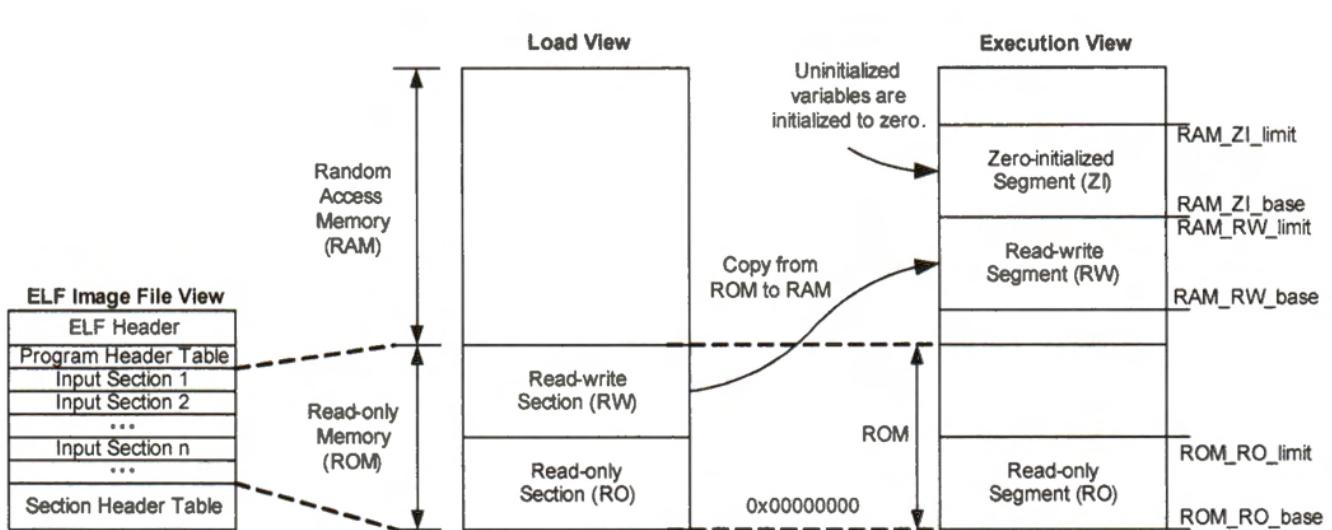


# Documentation- Chapter 1- Embedded Systems with ARM Cortex-M Microcontroller by Yifen Zhu

I have documented my learning as I often do in obsidian( obsidian is an application similar to Microsoft word). The below document **summarizes my understanding** of what I understand about ELF files, and how these elf files are loaded into the microcontroller. I have also tested practically verified the the disassembly of a sample code I had created by using **objdump** package.

It starts with the discussion of Executable file format (ELF) and how and where it is stored in the microcontroller

In ELF, similar data, symbols, and other information are grouped into many important input sections. The executable interface provides two separate logic views: the load view and the execution view, as shown in Figure 1-2.



**Figure 1-2. Interface of an executable binary file in the executable and linking format (ELF).**  
An ELF file provides *load view* and *execution view*. The load view specifies how to load data into the memory. The execution view instructs how to initialize data regions at runtime.

Some common questions that arise-

1. where exactly is the ELF stored
2. What do you mean by the executable interface
3. What do you mean by the load view and the execution view

According to google, executable interface means- An Executable interface defines a contract for a block of code that can be executed.

I found a good website explaining the same-

[How is a binary executable organized? Let's explore it!](#)

The writer of the website, used readELF to view the executable file in a readable format, and also viewing it in a readable format helps us actually see the different sections of the executable

But since windows only has a .exe file format, I couldn't run readELF by myself

But a simple code was executed by the author,

```
#include <stdio.h>

int main() {
    printf("Penguin!\n");
}
```

And upon running the readELF we could see-

```
gistfile1.txt
```

```
1
2 Symbol table '.dynsym' contains 4 entries:
3   Num: Value      Size Type  Bind  Vis    Ndx Name
4     0: 0000000000000000      0 NOTYPE LOCAL  DEFAULT  UND
5     1: 0000000000000000      0 FUNC   GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
6     2: 0000000000000000      0 FUNC   GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
7     3: 0000000000000000      0 NOTYPE WEAK   DEFAULT  UND __gmon_start__
8
9 Symbol table '.symtab' contains 64 entries:
10  Num: Value      Size Type  Bind  Vis    Ndx Name
11    0: 0000000000000000      0 NOTYPE LOCAL  DEFAULT  UND
12    1: 000000000400238     0 SECTION LOCAL  DEFAULT  1
13    2: 000000000400254     0 SECTION LOCAL  DEFAULT  2
14    3: 000000000400274     0 SECTION LOCAL  DEFAULT  3
15    4: 000000000400298     0 SECTION LOCAL  DEFAULT  4
16    5: 0000000004002b8     0 SECTION LOCAL  DEFAULT  5
17    6: 000000000400318     0 SECTION LOCAL  DEFAULT  6
18    7: 000000000400356     0 SECTION LOCAL  DEFAULT  7
19    8: 000000000400360     0 SECTION LOCAL  DEFAULT  8
20    9: 000000000400380     0 SECTION LOCAL  DEFAULT  9
21   10: 000000000400398     0 SECTION LOCAL  DEFAULT  10
22   11: 0000000004003c8     0 SECTION LOCAL  DEFAULT  11
23   12: 0000000004003e0     0 SECTION LOCAL  DEFAULT  12
24   13: 000000000400410     0 SECTION LOCAL  DEFAULT  13
25   14: 0000000004005e8     0 SECTION LOCAL  DEFAULT  14
26   15: 0000000004005f8     0 SECTION LOCAL  DEFAULT  15
27   16: 000000000400604     0 SECTION LOCAL  DEFAULT  16
28   17: 000000000400630     0 SECTION LOCAL  DEFAULT  17
29   18: 000000000600e28     0 SECTION LOCAL  DEFAULT  18
30   19: 000000000600e38     0 SECTION LOCAL  DEFAULT  19
31   20: 000000000600e48     0 SECTION LOCAL  DEFAULT  20
32   21: 000000000600e50     0 SECTION LOCAL  DEFAULT  21
```

I had a similar executable file for a C program increment.c stored in my PC which is a simple program that increments a number

```
//program to increment a number using call by reference
//23bec0384
#include <stdio.h>
void increment(int *num);
int main() {
    int value = 5;
    printf("Before increment: value = %d\n", value);
    increment(&value);
}
PS C:\Users\Tushar\Desktop\Code_test\C> objdump -t increment.exe
increment.exe:      file format pei-i386

SYMBOL TABLE:
[ 0](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 0) 0x000002a0 __mingw32_init_mainargs
[ 1](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 0) 0x000002e0 __mainCRTStartup
[ 2](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 0) 0x00000300 __WinMainCRTStartup
[ 3](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 0) 0x00000320 __atexit
[ 4](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 0) 0x00000330 __onexit
[ 5](sec 1)(fl 0x00)(ty 0)(scl 3) (nx 1) 0x00000000 .text
AUX scnlen 0x336 nreloc 42 nlno 0
[ 7](sec 5)(fl 0x00)(ty 0)(scl 3) (nx 1) 0x00000000 .bss
AUX scnlen 0x8 nreloc 0 nlno 0
[ 9](sec -2)(fl 0x00)(ty 0)(scl 103) (nx 1) 0x0000001e cygming-crtbegin.c
File
[ 11](sec 5)(fl 0x00)(ty 0)(scl 3) (nx 0) 0x00000008 __obj
[ 12](sec 2)(fl 0x00)(ty 0)(scl 3) (nx 0) 0x00000000 __deregister_frame_fn
[ 13](sec 2)(fl 0x00)(ty 0)(scl 3) (nx 0) 0x00000018 __JCR_LIST__
[ 14](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 1) 0x00000340 __gcc_register_frame
AUX tagndx 0 ttlsiz 0x0 lnnos 0 next 0
[ 16](sec 1)(fl 0x00)(ty 20)(scl 2) (nx 0) 0x00000430 __gcc_deregister_frame
[ 17](sec 1)(fl 0x00)(ty 0)(scl 3) (nx 1) 0x00000340 .text
```

the ELF file consists organized in the form of symbols, section and segments.

But **symbols** are usually the name of the functions in the program/global variables

And sections are basically a sup-part of the ELF file that gets loaded into the RAM for execution, and apparently .text and .data sections are the main parts of the ELF file that get loaded into RAM. Source: [Unix SE](#) And segments are grouping of one or more sections

And upon viewing the header of the sections using ObjDump we get to see and sort of verify that. At least we get to see the presence of .text and .data sections

```

increment.exe:      file format pei-i386

Sections:
Idx Name      Size    VMA      LMA      File off  Align
 0 .text     000002be4 00401000 00401000 000004000 2**4
              CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
 1 .data     00000001c 004040000 004040000 00003000 2**2
              CONTENTS, ALLOC, LOAD, DATA
 2 .rdata    000000030c 004050000 004050000 00003200 2**2
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .eh_frame 000000094 004060000 004060000 00003600 2**2
              CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .bss      000000070 004070000 004070000 00000000 2**2
              ALLOC
 5 .idata     00000005bc 004080000 004080000 00004000 2**2
              CONTENTS, ALLOC, LOAD, DATA
 6 .CRT       000000018 004090000 004090000 00004600 2**2
              CONTENTS, ALLOC, LOAD, DATA
 7 .tls      0000000020 0040a0000 0040a0000 00004800 2**2
              CONTENTS, ALLOC, LOAD, DATA
 8 .debug_aranges 000000038 0040b0000 0040b0000 00004a00 2**3
              CONTENTS, READONLY, DEBUGGING
 9 .debug_info 000001cff 0040c0000 0040c0000 00004c00 2**0
              CONTENTS, READONLY, DEBUGGING
10 .debug_abbrev 00000012f 0040e0000 0040e0000 00006a00 2**0
              CONTENTS, READONLY, DEBUGGING
11 .debug_line 0000001c8 0040f0000 0040f0000 00006c00 2**0
              CONTENTS, READONLY, DEBUGGING
12 .debug_frame 000000038 004100000 004100000 00006e00 2**2
              CONTENTS, READONLY, DEBUGGING

```

ELF executables - and the resulting processes - on a UNIX/Linux system have a uniform or similar memory layout. The *link editor* (a.k.a *static linker*), `ld(1)`, ensures that an executable's loadable code segment **always starts at a certain virtual address**. The value is architecture dependent. For x86, `0x08048000` (for 32-bit address spaces) and `0x400000` (for 64-bit address spaces).

## Difference between section and segment

Source: [exe - Object Files/Executables: What's the difference between a segment and a section? - Stack Overflow](#)

Segments contain information needed at runtime, while the sections contain information needed during linking.

A segment can contain 0 or more sections.

[elf\(5\) - Linux manual page](#)

The man page regarding the `elf.h` header file offers a better explanation

The header file `<elf.h>` defines the format of ELF executable binary files. Amongst these files are normal executable files, relocatable object files, core files, and shared objects.

They also describe the structure **Elfn\_Ehdr** which describes the different characteristics of a file the e\_ident array describes the file in a detailed format, containing information such as architecture for the binary(EI\_CLASS), to how is the data encoded in the binary(EI\_DATA), to which operating system the binary is targeted towards(EI\_OSABI)

This **leads me to infer** that the ELF header file contains information regarding how an executable can be created from a set of object files and how they must be "loaded" into the memory regardless of the machine the file was created in.

Basically ELF file serves as a sort of blueprint for different machines to actually load a program into memory

## Section header(Shdr)

```
typedef struct {
    uint32_t sh_name;
    uint32_t sh_type;
    uint32_t sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    uint32_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint32_t sh_addralign;
    uint32_t sh_entsize;
} Elf32_Shdr;
```

```
typedef struct {
    uint32_t    sh_name;
    uint32_t    sh_type;
    uint64_t    sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off   sh_offset;
    uint64_t    sh_size;
    uint32_t    sh_link;
    uint32_t    sh_info;
    uint64_t    sh_addralign;
    uint64_t    sh_entsize;
} Elf64_Shdr;
```

The ElfN\_Shdr struct gives an information about each section. One or more sections can be there in a segment.

A brief description about each member of the `ElfN_Shdr` is as follows

1. Sh\_name - gives the name of the section
2. Sh\_Type - gives the type of information held by the section, it could be a string table ( SHT\_STRTAB ), it could be a hash table( SHT\_HASH ), it could also contain a symbol table, SHT\_SYMTB , OR SHT\_PROGBITS - that particular section is sort of defined by the person who programmed it, like for ex. .bss contains uninitialized variables
3. Sh\_Flags - Contains flag which, if enabled helps us do certain operations on the section such as modifying it while execution of process( SHF\_WRITE ) and so on. Basically if the Sh\_Flag member of the struct is equal to the value(SHT\_Write) then, we can modify that section during runtime
4. Sh\_Addr - If the section shows up in the memory image of the process, then this contains the address of the first byte of the section
5. Sh\_offset - this tells us where is the first byte of the section is wrt to the beginning of the binary elf file(.o, .so, or .out file)
6. Sh\_link - not sure what this does
7. Sh\_info -
8. Sh\_AddrAlign - If 0
9. Sh\_entrsize

## Program Header

[Introduction to ELF Program Headers](#)

# ELF Program Header Structure (32-bit)

```
typedef struct {
    Elf32_Word p_type;           /* Segment type */          Size: 32 bytes (0x20)
    Elf32_Off  p_offset;         /* Segment file offset */   Byte Offset: 0 (0x0)
    Elf32_Addr p_vaddr;          /* Segment virtual address */ Byte Offset: 4 (0x4)
    Elf32_Addr p_paddr;          /* Segment physical address */ Byte Offset: 8 (0x8)
    Elf32_Word  p_filesz;        /* Segment size in file */  Byte Offset: 12 (0xC)
    Elf32_Word  p_memsz;         /* Segment size in memory */ Byte Offset: 16 (0x10)
    Elf32_Word  p_flags;          /* Segment flags */        Byte Offset: 20 (0x14)
    Elf32_Word  p_align;          /* Segment alignment */     Byte Offset: 24 (0x18)
} Elf32_Phdr;
Source: elf.h
                                         Byte Offset: 28 (0x1C)
```

```
typedef uint32_t Elf32_Word;
typedef uint32_t Elf32_Off;
typedef uint32_t Elf32_Addr;
```

```

dev@dev-VirtualBox:~/projects/hello_world$ readelf -l hello
Elf file type is EXEC (Executable file)
Entry point 0x1030c
There are 9 program headers, starting at offset 52
Program Headers:
  Type        Offset      VirtAddr     PhysAddr     FileSiz   MemSiz     Flg Align
 00 EXIDX      0x0004d0 0x000104d0 0x000104d0 0x00008 0x00008 R 0x4
 01 PHDR      0x0000034 0x00010034 0x00010034 0x00120 0x00120 R E 0x4
 02 INTERP    0x000154 0x00010154 0x00010154 0x00013 0x00013 R 0x1
    [Requesting program interpreter: /lib/ld-linux.so.3]
 03 LOAD       0x0000000 0x00010000 0x00010000 0x004dc 0x004dc R E 0x10000
 04 LOAD       0x0000f0c 0x00020f0c 0x00020f0c 0x00011c 0x000120 RW 0x10000
 05 DYNAMIC   0x0000f18 0x00020f18 0x00020f18 0x000e8 0x000e8 RW 0x4
 06 NOTE       0x0000168 0x00010168 0x00010168 0x00044 0x00044 R 0x4
 07 GNU_STACK 0x0000000 0x000000000 0x000000000 0x00000 0x00000 RW 0x10
 08 GNU_RELRO 0x0000f0c 0x00020f0c 0x00020f0c 0x000f4 0x000f4 R 0x1

```

## STM32 Executable Structure

When you compile and link code for STM32 (ARM Cortex-M), the output is typically an **ELF file**. This file contains several sections that get mapped into memory when the MCU boots:

- **.text** → program instructions (read-only, stored in Flash)
- **.rodata** → read-only constants (Flash)
- **.data** → initialized variables (copied from Flash to RAM at startup)
- **.bss** → uninitialized variables (zeroed in RAM at startup)
- **Setup / startup section** → special code and data that prepare the runtime environment before `main()` executes

Some other interesting observations upon using CoPilot,

## Relation to ELF Sections

- The **setup section** isn't a formal ELF section like `.text` or `.data`.
- Instead, it's the **startup code** that interprets ELF sections and prepares RAM.
- In linker scripts (`stm32f4xx.ld` etc.), you'll see definitions like:

ld

```

.data : {
    _sidata = LOADADDR(.data);
    _sdata = .;
    *(.data*)
    _edata = .;
} >RAM AT>FLASH

```

This tells the startup code where to copy `.data` from Flash to RAM

So from this we can see that the linker script decides what address each section must be loaded into

## ELF Segments vs. Sections

- **Sections** (`.text`, `.data`, `.bss`, etc.) are compiler/linker constructs.-
- **Sgments** (shown with `readelf -l`) are how the loader maps sections into memory.
- A **read/write segment (RW)** typically corresponds to RAM regions that hold variables.

## Typical Sections Inside RW Segments

When you run `readelf -l <file.elf>`, you'll see program headers like:

Code

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x0000000	0x080000000	0x080000000	0x0010000	0x0010000	R E	0x1000
LOAD	0x0010000	0x200000000	0x200000000	0x0002000	0x0004000	RW	0x1000

- The **first LOAD segment (R E)** → Flash, contains `.text`, `.rodata`, vector table.
- The **second LOAD segment (RW)** → RAM, contains:

## ● Sections inside RW segment

- **.data**
  - Initialized global/static variables.
  - Copied from Flash to RAM at startup.
- **.bss**
  - Uninitialized global/static variables.
  - Zeroed in RAM at startup.
- **.heap** (optional, depending on linker script)
  - Dynamic memory allocation region.
- **.stack** (optional, depending on linker script)
  - Stack space for function calls.
- **Other custom RAM sections** (e.g., `.noinit`, `.dma_buffer`)
  - Defined in linker script for special use cases.

## ✓ Summary

In STM32 ELF executables, the **read/write segment (RW)** usually contains:

- `.data` (initialized variables)
- `.bss` (uninitialized variables)
- `.heap` (malloc/new)
- `.stack` (runtime stack)
- Any custom RAM sections defined in the linker script

These are the sections that require **read/write access** at runtime, unlike `.text` or `.rodata` which are read-only in Flash.

Some more questions to be answered-

1. How can the processor concurrently fetch-decode and execute ?

Inside the processor, for each of fetch-decode-execute operation, there exist a subunit- fetch unit, decode unit and execute unit(ALU) which, so in a single clock cycle while 1st instruction is executing, the fetch unit can fetch the next instruction, thereby achieving concurrency

peripheral devices to the chip.

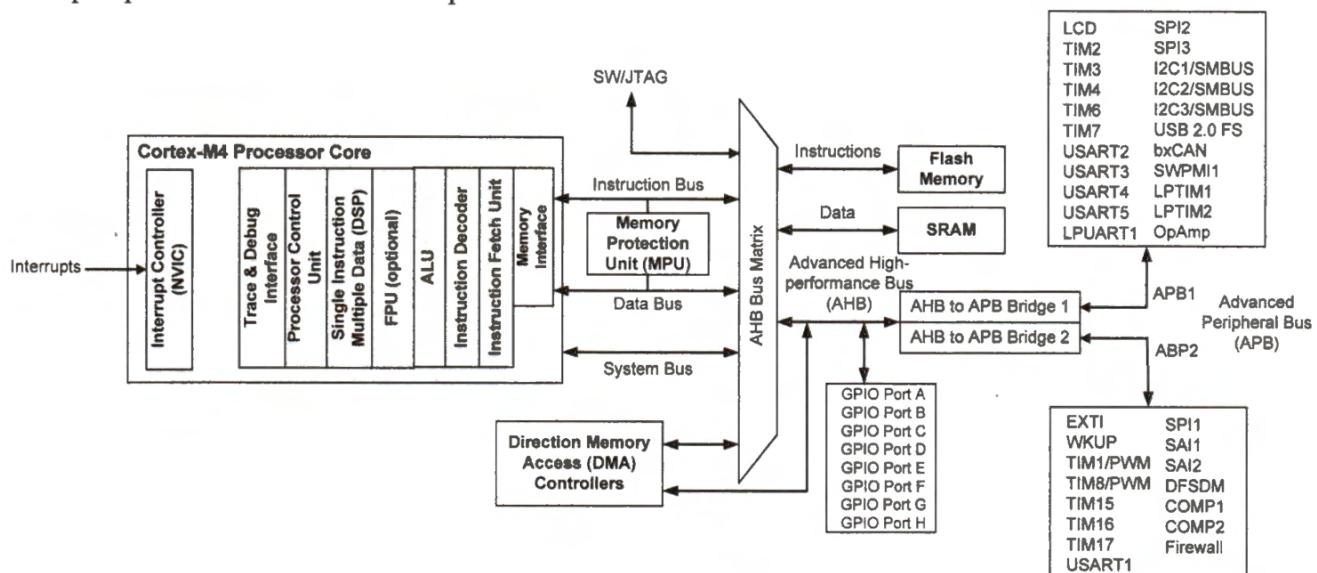


Figure 3-3. Organization of STM32L4 ARM Cortex-M4 processor