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.
```

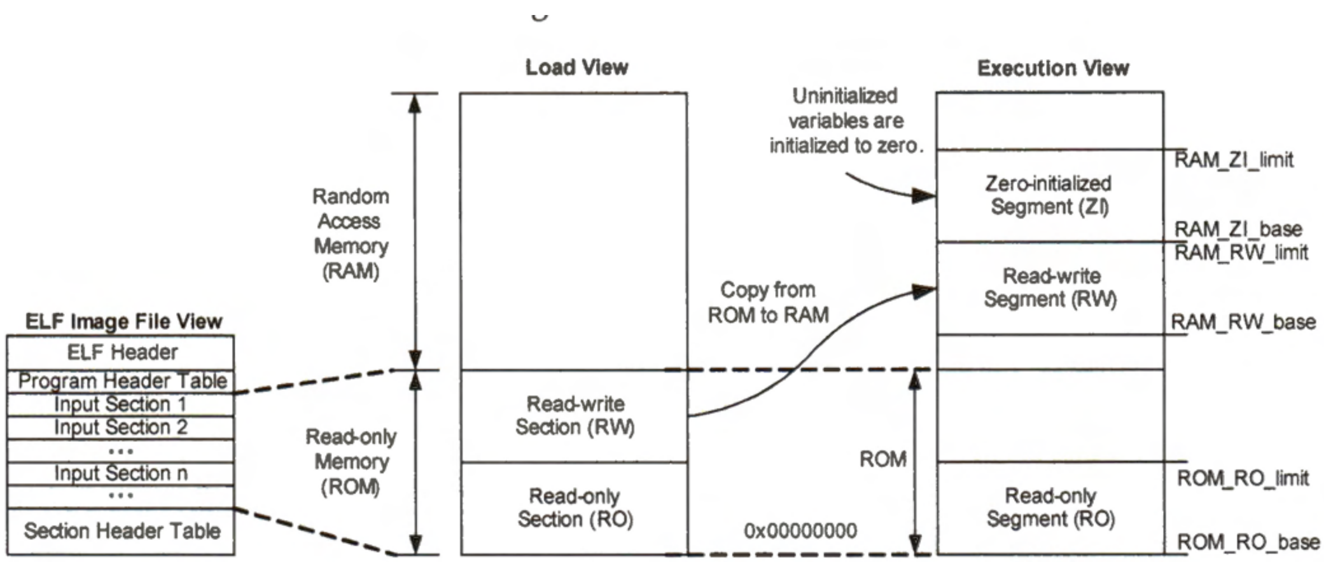


Figure 1-2. Interface of an executable binary file in the executable and linking format (ELF). An ELF file provides *load view* and *execution view*. The load view specifies how to load data into the memory. The execution view instructs how to initialize data regions at runtime.

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");
}
```

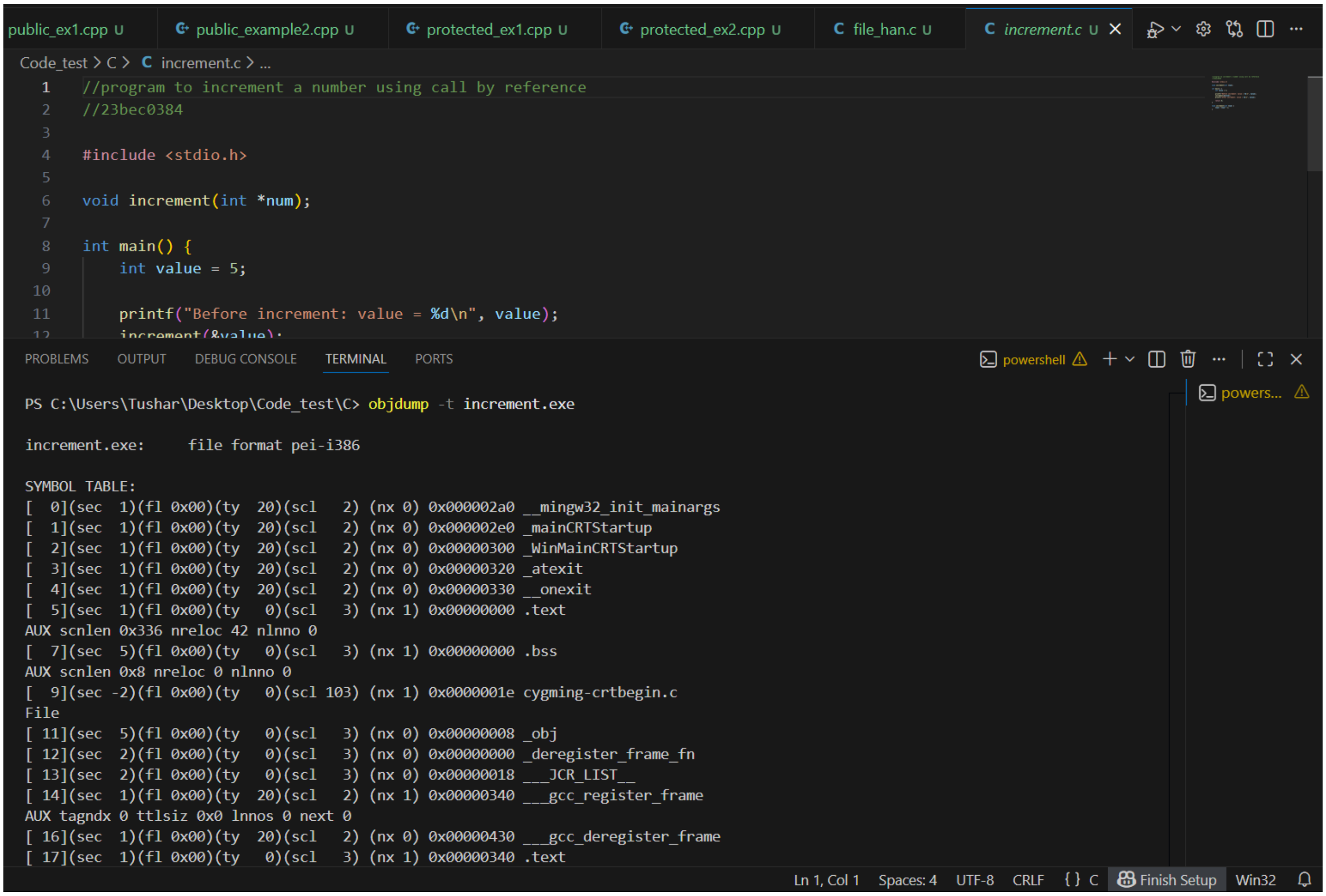
And upon running the readELF we could seegistfile1.txt

```

Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 64 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400274     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400298     0 SECTION LOCAL  DEFAULT    4
     5: 00000000004002b8     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400318     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400356     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000400360     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000400380     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000400398     0 SECTION LOCAL  DEFAULT   10
    11: 00000000004003c8     0 SECTION LOCAL  DEFAULT   11
    12: 00000000004003e0     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000400410     0 SECTION LOCAL  DEFAULT   13
    14: 00000000004005e8     0 SECTION LOCAL  DEFAULT   14
    15: 00000000004005f8     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000400604     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000400630     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000600e28     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000600e38     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000600e48     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000600e50     0 SECTION LOCAL  DEFAULT   21
```

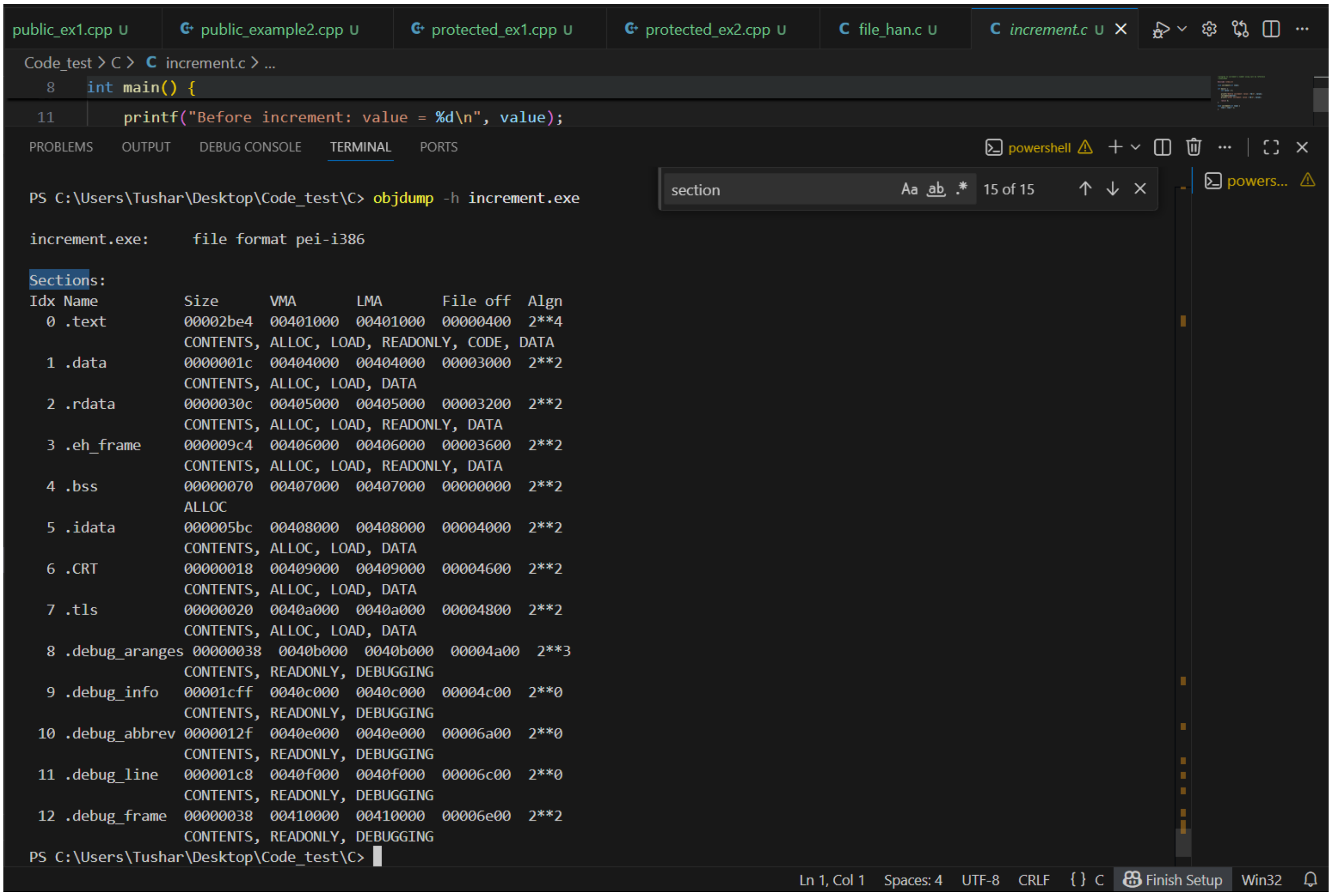
I had a similar executable file for a C program increment.c stored in my PC which is a simple program that increments a number



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



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 ellf 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;
```

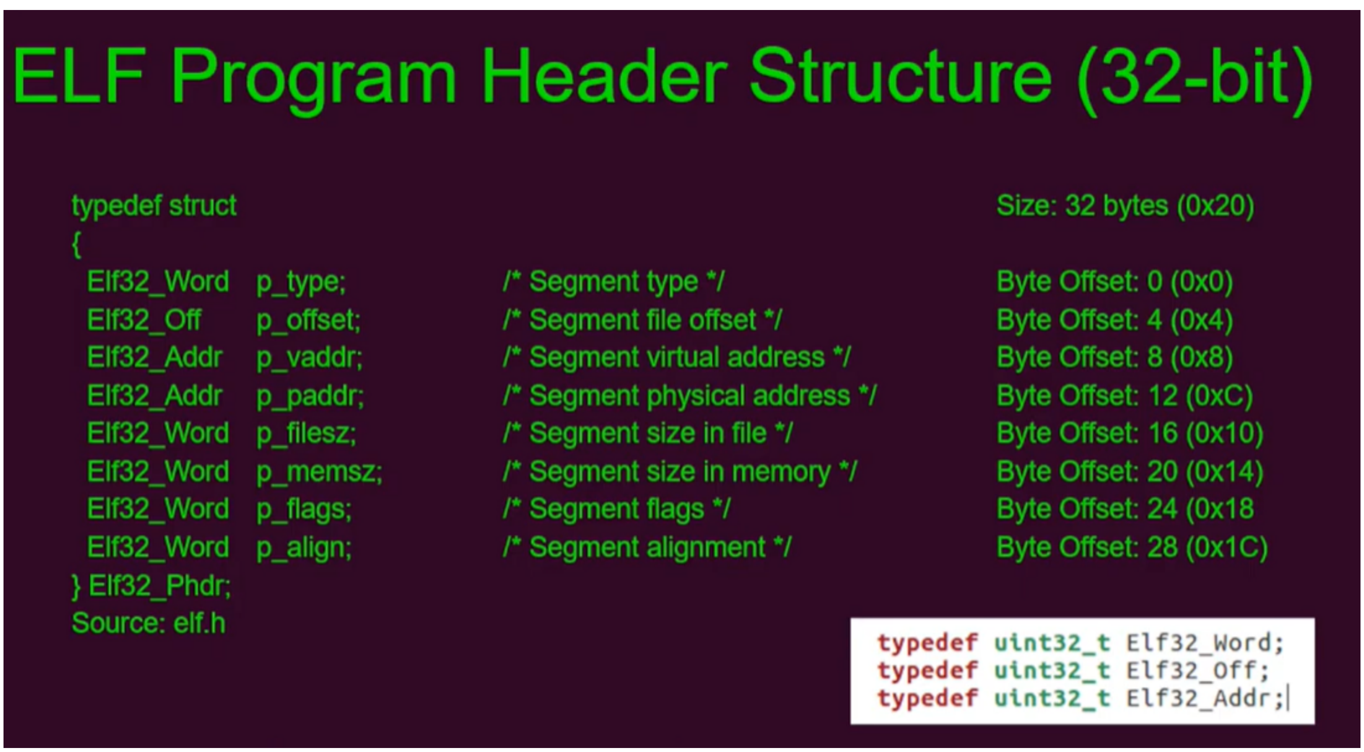
The ElfN_Shdr struct gives an information about each section. One or more sections can be there in a segment.

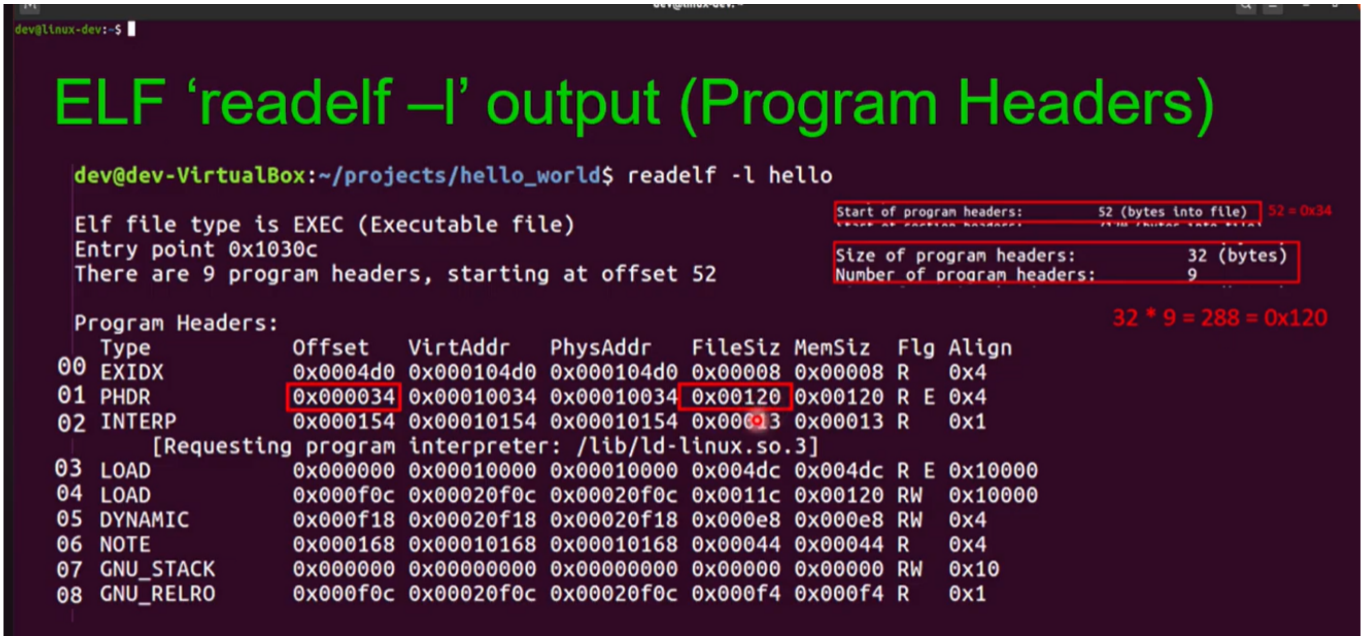
A brief description about each member of the `ElfN_Shdr` is as follows

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]





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.-
- **Segments** (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           0x000000 0x08000000 0x08000000 0x001000 0x001000 R E 0x1000
LOAD           0x001000 0x20000000 0x20000000 0x000200 0x000400 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.

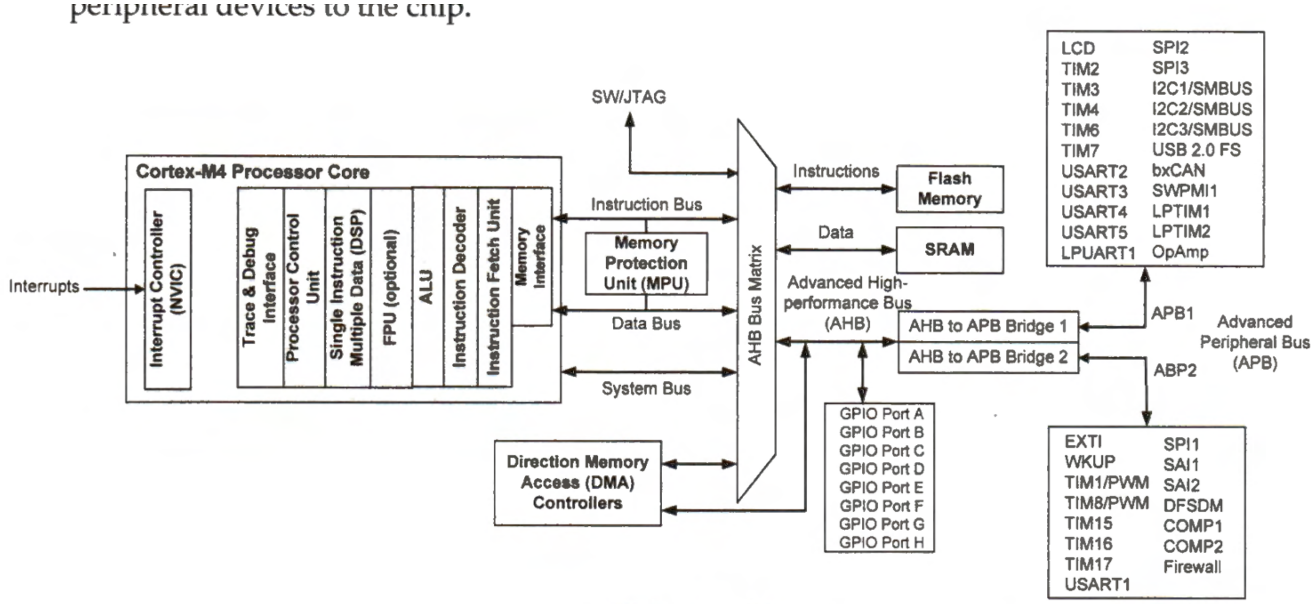


Figure 3-3. Organization of STM32L4 ARM Cortex-M4 processor