Nano-Kernel : A Bare Metal OS

## Part 11 - Reading ELF Files

Loading and running new tasks is a key function of any operating system. The Executable and Linker Format (ELF) file has become the standard for most operating systems, including Linux, OSX, and BSD variants. When a compiler creates the executable file it writes out the binary encoded machine language instructions and any compiler generated data to a file. Every executable format does that too. The ELF file also includes information about the executable, the expected organization of memory, and where the first instruction is in the loaded image.

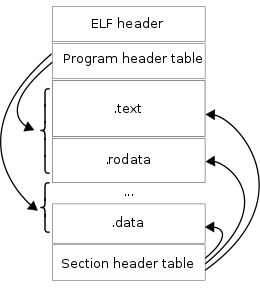


Figure 2 - ELF File Structure from  
<https://en.wikipedia.org/wiki/Executable_and_Linkable_Format>

The ELF file starts with a header that identifies information about the executable, such as what CPU / instruction set it is for (e.g. Intel 32-bit or AMD 64-bit), and whether it is little- or big-endian (e.g. MIPS and ARM can be either).

After that header there is a program header table that contains a list of section types and sizes. As the diagram in Figure 2 shows, the table includes pointers to each of the various sections in the executable file itself. The table also include where these sections are to start in memory.

Lastly, the ELF file includes information about dynamic link libraries that this executable depends on and symbolic debugging information that can be used by tools like GDB.

## Reading ELF Files

The best strategy here is to create an “elf.c” file outside your kernel so that you can test and debug this part separately from the Nano Kernel. However, it is also important to “test what you fly, and fly what you test.” So, I also created an “elf-test.c” file that included the junk that won’t be in the kernel; and then I added some preamble to the “elf.c” that selected the right header files and defined debugging “printfs” that I could turn-on or off, and direct to either the actual “printf” or the “kprintf” as needed. I’ve included this in Figure 3.

The ELF has a bunch of structures and enumerations. Normally we have a single “.h” file for a “.c” that exposes the parts of the “.c” file that should be used by the rest of the program. In this case, having all of these ELF-stuff in the “.c” really jumbled things up. Instead, we can create another header file to move all of this cruft out of the “.c”, but not pollute the “.h” that will be used by the rest of the kernel. I called mine “elf-struct.h.”

Figure 3 – Header file for “elf.c” – can be built as part of NanoKernel or used in a stand-alone test.

#include <stdint.h>

#include <string.h>

#ifdef DEBUG

#include <stdio.h>

#include <stdlib.h>

#define DBPRINT(...) do{ fprintf( stderr, \_\_VA\_ARGS\_\_ ); } while( 0 )

#define MEMCPY memcpy

#else

#include "../kstdlib.h"

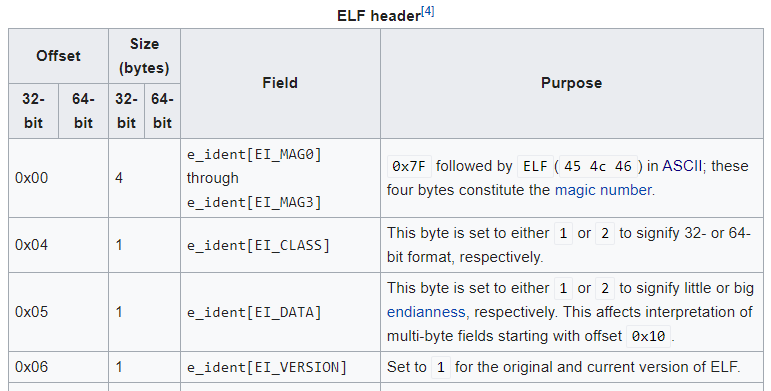
#define DBPRINT(...) do{ } while ( 0 )

#define MEMCPY kmemcpy

#endif

### Build the ELF Structures

Take a look at Wikipedia’s entry on the ELF file at <https://tinyurl.com/3p3ltom>, especially starting with the ELF header. The Wikipedia entry shows what each of the fields are, an excerpt is included here. This provides all of the information necessary to create a C-struct:



struct elf\_header {  
 uint8\_t elf\_magic[4];  
 uint8\_t elf\_class;  
 uint8\_t elf\_data;  
 uint8\_t elf\_version;  
 …  
}

Later, we can either *fread* or *memcpy* from the ELF file into the struct. As we saw with the Multiboot header, so long as the bytes line up, the information stored in the ELF file will be automagically available in the C struct.

In addition to the ELF file structures, many of the fields have a range of valid values. For example, the ELF class, data, and version use the value 1 or 2 to indicate one or another number. These are most definitely *magic numbers*, and should be eliminated. The proper way to do this is through *enumerations*. In addition to the C-structs for the tables, its also good practice to create enums for each of the values:

enum elf\_class {

ELF\_BIT32 = 1, ELF\_BIT64 = 2

};

### Step 1 – Create Structs and Enums for the ELF File Parser

* Structs for:
  + ELF header,
  + program header, and
  + section header
* Enums for:
  + ELF class
  + ELF endianness
  + ELF ABI
  + ELF Type
  + ELF ISA
  + ELF program table type
  + ELF alignment
  + ELF section type
  + ELF section flags

A good test is to use the “sizeof” operator to make sure that your three structs’ sizes match the Wiki page, e.g.

int main() { assert(sizeof(elf\_header\_t) == 52); }

### Step 2 – Create a Test Program

Create a C language test program, like “elf-test.c” that will read an ELF file from Linux and display various fields. Since we *know* that the OS isn’t going to read files from the file system, we need to make an ELF parser that can take a pointer to a block of memory.

Your test program can either: open the file, get the size, malloc a chunk of memory, and read the bytes of the file into that chunk, or it can be better than that and use the “mmap()” function.

Now that we’ve just finished the paging section, you can better understand how mmap() works. Instead of connecting virtual pages of memory to physical pages of memory, the OS maps virtual pages of memory on the promise that they will be there. When you actually try to read from the virtual memory, the OS will actually read the data on demand. This is much more efficient than malloc’ing the whole file.

Here is some sample code:

int fid = open("simple", O\_RDONLY);

assert(fid > 0);

struct stat statbuf;

int rc = fstat(fid, &statbuf);

assert(rc >= 0);

void \*ptr = mmap(NULL, statbuf.st\_size, PROT\_READ, MAP\_PRIVATE, fid, 0);

assert(ptr != NULL);

elf\_header\_t \*header = (elf\_header\_t \*)ptr;

if ((header->magic[0] == 0x7F) && (header->magic[1] == ‘E’) …)   
 DBPRINTF(“ELF File is Found\n”);

Note how the mmap() function locks the contents of the file into virtual memory, and from then on, our program only need to play with pointers.

### Step 3 – Validate the ELF File Header

Write a function that, given a pointer to an ELF file will return “1” if the ELF file meets the criteria for the NanoKernel, and “0” if it doesn’t. The criteria are:

* Magic must match
* 32-bit
* little endian
* version 1
* System V ABI
* Executable file
* X86 machine type
* Executable version 1
* Executable flags = 0
* Ident. Header = 52 bytes
* Program header size matches your struct size
* Section header size matches your struct size

### Step 4 – Read the Program Headers

In this step, given a program header, the goal is to load the program header into memory. This can be a bit tricky, so I’m providing my code for this function. Lets dissect whats going on.

static int handle\_program\_header(

const char \*elf\_start,

const elf\_program\_header\_t \*pheader,

void \*(\*virt\_alloc)(uint32\_t virt\_start, uint32\_t length))

{

if (pheader->p\_type == PT\_LOAD) {

void \*ptr = virt\_alloc(pheader->p\_vaddr, pheader->p\_memsz);

MEMCPY(ptr, elf\_start + pheader->p\_offset, pheader->p\_filesz);

}

return 0;

}

The function arguments include:

* “const char \*elf\_start” – a pointer to the memory location of the entire ELF file, i.e. elf\_start[0] == 0x7F, elf\_start[1] = ‘E’, …
* “const elf\_program\_header\_t \*” – is a pointer to the current program header. If you read through the Wikipedia page, you’ll find that the program header includes the *virtual address,* which is the address that this program expects this program section to be given.
* “void \*(\*virt\_alloc)(uint32\_t virt\_start, uint32\_t length)” – is a pointer to a function that returns an allocated virtual address. The allocation will need to allocate starting at virt\_start, and allocating for length bytes.

The body of the function checks to see if this program header is “loadable.” If it is, I will allocate the block of memory and then memcpy the ELF file contents into it.

The function pointer that actually allocates the memory will need to be written, but its not hard to imagine that I will eventually need to call “

void page\_set\_map(page\_directory\_t \*directory, uint32\_t virtual, uint32\_t physical)

Finally, all that remains is to loop over the program headers contained in the program headers table:

// load headers

elf\_program\_header\_t \*pheaders =

(elf\_program\_header\_t \*) (elf\_start + header->header\_offset);

int i;

for (i = 0; i < header->program\_header\_num; i++) {

handle\_program\_header(elf\_start, &pheaders[i], virt\_alloc);

}

### Step 5 – Create an ELF File to Load

This is probably one of the easiest steps in the entire project! Our goal is to create an “idle” task. Lets be clear – this is not the typical strategy for adding an idle loop, but this is the simplest ELF file that can be imagined, so it’s a great place to start.

Write the following C file:

void \_start(void) {

while(1) asm("nop;");

}

Seriously, that’s it. Modify your Makefile to build the “idle.c”, but it needs some special flags:

NOOSCFLAGS := -ffreestanding -O0 -Wall -Wextra -nostdlib -nostartfiles -nodefaultlibs -Wl,--build-id=none -ggdb

NOOSCPPFLAGS:=$(CFLAGS)

idle: idle.c

i686-elf-gcc $(NOSSCFLAGS) -o idle.bin idle.c

### Step 6 – Include the “idle” ELF into Your Kernel Image

Create a new Assembler file: “builtin.S”, that includes the following lines:

# built-in executables to the kernel

# these run in a different, virtual, address space from the kernel

.section .builtin

.global \_idletask\_start

.incbin "idle.bin"

Unpacking this, we:

* Create a new linker section calleld “.builtin”
* Create a new global symbol called “\_idletask\_start”
* Include the binary file “idle.bin” in the executable image.

Yes, if you’re wondering, we are embedding one ELF in another ELF.

Finally, back in the “linker.ld” file, modify the .rodata section to include anything marked as belonging to the “.builtin” section.

.rodata BLOCK(4K) : ALIGN(4K)

{

\*(.rodata)

/\* built-in executables \*/

**\*(.builtin\*)**

}

Note: this same technique could be used to include logos and graphics in your kernel which can be displayed at boot.

### Step 7 – Create a Task from the ELF file

Use the task creation code from parts 9 and 10 to wrap a task around the virtual memory allocated for the ELF file, and run the program. Of course, you cannot just run any program – it has to be compatible with the system calls you made in part 10.

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. I want to see your code
2. Demonstrate how your OS runs the tasks that are built in.

Points: \_\_\_\_\_\_\_\_\_ / 40