsg(-1));

    shdr->sh_name = 1;
    shdr->sh_type = SHT_HASH;
    shdr->sh_flags = SHF_ALLOC;
    shdr->sh_entsize = 0;

    if ((scn = elf_newscn(e)) == NULL) 9
        errx(EX_SOFTWARE, "elf_newscn()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((data = elf_newdata(scn)) == NULL)
        errx(EX_SOFTWARE, "elf_newdata()␣failed:␣%s.",
            elf_errmsg(-1));

    data->d_align = 1;
    data->d_buf = string_table;
    data->d_off = 0LL;
    data->d_size = sizeof(string_table);
    data->d_type = ELF_T_BYTE;
    data->d_version = EV_CURRENT;

    if ((shdr = elf32_getshdr(scn)) == NULL)
        errx(EX_SOFTWARE, "elf32_getshdr()␣failed:␣%s.",
            elf_errmsg(-1));

    shdr->sh_name = 6;
    shdr->sh_type = SHT_STRTAB;
    shdr->sh_flags = SHF_STRINGS | SHF_ALLOC;
    shdr->sh_entsize = 0;

    elf_setshstrndx(e, elf_ndxscn(scn)); 10

    if (elf_update(e, ELF_C_NULL) < 0) 11
        errx(EX_SOFTWARE, "elf_update(NULL)␣failed:␣%s.",
            elf_errmsg(-1));

    phdr->p_type = PT_PHDR;
    phdr->p_offset = ehdr->e_phoff;
    phdr->p_filesz = elf32_fsize(ELF_T_PHDR, 1, EV_CURRENT);

    (void) elf_flagphdr(e, ELF_C_SET, ELF_F_DIRTY);

    if (elf_update(e, ELF_C_WRITE) < 0) 12
        errx(EX_SOFTWARE, "elf_update()␣failed:␣%s.",
            elf_errmsg(-1));

    (void) elf_end(e);
    (void) close(fd);

    exit(EX_OK);
}
```

1 We include `libelf.h` to bring in prototypes for `libelf`'s functions.

2 We will create an ELF section containing 'hash' values. These values are present in host-native order in the array `hash_words`. These values will be translated to the appropriate byte order by the `libelf` library when the object file is created.

3 We use a pre-fabricated ELF string table to hold section names. See section 5.1.1 on page 37 for more information on the layout of ELF string tables.

4 The first step to create an ELF object is to obtain a file descriptor from the OS that is opened for writing.

5 By passing parameter `ELF_C_WRITE` to function `elf_begin`, we obtain an ELF descriptor suitable for creating new ELF objects.

$\boxed{6}$ We allocate an ELF executable header and set the `EI_DATA` byte in its `e_ident` member. The machine type is set to `EM_PPC` denoting the PowerPC architecture, and the object is marked as an ELF executable.

$\boxed{7}$ We allocate an ELF program header table with one entry. At this point of time we do not know how the ELF object will be laid out so we don't know where the ELF program header table will reside. We will update this entry later.

$\boxed{8}$ We create a section descriptor for the section containing the 'hash' values, and associate the data in the `hash_words` array with this descriptor. The type of the section is set to `SHT_HASH`. The library will compute its size and location in the final object and will byte-swap the values when creating the ELF object.

$\boxed{9}$ We allocate another section for holding the string table. We use the prefabricated string table in variable `string_table`. The type of the section is set to `SHT_STRTAB`. Its offset and size in the file will be computed by the library.indexsections!string table!allocation of

$\boxed{10}$ We set the string table index field in the ELF executable header using the function `elf_setshstrndx`.

$\boxed{11}$ Calling function `elf_update` with parameter `ELF_C_NULL` indicates that the `libelf` library is to compute the layout of the object, updating all internal data structures, but *not* write it out. We can thus fill in the values in the ELF program header table entry that we had allocated using the new values in the executable header after this call to `elf_update`. The program header table is then marked "dirty" using a call to function `elf_flagdata`, so that a subsequent call to `elf_update` will use the new contents.

$\boxed{12}$ A call to function `elf_update` with parameter `ELF_C_WRITE` causes the object file to be written out.

Save the program in listing 6.1 on page 44 to file `prog5.c` and then compile and run it as shown in listing 6.2.

Listing 6.2: Compiling and Running prog5

```
% cc -o prog5 prog5.c -lelf [1]
% ./prog5 foo

% file foo [2]
foo: ELF 32-bit MSB executable, PowerPC or cisco 4500, \
     version 1 (SYSV), statically linked, stripped

% readelf -a foo [3]
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           PowerPC
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          112 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         3
  Section header string table index: 2
...etc...
```

$\boxed{1}$ Compile, link and run the program in the standard way.

$\boxed{2}$ $\boxed{3}$ We use the **file** and **readelf** programs to examine the object that we have created.

6.2 The finer points in creating ELF objects

Some of the finer points in creating ELF objects using the `libelf` library are examined below. We cover memory management rules, ELF data structure lifetimes, and how an application can take full control over an object's layout. We also briefly cover how to modify an existing ELF object.

6.2.1 Controlling ELF Layout

By default, the `libelf` library will lay out your ELF objects for you. The default layout is shown in figure 3.1 on page 16.An application may request fine-grained control over the ELF object's layout by setting the flag `ELF_F_LAYOUT` on the ELF descriptor using function `elf_flagelf`.

Once an ELF descriptor has been flagged with flag `ELF_F_LAYOUT` the following members of the ELF data structures come under application control:

- The `e_phoff` and `e_shoff` fields, which determine whether the ELF program header table and section header table start.

- For each section, the `sh_addralign`, `sh_offset`, and `sh_size` fields in its section header.

These fields must set prior to calling function `elf_update`.

The library will fill "gaps" between parts of the ELF file with a *fill character*. An application may set the fill character using the function `elf_fill`. The default fill character is a zero byte.

6.2.2 Memory Management

Applications pass pointers to allocated memory to the `libelf` library by setting the `d_buf` members of `Elf_Data` structures passed to the library. The `libelf` library also passes data back to the application using the same mechanism. In order to keep tracking memory ownership simple, the `libelf` library follows the rule that it will never attempt to free data that it did not allocate. Conversely, the application is also not to free memory allocated by the `libelf` library.

6.2.3 `libelf` data structure lifetimes

As part of the process of writing out an ELF object, the `libelf` library may release or reallocate its internal bookkeeping structures.

A rule to be followed when using the `libelf` library is that all pointers to returned data structures (e.g., pointers to `Elf_Scn` and `Elf_Data` structures or to other ELF headers *become invalid* after a call to function `elf_update` with parameter `ELF_C_WRITE`.

After a successful call to function `elf_update` all ELF data structures will need to be retrieved afresh.

6.2.4 Modifying existing ELF objects

The `libelf` library also allows existing ELF objects to be modified. The process is similar to that for creating ELF objects, the differences being:

- The underlying file object would need to be opened for reading and writing, and the call to function `elf_begin` would use parameter `ELF_C_RDWR` instead of `ELF_C_WRITE`.
- The application would use the `elf_get*` APIs to retrieve existing ELF data structures in addition to the `elf_new*` APIs used for allocating new data structures. The `libelf` library would be informed of modifications to ELF data structures by calls to the appropriate `elf_flag*` functions.

The rest of the program flow would be similar to the object creation case.

Chapter 7

Processing ar(1) archives

The `libelf` library also offers support for reading archives members in an ar(1) archive. This support is “read-only”; you cannot create new ar(1) archives or update members in an archive using these functions. The `libelf` library supports both random and sequential access to the members of an ar(1) archive.

7.1 Archive structure

Each ar(1) archive starts with a sequence of 8 signature bytes (see the constant `ARMAG` defined in the system header `ar.h`). The members of the archive follow, each member preceded by an *archive header* describing the metadata associated with the member. Figure 7.1 on the following page depicts the structure of an ar(1) archive pictorially.

Each archive header is a collection of fixed size ASCII strings. Archive headers are required to reside at even offsets in the archive file. Figure 7.1 shows the layout of the archive header as a C structure.

Listing 7.1: Archive Header Layout

```
struct ar_hdr {
    char ar_name[16]; /* file name */
    char ar_date[12]; /* file modification time */
    char ar_uid[6];   /* creator user id */
    char ar_gid[6];   /* creator group id */
    char ar_mode[8];  /* octal file permissions */
    char ar_size[10]; /* size in bytes */
#define        ARFMAG   "`\n"
    char ar_fmag[2];  /* consistency check */
} __packed;
```

The initial members of an ar(1); archive may be special:

- An archive member with name “/” is an *archive symbol table*. An archive symbol table maps program symbols to archive members in an archive. It is usually maintained by tools like **ranlib** and **ar**.
- An archive member with name “//” is an *archive string table*. The members of an ar(1) header only contain fixed size ASCII strings with space

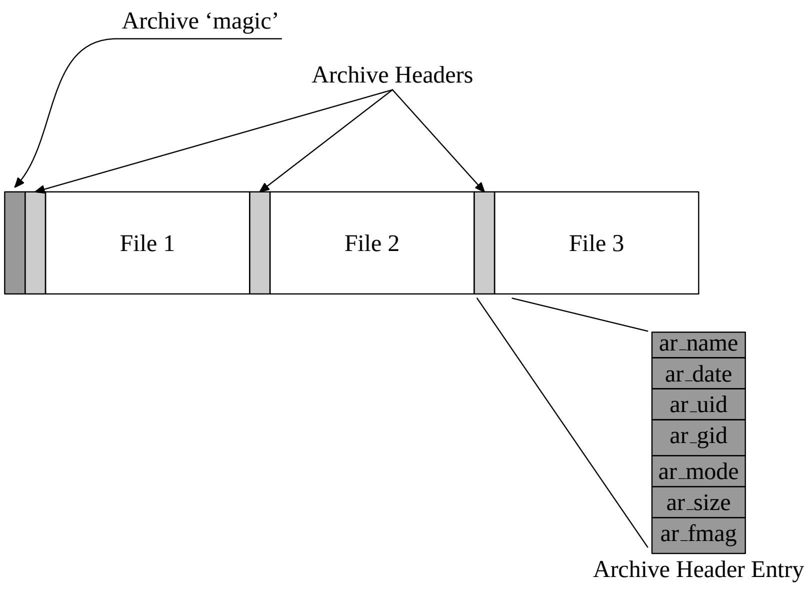


Figure 7.1: The structure of ar(1) archives

and '/' characters being used for string termination. File names that exceed the length limits of the `ar_name` member are handled by placing them in a special string table (not to be confused with ELF string tables) and storing the offset of the file name in the `ar_name` member as a string of decimal digits.

The archive handling functions offered by the `libelf` library insulate the application from these details of the layout of ar(1) archives.

7.2 Example: Stepping through an ar(1) archive

We now illustrate (listing 7.2) how an application may iterate through the members of an ar(1) archive. The steps involved are:

1. Archives are opened using `elf_begin` in the usual way.
2. Each archive managed by the `libelf` library tracks the next member to opened. This information is updated using the functions `elf_next` and `elf_rand`.
3. Nested calls to function `elf_begin` retrieve ELF descriptors for the members in the archive.

Figure 7.2 on the next page pictorially depicts how functions `elf_begin` and `elf_next` are used to step through an ar(1) archive.

We now look at an example program that illustrates these concepts.

Listing 7.2: Program 6

```
/*
```

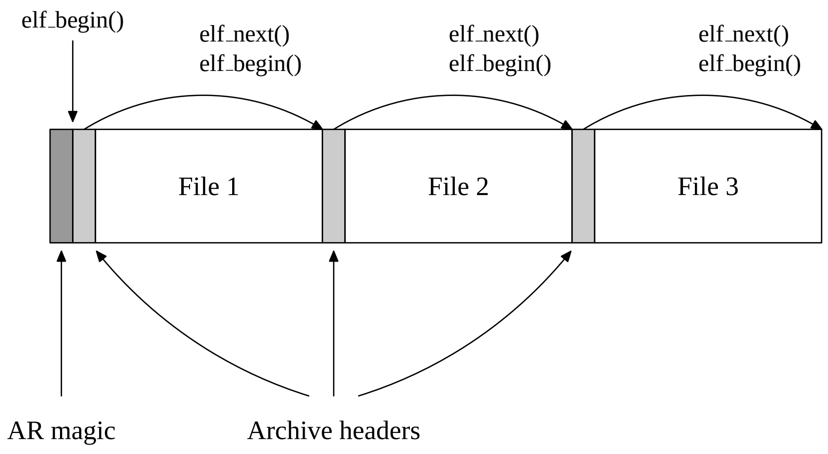


Figure 7.2: Iterating through ar(1) archives with `elf_begin` and `elf_next`

```
 * Iterate through an ar(1) archive.
 */

#include <err.h>
#include <fcntl.h>
#include <libelf.h>
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>

int
main(int argc, char **argv)
{
    int fd;
    Elf *ar, *e;
    Elf_Arhdr *arh;

    if (argc != 2)
        errx(EX_USAGE, "usage:␣%s␣file-name", argv[0]);

    if (elf_version(EV_CURRENT) == EV_NONE)
        errx(EX_SOFTWARE, "ELF␣library␣initialization␣"
            "failed:␣%s", elf_errmsg(-1));

    if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
        err(EX_NOINPUT, "open␣\%s\"␣failed", argv[1]);

    if ((fd = open(argv[1], O_RDONLY, 0)) < 0) 1
        err(EX_NOINPUT, "open␣\%s\"␣failed", argv[1]);

    if ((ar = elf_begin(fd, ELF_C_READ, NULL)) == NULL) 2
        errx(EX_SOFTWARE, "elf_begin()␣failed:␣%s.",
            elf_errmsg(-1));

    if (elf_kind(ar) != ELF_K_AR)
        errx(EX_DATAERR, "%s␣is␣not␣an␣ar(1)␣archive.",
            argv[1]);

    while ((e = elf_begin(fd, ELF_C_READ, ar)) != NULL) { 3
        if ((arh = elf_getarhdr(e)) == NULL) 4
            errx(EX_SOFTWARE, "elf_getarhdr()␣failed:␣%s.",
                elf_errmsg(-1));

        (void) printf("%20s␣%d\n", arh->ar_name,
            arh->ar_size);

        (void) elf_next(e); 5
        (void) elf_end(e); 6
    }

    (void) elf_end(ar);
    (void) close(fd);
    exit(0);
}
```

1 2 We open the ar(1) archive for reading and obtain a descriptor in the usual manner.

3 Function **elf_begin** is used to the iterate through the members of the archive. The third parameter in the call to **elf_begin** is a pointer to the descriptor for the archive itself. The return value of function **elf_begin** is a descriptor that references an archive member.

4 We retrieve the translated ar(1) header using function **elf_getarhdr**. We then print out the name and size of the member. Note that function **elf_getarhdr** translates names to null-terminated C strings suitable for use with `printf`.

Figure 7.3 shows the translated information returned by **elf_getarhdr**.

Listing 7.3: The `Elf_Arhdr` Structure

```
typedef struct {
        time_t      ar_date;     /* time of creation */
        char        *ar_name;    /* archive member name */
        gid_t       ar_gid;      /* creator's group */
        mode_t      ar_mode;     /* file creation mode */
        char        *ar_rawname; /* 'raw' member name */
        size_t      ar_size;     /* member size in bytes */
        uid_t       ar_uid;      /* creator's user id */
} Elf_Arhdr;
```

$\boxed{5}$ The `elf_next` function sets up the *parent* archive descriptor (referenced by variable `ar` in this example) to return the next archive member on the next call to function `elf_begin`.

$\boxed{6}$ It is good programming practice to call `elf_end` on descriptors that are no longer needed.

Save the program in listing 7.2 on page 52 to file `prog6.c` and then compile and run it as shown in listing 7.4.

Listing 7.4: Compiling and Running prog6

```
% cc -o prog6 prog6.c -lelf [1]
% ./prog6 /usr/lib/librt.a [2]
             timer.o 7552
                mq.o 8980
               aio.o 8212
      sigev_thread.o 15528
```

$\boxed{1}$ Compile and link the program in the usual fashion.

$\boxed{2}$ We run the program against a small library and get a list of its members.

7.2.1 Random access in an ar(1) archive

Random access in the archive is supported by the function `elf_rand`. However, in order to use this function you need to know the file offsets in the archive for the desired archive member. For archives containing object files this information is present in the archive symbol table.

If an archive has an archive symbol table, it can be retrieved using the function `elf_getarsym`. Function `elf_getarsym` returns an array of `Elf_Arsym` structures. Each `Elf_Arsym` structure (figure 7.5) maps one program symbol to the file offset inside the ar(1) archive of the member that contains its definition.

Listing 7.5: The `Elf_Arsym` structure

```
typedef struct {
  off_t         as_off;   /* byte offset to member header */
  unsigned long as_hash;  /* elf_hash() value for name */
  char          *as_name; /* null terminated symbol name */
} Elf_Arsym;
```

Once the file offset of the member is known, the function `elf_rand` can be used to set the parent archive to open the desired archive member at the next call to `elf_begin`.

Chapter 8

Conclusion

This tutorial covered the following topics:

- We gained an overview of the facilities for manipulating ELF objects offered by the ELF(3) and GELF(3) API sets.
- We studied the basics of the ELF format, including the key data structures involved and their layout inside ELF objects.
- We looked at example programs that retrieve ELF data structures from existing ELF objects.
- We looked at how to create new ELF objects using the ELF(3) library.
- We looked at accessing information in the ar(1) archives.

8.1 Further Reading

There are very few books today on the topic of linking and loading. John Levine's "Linkers and Loaders" is a readable book that offers a overview of the concepts involved in the process of linking and loading object files.

On the Web, Peter Seebach's DeveloperWorks article "An unsung hero: The hardworking ELF" covers the history and features of the ELF format. Other tutorials include Hongjiu Liu's "ELF: From The Programmer's Perspective", which covers GCC and GNU ld, and Michael L. Haung's "The Executable and Linking Format (ELF)".

Neelakanth Nadgir's tutorial on ELF(3) and GELF(3) is a readable and brief introduction to the ELF3 and GELF3 APIs for Solaris™.

The Linkers and Libraries Guide from Sun Microsystems® describes linking and loading tools in Solaris™. Chapter 7 of this book, "Object File Format" contains a readable introduction to the ELF format.

The current specification of the ELF format, the "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2" is freely available to download.

Index