Startup file in C Language

Table of Contents

In this article we are going to implement the startup file using C language, according to this we will introduce some topics like:

1. Memory Layout and linker script file.
2. Arm-toolchain attributes (weak, alias).
3. Copy data section from the FLASH to SRAM.
4. Startup code in C Language.

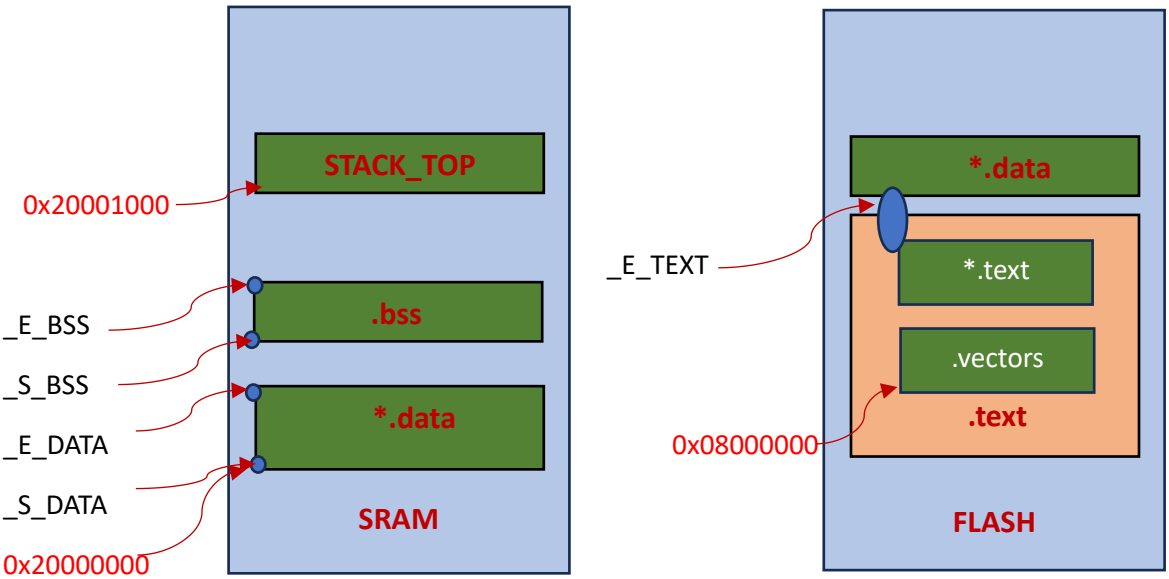
1. Memory Layout and Linker Script

According to the boot sequence for Cortex-M3 processor:

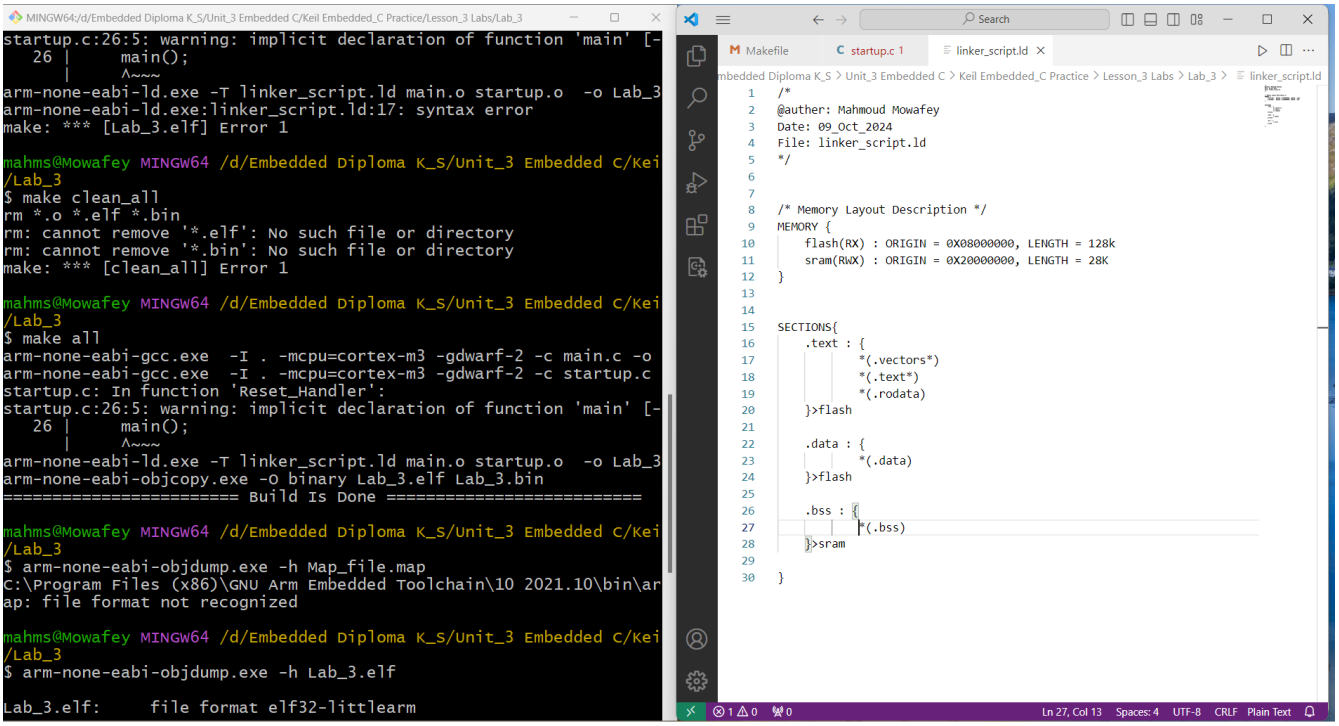
We need to put the value of the Stack_Top in the first location of the Flash memory.

- So, we will create a section called .text and include the .vectors, .text, and .rodata sections.
- Then save the locator operator [.] address inside the _E_TEXT symbol → will be used later.
- Make the virtual address and loading address is the FLASH memory.

```
SECTIONS{
  .text : {
        *(.vectors*)
        *(.text*)
        *(.rodata)
        _E_TEXT = .;
  }>flash
```



- Open the terminal, make clean_all and make all again.



Now we need to create the startup file and put the Stack_Top address inside the first location of the FLASH memory.

Our task this time is to save the value of the Stack_Top address inside the FLASH memory, first of all we will create a global variable to be saved in the .data section.

- Make sure that we will save the address of the Stack_Top as the first variable.
- After that we can store the addresses of the Handlers (Reset, NMI, Bus_Fault).

```
/* Define the Stack_top at 0x20001000 */
#define STACK_TOP 0X20001000
uint32_t vectors[]=
  {
    /* put the stack_top address as the first element of the array to be
      written inside first location of the flash */
    (uint32_t)STACK_TOP,
    };
```

Then we use __attribute__ () command from the Toolchain that's allowing us to save this global variables in a certain memory section.

__attribute__ () it provides additional information to the compiler like #pragmas.

We will ask the compiler to save this global array into the .vectors section inside the .text section of the FLASH memory, by this way the Stack_Top address is saved into the first location of the FLASH at address 0X08000000 as required.

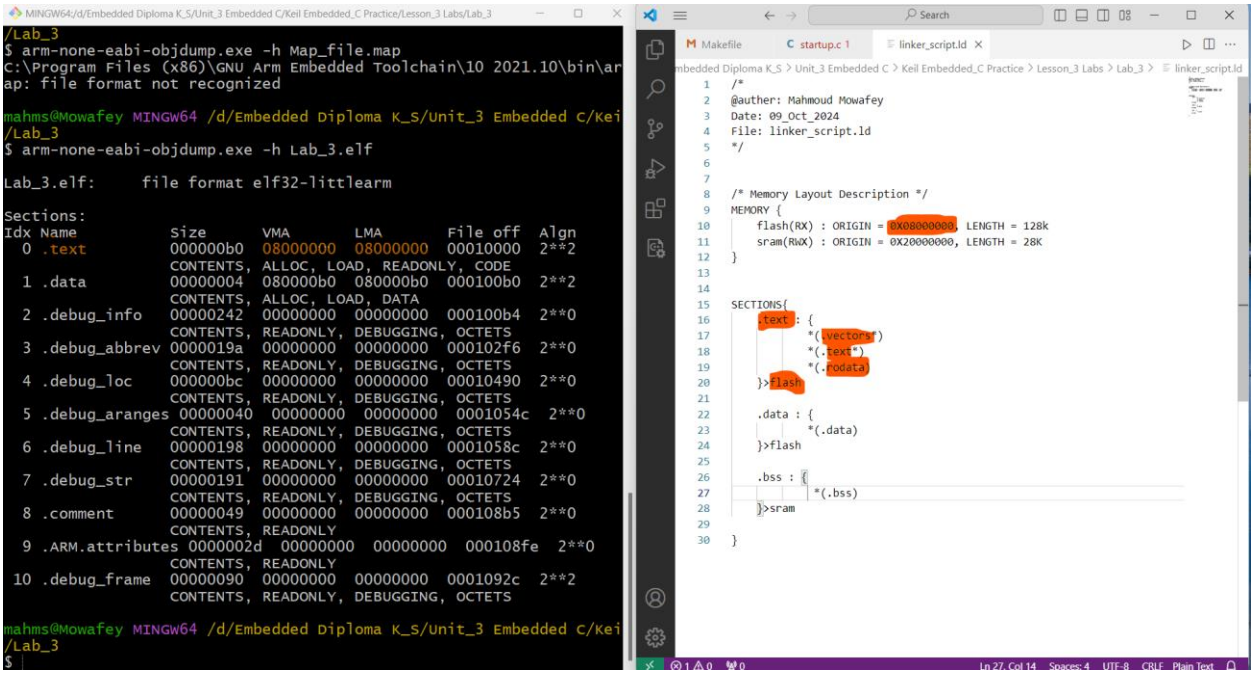
Also include the Handlers to be stored in the .vectors section.

```
uint32_t vectors[] __attribute__((section(".vectors"))) =
  {
    /* put the stack_top address as the first element of the array to
      be written inside first location of the flash */
    (uint32_t)STACK_TOP,
    /* write the Handlers according to the Interrupt Vector Table */
    (uint32_t)&Reset_Handler,
    (uint32_t)&NMI_Handler,
    (uint32_t)&Bus_Fault_Handler
        };
```

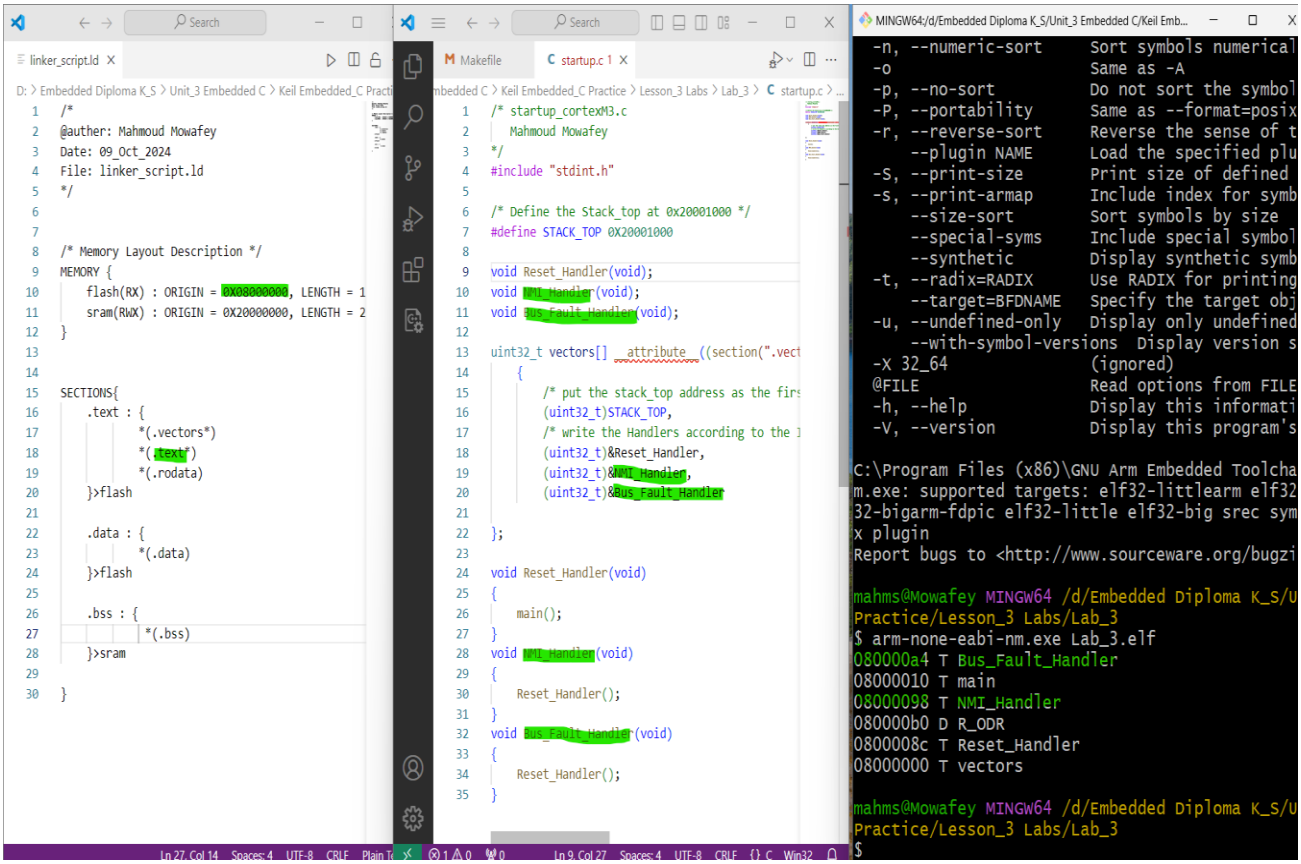
Let's compile again.

After compilation we can see that:

1- The beginning of the .vectors section is at the address 0x08000000 as required.



2- Handlers_Addresses are different, this will lead to memory wastage, we need to solve this.



2. Arm-toolchain attributes (weak, alias)

Now we need to solve the problem of memory wastage that is coming from multiple handler definitions.

In the Arm-Toolchain we have two keywords (weak, alias) are designed to provide more info to the compiler so that we can reduce the memory used for handlers.

Weak keyword means that we can override this place and execute the new definition for the handler, and this definition maybe defined in another file by the user.

For example ISR_Handler() in atmega, it has a default definition, but the user has the ability to override this definition and execute another code as he likes, the linker can handle this without any errors.

- This keyword instructs the compiler to export symbols weakly.
- The __weak keyword can be applied to function and variable declarations, and to function definitions.

Weak symbols theoretically work like this:

- There is one .o file with __weak__ symbol.
- There is another .o file with a normal symbol.
- Linker sees both symbols and picks the non-weak symbol.

The .o object files are generated from .c files. The definition of the symbol inside .c should be weak. Attributes applied to declaration are applied to definitions that see it. Doing __weak__ in a header in a declaration marks **all** definitions that see that declaration of this symbol as __weak__.

alias This function attribute enables you to specify multiple aliases for functions.

- Where a function is defined in the current translation unit, the alias call is replaced by a call to the function, and the alias is emitted alongside the original name.
- Where a function is not defined in the current translation unit, the alias call is replaced by a call to the real function.
- Where a function is defined as static, the function name is replaced by the alias name and the function is declared external if the alias name is declared external.

__attribute__((alias)) function attribute

Now we will apply those two keywords in our code:
-We are going to define a Default_Handler and mapping all unused handlers to this one, so that they can share the same address.

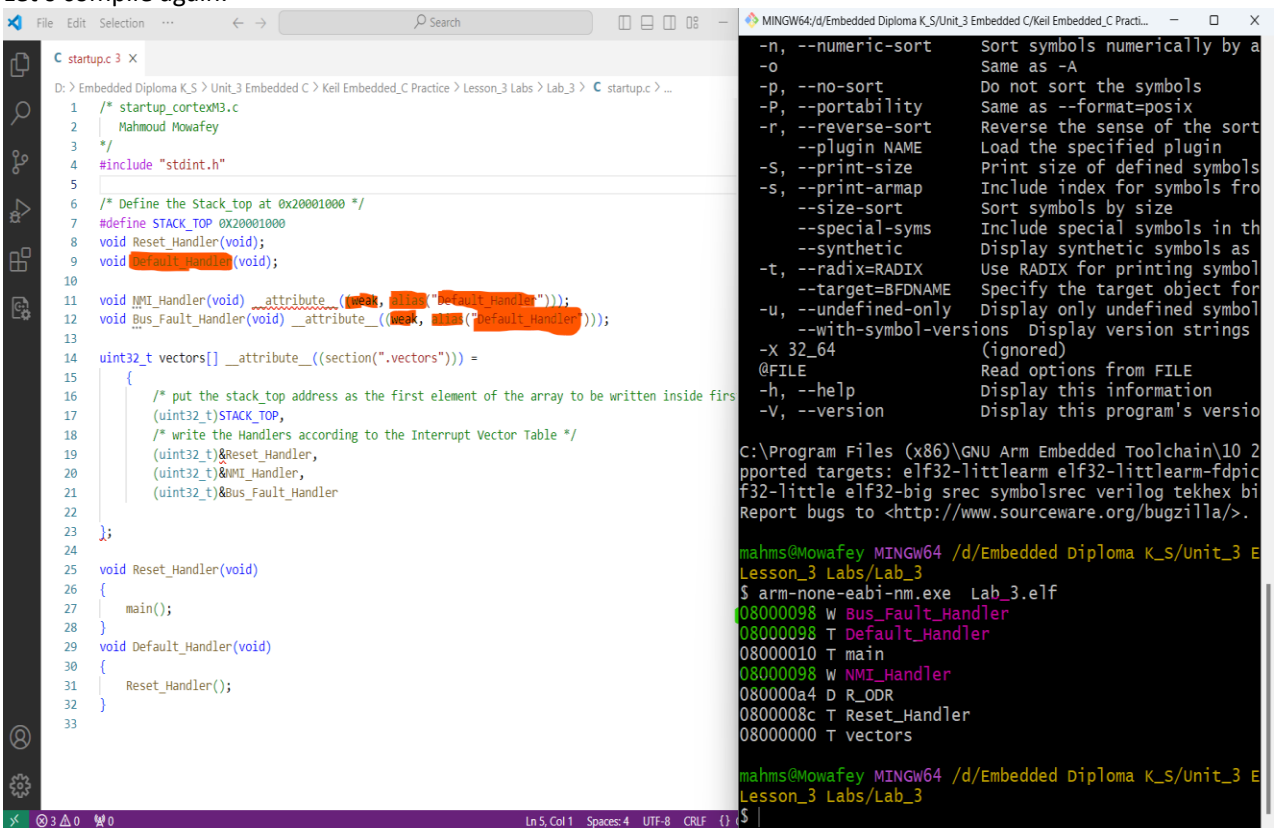
```
void Default_Handler(void);
void Bus_Fault_Handler(void) __attribute__((weak, alias("Default_Handler")));
void NMI_Handler(void) __attribute__((weak, alias("Default_Handler")));
```

-We will jump to the reset_handler as a default in our lab.

```
void Default_Handler(void)
{
  Reset_Handler();
}
```

-Wehenever we need to use those unused handlers the compiler gives a new address for it and override the definition.

Let's compile again.



As you can see now that the NMI_Handler & Bus_Default_Handler are sharing the same address of the Default_Handler which is 0x08000098

3. Copy data section from the FLASH to SRAM.

To copy data using C language, we need to know the boundaries so that we can copy the data safely.

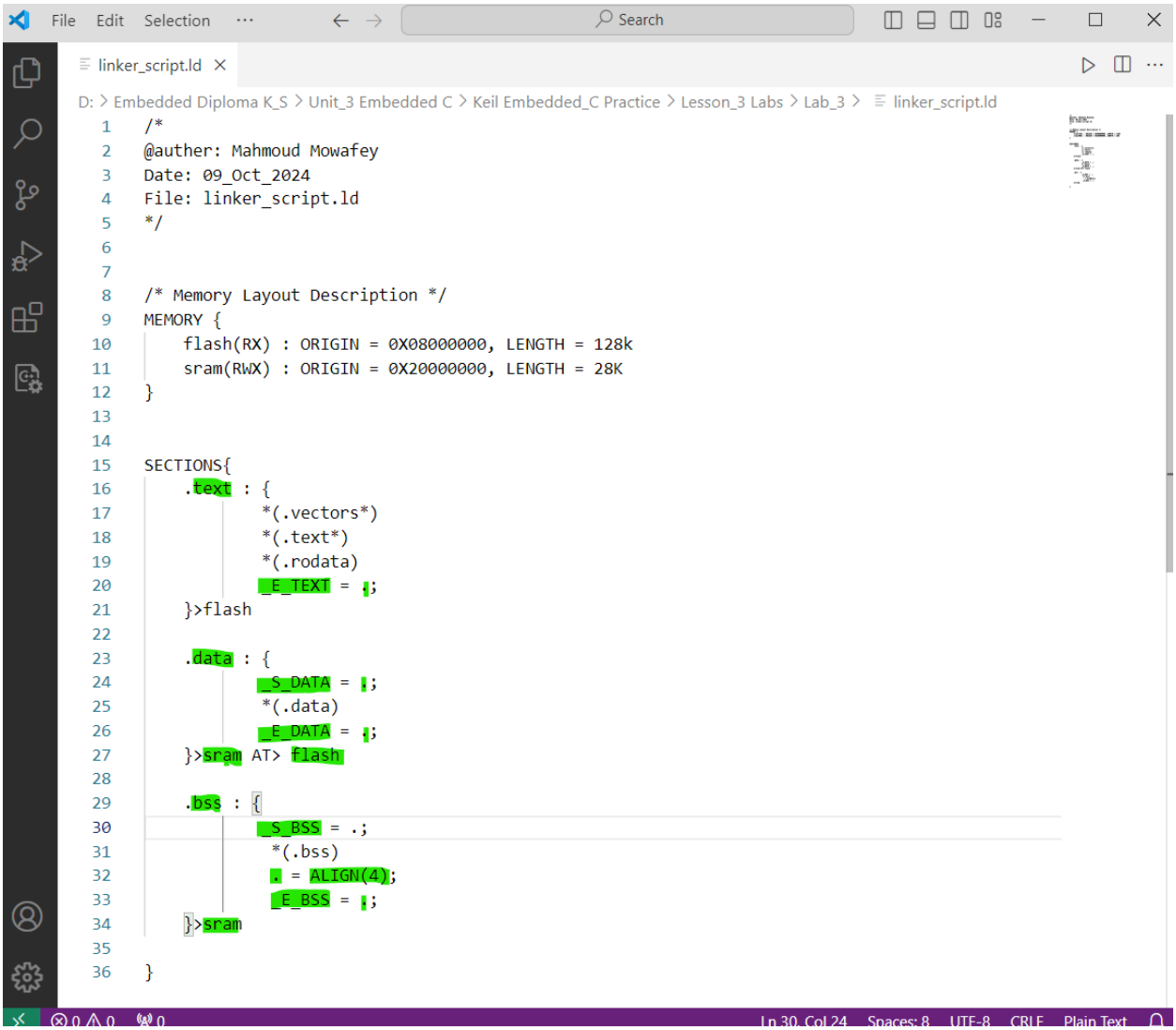
For that we need to specify the beginning and the end for .data, .bss, and specify the end of the .text section.

This can be achieved using the increment locator in the Linker_Script file.

We will add a symbols:

- to save the end of .text section → _E_TEXT
- to save the start and the end of the .data section → _S_DATA & _E_DATA
- to save the start and the end of the .bss section → _S_BSS & _E_BSS

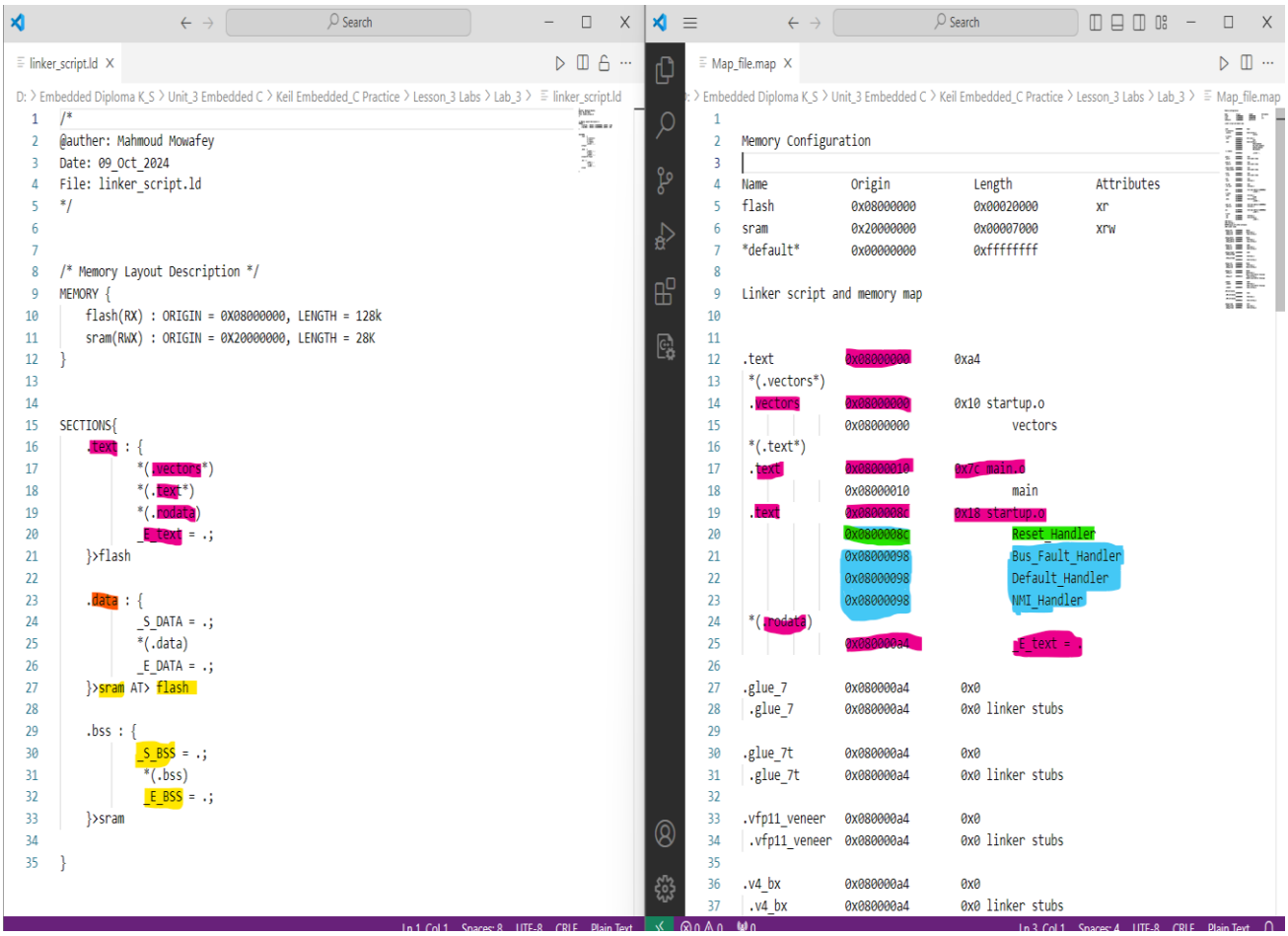
Also, we need to take care about the alignment of the memory, and the Stack_Size.



Also, take in mind the VA & LA for each section where:

- The .text section must be in the FLASH in both VA & LA → FLASH
- The .data section need to be loaded from the FLASH to SRAM → SRAM AT> FLASH
- The .bss section need to be only in the SRAM → SRAM

Now we are ready to compile and check the map_file to see where our data boundaries are set.

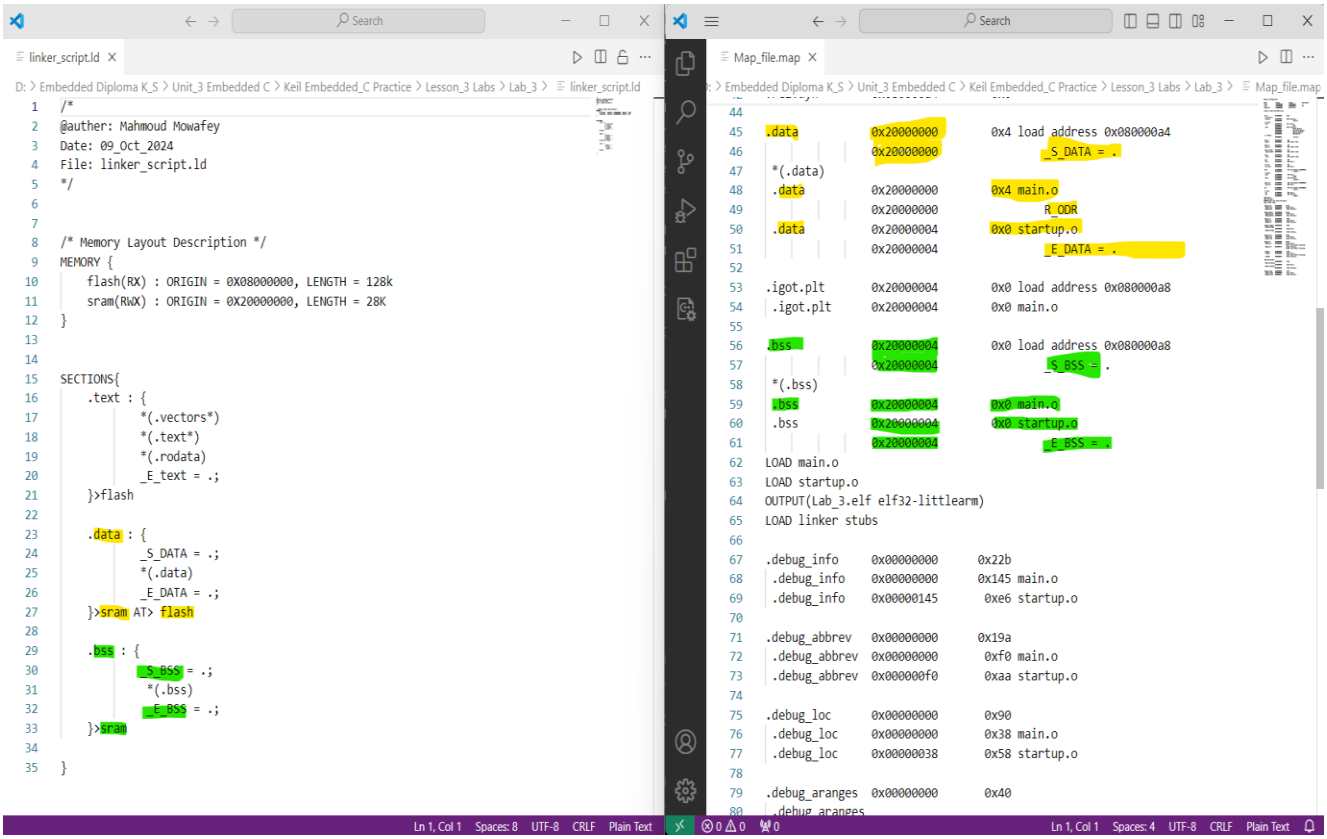


```
/*
@auther: Mahmoud Mowafey
Date: 09_Oct_2024
File: linker_script.ld
*/


/* Memory Layout Description */
MEMORY {
    flash(RX) : ORIGIN = 0X08000000, LENGTH = 128k
    sram(RWX) : ORIGIN = 0X20000000, LENGTH = 28K
}


SECTIONS{
    .text : {
            *(.vectors*)
            *(.text*)
            *(.rodata)
            _E_text = .;
    }>flash

    .data : {
            _S_DATA = .;
            *(.data)
            _E_DATA = .;
    }>sram AT> flash

    .bss : {
            _S_BSS = .;
             *(.bss)
            _E_BSS = .;
    }>sram

}
```

```
Memory Configuration

Name            Origin          Length          Attributes
flash           0x08000000      0x00020000      xr
sram            0x20000000      0x00007000      xrw
*default*       0x00000000      0xffffffff

Linker script and memory map


.text           0x08000000      0xa4
 *(.vectors*)
 .vectors       0x08000000      0x10 startup.o
                0x08000000              vectors
 *(.text*)
 .text          0x08000010      0x7c main.o
                0x08000010              main
 .text          0x0800008c      0x18 startup.o
                0x0800008c              Reset_Handler
                0x08000098              Bus_Fault_Handler
                0x08000098              Default_Handler
                0x08000098              NMI_Handler
 *(.rodata)
                0x080000a4              _E_text = .

.glue_7         0x080000a4      0x0
 .glue_7        0x080000a4      0x0 linker stubs

.glue_7t        0x080000a4      0x0
 .glue_7t       0x080000a4      0x0 linker stubs

.vfp11_veneer   0x080000a4      0x0
 .vfp11_veneer  0x080000a4      0x0 linker stubs

.v4_bx          0x080000a4      0x0
 .v4_bx         0x080000a4      0x0 linker stubs
```

the text section beginning from 0x08000000 at flash, vectors are first elements starting with this location, which is reserved by the startup, then reserve memory for text from the main file, then reserve memory for text from startup for handlers definitions increment_locator is saved to the _E_TEXT symbol.

The same also we can see the .data section & the .bss section via the map_file.

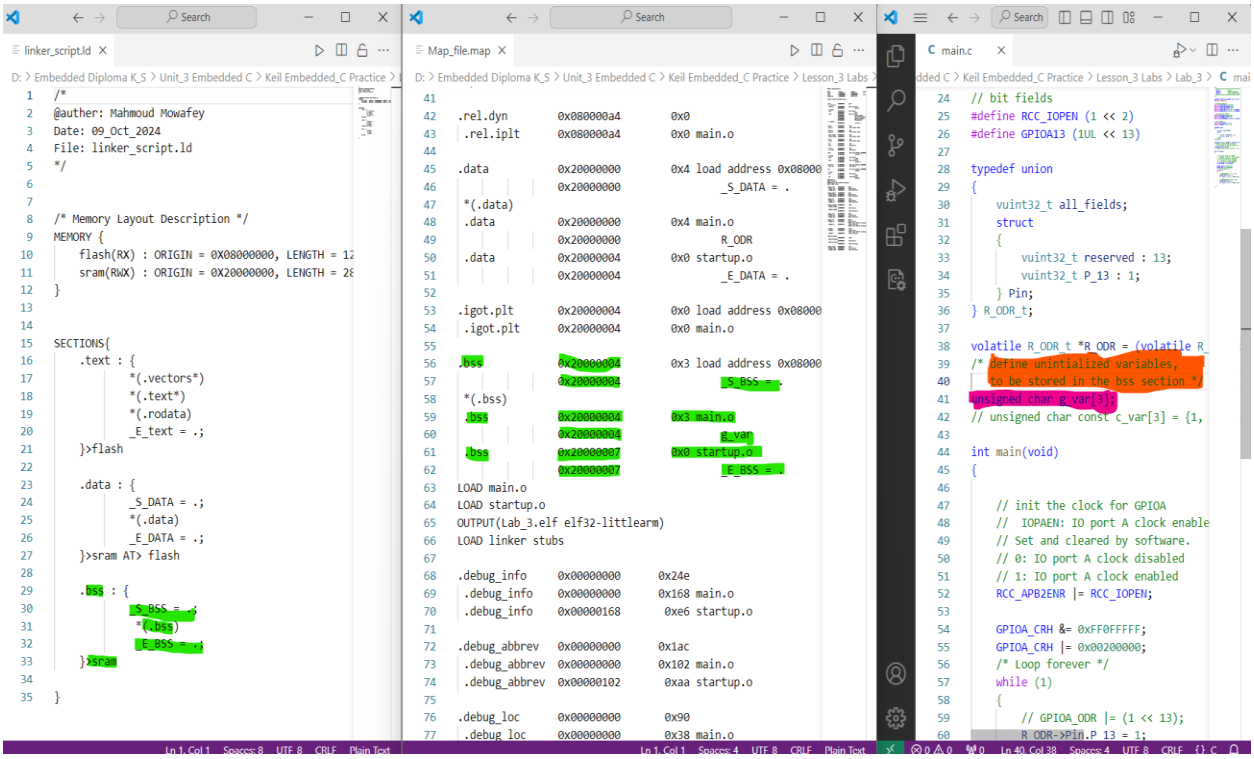


The .data section starting from address 0X20000000, clarification for data section begining and end, also the bss section but for the time being we did not define uninitialized variables to be stored inside the .bss section.

Let's now define some global_uninitialized variables inside the main.c file and check the .bss section again.

```
/* define unintialized variables,
   to be stored in the bss section */
unsigned char g_var[3];
```

the compiler will reserve 3 location inside the.bss section and initialized them by ZEROs.

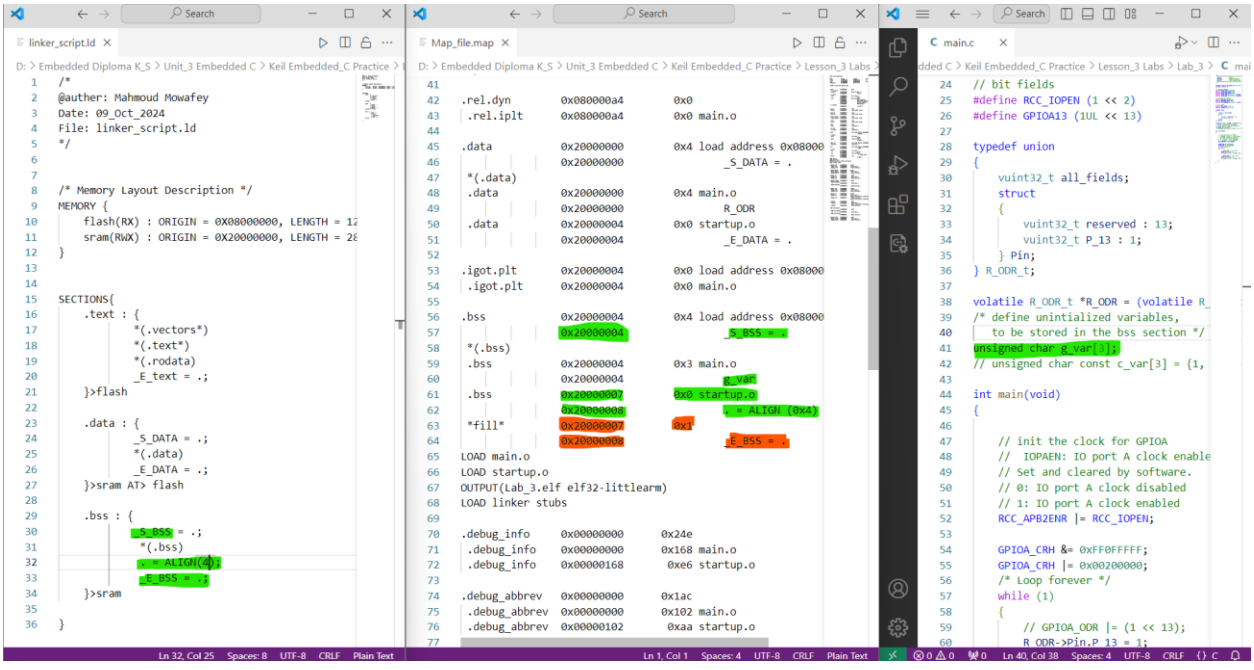


As you can see the beginning and the end of the .bss section is not the same where the:

_S_BSS is at the address of 0X20000004

_E_BSS is at the address of 0X20000007

But we have a problem so that we can not achieve the memory alignment due to the end of .bss section is not aligned, so we need to fill one byte then save the _E_BSS again.



4. Startup code in C Language

Finally, we can write our complete code and test it through proteus.