libelf by Example

Joseph Koshy

Contents

Preface

This tutorial introduces the `libelf` library being developed at the ElfToolChain Project on SourceForge.Net. It uses simple example programs to show how to create ELF processing tools using the `libelf` library. Along the way the tutorial covers the ELF file format to the extent needed to understand the example programs being discussed.

This tutorial would be of interest to software developers who wish to write software that processes ELF files.

The tutorial's example programs are written using the C programming language. They should compile out of the box on FreeBSD,™ NetBSD® and other BSD-family operating systems. They should also compile on Debian GNU/Linux® with the `libbsd-dev` package installed.

Legal Notice

Copyright © 2006–2020 Joseph Koshy. All rights reserved.

Redistribution and use in source (LaTeX format) and "compiled" forms (EPUB, HTML, PDF, Postscript, RTF, etc), with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code (LaTeX format) must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in compiled form (transformed to other documentation formats, converted to EPUB, HTML, PDF, Postscript, RTF, etc) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The names of the author and contributors may not be used to endorse or promote products derived from this documentation without specific prior written permission.

THIS DOCUMENTATION IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and the author and contributors were aware of the trademark claim, the designations have been followed by the "™" or the "®" symbol.

Document Identifier

You are reading the following version of this tutorial:

```
elftoolchain HEAD svn:3854
```

Acknowledgements

The following people (names in alphabetical order) offered constructive criticism of this tutorial: Cherry George Mathew, Douglas Fraser, Hyogeol Lee, Kai Wang, Prashanth Chandra, Ricardo Nabinger Sanchez, Sam Arun Raj, Wei-Jen Chen and Y. Giridhar Appaji Nag. Thank you, all.

Chapter 1

Introduction

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

The ELF specification was released to the public in 1990 by a group of vendors as an "open standard". ELF has been widely adopted by industry and by the open-source community on account of its availability and modern features. The ELF standard supports big and little-endian machine architectures with 32-bit and 64-bit word widths. It supports cross-compilation, the use of dynamic shared libraries, and the special compilation needs of the C++ language.

Among the open-source operating systems, the RedHat™ RHL 2.0 Beta release (late summer 1995) and the Slackware v3.0 (November 1995) release were among the first Linux®-based operating systems to use ELF. The first ELF based release for NetBSD® was for the DEC Alpha™ architecture, in release 1.3 (January 1998). FreeBSD™ switched to using ELF as its object format in FreeBSD 3.0 (October 1998).

The `libelf` library implements a large set of APIs, known as the ELF(3) & GELF(3) APIs. These APIs allow you to write software that can run on one kind of machine architecture, while manipulating ELF objects meant for another.

There are multiple implementations of the ELF(3)/GELF(3) API set in the open-source world. This tutorial is based on the `libelf` library being developed as part of the ElfToolChain Project on SourceForge.Net.

The ELF(3)/GELF(3) API set contains over 80 callable functions, and can be daunting to learn. This tutorial offers a gentle introduction to this API set.

1.1 What to Expect From This Tutorial

This tutorial looks at a series of simple but complete programs, with each program illustrating a different aspect of the ELF format and the `libelf` library's APIs. Along the way, it introduces the concepts and data structures needed to understand these example programs.

Chapter 2 covers getting started with the `libelf` library. This chapter looks at how to compile and link an application that uses the library, how to establish a working ELF version number, how to obtain a handle to a ELF object, and how to handle errors reported by the `libelf` library.

Chapter 3 looks at how ELF executables, relocatable objects and shared objects are laid out. This chapter describes how the ELF data structure known as the Executable Header specifies the layout of the rest of the file. The chapter introduces the notions of the "file representation" and "memory representation" of ELF data types, and explains how the `libelf` library enables programs to work on ELF objects whose word size and endianness differ from their "native" size and endianness. The example program in this chapter displays the contents of the ELF Executable Header for an ELF object.

Chapter 4 studies ELF segments, and how these segments are described by entries in the ELF Program Header Table. The example program in this chapter reads and displays the program header table entries in an ELF executable.

Chapter 5 looks at how data is stored in ELF sections. It looks at how ELF sections are described by the ELF Section Header Table, and how ELF string tables (a special kind of ELF section) are structured. The example program in this chapter traverses an ELF object, printing out the names of its sections.

Chapter 6 demonstrates how to create new ELF objects using the `libelf` library. This chapter covers the correct ordering of calls to `libelf`'s functions when creating ELF objects, the default layout used by the library, and the facilities offered by `libelf` for creating ELF objects with custom layouts. The example program in this chapter creates a new ELF object from scratch.

The tutorial then moves on to **ar** archives. Chapter 7 looks at the structure of **ar** archives and covers `libelf`'s facilities for reading these archives. The example program in this chapter lists the names of the files present in an **ar** archive.

Finally, chapter 8 concludes the tutorial with suggestions for further reading.

Figure 1.1 on the facing page offers a graphical overview of the concepts covered by this tutorial.

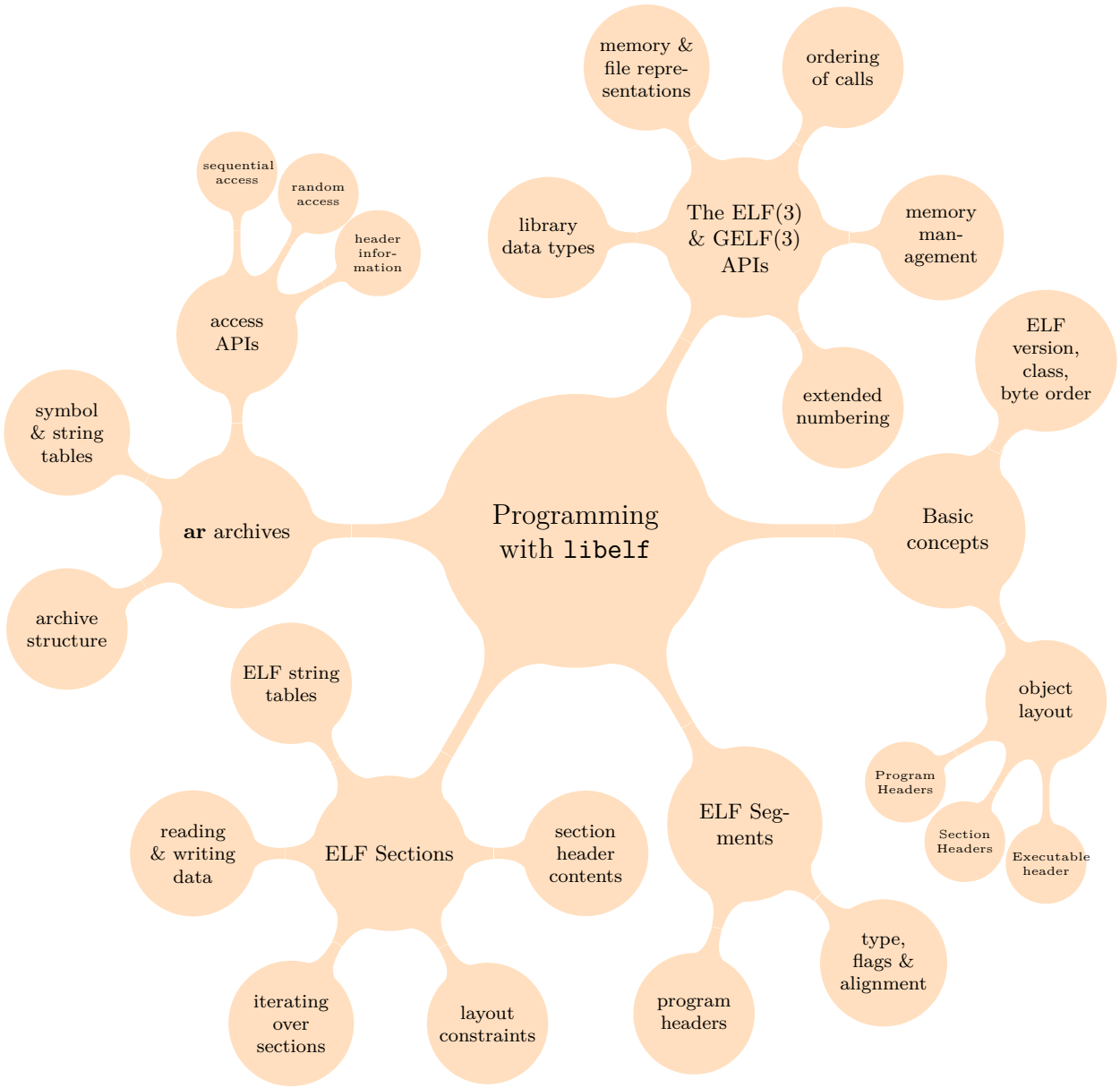

Figure 1.1: An overview of the concepts covered in this tutorial.

Chapter 2

Getting Started

It is time to get a taste of programming with `libelf`.

Our first example program (Program 1, listing 2.1) opens the file named on its command line, and displays the file's type as recognized by the ELF library. This example covers the basics of using `libelf`: how to compile a program that uses `libelf`, how to initialize the library, how to handle any errors reported by the library, and how to release resources cleanly when done.

Listing 2.1: Program 1

```
/*
 * Getting started with libelf.
 *
 * $Id: prog1.txt 2133 2011-11-10 08:28:22Z jkoshy $
 */

#include <err.h>
#include <fcntl.h>
#include <libelf.h> 1
#include <stdio.h>
#include <stdlib.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(EXIT_FAILURE, "usage: %s file-name", argv[0]);

    if (elf_version(EV_CURRENT) == EV_NONE) 4
        errx(EXIT_FAILURE, "ELF library initialization "
            "failed: %s", elf_errmsg(-1));
```

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

    if ((e = elf_begin(fd, ELF_C_READ, NULL)) == NULL) [5]
        errx(EXIT_FAILURE, "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(EXIT_SUCCESS);
}
```

[1] The functions and dataypes that make up the ELF(3) API are declared in the header `libelf.h`. This file must be included by every source file that uses the `libelf` library.

[2] The library uses an opaque C type named `Elf` as a handle to an ELF object.

[4] Before the functions in the library can be invoked, every application that uses `libelf` must indicate the version of the ELF specification that it expects to use. This is done by calling the `elf_version` function.

Calling `elf_version` is mandatory—most of `libelf`'s APIs will return an error if invoked before the expected ELF version is set.

Multiple ELF specification versions could come into play when an application reads or writes an ELF object. In figure 2.1 on the facing page the application program using `libelf` expects to work with files conforming to version v_1 of the ELF specification. The ELF object file however conforms to ELF specification version v_2. The `libelf` library understands

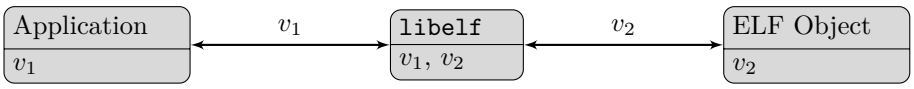

Figure 2.1: Handling ELF versioning.

the semantics of both specification versions v_1 and v_2, and so would (in theory) be able to mediate between the application and the ELF object.

In practice, the ELF specification's version has not changed since its inception; the current version (denoted by the symbol `EV_CURRENT`) is 1.

5 The `elf_begin` function takes an open file descriptor and converts it an `Elf` handle.

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 file opening mode for the file descriptor `fd` should be compatible with this parameter: the file descriptor should be opened for reading for use with `ELF_C_READ`, for writing for use with `ELF_C_WRITE`, and for updating for use with `ELF_C_RDWR`.

The third parameter to `elf_begin` is only used when reading **ar** archives. Chapter 7 on page 51 covers **ar** archive processing.

6 When the ELF library encounters an error, it will record an error number in an internal location and return a sentinel value (e.g., the `NULL` value from functions that return pointers). The saved error number indicates the specific class of error that was encountered. This number can be retrieved using the `elf_errno` function.

Numeric error numbers are however not user-friendly. The `elf_errmsg` function returns a human-readable string describing the error number passed to it. As a programming convenience, an error number of -1 denotes the most recent error number that had been recorded by the library.

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`. Currently, the library only recognizes ELF files and **ar** archives.

8 The `elf_end` function releases the resources held by an `Elf` handle.

It is now time to compile and run our first example program.

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

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
```

[1] The `-lelf` option to the **cc** comand informs it to link **prog1** with the `libelf` library.

[2] We invoke **prog1** on itself. If all went well it should recognize its own executable file to be an ELF object.

[3] **prog1** should recognize an **ar** archive correctly.

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

Chapter 3

Peering Inside an ELF Object

This chapter takes a deeper look at the structure of ELF objects.

3.1 ELF Object Kinds

ELF supports multiple kinds of objects:

- *Relocatable objects* contain compiled code along with extra information used by tools like linkers. Relocatable objects are usually created by compilers when source code is compiled.
- *Executables* contain code in a form that an operating system can directly use to launch a process. The process of forming executables from collections of relocatable objects is called *linking*.
- *Dynamically loadable objects* contain code in a form suitable for loading into a running process. Shared libraries are examples of dynamically loadable objects.
- *Core files* contain the memory image of a process. Core files are usually generated when programs crash.

Each of these kinds of objects has a different internal structure.

3.2 ELF File Layout

Figure 3.1 on the following page shows the layout of a typical ELF object.

Every ELF object starts with a mandatory data structure known as the ELF Executable Header. This header is followed by optional content—depending on the kind of ELF object, this could be an optional ELF Program Header Table and zero or more ELF Sections:

- The ELF Program Header Table is found in executables and dynamically loadable objects. This data structure contains information that is used

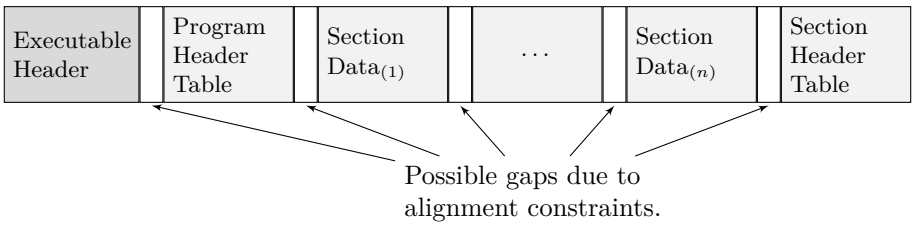

Figure 3.1: The layout of a typical ELF File.

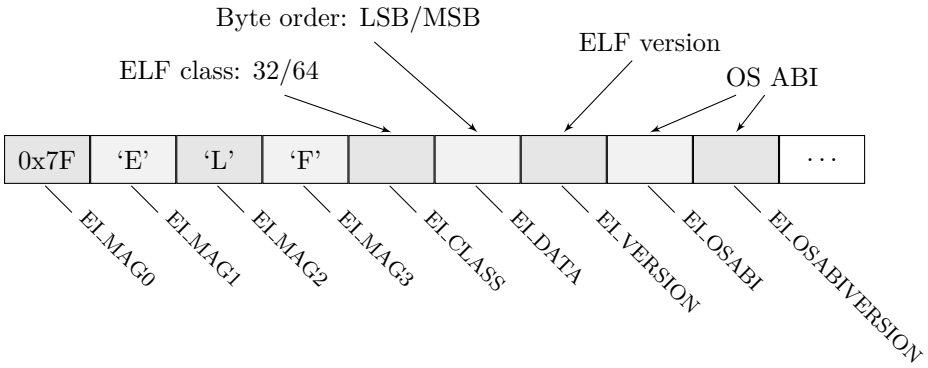

Figure 3.2: The layout of the `e_ident` array.

when the ELF object is loaded into a process. We look at the Program Header Table more closely in chapter 4 on page 25.

- ELF Sections are present in most ELF files. Sections are contiguous regions inside the ELF object holding data of a specific kind. ELF sections are described by entries in an ELF Section Header Table. Chapter 5 on page 33 describes ELF Sections and the Section Header Table in further detail.

The optional elements of an ELF object are shown with a lighter background in figure 3.1.

The ELF Executable Header

Table 3.1 on the facing page describes the layout of an ELF Executable Header using a "C-like" notation that shows the sizes and ordering of its members. In an actual ELF object the data in the header would be stored using the ELF object's "native" byte ordering; this ordering could differ from the byte ordering used by the program reading the header. 32-bit and 64-bit ELF Executable Header structures also have slightly different layouts due to the differing sizes of their members.

1 Figure 3.2 shows the contents of the first 16 bytes of the ELF header (the `e_ident` array).

	32 bit Executable Header		64 bit Executable Header	
	`typedef struct {`		`typedef struct {`	
1	`unsigned char`	`e_ident[16];`	`unsigned char`	`e_ident[16];`
2	`uint16_t`	`e_type;`	`uint16_t`	`e_type;`
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;`
4	`uint32_t`	`e_phoff;`	`uint64_t`	`e_phoff;`
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;`
6	`uint16_t`	`e_phnum;`	`uint16_t`	`e_phnum;`
7	`uint16_t`	`e_shnum;`	`uint16_t`	`e_shnum;`
8	`uint16_t`	`e_shstrndx;`	`uint16_t`	`e_shstrndx;`
	`} Elf32_Ehdr;`		`} Elf64_Ehdr;`	

Table 3.1: The ELF Executable Header.

The first 4 bytes of an ELF object always contain 0x7F, 0x45 (ASCII 'E'), 0x4c (ASCII 'L') and 0x46 (ASCII 'F').

The next three bytes specify:

- The ELF class of the object—whether it is a 32 bit ELF object (`ELFCLASS32`) or a 64 bit (`ELFCLASS64`) one.
- The endianness of the object—whether little-endian (`ELFDATA2LSB`) or big-endian (`ELFDATA2MSB`).
- The ELF specification version number that the object conforms to. ELF object versioning was discussed in chapter 2 on page 11.

With this information on hand, the `libelf` library can then interpret the rest of the ELF Executable Header correctly.

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

3 The `e_machine` member describes the machine architecture for the ELF object. Example values are '3' (`EM_386`) for the Intel® i386™ architecture, and '20' (`EM_PPC`) for the 32-bit PowerPC™ architecture.

4 **5** The ELF Executable Header also describes where to find the ELF Program Header Table and the Section Header Table, if these data structures are present in the ELF object (Figure 3.3 on the next page).

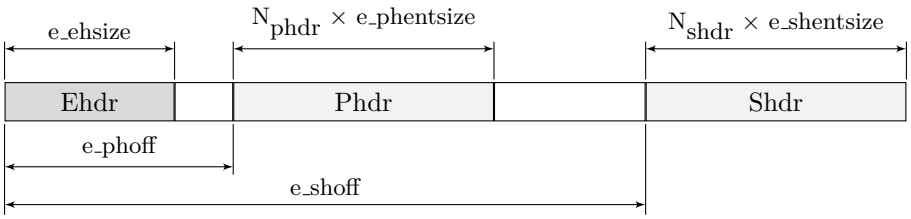

Figure 3.3: The ELF Executable Header describes the layout of the rest of the ELF object.

The `e_phoff` and `e_shoff` members contain the file offsets at which the ELF Program Header Table and the ELF Section Header Table reside in the ELF object. These members are zero if the file does not contain the corresponding data structures. The sizes of these tables are determined by the `e_phentsize` and `e_shentsize` members of the executable header, in conjunction with the number of entries in these tables.

The ELF Executable Header describes its own size (in bytes) in member `e_ehsize`.

6 7 The `e_phnum` and `e_shnum` members contain the number of ELF program header table entries and section header table entries respectively.

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 3.3) is used to encode the actual number of sections or program header table entries. When extended numbering is in use these fields will contain special values instead of actual counts.

8 When an ELF object contains sections, the names of these sections are stored in a separate string table section. ELF string tables will be covered in more detail in section 5.3 on page 37.

The `e_shstrndx` member stores the section index of this string table (possibly using "Extended Numbering", see section 3.3). This allows tools processing the ELF object to use the correct string table for looking up section names.

The `e_entry` and `e_flags` members are used for executables. These members are placed in the executable header for easy access at program load time. This tutorial will not discuss these members any further.

3.3 Extended Numbering

The `e_shnum`, `e_phnum` and `e_shstrndx` members of the ELF Executable Header are 2 bytes long and are not wide enough to represent numbers larger than 65535. We would therefore need a different way of encoding these numbers for ELF objects with a large number of sections or segments.

When extended numbering is in use, the actual values of these members will be stored in the normally unused zeroth section header table entry.[1]

- The true number of sections will stored in the `sh_size` field of the zeroth section header table entry, while the `e_shnum` member of the ELF executable header will be set to zero.
- The actual number of program header table entries will be stored in the `sh_info` field of the zeroth section header table entry, while the `e_phnum` member of the executable header will be set to the value `PN_XNUM` (0xFFFF).
- The true index of the section name string table will be stored in the `sh_link` field of the zeroth entry of the section header table, while the `e_shstrndx` member of the executable header will be set to the value `SHN_XINDEX` (0xFFFF).

You should always use the functions `elf_getphdrnum`, `elf_getshdrnum` and `elf_getshdrstrndx` to read the value of these members from an `Elf` descriptor. Directly using the values of the `e_phnum`, `e_shnum` and `e_shstrndx` members of the executable header is likely to lead to incorrect program behavior.

3.4 The Elf32, Elf64 and GElf APIs

The ELF(3) API is defined in terms of ELF class-dependent types (`Elf32_Ehdr`, `Elf64_Shdr`, etc). Consequently many operations on ELF handles in the ELF(3) API have both 32- and 64- bit variants.

For example, in order to retrieve an ELF executable header from a 32 bit ELF object we would use the function `elf32_getehdr`, which would return a pointer to an `Elf32_Ehdr` structure. For a 64-bit ELF object, we would need to use the function `elf64_getehdr`, which would return a pointer to an `Elf64_Ehdr` structure.

This duplication is awkward when you want to write applications that need to process either class of ELF objects..

The GELF(3) APIs provide a way to write applications that can handle objects of both ELF classes without code duplication. These APIs are defined in terms of "generic" C types that are large enough to hold their corresponding 32-bit and 64- bit ELF types. The GELF(3) data types have names that start with `GElf_`, and the functions have names that start with `gelf_`. You can freely mix calls to GELF(3) and ELF(3) functions in your code.

The downside to using the GELF(3) APIs is the small cost of the copying and conversion that happens behind the scenes inside `libelf`. This overhead is usually insignificant for most programs.

3.5 File and Memory Representations

ELF objects use the native word width, enddianness and data alignment rules of the machine they are intended for. These could be different from the native

[1] Section header table entries are covered in more detail in section 5 on page 33.

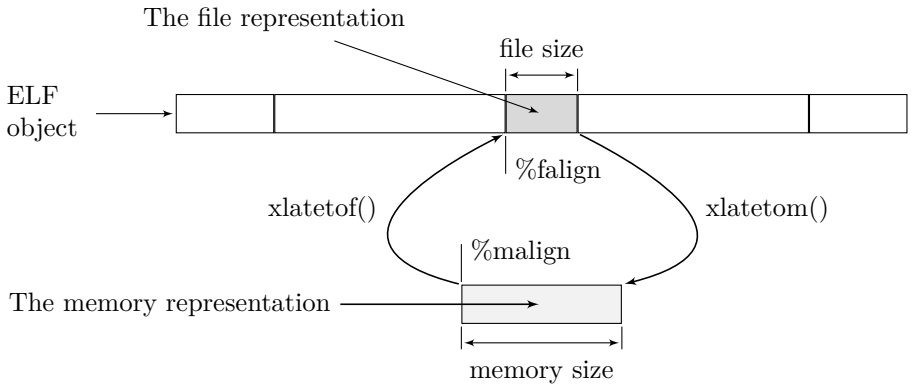

Figure 3.4: The relationship between the file and memory representation of an ELF data structure.

word width, enddianness and data alignment rules for the machine that the program reading the ELF object is running on.

ELF data structures therefore have two distinct representations:

- An *in-memory representation* that obeys the constraints for the machine architecture that the program handling the ELF object is running on.
- An *in-file representation* that corresponds to the target architecture for the ELF object.

Figure 3.4 depicts the relationship between the in-file and in-memory representations of an ELF data structure. This figure shows that:

- The size of an ELF data structure in the file could be different from its size in memory.
- The alignment restrictions placed on the data structure (denoted by `%falign` and `%malign` in the figure) could differ.
- The byte ordering of data in the file could be different from that in memory.

When using `libelf` you do not need to handle these differences in your code—`libelf` will handle the conversions of in-memory ELF data structures to and from their in-file representations automatically. For example, in program 3.1 below, the `libelf` library will automatically do the necessary byteswapping and alignment adjustments when reading in the ELF executable header.

If you need finer-grain control over the conversion process, the `libelf` library offers the class-dependent `elf`*NN*`_xlatetof` and `elf`*NN*`_xlatetom` functions. This introductory tutorial does not discuss these functions further.

3.6 Example: Reading an ELF Executable Header

Let us now examine a program that will print out the ELF Executable Header present in an ELF object. Our example program will use the class-independent GELF(3) APIs.

Listing 3.1: Program 2

```
/*
 * Print the ELF Executable Header from an ELF object.
 *
 * $Id: prog2.txt 3830 2020-03-01 14:15:53Z jkoshy $
 */

#include <err.h>
#include <fcntl.h>
#include <gelf.h> 1
#include <stdio.h>
#include <stdint.h>
#include <stdlib.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(EXIT_FAILURE, "usage:␣%s␣file-name", argv[0]);

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

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

    if ((e = elf_begin(fd, ELF_C_READ, NULL)) == NULL) 3
        errx(EXIT_FAILURE, "elf_begin()␣failed:␣%s.",
            elf_errmsg(-1));

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

    if (gelf_getehdr(e, &ehdr) == NULL) 4
        errx(EXIT_FAILURE, "getehdr()␣failed:␣%s.",
            elf_errmsg(-1));

    if ((i = gelf_getclass(e)) == ELFCLASSNONE) 5
        errx(EXIT_FAILURE, "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) [6]
        errx(EXIT_FAILURE, "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); [7]
    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) [8]
        errx(EXIT_FAILURE, "getshdrnum()␣failed:␣%s.",
            elf_errmsg(-1));
    (void) printf(PRINT_FMT, "(shnum)", (uintmax_t) n);

    if (elf_getshdrstrndx(e, &n) != 0) [9]
        errx(EXIT_FAILURE, "getshdrstrndx()␣failed:␣%s.",
            elf_errmsg(-1));
    (void) printf(PRINT_FMT, "(shstrndx)", (uintmax_t) n);

    if (elf_getphdrnum(e, &n) != 0) [10]
        errx(EXIT_FAILURE, "getphdrnum()␣failed:␣%s.",
            elf_errmsg(-1));
    (void) printf(PRINT_FMT, "(phnum)", (uintmax_t) n);

    (void) elf_end(e);
    (void) close(fd);
    exit(EXIT_SUCCESS);
```

```
}
```

1 Source code that uses the GELF(3) APIs should include the `gelf.h` header file.

2 The `GElf_Ehdr` type used here has fields that are large enough to contain values for a 64 bit ELF executable header.

3 The `elf_begin` function allocates an `Elf` handle opened for reading.

4 The function `gelf_getehdr` retrieves the executable header present in the ELF object.

This function translates the ELF executable header in the file to its corresponding in-memory representation in the C type `GElf_Ehdr`. For example, if a 32-bit ELF object is being examined, then the values in its executable header would be appropriately expanded and/or byte swapped by this function.

5 The `gelf_getclass` function retrieves the ELF class of the object being examined.

6 The `elf_getident` function retrieves the contents of the `e_ident` array from the ELF descriptor. These bytes would also be present in the `e_ident` member of the Executable Header structure. We print the first few bytes of the `e_ident` byte array.

7 After printing out the values of the bytes in the `e_ident` array, we print the values of some of the fields of the ELF executable header structure.

8 The function `elf_getphdrnum` retrieves the count of program header table entries in the ELF object.

9 The `elf_getshdrnum` function retrieves the number of sections in the ELF object.

10 The function `elf_getshdrstrndx` function retrieves the index of the section name string table in the object.

Save the program in listing 3.1 on page 21 to a file named `prog2.c`, and 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
```

1 The process for compiling and linking a GELF(3) application is the same as for other `libelf` based programs.

2 We run our program on itself. This listing in this tutorial was generated on an AMD64™ 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 system memory. The technical name for this process is "loading".

When loading an ELF program into memory, the operating system views it as comprising of distinct parts, where each part has a particular characteristic. For example, one part of the program could contain read-only data that needs to be loaded at a specific virtual memory address. Another part could contain executable code. Each such part of the ELF object is called an ELF Segment.

By way of an example, the FreeBSD™ operating system expects programs to contain a segment containing executable code. This segment is called the program's "`text`" segment. A `text` segment would usually be loaded into memory with 'read' and 'execute' permissions. Multiple processes using the same executable could potentially share the same `text` segment. FreeBSD programs would usually have `data` segments too; these segments are placed in memory with 'read' and 'write' permissions, and made private to each process.

Like executables, dynamically linked objects can be viewed as comprising segments.

The segments present in an ELF object are described by a data structure known as the ELF Program Header Table. We will study this data structure in this chapter, and write an example program that displays the Program Header Table present in an ELF object.

4.1 The ELF Program Header Table

An ELF Program Header Table is a contiguous array of Program Header Table Entry structures. Every segment present in the ELF object would have a corresponding entry in the program header table.

The location of the program header table within the ELF object is given by the `e_phoff` member of the ELF executable header (see figure 3.3 in section 3.2). This member holds the offset in bytes from the start of the ELF object to the start of its program header table.

Table 4.1 on the next page lists the members of a Program Header Table Entry structure. Figure 4.1 on the following page illustrates how these members

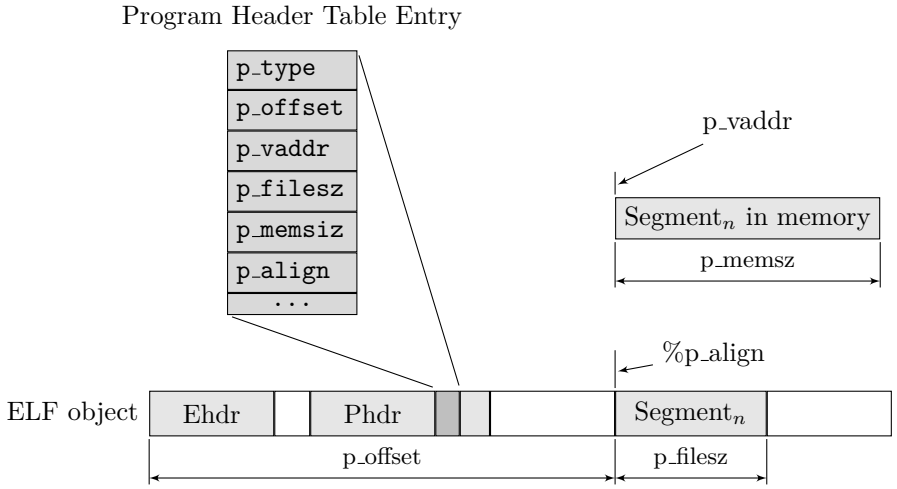

Figure 4.1: ELF Segment Placement.

specify the segment's placement both in memory and within the ELF object.

	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;`	

Table 4.1: ELF Program Header Table Entries.

1 The `p_type` member of the program specifies the type of the ELF segment. The type of the segment is specified by one of the `PT_*` constants in the programming API. Examples include:

- A segment of type `PT_LOAD` contains data that needs to be placed in memory.
- A segment of type `PT_PHDR` describes the ELF Program Header Table itself.
- A segment of type `PT_INTERP` contains a path to the runtime linker used by dynamically linked executables.

- A segment of type `PT_NOTE` contains auxiliary information.

2 The `p_offset` member holds the offset from the start of the ELF object to the start of the segment being described by this table entry.

3 The `p_vaddr` member specifies the virtual address that this segment should be placed at.

4 The `p_paddr` member specifies the physical memory address this segment should be loaded at.

5 The `p_filesz` member specifies the size of the segment in the file. This number can be zero if the segment does not use data from file (for example, if the segment is a memory-only segment).

6 The `p_memsz` member specifies the number of bytes of memory the segment would use.

7 The `p_flags` member specifies additional segment properties. For example, the value `PF_X` specifies that the segment should be made executable, the value `PF_W` specifies that the segment should be writable, and so on.

8 The `p_align` member specifies the alignment requirements of the segment in memory and in the file. This member holds a number that is a power of two.

The file representation of a Program Header Table uses the ELF object's native endianness. The `libelf` library will handle the translation between the in-file and in-memory representations of program header table entries for you. Please see section 3.5 on page 19 for more information on in-memory and in-file representations of ELF data structures.

4.2 Example: Reading a Program Header Table

The example program in this chapter will read and print out the program header table in an ELF object. This example, like the previous one, uses the class-agnostic GELF(3) APIs.

Listing 4.1: Program 3

```
/*
 * Print the ELF Program Header Table in an ELF object.
 *
 * $Id: prog3.txt 3834 2020-03-03 21:40:31Z jkoshy $
 */

#include <err.h>
#include <fcntl.h>
#include <gelf.h> 1
#include <stdio.h>
```

```
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>

void
print_ptype(size_t pt) [2]
{
    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; [3]

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

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

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

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

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


    if (elf_getphdrnum(e, &n) != 0) [4]
```

```
        errx(EXIT_FAILURE, "elf_getphdrnum() failed: %s.",
            elf_errmsg(-1));

    for (i = 0; i < n; i++) { [5]
        if (gelf_getphdr(e, i, &phdr) != &phdr) [6]
            errx(EXIT_FAILURE, "getphdr() failed: %s.",
                elf_errmsg(-1));

        (void) printf("PHDR %d:\n", i);
#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); [7]
        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(EXIT_SUCCESS);
}
```

[1] Source code that uses the GELF(3) functions needs to include the `gelf.h` header file.

[2] `print_ptype` is a helper function that translates the value of the `p_type` member to human-readable form.

[3] The GELF(3) functions in this example will use the `GElf_Phdr` C type. This type has members that are large enough for both the 32-bit (`Elf32_Phdr`) and 64-bit (`Elf64_Phdr`) header table entries.

[4] The function `elf_getphdrnum` will retrieve the number of program header table entries in the ELF object.

5 This `for` loop iterates over the valid indices for the Program Header Table.

6 The `gelf_getphdr` function retrieves the program header table entry at a specified index. If successful it will return the `GElf_Phdr` pointer that was passed to it.

7 The remaining lines of the loop's body print out the contents of the returned `GElf_Phdr` entry.

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 with `libelf`, as before.

2 We run our program on itself, and have it print out its own program header table. This listing was generated on an i386™ machine running FreeBSD.™

3 The very first entry of this particular program header table describes the object's Program Header Table itself.

4 The program **prog3** contains a header entry of type `PT_INTERP` because it is dynamically linked. A segment of type `PT_INTERP` contains the path name to the "interpreter" that the kernel should use when executing the program. This is usually the runtime loader (the file `/libexec/ld-elf.so.1` on FreeBSD systems).

5 **6** This object contains two loadable segments. The first segment requires execute and read permissions, and the second read and write permissions. Both segments require page (4096 byte) alignment.

You should try runing **prog3** on other object files.

- Try running **prog3** on a relocatable object created by a **cc -c** invocation. Does this object have a program header table?
- Try running **prog3** on a shared library. What does the Program Header Table look like for a shared library?
- Can you find ELF objects on your system that contain program header table entries of type `PT_TLS`?

Chapter 5

Looking at Sections

Compilers and linkers view ELF objects differently than operating systems do. These tools treat ELF files as a collection of ELF Sections.

An ELF Section is contiguous region of an ELF object holding one kind of data. For example, an ELF relocatable object could have sections with executable code, symbol tables, code relocation entries, and so on. Non-empty sections do not overlap in the ELF object.

5.1 The Section Header Table

The sections of an ELF object are described by a data structure known as the ELF Section Header Table. The Section Header Table is usually found at the very end of the ELF object (see figure 3.1 on page 16). The `e_shoff` member in the ELF Executable Header for the object specifies the location of the Section Header Table.

Every ELF Section present in an ELF object is described by an ELF Section Header Table Entry (see table 5.1 on the next page). Figure 5.1 shows how the fields of an ELF Section Header Entry specify the section's placement within the ELF object.

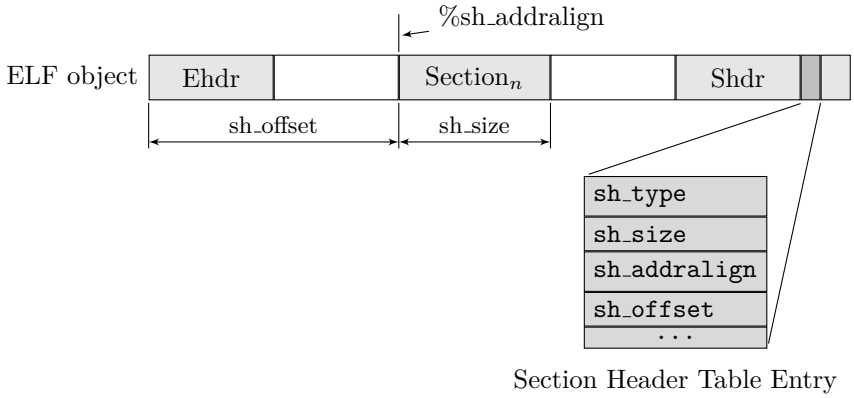

Figure 5.1: Section layout.

	32 bit SHDR Table Entry		64 bit SHDR Table Entry	
	typedef struct {		typedef struct {	
1	Elf32_Word	sh_name;	Elf64_Word	sh_name;
2	Elf32_Word	sh_type;	Elf64_Word	sh_type;
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;
4	Elf32_Xword	sh_size;	Elf64_Xword	sh_size;
5	Elf32_Word	sh_link;	Elf64_Word	sh_link;
6	Elf32_Word	sh_info;	Elf64_Word	sh_info;
7	Elf32_Word	sh_addralign;	Elf64_Word	sh_addralign;
8	Elf32_Word	sh_entsize;	Elf64_Word	sh_entsize;
	} Elf32_Shdr;		} Elf64_Shdr;	

Table 5.1: ELF Section Header Table Entries.

1 The `sh_name` member encodes the section's name. Because section names can be of variable length, they are not kept in the section header table entry itself. Instead, all section names are placed in a common "section name string table", and the `sh_name` member in the section header entry stores the byte offset of the section's name in that string table. The ELF Executable Header has an `e_shstrndx` member that contains the section index of the section name string table itself. We will look at ELF string tables in greater detail in section 5.3 on page 37.

2 The `sh_type` member specifies the section's type. Section types are defined by the `SHT_*` constants defined in the system's ELF headers. For example, a section of type `SHT_PROGBITS` would contain executable code, and a section of type `SHT_SYMTAB` would hold a symbol table.

3 The flags field indicates whether the section has specific properties; for example, whether it contains writable data, whether it has special link ordering requirements, and so on.

4 The `sh_size` member specifies the size of the section in bytes.

5 **6** The `sh_link` and `sh_info` members contain additional section-specific information. We do not look at these members further in this tutorial.

7 For sections with specific alignment requirements, the `sh_addralign` member holds the required alignment. Its value would be a power of two.

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 quirks to keep mind when handling ELF sections:

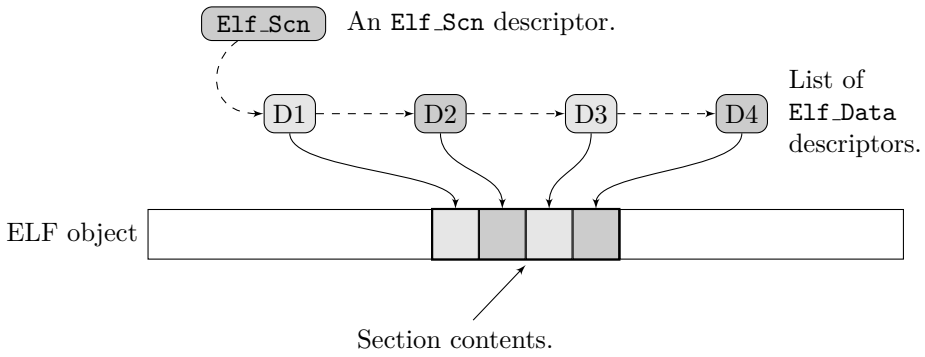

Figure 5.2: Coverage of an ELF section by `Elf_Scn` and `Elf_Data` descriptors.

- First, the section header table entry at index '0' (`SHN_UNDEF`) is special: it is always of type `SHT_NULL`. When extended numbering is not in use this entry has its members set to zero. When extended numbering is in use, the fields of this entry could be non-zero; please see section 3.3 on page 18 for a discussion of extended numbering.

- Next, valid section indices range from `SHN_UNDEF` (0) up to `SHN_LORESERVE`−1 (0xFEFF). Section indices between 0xFF00 (`SHN_LORESERVE`) and 0xFFFF (`SHN_HIRESERVE`) have special meanings. If an ELF object has more than 65279 (0xFEFF) sections, then it will need to use extended section numbering.

5.2 ELF Section Handling With `libelf`

The `libelf` library offers APIs to retrieve section header table entries and the contents of sections.[1] These APIs take care of translating between the in-file and in-memory representations of data in the ELF object; your application can then directly work with the in-memory representation of data.

ELF sections are represented by `libelf` using a type named `Elf_Scn`. This type is meant to be opaque to application code—the only way to allocate an `Elf_Scn` is by calling one of `libelf`'s APIs..

Notable functions in the API that operate on sections include:

- The function `elf_getscn` retrieves section information for a specified section index.

- The function `elf_nextscn` is used to iterate through the sections in the ELF object..

- The function `gelf_getshdr` retrieves the section header table entry for a section.

- The functions `elf_getdata` retrieves the contents of the section.

[1] We will cover the adding new sections to ELF objects in chapter 6 on page 43.

An `Elf_Scn` descriptor is associated with zero or more `Elf_Data` descriptors. Each `Elf_Data` descriptor describes a region of application memory containing data for the ELF section. Figure 5.2 on the preceding page shows how the `Elf_Data` descriptors for an `Elf_Scn` descriptor could cover the content of a section.

Listing 5.1 shows the C definition of the `Elf_Scn` and `Elf_Data` types.

Listing 5.1: The `Elf_Data` and `Elf_Scn` types

```
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
        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 the alignment of the data referenced in the `Elf_Data` with respect to its containing section.

3 The `d_buf` member points to a contiguous region of application memory containing the section's data.

4 The `d_off` member contains the file offset from the start of the section for the data in this buffer.

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

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

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. Please see chapter 2 for more information on ELF version numbers.

Figure 5.3 on the facing page shows how the members of the `Elf_Data` descriptor describe a region of application memory containing section data. As seen in the figure, the in-memory representation of this data might have a different size and different endianness than its in-file representation.

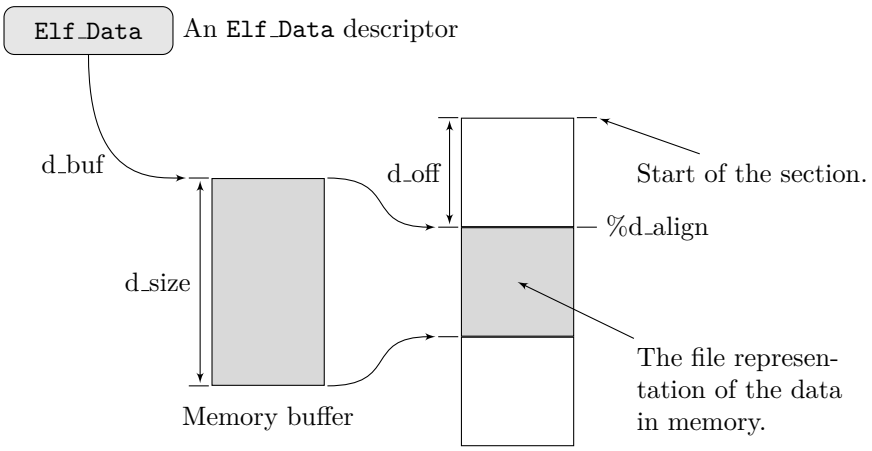

Figure 5.3: How `Elf_Data` descriptors work.

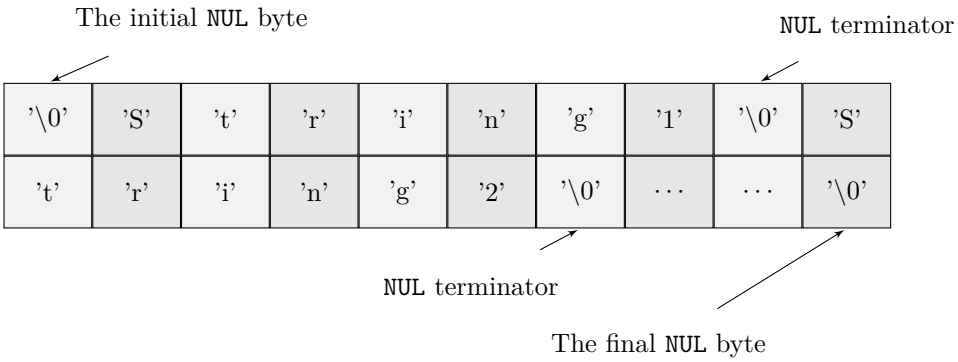

Figure 5.4: String Table Layout.

5.3 ELF String Tables

ELF string tables hold variable length strings. Other ELF data structures refer to the strings stored in these tables by using byte offsets from the start of the string table.

Figure 5.4 shows the layout of a string table.

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

Sections containing string tables have section type `SHT_STRTAB`.

An ELF file can have multiple string tables. For example, the names of the sections could be kept in a section name string table while the names of program symbols could be kept in a symbol name string table.

The `elf_strptr` function in the `libelf` library converts string table offsets into `char *` pointers usable by C code.

5.4 Example: Listing Section Names

Let us now write an example program that prints the names of the sections in an ELF object.

Listing 5.2: Program 4

```
/*
 * Print the names of ELF sections.
 *
 * $Id: prog4.txt 3687 2019-02-22 07:55:09Z jkoshy $
 */

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

int
main(int argc, char **argv)
{
    int fd;
    Elf *e;
    Elf_Scn *scn;
    Elf_Data *data;
    GElf_Shdr shdr;
    size_t n, shstrndx, sz;
    char *name, *p, pc[(4 * sizeof(char)) + 1];

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

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

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

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

    if (elf_kind(e) != ELF_K_ELF)
```

```
        errx(EXIT_FAILURE, "%s␣is␣not␣an␣ELF␣object.",
            argv[1]);

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

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

    if (gelf_getshdr(scn, &shdr) != &shdr)
        errx(EXIT_FAILURE, "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(EXIT_SUCCESS);
}
```

1 The function `elf_getshdrstrndx` retrieves the section index of the section name string table. Using this function allows our program to work correctly when the object being examined uses extended numbering.

2 The function `elf_nextscn` has the useful property that it will return the pointer to the `Elf_Scn` descriptor for section number 1 if a NULL pointer is passed in to it. Section number 0 is always of type `SHT_NULL`, and is not for use by applications.

3 This `while` loop iterates through the sections in the ELF object. Function `elf_nextscn` will return NULL after the last section has been traversed, giving us a convenient way to exit the loop.

4 The function `gelf_getshdr` retrieves the section header table entry for an `Elf_Scn` descriptor. The `sh_name` member of the returned section header table entry holds the byte offset of the section's name inside the section name string table.

5 The `elf_strptr` function converts the byte offset in the `sh_name` member to a `char *` pointer. The pointed-to C string can then be printed using `printf`.

6 Next, the contents of the section name string table itself are printed out. The call to `elf_getscn` returns the `Elf_Scn` descriptor for the section name string table itself.

7 This code cycles through the `Elf_Data` descriptors for the section, printing out the characters in each `Elf_Data` buffer for the section.

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.

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...
```

1 Compile and link the program in the standard way.

2 The program is invoked on itself, to print the names of its own sections.

3 The section name string table is called `.shstrtab` by convention.

4 This is the content of the section name string table in this object.

Chapter 6

Creating New ELF Objects

This chapter shows how to use the `libelf` library to create new ELF objects.

6.1 Example: Creating an ELF Object

The example program in listing 6.1 creates an ELF file with the following content:

- A section named "`.foo`" containing data in the form of 32-bit words that may need byte-swapping. We had discussed `libelf`'s handling of data needing byte-swapping in section 3.5 on page 19.
- A section named "`.shstrtab`" containing the section name string table for our ELF object. We covered string tables in section 5.3 on page 37.
- A Program Header Table with a single Program Header Table Entry covering the program header table itself. We studied the ELF Program Header Table in chapter 4.

The new ELF object will be marked as a 32-bit PowerPC™ executable, and will use big-endian data ordering.

Listing 6.1: Program 5

```
/*
 * Create an ELF object.
 *
 * $Id: prog5.txt 2133 2011-11-10 08:28:22Z jkoshy $
 */

#include <err.h>
#include <fcntl.h>
#include <libelf.h> 1
#include <stdio.h>
#include <stdlib.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(EXIT_FAILURE, "usage: %s file-name", argv[0]);

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

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

    if ((e = elf_begin(fd, ELF_C_WRITE, NULL)) == NULL) 5
        errx(EXIT_FAILURE, "elf_begin() failed: %s.",
            elf_errmsg(-1));

    if ((ehdr = elf32_newehdr(e)) == NULL) 6
        errx(EXIT_FAILURE, "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(EXIT_FAILURE, "elf32_newphdr() failed: %s.",
            elf_errmsg(-1));

    if ((scn = elf_newscn(e)) == NULL) 8
```

```
        errx(EXIT_FAILURE, "elf_newscn() failed: %s.",
            elf_errmsg(-1));

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

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

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

    exit(EXIT_SUCCESS);
}
```

1 The header file `libelf.h` brings in function prototypes for `libelf`'s functions.

2 The `hash_words` array holds 32-bit words. The values in the array would need to be written using big-endian byte ordering when the section is written to file.

3 The `string_table` array holds a pre-fabricated string table containing the names of our two sections, "`.foo`" and "`.shstrtab`".

4 The first step in creating a new ELF object is to obtain a file descriptor opened for writing.

5 The call to function `elf_begin` allocates an `Elf` handle. The parameter `ELF_C_WRITE` informs `libelf` of our intent to create a brand new ELF object.

6 The function `elf32_newehdr` allocates an ELF Executable Header. An Executable Header is always needed for an ELF object.

The next few lines populate the executable header:

1. The `EI_DATA` byte in the `e_ident` member is set to the desired endianness (big-endian in our case).
2. The machine type is set to the constant `EM_PPC`, for the PowerPC™ architecture.
3. The object is marked as an ELF executable.

The new ELF object will be a 32-bit object, since its Executable Header had been allocated using the 32-bit `elf32_newehdr` function.

7 The call to `elf_newphdr` allocates an ELF Program Header Table containing a single entry. This entry is meant to cover the Program Header Table itself.

At this point in our program we do not know the file offset at which the ELF Program Header Table will be placed inside our new ELF object. This offset would only be known after the object is laid out. We need to defer filling in our program header table entry until `libelf` has computed an object layout for us (please see step 11 below).

8 The call to `elf_newscn` allocates an `Elf_Scn` descriptor for the ELF section that will hold the values in the `hash_words` array.

To actually associate data with our new section we allocate an `Elf_Data` descriptor and set its fields to map the `hash_words` array.

A call to `elf32_getshdr` then returns the Section Header Table Entry for the new section.

1. The type of the new section is set to `SHT_HASH`. The `libelf` library knows how to byte-swap sections of this type.
2. The section is marked as containing content in the file by setting its `sh_flags` field to the constant `SHF_ALLOC`.

9 The next call to `elf_newscn` allocates another section descriptor; this descriptor will be used for the section name string table. The code then allocates an `Elf_Data` descriptor for this section, and sets its members to map the pre-fabricated string table in the array `string_table`.

The call to `elf32_getshdr` retrieves the Section Header Table Entry for this section. The members of this section header table entry are set as follows:

1. The type of the section is set to `SHT_STRTAB`, the section type for string tables.
2. The section flags are set to indicate that the section contains data in the file, and that it contains `NUL`-terminated strings.

10 The function `elf_ndxscn` retrieves the section index for the string table section. The call to function `elf_setshstrndx` then sets the section name string table index field in the ELF Executable Header.

11 The call of the function `elf_update` with the parameter `ELF_C_NULL` requests the `libelf` library to compute a layout for an ELF object without writing the object out..

After the call to `elf_update` returns, the code examines the ELF object's Executable Header to determine where `libelf` had placed the object's Program Header Table. It then updates the Program Header Table Entry created in step 7 to cover the program header table's file location.

The call to function `elf_flagdata` marks the Program Header Table as having been modified.

12 Finally, the function `elf_update` is called with parameter `ELF_C_WRITE` in order to write the ELF object out to file.

If this example program is run on a little-endian host the `libelf` library will byte-swap the sections that need byte-swapping when the ELF object is written out to file.

Save the program in listing 6.1 on page 43 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...
```

[1] Compile, link and run the program as in our previous examples.

[2] [3] We can use the **file** and **readelf** programs to examine the object that we have created.

6.2 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.

You can request fine-grained control over the ELF object's layout by setting the `ELF_F_LAYOUT` flag on its `Elf` descriptor. This flag is set using the function `elf_flagelf`.

After setting the `ELF_F_LAYOUT` flag on an `Elf` descriptor, you can control the layout of the ELF object using the following parameters:

- You can set the values of the `e_phoff` and `e_shoff` members of the Executable Header. These values determine where the ELF Program Header Table and Section Header Table would be placed in the ELF object.
- For each section you can set the `sh_addralign`, `sh_offset`, and `sh_size` members of the section's header table entry. These members control the placement of the section within the ELF object.

These members must be set prior to calling the `elf_update` function.

6.3 Fill Characters

The `libelf` library will fill the gaps between the parts of the ELF object with a fill character. These gaps may arise due to the alignment constraints on adjacent sections.

You can set the fill character to use by calling the function `elf_fill` before calling `elf_update`. The default fill character is a zero byte.

6.4 Memory Ownership

Some of the APIs implemented by `libelf` return pointers to memory arenas; other APIs accept pointers to memory arenas as input. Knowing when you should (or should not) call `free()` on a pointer is essential to avoid corrupting memory.

The `libelf` library follows a simple rule: it will not free data that it did not allocate. Conversely, it *will* free memory that it had allocated.

You should not free pointers returned by `libelf`, such as the `Elf_Scn *` pointers returned by calls to `elf_getscn` and the `Elf_Data *` pointers returned by calls to `elf_newdata`. Conversely, if you had allocated a memory arena mapped by an `Elf_Data` structure, then you should release the arena once you are done with it.

6.5 Data Structure Lifetimes

Many of `libelf`'s APIs return pointers to its internal data structures. In objects opened for writing these pointers have a limited lifetime—they are only valid up till the time the ELF object is written out to file by a call to function `elf_update`.

This is because when `libelf` writes out an ELF object, it releases and reallocates some of its internal bookkeeping structures.

After calling function `elf_update` with parameter `ELF_C_WRITE` you should treat any prior pointers returned by `libelf`, such as pointers to `Elf_Scn` and `Elf_Data` structures, as invalid. If you wish to continue working with your ELF object, you should retrieve these pointers afresh from `libelf`.

6.6 Modifying Existing ELF Objects

You can use the `libelf` library to modify existing ELF objects.

The process to update an ELF object is similar to that for creating ELF objects, with the following differences:

- You would open the underlying file for both reading and writing, i.e., with mode `O_RDWR`.
- You would need an `Elf` descriptor that is valid for updates. You can allocate a suitable descriptor by calling function `elf_begin` using the parameter `ELF_C_RDWR`.
- You can use functions such as `elf_newscn`, `elf32_newphdr` and `elf64_newphdr` to add new data structures to your object. You can also retrieve existing ELF data structures in the file using APIs such as those discussed in the previous chapters of this tutorial. You can add new data to existing sections using the function `elf_newdata`.
- After you modify the fields of a data structure retrieved from the ELF object, you should call the appropriate `elf_flag` functions to inform `libelf` about your change.

A caution: when you update an ELF object, you should take care to ensure that the resulting object remains a valid ELF object. For example, if you move the sections of an ELF executable around, then you should also keep the relevant offsets in its Program Header Table entries updated. We will not however explore this topic further in this introductory tutorial.

Chapter 7

Processing ar Archives

During program development the **ar** archiver is used to manage "libraries" of object files.

You can use the `libelf` library to read these archives. The `libelf` library's APIs are however 'read-only'—**ar** archives cannot be created or modified using these APIs.[1]

In this chapter we will build an example program that takes an **ar** archive as input and lists the names and sizes of the files contained within it.

7.1 The Structure of ar Archives

Every **ar** archive starts with a signature sequence of 8 bytes (please see the constant `ARMAG` defined in the system header `ar.h`). The members of the archive follow this signature.

Figure 7.1 on the following page shows the structure of an **ar** archive.

7.1.1 The Archive Header

Each member of an **ar** archive is preceded by an archive header that describes the member's attributes. The archive header is a collection of fixed-size ASCII strings that resides at an even offset within the archive file. Listing 7.1 shows the layout of the archive header as a C `struct`.

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;
```

[1] The `libarchive` library could be used instead to create or modify **ar** archives.

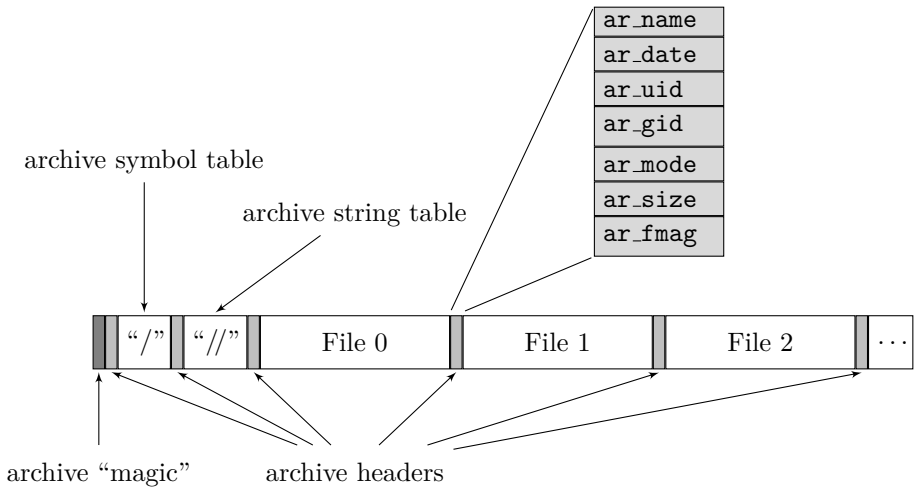

Figure 7.1: The structure of **ar** archives.

7.2 Special Archive Members

The initial members of an **ar** archive may be special:

- An archive member with the name "/" is an archive symbol table. This symbol table maps program symbols to archive members in the archive. It is usually maintained by tools like **ranlib** and **ar**.
- An archive member with the name "//" is an archive string table.

  The **ar** archive header can only contain fixed size ASCII strings. Member file names that exceed the length limits of the `ar_name` archive header field would need to be placed in a special string table.[2] The `ar_name` field of the archive header would then hold the offset within the archive string table of the real file name, encoded as a decimal number.

7.3 Archive Flavors

ar archives come in two flavors mainly: BSD and SVR4. These flavors are different in many respects—for example, SVR4 archives use a '/' character to terminate file names in the archive header, whereas BSD format archives use a ASCII space character as a terminator. The way the two formats handle long file names is also different. The archive handling APIs offered by the `libelf` library will insulate your code from the differences between the archive formats.

7.4 Archive Symbol Tables

An archive symbol table helps linkers to quickly locate the ELF objects in an archive. The BSD and SVR4 archive flavors have their own archive symbol table

[2] Archive string tables are not to be confused with ELF string tables. ELF string tables were examined in section 5.3.

formats.

If an archive symbol table is present in an **ar** archive, it will be the archive's first member.

ar archive symbol tables are read using the function `elf_getarsym`. This function returns an array of `Elf_Arsym` structures, where each `Elf_Arsym` structure maps a program symbol to a file offset within the **ar** archive. These file offsets can then be used with the `elf_rand` function to retrieve the ELF object in question (please see section 7.5 below).

Listing 7.2 contains the C definition of an `Elf_Arsym` data type.

Listing 7.2: 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;
```

7.5 Random Archive Access Using `elf_rand`

Instead of iterating over the members of an **ar** archive in sequence, you can also directly access specific members in the archive using the `elf_rand` function.

This function configures the parent archive's `Elf` descriptor to open the desired archive member on the next call to `elf_begin`.

The `elf_rand` function takes the file offset to an archive header as its input parameter. This means that the function is only useful when the file offset to the desired member's archive header is already known. If an archive contains an archive symbol table then the function `elf_getarsym` could be used to retrieve the relevant file offsets to its member's headers.

The `elf_getarsym` function was described in section 7.4 above.

7.6 Example: Stepping Through an `ar` Archive

Listing 7.3 on the next page contains a program that traverses an **ar** archive, printing out the file names and byte sizes of its members.

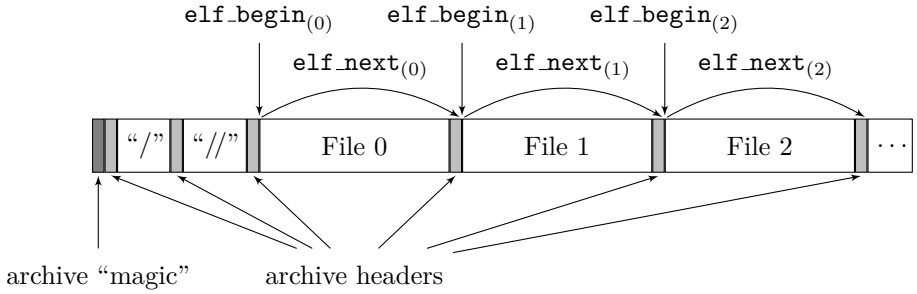

Figure 7.2: Iterating through **ar** archives with `elf_begin` and `elf_next`.

Listing 7.3: Program 6

```
/*
 * Iterate through an ar(1) archive.
 *
 * $Id: prog6.txt 3840 2020-03-14 21:19:09Z jkoshy $
 */

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

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

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

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

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

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

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

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

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

        cmd = elf_next(e); [5]
```

```
        (void) elf_end(e); 6
    }

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

1 The call to `open()` opens the archive for reading.

2 The function `elf_begin` is used to obtain an `Elf` descriptor. The code then checks that `libelf` library has recognized the file as an **ar** archive.

3 The call of `elf_begin` returns a nested `Elf` descriptor to an archive member. The third parameter passed to `elf_begin` is a pointer to the `Elf` descriptor for the archive itself.

4 The function `elf_getarhdr` retrieves the archive header for the current archive member. This function translates the (possibly encoded) file names in the archive header to `NUL`-terminated strings suitable for use with `printf`.

Figure 7.4 shows the translated information returned by the `elf_getarhdr` function.

Listing 7.4: 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;
```

The code then prints out the name and the size of the archive member using the `ar_name` and `ar_size` fields of the returned `Elf_Arhdr` structure.

5 The call of the `elf_next` function sets up the parent archive descriptor (held in the variable `ar` in our example) to return the next archive member on a subsequent call to function `elf_begin`.

The `elf_next` function will return the value `ELF_C_READ` as long as the traversal of the archive can continue. When called with a descriptor to the last member of an archive the `elf_next` function will return the value `ELF_C_NULL`. This value will cause the subsequent call to function `elf_begin` at step 3 to return `NULL`, thereby terminating the loop.

Figure 7.2 on page 53 shows how the functions `elf_begin` and `elf_next` work together to step through an **ar** archive.

6 The `elf_end` function releases the resources held by `Elf` descriptors.

Save the program in listing 7.3 to a file named `prog6.c`, and compile and run it as shown.

Listing 7.5: 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
```

1 We compile and link the program with `libelf`.

2 We run the program on an archive and obtain a listing of the archive's contents.

Chapter 8

Conclusion

This tutorial covered the following topics:

- We studied the basics of the ELF format. We looked at a few key ELF data structures, and at their layout inside ELF objects.
- We covered the facilities offered by the `libelf` library for manipulating ELF objects.
- We wrote example programs that retrieved and displayed the ELF data structures present in a few ELF objects.
- We studied how to create new ELF objects using the `libelf` library.
- We looked at how to read **ar** archives using `libelf`.

8.1 Further Reading

8.1.1 On the Web

Peter Seebach's DeveloperWorks article "An unsung hero: The hardworking ELF" covers the history and features of the ELF format. Hongjiu Liu's "ELF: From The Programmer's Perspective" describes how to use the features of ELF with GCC and GNU ld. The paper "Extending Sim286 to the Intel386 Architecture with 32-bit processing and Elf Binary input by Michael L. Haungs and Brian A. Malloy contains a description of the ELF format in the chapter "Executable and Linking Format (ELF)".

Neelakanth Nadgir's tutorial "LibElf and GElf - A Library to Manipulate ELf Files" is a readable introduction to the ELF(3) and GELF(3) APIs in Solaris™.

The Linkers and Libraries Guide from Oracle® describes the linking and loading tools present in Solaris™. Chapter 14 of this book, titled "Object File Format", contains a readable introduction to the ELF format.

8.1.2 More Example Programs

The source code for the tools being developed at the ElfToolChain Project at SourceForge.Net show the use of the ELF(3)/GELF(3) APIs in useful programs.

For readers looking for smaller programs to study, Emmanuel Azencot offers a website with example programs.

8.1.3 Books

John Levine's "Linkers and Loaders" is a readable book offering a overview of the process of linking and loading object files.

8.1.4 Standards

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 be downloaded.

8.2 Getting Further Help

If you have further questions about the use of `libelf`, please feel free to use our discussion list: `elftoolchain-developers@lists.sourceforge.net`.

Index