# ELF Hello World Tutorial



Introductory analysis of a simple example of the Executable and Linkable File format.

Extracted from my Stack Overflow answer (http://stackoverflow.com/a/30648229/895245).



# Introduction

ELF is the dominating file format for Linux. It competes with Mach-O for OS X and PE for Windows.

ELF supersedes `.coff`, which supersedes `a.out`.

## Standards

ELF is specified by the LSB (https://en.wikipedia.org/wiki/Linux_Standard_Base):

- core generic: http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html (http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/elf-generic.html)
- core AMD64: http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-AMD64/LSB-Core-AMD64/book1.html (http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-AMD64/LSB-Core-AMD64/book1.html)

The LSB basically links to other standards with minor extensions, in particular:

- Generic (both by SCO (https://en.wikipedia.org/wiki/Santa_Cruz_Operation)):
    - System V ABI 4.1 (1997) http://www.sco.com/developers/devspecs/gabi41.pdf (http://www.sco.com/developers/devspecs/gabi41.pdf), no 64 bit, although a magic number is reserved for it. Same for core files. *This* is the first document you should look at when searching for information.
    - System V ABI Update DRAFT 17 (2003) http://www.sco.com/developers/gabi/2003-12-17/contents.html (http://www.sco.com/developers/gabi/2003-12-17/contents.html), adds 64 bit. Only updates chapters 4 and 5 of the previous document: the others remain valid and are still referenced.
- Architecture specific (by the processor vendor):
    - IA-32: http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-IA32/LSB-Core-IA32/elf-ia32.html (http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-IA32/LSB-Core-IA32/elf-ia32.html), points mostly to http://www.sco.com/developers/devspecs/abi386-4.pdf (http://www.sco.com/developers/devspecs/abi386-4.pdf)
    - AMD64: http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-AMD64/LSB-Core-AMD64/elf-amd64.html (http://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-AMD64/LSB-Core-AMD64/elf-amd64.html), points mostly to http://www.x86-64.org/documentation/abi.pdf (http://www.x86-64.org/documentation/abi.pdf)

A handy summary can be found at:

```
man elf
```

## How to learn

Spin like mad between:

- standards
- high level generators. We use the assembler `as` and linker `ld`.
- hexdumps
- file decompilers. We use `readelf`. It makes it faster to read the ELF file by turning it into human readable output. But you must have seen one byte-by-byte example first, and think how `readelf` output maps to the standard.
- low-level generators: stand-alone libraries that let you control every field of the ELF files you generated. https://github.com/BR903/ELFkickers (https://github.com/BR903/ELFkickers), https://github.com/sqall01/ZwoELF (https://github.com/sqall01/ZwoELF) and many more on GitHub.
- consumer: the `exec` system call of the Linux kernel can parse ELF files to starts processes: https://github.com/torvalds/linux/blob/v4.11/fs/binfmt_elf.c (https://github.com/torvalds/linux/blob/v4.11/fs/binfmt_elf.c), http://stackoverflow.com/questions/8352535/how-does-kernel-get-an-executable-binary-file-running-under-linux/31394861#31394861 (http://stackoverflow.com/questions/8352535/how-does-kernel-get-an-executable-binary-file-running-under-linux/31394861#31394861)

## Specified file formats

The ELF standard specifies multiple file formats:

- Object files ( `.o` ).

  Intermediate step to generating executables and other formats:

  ```
  Source code

      |
      | Compilation
      |
      v

  Object file

      |
      | Linking
      |
      v

  Executable
  ```

  Object files exist to make compilation faster: with `make`, we only have to recompile the modified source files based on timestamps.

  We have to do the linking step every time, but it is much less expensive.

- Executable files (no standard Linux extension).

  This is what the Linux kernel can actually run.

- Archive files ( `.a` ).

  Libraries meant to be embedded into executables during the Linking step.

- Shared object files ( `.so` ).

  Libraries meant to be loaded when the executable starts running.

- Core dumps.

  Such files may be generated by the Linux kernel when the program does naughty things, e.g. segfault.

  They exist to help debugging the program.

In this tutorial, we consider only object and executable files.

## Implementations

- Compiler toolchains generate and read ELF files.

  Sane compilers should use a separate standalone library to do the dirty work. E.g., Binutils uses BFD (in-tree and canonical source).

- Operating systems read and run ELF files.

  Kernels cannot link to a library nor use the C stlib, so they are more likely to implement it themselves.

This is the case of the Linux kernel 4.2 which implements it in th file `fs/binfmt_elf.c` .

- Specialized libraries. Examples:
  - https://github.com/eliben/pyelftools (https://github.com/eliben/pyelftools). By a hardcore Googler: https://plus.google.com/+EliBenderskyGplus/posts (https://plus.google.com/+EliBenderskyGplus/posts)
  - https://sourceforge.net/projects/elftoolchain (https://sourceforge.net/projects/elftoolchain)

# Minimal ELF file

It is non-trivial to determine what is the smallest legal ELF file, or the smaller one that will do something trivial in Linux.

Some impressive attempts:

- http://codegolf.stackexchange.com/questions/5696/shortest-elf-for-hello-world-n (http://codegolf.stackexchange.com/questions/5696/shortest-elf-for-hello-world-n)
- http://www.muppetlabs.com/~breadbox/software/tiny/ (http://www.muppetlabs.com/~breadbox/software/tiny/)
- http://timelessname.com/elfbin/ (http://timelessname.com/elfbin/)

In this example we will consider a saner `hello world` example that will better capture real life cases.

# Generate the example

Let's break down a minimal runnable Linux x86-64 example:

```
section .data
    hello_world db "Hello world!", 10
    hello_world_len  equ $ - hello_world
section .text
    global _start
    _start:
        mov rax, 1
        mov rdi, 1
        mov rsi, hello_world
        mov rdx, hello_world_len
        syscall
        mov rax, 60
        mov rdi, 0
        syscall
```

Compiled with:

```
nasm -w+all -f elf64 -o 'hello_world.o' 'hello_world.asm'
ld -o 'hello_world.out' 'hello_world.o'
```

TODO: use a minimal linker script with `-T` to be more precise and minimal.

Versions:

- NASM 2.10.09
- Binutils version 2.24 (contains `ld` )
- Ubuntu 14.04

We don't use a C program as that would complicate the analysis, that will be level 2 :-)

# Object hd

```
hd hello_world.o
```

Gives:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 07 00 03 00  |....@.....@.....|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000080  01 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00  |................|
00000090  00 00 00 00 00 00 00 00  00 02 00 00 00 00 00 00  |................|
000000a0  0d 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  07 00 00 00 01 00 00 00  06 00 00 00 00 00 00 00  |................|
000000d0  00 00 00 00 00 00 00 00  10 02 00 00 00 00 00 00  |................|
000000e0  27 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |'...............|
000000f0  10 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  0d 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00  40 02 00 00 00 00 00 00  |........@.......|
00000120  32 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |2...............|
00000130  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000140  17 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
00000150  00 00 00 00 00 00 00 00  80 02 00 00 00 00 00 00  |................|
00000160  a8 00 00 00 00 00 00 00  05 00 00 00 06 00 00 00  |................|
00000170  04 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
00000180  1f 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
00000190  00 00 00 00 00 00 00 00  30 03 00 00 00 00 00 00  |........0.......|
000001a0  34 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |4...............|
000001b0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001c0  27 00 00 00 04 00 00 00  00 00 00 00 00 00 00 00  |'...............|
000001d0  00 00 00 00 00 00 00 00  70 03 00 00 00 00 00 00  |........p.......|
000001e0  18 00 00 00 00 00 00 00  04 00 00 00 02 00 00 00  |................|
000001f0  04 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
00000200  48 65 6c 6c 6f 20 77 6f  72 6c 64 21 0a 00 00 00  |Hello world!....|
00000210  b8 01 00 00 00 bf 01 00  00 00 48 be 00 00 00 00  |..........H.....|
00000220  00 00 00 00 ba 0d 00 00  00 0f 05 b8 3c 00 00 00  |............<...|
00000230  bf 00 00 00 00 0f 05 00  00 00 00 00 00 00 00 00  |................|
00000240  00 2e 64 61 74 61 00 2e  74 65 78 74 00 2e 73 68  |..data..text..sh|
00000250  73 74 72 74 61 62 00 2e  73 79 6d 74 61 62 00 2e  |strtab..symtab..|
00000260  73 74 72 74 61 62 00 2e  72 65 6c 61 2e 74 65 78  |strtab..rela.tex|
00000270  74 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |t...............|
00000280  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000290  00 00 00 00 00 00 00 00  01 00 00 00 04 00 f1 ff  |................|
000002a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002b0  00 00 00 00 03 00 01 00  00 00 00 00 00 00 00 00  |................|
000002c0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 02 00  |................|
000002d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002e0  11 00 00 00 00 00 01 00  00 00 00 00 00 00 00 00  |................|
000002f0  00 00 00 00 00 00 00 00  1d 00 00 00 00 00 f1 ff  |................|
00000300  0d 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000310  2d 00 00 00 10 00 02 00  00 00 00 00 00 00 00 00  |-...............|
00000320  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000330  00 68 65 6c 6c 6f 5f 77  6f 72 6c 64 2e 61 73 6d  |.hello_world.asm|
00000340  00 68 65 6c 6c 6f 5f 77  6f 72 6c 64 00 68 65 6c  |.hello_world.hel|
00000350  6c 6f 5f 77 6f 72 6c 64  5f 6c 65 6e 00 5f 73 74  |lo_world_len._st|
00000360  61 72 74 00 00 00 00 00  00 00 00 00 00 00 00 00  |art.............|
00000370  0c 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00  |................|
00000380  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000390
```

# Executable hd

```
hd hello_world.out
```

Gives:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  b0 00 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  10 01 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  02 00 40 00 06 00 03 00  |....@.8...@.....|
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  d7 00 00 00 00 00 00 00  d7 00 00 00 00 00 00 00  |................|
00000070  00 00 20 00 00 00 00 00  01 00 00 00 06 00 00 00  |.. .............|
00000080  d8 00 00 00 00 00 00 00  d8 00 60 00 00 00 00 00  |..........`.....|
00000090  d8 00 60 00 00 00 00 00  0d 00 00 00 00 00 00 00  |..`.............|
000000a0  0d 00 00 00 00 00 00 00  00 00 20 00 00 00 00 00  |.......... .....|
000000b0  b8 01 00 00 00 bf 01 00  00 00 48 be d8 00 60 00  |..........H...`.|
000000c0  00 00 00 00 ba 0d 00 00  00 0f 05 b8 3c 00 00 00  |............<...|
000000d0  bf 00 00 00 00 0f 05 00  48 65 6c 6c 6f 20 77 6f  |........Hello wo|
000000e0  72 6c 64 21 0a 00 2e 73  79 6d 74 61 62 00 2e 73  |rld!...symtab..s|
000000f0  74 72 74 61 62 00 2e 73  68 73 74 72 74 61 62 00  |trtab..shstrtab.|
00000100  2e 74 65 78 74 00 2e 64  61 74 61 00 00 00 00 00  |.text..data.....|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000150  1b 00 00 00 01 00 00 00  06 00 00 00 00 00 00 00  |................|
00000160  b0 00 40 00 00 00 00 00  b0 00 00 00 00 00 00 00  |..@.............|
00000170  27 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |'...............|
00000180  10 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000190  21 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00  |!...............|
000001a0  d8 00 60 00 00 00 00 00  d8 00 00 00 00 00 00 00  |..`.............|
000001b0  0d 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001c0  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001d0  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000001e0  00 00 00 00 00 00 00 00  e5 00 00 00 00 00 00 00  |................|
000001f0  27 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |'...............|
00000200  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000210  01 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
00000220  00 00 00 00 00 00 00 00  90 02 00 00 00 00 00 00  |................|
00000230  08 01 00 00 00 00 00 00  05 00 00 00 07 00 00 00  |................|
00000240  08 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
00000250  09 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
00000260  00 00 00 00 00 00 00 00  98 03 00 00 00 00 00 00  |................|
00000270  4c 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |L...............|
00000280  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002a0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 01 00  |................|
000002b0  b0 00 40 00 00 00 00 00  00 00 00 00 00 00 00 00  |..@.............|
000002c0  00 00 00 00 03 00 02 00  d8 00 60 00 00 00 00 00  |..........`.....|
000002d0  00 00 00 00 00 00 00 00  01 00 00 00 04 00 f1 ff  |................|
000002e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002f0  11 00 00 00 00 00 02 00  d8 00 60 00 00 00 00 00  |..........`.....|
00000300  00 00 00 00 00 00 00 00  1d 00 00 00 00 00 f1 ff  |................|
00000310  0d 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000320  00 00 00 00 04 00 f1 ff  00 00 00 00 00 00 00 00  |................|
00000330  00 00 00 00 00 00 00 00  2d 00 00 00 10 00 01 00  |........-.......|
00000340  b0 00 40 00 00 00 00 00  00 00 00 00 00 00 00 00  |..@.............|
00000350  34 00 00 00 10 00 02 00  e5 00 60 00 00 00 00 00  |4.........`.....|
00000360  00 00 00 00 00 00 00 00  40 00 00 00 10 00 02 00  |........@.......|
00000370  e5 00 60 00 00 00 00 00  00 00 00 00 00 00 00 00  |..`.............|
00000380  47 00 00 00 10 00 02 00  e8 00 60 00 00 00 00 00  |G.........`.....|
00000390  00 00 00 00 00 00 00 00  00 68 65 6c 6c 6f 5f 77  |.........hello_w|
000003a0  6f 72 6c 64 2e 61 73 6d  00 68 65 6c 6c 6f 5f 77  |orld.asm.hello_w|
000003b0  6f 72 6c 64 00 68 65 6c  6c 6f 5f 77 6f 72 6c 64  |orld.hello_world|
000003c0  5f 6c 65 6e 00 5f 73 74  61 72 74 00 5f 5f 62 73  |_len._start.__bs|
000003d0  73 5f 73 74 61 72 74 00  5f 65 64 61 74 61 00 5f  |s_start._edata._|
000003e0  65 6e 64 00                                       |end.|
000003e4
```

# Global file structure

An ELF file contains the following parts:

- ELF header. Points to the position of the section header table and the program header table.
- Section header table (optional on executable). Each has `e_shnum` section headers, each pointing to the position of a section.
- N sections, with `N <= e_shnum` (optional on executable)
- Program header table (only on executable). Each has `e_phnum` program headers, each pointing to the position of a segment.
- N segments, with `N <= e_phnum` (only on executable)

The order of those parts is *not* fixed: the only fixed thing is the ELF header that must be the first thing on the file: Generic docs say:

> Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

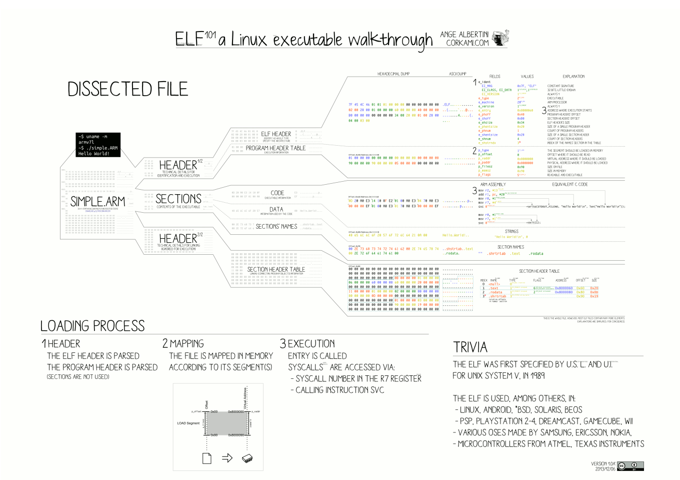
In pictures: sample object file with three sections:

```
            +-------------------+
            | ELF header        |---+
+---------> +-------------------+   | e_shoff
|           |                   |<--+
| Section   | Section header 0  |
|           |                   |---+ sh_offset
| Header    +-------------------+   |
|           | Section header 1  |---|--+ sh_offset
| Table     +-------------------+   |  |
|           | Section header 2  |---|--|--+
+---------> +-------------------+   |  |  |
            | Section 0         |<--+  |  |
            +-------------------+      |  | sh_offset
            | Section 1         |<-----+  |
            +-------------------+         |
            | Section 2         |<--------+
            +-------------------+
```

But nothing (except sanity) prevents the following topology:

```
            +-------------------+
            | ELF header        |---+ e_shoff
            +-------------------+   |
            | Section 1         |<--|--+
+---------> +-------------------+   |  |
|           |                   |<--+  | sh_offset
| Section   | Section header 0  |      |
|           |                   |------|---------+
| Header    +-------------------+      |         |
|           | Section header 1  |------+         |
| Table     +-------------------+                |
|           | Section header 2  |---+            | sh_offset
+---------> +-------------------+   | sh_offset  |
            | Section 2         |<--+            |
            +-------------------+                |
            | Section 0         |<---------------+
            +-------------------+
```

But some newbies may prefer PNGs :-)




(/elf101.png)

Image source (https://github.com/corkami/pics/blob/28cb0226093ed57b348723bc473cea0162dad366/binary/elf101/elf101.pdf).

# Section vs segment

We will get into more detail later, but it is good to have it in mind now:

- section: exists before linking, in object files.

  One ore more sections will be put inside a single segment by the linker.

  Major information sections contain for the linker:

  - is this section
    - raw data to be loaded into memory, e.g. `.data` , `.text` , etc.
    - or metadata about other sections, that will be used by the linker, but disappear at runtime e.g. `.symtab` , `.srttab` , `.rela.text`
- segment: exists after linking, in the executable file.

  Contains information about how each segment should be loaded into memory by the OS, notably location and permissions.

See also:

- http://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-segment-in-elf-file-format (http://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-segment-in-elf-file-format)
- http://stackoverflow.com/questions/23379880/difference-between-program-header-and-section-header-in-elf (http://stackoverflow.com/questions/23379880/difference-between-program-header-and-section-header-in-elf)

# ELF header

`readelf -h hello_world.o` :

```
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              REL (Relocatable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x0
Start of program headers:          0 (bytes into file)
Start of section headers:          64 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           0 (bytes)
Number of program headers:         0
Size of section headers:           64 (bytes)
Number of section headers:         7
Section header string table index: 3
```

`readelf -h hello_world.out` :

```
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x4000b0
Start of program headers:          64 (bytes into file)
Start of section headers:          272 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         2
Size of section headers:           64 (bytes)
Number of section headers:         6
Section header string table index: 3
```

Bytes in the object file:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 07 00 03 00  |....@.....@.....|
```

Executable:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  b0 00 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  10 01 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  02 00 40 00 06 00 03 00  |....@.8...@.....|
```

Structure represented:

```
# define EI_NIDENT 16

typedef struct {
    unsigned char   e_ident[EI_NIDENT];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
    Elf64_Half      e_phentsize;
    Elf64_Half      e_phnum;
    Elf64_Half      e_shentsize;
    Elf64_Half      e_shnum;
    Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

Manual breakdown:

- 0 0: `EI_MAG = 7f 45 4c 46 = 0x7f 'E', 'L', 'F'` : ELF magic number
- 0 4: `EI_CLASS = 02 = ELFCLASS64` : 64 bit elf
- 0 5: `EI_DATA = 01 = ELFDATA2LSB` : little endian data
- 0 6: `EI_VERSION = 01` : format version
- 0 7: `EI_OSABI` (only in 2003 Update) = `00 = ELFOSABI_NONE` : no extensions.
- 0 8: `EI_PAD` = 8x `00` : reserved bytes. Must be set to 0.
- 1 0: `e_type = 01 00` = 1 (big endian) = `ET_REl` : relocatable format

  On the executable it is `02 00` for `ET_EXEC` .
- 1 2: `e_machine = 3e 00 = 62 = EM_X86_64` : AMD64 architecture
- 1 4: `e_version = 01 00 00 00` : must be 1
- 1 8: `e_entry` = 8x `00` : execution address entry point, or 0 if not applicable like for the object file since there is no entry point.

  On the executable, it is `b0 00 40 00 00 00 00 00` . The kernel puts the RIP directly on that value when executing. It can be configured by the linker script or `-e` . But it will segfault if you set it too low: http://stackoverflow.com/questions/2187484/why-is-the-elf-execution-entry-point-virtual-address-of-the-form-0x80xxxxx-and-n (http://stackoverflow.com/questions/2187484/why-is-the-elf-execution-entry-point-virtual-address-of-the-form-0x80xxxxx-and-n)
- 2 0: `e_phoff` = 8x `00` : program header table offset, 0 if not present.

  `40 00 00 00` on the executable, i.e. it starts immediately after the ELF header.
- 2 8: `e_shoff = 40` 7x `00 = 0x40` : section header table file offset, 0 if not present.
- 3 0: `e_flags = 00 00 00 00` Arch specific. `i386` docs say:

  > The Intel386 architecture defines no flags; so this member contains zero.

- 3 4: `e_ehsize = 40 00` : size of this elf header. TODO why this field needed? Isn't the size fixed?
- 3 6: `e_phentsize = 00 00` : size of each program header, 0 if not present.

  `38 00` on executable: it is 56 bytes long
- 3 8: `e_phnum = 00 00` : number of program header entries, 0 if not present.

  `02 00` on executable: there are 2 entries.
- 3 A: `e_shentsize` and `e_shnum = 40 00 07 00` : section header size and number of entries
- 3 E: `e_shstrndx` ( `Section Header STRing iNDeX` ) = `03 00` : index of the `.shstrtab` section.

# Section header table

Array of `Elf64_Shdr` structs.

Each entry contains metadata about a given section.

`e_shoff` of the ELF header gives the starting position, 0x40 here.

`e_shentsize` and `e_shnum` from the ELF header say that we have 7 entries, each `0x40` bytes long.

So the table takes bytes from 0x40 to `0x40 + 7 + 0x40 - 1` = 0x1FF.

Some section names are reserved for certain section types: http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html#special_sections (http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html#special_sections) e.g. `.text` requires a `SHT_PROGBITS` type and `SHF_ALLOC` + `SHF_EXECINSTR`

`readelf -S hello_world.o` :

```
There are 7 section headers, starting at offset 0x40:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .data             PROGBITS         0000000000000000  00000200
       000000000000000d  0000000000000000  WA       0     0     4
  [ 2] .text             PROGBITS         0000000000000000  00000210
       0000000000000027  0000000000000000  AX       0     0     16
  [ 3] .shstrtab         STRTAB           0000000000000000  00000240
       0000000000000032  0000000000000000           0     0     1
  [ 4] .symtab           SYMTAB           0000000000000000  00000280
       00000000000000a8  0000000000000018           5     6     4
  [ 5] .strtab           STRTAB           0000000000000000  00000330
       0000000000000034  0000000000000000           0     0     1
  [ 6] .rela.text        RELA             0000000000000000  00000370
       0000000000000018  0000000000000018           4     2     4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

`struct` represented by each entry:

```
typedef struct {
    Elf64_Word  sh_name;
    Elf64_Word  sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr  sh_addr;
    Elf64_Off   sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word  sh_link;
    Elf64_Word  sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

# Sections

## Index 0 section

Contained in bytes 0x40 to 0x7F.

The first section is always magic: http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html (http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html) says:

> If the number of sections is greater than or equal to SHN_LORESERVE (0xff00), e_shnum has the value SHN_UNDEF (0) and the actual number of section header table entries is contained in the sh_size field of the section header at index 0 (otherwise, the sh_size member of the initial entry contains 0).

There are also other magic sections detailed in `Figure 4-7: Special Section Indexes` .

### SHT_NULL

In index 0, `SHT_NULL` is mandatory. Are there any other uses for it: http://stackoverflow.com/questions/26812142/what-is-the-use-of-the-sht-null-section-in-elf (http://stackoverflow.com/questions/26812142/what-is-the-use-of-the-sht-null-section-in-elf) ?

## .data section

`.data` is section 1:

```
00000080  01 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00  |................|
00000090  00 00 00 00 00 00 00 00  00 02 00 00 00 00 00 00  |................|
000000a0  0d 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000b0  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

- 80 0: `sh_name = 01 00 00 00` : index 1 in the `.shstrtab` string table

  Here, `1` says the name of this section starts at the first character of that section, and ends at the first NUL character, making up the string `.data` .

  `.data` is one of the section names which has a predefined meaning http://www.sco.com/developers/gabi/2003-12-17/ch4.strtab.html (http://www.sco.com/developers/gabi/2003-12-17/ch4.strtab.html)

  > These sections hold initialized data that contribute to the program's memory image.

- 80 4: `sh_type = 01 00 00 00` : `SHT_PROGBITS` : the section content is not specified by ELF, only by how the program interprets it. Normal since a `.data` section.

- 80 8: `sh_flags = 03` 7x `00` : `SHF_ALLOC` and `SHF_EXECINSTR` : http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html#sh_flags (http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html#sh_flags), as required from a `.data` section

- 90 0: `sh_addr` = 8x `00` : TODO: standard says:

  > If the section will appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

  but I don't understand it very well yet.

- 90 8: `sh_offset = 00 02 00 00 00 00 00 00 = 0x200` : number of bytes from the start of the program to the first byte in this section

- a0 0: `sh_size = 0d 00 00 00 00 00 00 00`

  If we take 0xD bytes starting at `sh_offset` 200, we see:

  ```
  00000200  48 65 6c 6c 6f 20 77 6f  72 6c 64 21 0a 00        |Hello world!..  |
  ```

  AHA! So our `"Hello world!"` string is in the data section like we told it to be on the NASM.

  Once we graduate from `hd` , we will look this up like:

  ```
  readelf -x .data hello_world.o
  ```

  which outputs:

  ```
  Hex dump of section '.data':
    0x00000000 48656c6c 6f20776f 726c6421 0a       Hello world!.
  ```

  NASM sets decent properties for that section because it treats `.data` magically: http://www.nasm.us/doc/nasmdoc7.html#section-7.9.2 (http://www.nasm.us/doc/nasmdoc7.html#section-7.9.2)

  Also note that this was a bad section choice: a good C compiler would put the string in `.rodata` instead, because it is read-only and it would allow for further OS optimizations.

- a0 8: `sh_link` and `sh_info` = 8x 0: do not apply to this section type. http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html#special_sections (http://www.sco.com/developers/gabi/2003-12-17/ch4.sheader.html#special_sections)

- b0 0: `sh_addralign = 04` = TODO: why is this alignment necessary? Is it only for `sh_addr` , or also for symbols inside `sh_addr` ?

- b0 8: `sh_entsize = 00` = the section does not contain a table. If != 0, it means that the section contains a table of fixed size entries. In this file, we see from the `readelf` output that this is the case for the `.symtab` and `.rela.text` sections.

## .text section

Now that we've done one section manually, let's graduate and use the `readelf -S` of the other sections.

```
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 2] .text             PROGBITS         0000000000000000  00000210
       0000000000000027  0000000000000000  AX       0     0     16
```

`.text` is executable but not writable: if we try to write to it Linux segfaults. Let's see if we really have some code there:

```
objdump -d hello_world.o
```

gives:

```
hello_world.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:       b8 01 00 00 00          mov    $0x1,%eax
   5:       bf 01 00 00 00          mov    $0x1,%edi
   a:       48 be 00 00 00 00 00    movabs $0x0,%rsi
  11:       00 00 00
  14:       ba 0d 00 00 00          mov    $0xd,%edx
  19:       0f 05                   syscall
  1b:       b8 3c 00 00 00          mov    $0x3c,%eax
  20:       bf 00 00 00 00          mov    $0x0,%edi
  25:       0f 05                   syscall
```

If we grep `b8 01 00 00` on the `hd`, we see that this only occurs at `00000210`, which is what the section says. And the Size is 27, which matches as well. So we must be talking about the right section.

This looks like the right code: a `write` followed by an `exit`.

The most interesting part is line `a` which does:

```
movabs $0x0,%rsi
```

to pass the address of the string to the system call. Currently, the `0x0` is just a placeholder. After linking happens, it will be modified to contain:

```
4000ba: 48 be d8 00 60 00 00    movabs $0x6000d8,%rsi
```

This modification is possible because of the data of the `.rela.text` section.

# SHT_STRTAB

Sections with `sh_type == SHT_STRTAB` are called *string tables*.

They hold a null separated array of strings.

Such sections are used by other sections when string names are to be used. The using section says:

- which string table they are using
- what is the index on the target string table where the string starts

So for example, we could have a string table containing:

```
Data: \0 a b c \0 d e f \0
Index: 0 1 2 3  4 5 6 7  8
```

The first byte must be a 0. TODO rationale?

And if another section wants to use the string `d e f`, they have to point to index `5` of this section (letter `d`).

Notable string table sections:

- `.shstrtab`
- `.strtab`

# .shstrtab

Section type: `sh_type == SHT_STRTAB`.

Common name: *section header string table*.

The section name `.shstrtab` is reserved. The standard says:

> This section holds section names.

This section gets pointed to by the `e_shstrnd` field of the ELF header itself.

String indexes of this section are are pointed to by the `sh_name` field of section headers, which denote strings.

This section does not have `SHF_ALLOC` marked, so it will not appear on the executing program.

```
readelf -x .shstrtab hello_world.o
```

Gives:

```
Hex dump of section '.shstrtab':
  0x00000000 002e6461 7461002e 74657874 002e7368 ..data..text..sh
  0x00000010 73747274 6162002e 73796d74 6162002e strtab..symtab..
  0x00000020 73747274 6162002e 72656c61 2e746578 strtab..rela.tex
  0x00000030 7400                                t.
```

The data in this section has a fixed format: http://www.sco.com/developers/gabi/2003-12-17/ch4.strtab.html (http://www.sco.com/developers/gabi/2003-12-17/ch4.strtab.html)

If we look at the names of other sections, we see that they all contain numbers, e.g. the `.text` section is number `7`.

Then each string ends when the first NUL character is found, e.g. character `12` is `\0` just after `.text\0`.

# .symtab

Section type: `sh_type == SHT_SYMTAB`.

Common name: *symbol table*.

First the we note that:

- `sh_link = 5`
- `sh_info = 6`

For `SHT_SYMTAB` sections, those numbers mean that:

- strings that give symbol names are in section 5, `.strtab`
- the relocation data is in section 6, `.rela.text`

A good high level tool to disassemble that section is:

```
nm hello_world.o
```

which gives:

```
0000000000000000 T _start
0000000000000000 d hello_world
000000000000000d a hello_world_len
```

This is however a high level view that omits some types of symbols and in which the symbol types . A more detailed disassembly can be obtained with:

```
readelf -s hello_world.o
```

which gives:

```
Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello_world.asm
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     4: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    1 hello_world
     5: 000000000000000d     0 NOTYPE  LOCAL  DEFAULT  ABS hello_world_len
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    2 _start
```

The binary format of the table is documented at http://www.sco.com/developers/gabi/2003-12-17/ch4.symtab.html (http://www.sco.com/developers/gabi/2003-12-17/ch4.symtab.html)

The data is:

```
readelf -x .symtab hello_world.o
```

Which gives:

```
Hex dump of section '.symtab':
  0x00000000 00000000 00000000 00000000 00000000 ................
  0x00000010 00000000 00000000 01000000 0400f1ff ................
  0x00000020 00000000 00000000 00000000 00000000 ................
  0x00000030 00000000 03000100 00000000 00000000 ................
  0x00000040 00000000 00000000 00000000 03000200 ................
  0x00000050 00000000 00000000 00000000 00000000 ................
  0x00000060 11000000 00000100 00000000 00000000 ................
  0x00000070 00000000 00000000 1d000000 0000f1ff ................
  0x00000080 0d000000 00000000 00000000 00000000 ................
  0x00000090 2d000000 10000200 00000000 00000000 -...............
  0x000000a0 00000000 00000000                   ........
```

The entries are of type:

```
typedef struct {
    Elf64_Word  st_name;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf64_Half  st_shndx;
    Elf64_Addr  st_value;
    Elf64_Xword st_size;
} Elf64_Sym;
```

Like in the section table, the first entry is magical and set to a fixed meaningless values.

## STT_FILE

Entry 1 has `ELF64_R_TYPE == STT_FILE`. `ELF64_R_TYPE` is continued inside of `st_info`.

Byte analysis:

- 10 8: `st_name = 01000000` = character 1 in the `.strtab`, which until the following `\0` makes `hello_world.asm`

  This piece of information file may be used by the linker to decide on which segment sections go: e.g. in `ld` linker script we write:

  ```
  segment_name :
  {
      file(section)
  }
  ```

  to pick a section from a given file.

  Most of the time however, we will just dump all sections with a given name together with:

  ```
  segment_name :
  {
      *(section)
  }
  ```

- 10 12: `st_info = 04`

  Bits 0-3 = `ELF64_R_TYPE` = Type = `4 = STT_FILE`: the main purpose of this entry is to use `st_name` to indicate the name of the file which generated this object file.

  Bits 4-7 = `ELF64_ST_BIND` = Binding = `0 = STB_LOCAL`. Required value for `STT_FILE`.

- 10 13: `st_shndx` = Symbol Table Section header Index = `f1ff = SHN_ABS`. Required for `STT_FILE`.
- 20 0: `st_value` = 8x `00`: required for value for `STT_FILE`
- 20 8: `st_size` = 8x `00`: no allocated size

Now from the `readelf`, we interpret the others quickly.

## STT_SECTION

There are two such entries, one pointing to `.data` and the other to `.text` (section indexes `1` and `2`).

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
  3: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
```

TODO what is their purpose?

## STT_NOTYPE

Then come the most important symbols:

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  4: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    1 hello_world
  5: 000000000000000d     0 NOTYPE  LOCAL  DEFAULT  ABS hello_world_len
  6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    2 _start
```

`hello_world` string is in the `.data` section (index 1). It's value is 0: it points to the first byte of that section.

`_start` is marked with `GLOBAL` visibility since we wrote:

```
global _start
```

in NASM. This is necessary since it must be seen as the entry point. Unlike in C, by default NASM labels are local.

#### SHN_ABS

`hello_world_len` points to the special `st_shndx == SHN_ABS == 0xF1FF`.

`0xF1FF` is chosen so as to not conflict with other sections.

`st_value == 0xD == 13` which is the value we have stored there on the assembly: the length of the string `Hello World!`.

This means that relocation will not affect this value: it is a constant.

This is small optimization that our assembler does for us and which has ELF support.

If we had used the address of `hello_world_len` anywhere, the assembler would not have been able to mark it as `SHN_ABS`, and the linker would have extra relocation work on it later.

### SHT_SYMTAB on the executable

By default, NASM places a `.symtab` on the executable as well.

This is only used for debugging. Without the symbols, we are completely blind, and must reverse engineer everything.

You can strip it with `objcopy`, and the executable will still run. Such executables are called *stripped executables*.

## .strtab

Holds strings for the symbol table.

This section has `sh_type == SHT_STRTAB`.

It is pointed to by `sh_link == 5` of the `.symtab` section.

```
readelf -x .strtab hello_world.o
```

Gives:

```
Hex dump of section '.strtab':
  0x00000000 0068656c 6c6f5f77 6f726c64 2e61736d .hello_world.asm
  0x00000010 0068656c 6c6f5f77 6f726c64 0068656c .hello_world.hel
  0x00000020 6c6f5f77 6f726c64 5f6c656e 005f7374 lo_world_len._st
  0x00000030 61727400                            art.
```

This implies that it is an ELF level limitation that global variables cannot contain NUL characters.

## .rela.text

Section type: `sh_type == SHT_RELA`.

Common name: *relocation section*.

`.rela.text` holds relocation data which says how the address should be modified when the final executable is linked. This points to bytes of the text area that must be modified when linking happens to point to the correct memory locations.

Basically, it translates the object text containing the placeholder 0x0 address:

```
   a:       48 be 00 00 00 00 00    movabs $0x0,%rsi
  11:       00 00 00
```

to the actual executable code containing the final 0x6000d8:

```
4000ba: 48 be d8 00 60 00 00    movabs $0x6000d8,%rsi
4000c1: 00 00 00
```

It was pointed to by `sh_info = 6` of the `.symtab` section.

`readelf -r hello_world.o` gives:

```
Relocation section '.rela.text' at offset 0x3b0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000000c  000200000001 R_X86_64_64       0000000000000000 .data + 0
```

The section does not exist in the executable.

The actual bytes are:

```
00000370  0c 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00  |................|
00000380  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

The `struct` represented is:

```
typedef struct {
    Elf64_Addr  r_offset;
    Elf64_Xword r_info;
    Elf64_Sxword    r_addend;
} Elf64_Rela;
```

So:

- 370 0: `r_offset` = 0xC: address into the `.text` whose address this relocation will modify
- 370 8: `r_info` = 0x200000001. Contains 2 fields:
    - `ELF64_R_TYPE` = 0x1: meaning depends on the exact architecture.
    - `ELF64_R_SYM` = 0x2: index of the section to which the address points, so `.data` which is at index 2.

  The AMD64 ABI says that type `1` is called `R_X86_64_64` and that it represents the operation `S + A` where:

    - `S` : the value of the symbol on the object file, here `0` because we point to the `00 00 00 00 00 00 00 00` of `movabs $0x0,%rsi`
    - `A` : the addend, present in field `r_added`

  This address is added to the section on which the relocation operates.

  This relocation operation acts on a total 8 bytes.

- 380 0: `r_addend` = 0

So in our example we conclude that the new address will be: `S + A = .data + 0` , and thus the first thing in the data section.

## .rel.text

Besides `sh_type == SHT_RELA` , there also exists `SHT_REL` , which would have section name `.text.rel` (not present in this object file).

Those represent the same `struct` , but without the addend, e.g.:

```
typedef struct {
    Elf64_Addr  r_offset;
    Elf64_Xword r_info;
} Elf64_Rela;
```

The ELF standard says that in many cases the both can be used, and it is just a matter of convenience.

# Program header table

Only appears in the executable.

Contains information of how the executable should be put into the process virtual memory.

The executable is generated from object files by the linker. The main jobs that the linker does are:

- determine which sections of the object files will go into which segments of the executable.

  In Binutils, this comes down to parsing a linker script, and dealing with a bunch of defaults.

  You can get the linker script used with `ld --verbose` , and set a custom one with `ld -T` .

- do relocation according to the `.rela.text` section. This depends on how the multiple sections are put into memory.

`readelf -l hello_world.out` gives:

```
Elf file type is EXEC (Executable file)
Entry point 0x4000b0
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000d7 0x00000000000000d7  R E    200000
  LOAD           0x00000000000000d8 0x00000000006000d8 0x00000000006000d8
                 0x000000000000000d 0x000000000000000d  RW     200000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data
```

On the ELF header, `e_phoff` , `e_phnum` and `e_phentsize` told us that there are 2 program headers, which start at `0x40` and are `0x38` bytes long each, so they are:

```
00000040  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  d7 00 00 00 00 00 00 00  d7 00 00 00 00 00 00 00  |................|
00000070  00 00 20 00 00 00 00 00                           |.. .....        |
```

and:

```
00000070                           01 00 00 00 06 00 00 00  |        ........|
00000080  d8 00 00 00 00 00 00 00  d8 00 60 00 00 00 00 00  |..........`.....|
00000090  d8 00 60 00 00 00 00 00  0d 00 00 00 00 00 00 00  |..`.............|
000000a0  0d 00 00 00 00 00 00 00  00 00 20 00 00 00 00 00  |.......... .....|
```

Structure represented http://www.sco.com/developers/gabi/2003-12-17/ch5.pheader.html (http://www.sco.com/developers/gabi/2003-12-17/ch5.pheader.html):

```
typedef struct {
    Elf64_Word  p_type;
    Elf64_Word  p_flags;
    Elf64_Off   p_offset;
    Elf64_Addr  p_vaddr;
    Elf64_Addr  p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
} Elf64_Phdr;
```

Breakdown of the first one:

- 40 0: `p_type = 01 00 00 00 = PT_LOAD` : this is a regular segment that will get loaded in memory.
- 40 4: `p_flags = 05 00 00 00` = execute and read permissions. No write: we cannot modify the text segment. A classic way to do this in C is with string literals: http://stackoverflow.com/a/30662565/895245 (http://stackoverflow.com/a/30662565/895245) This allows kernels to do certain optimizations, like sharing the segment amongst processes.
- 40 8: `p_offset` = 8x `00` TODO: what is this? Standard says:

  > This member gives the offset from the beginning of the file at which the first byte of the segment resides.

  But it looks like offsets from the beginning of *segments*, not file?
- 50 0: `p_vaddr = 00 00 40 00 00 00 00 00` : initial virtual memory address to load this segment to
- 50 8: `p_paddr = 00 00 40 00 00 00 00 00` : unspecified effect. Intended for systems in which physical addressing matters. TODO example?
- 60 0: `p_filesz = d7 00 00 00 00 00 00 00` : size that the segment occupies in memory. If smaller than `p_memsz` , the OS fills it with zeroes to fit when loading the program. This is how BSS data is implemented to save space on executable files. i368 ABI says on `PT_LOAD` :

  > The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the ''extra'' bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size.

- 60 8: `p_memsz = d7 00 00 00 00 00 00 00` : size that the segment occupies in memory
- 70 0: `p_align = 00 00 20 00 00 00 00 00` : 0 or 1 mean no alignment required. TODO why is this required? Why not just use `p_addr` directly, and get that right? Docs also say:

  > p_vaddr should equal p_offset, modulo p_align

The second segment ( `.data` ) is analogous. TODO: why use offset `0x0000d8` and address `0x00000000006000d8` ? Why not just use `0` and `0x00000000006000d8` ?

Then the:

```
Section to Segment mapping:
```

section of the `readelf` tells us that:

- 0 is the `.text` segment. Aha, so this is why it is executable, and not writable
- 1 is the `.data` segment.

TODO where does this information come from? http://stackoverflow.com/questions/23018496/section-to-segment-mapping-in-elf-files (http://stackoverflow.com/questions/23018496/section-to-segment-mapping-in-elf-files)

# Backlinks

- 2017-05: https://news.ycombinator.com/item?id=14359159 (https://news.ycombinator.com/item?id=14359159) https://web.archive.org/web/20170517174951/https://news.ycombinator.com/news (https://web.archive.org/web/20170517174951/https://news.ycombinator.com/news)

# Comments

**2 Comments** **Ciro Santilli's Home Page**

1 Login

Recommend 7 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

**rax** • 2 months ago

Bug: Elf header -> Manual breakdown
0 5: EI_DATA = 01 = ELFDATA2LSB: big endian data

should be little endian data

• Reply • Share ›

**Ciro Santilli** Mod rax • 2 months ago

Thanks, fixed: https://github.com/cirosant...

• Reply • Share ›

ALSO ON **CIRO SANTILLI'S HOME PAGE**

**Professional Interests - Ciro Santilli**

2 comments • 3 years ago•

best college essay help — Many people will be going to admire your likes for your life which might also help them to get inspired by your work and …

**Skills - Ciro Santilli**

20 comments • 3 years ago•

Jason Young — Hi Man, we have so much in common! I am a chinese and I hope we can learn from each other. And also "je parle a petit peu …

**Markdown Style Guide - Ciro Santilli**

2 comments • 3 years ago•

Ciro Santilli — I'm not sure I understand what you mean by references: do you mean "proof that this is popular"? If so: 1) This is currently not a popular …

**x86 Paging Tutorial - Ciro Santilli**

8 comments • 3 years ago•

vp — The bottom half on the "64-bit architectures" chaptershouldn't have been----------------- 00007FFF FFFFFFFFBottom half----------------- 00000000 …