

Startup file in C Language

Table of Contents

1. Memory Layout and Linker Script	2
2. Arm-toolchain attributes (weak, alias)	6
3. Copy data section from the FLASH to SRAM.....	8
4. Startup code in C Language	12

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.

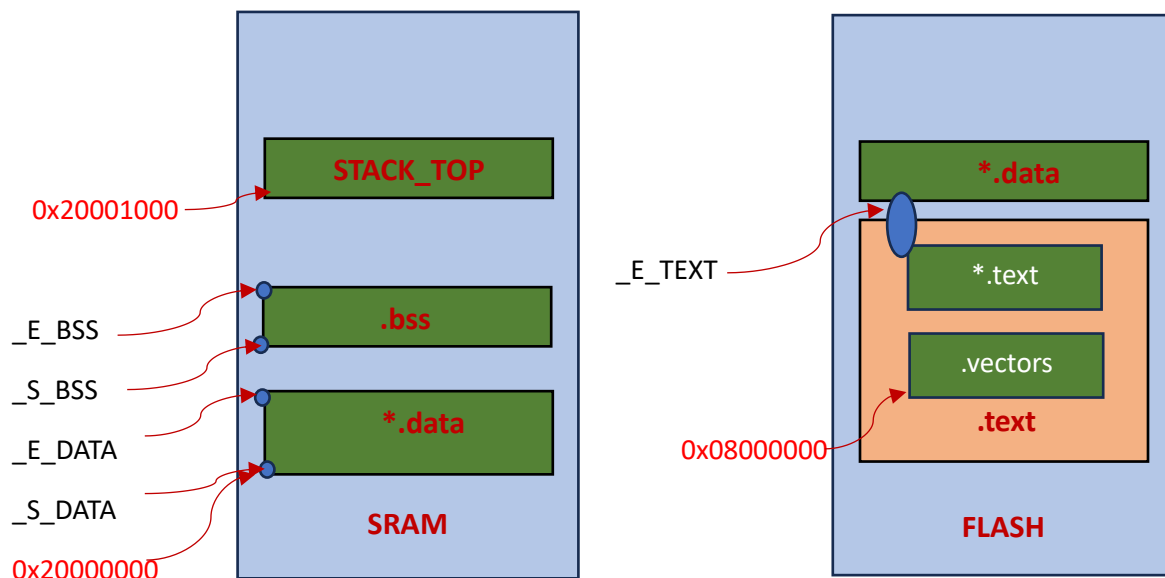
1. Memory Layout and Linker Script

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.

```

MINGW64/d/Embedded Diploma K_S/Unit_3 Embedded C/Keil Embedded_C Practice/Lesson_3 Labs/Lab_3
startup.c:26:5: warning: implicit declaration of function 'main' [-Wimplicit-declaration]
    26 |     main();
        |     ^~~~~
arm-none-eabi-ld.exe -T linker_script.ld main.o startup.o -o Lab_3
arm-none-eabi-ld.exe:linker_script.ld:17: syntax error
make: *** [Lab_3.elf] Error 1

mahms@Mowafey MINGW64 /d/Embedded Diploma K_S/Unit_3 Embedded C/Keil Embedded_C Practice/Lesson_3 Labs/Lab_3
$ make clean_all
rm *.o *.elf *.bin
rm: cannot remove '*.elf': No such file or directory
rm: cannot remove '*.bin': No such file or directory
make: *** [clean_all] Error 1

mahms@Mowafey MINGW64 /d/Embedded Diploma K_S/Unit_3 Embedded C/Keil Embedded_C Practice/Lesson_3 Labs/Lab_3
$ make all
arm-none-eabi-gcc.exe -I . -mcpu=cortex-m3 -gdwarf-2 -c main.c -o main.o
arm-none-eabi-gcc.exe -I . -mcpu=cortex-m3 -gdwarf-2 -c startup.c -o startup.o
startup.c: In function 'Reset_Handler':
startup.c:26:5: warning: implicit declaration of function 'main' [-Wimplicit-declaration]
    26 |     main();
        |     ^~~~~
arm-none-eabi-ld.exe -T linker_script.ld main.o startup.o -o Lab_3
arm-none-eabi-objcopy.exe -O binary Lab_3.elf Lab_3.bin
===== Build Is Done =====

mahms@Mowafey MINGW64 /d/Embedded Diploma K_S/Unit_3 Embedded C/Keil Embedded_C Practice/Lesson_3 Labs/Lab_3
$ arm-none-eabi-objdump.exe -h Map_file.map
C:\Program Files (x86)\GNU Arm Embedded Toolchain\10.2021.10\bin\arm-none-eabi-objdump.exe: file format not recognized

mahms@Mowafey MINGW64 /d/Embedded Diploma K_S/Unit_3 Embedded C/Keil Embedded_C Practice/Lesson_3 Labs/Lab_3
$ arm-none-eabi-objdump.exe -h Lab_3.elf
Lab_3.elf:      file format elf32-littlearm
  
```

```

1  /*
2  @author: Mahmoud Mowafey
3  Date: 09_Oct_2024
4  File: linker_script.ld
5  */
6
7  /* Memory Layout Description */
8  MEMORY {
9      flash(RX) : ORIGIN = 0x08000000, LENGTH = 128k
10     sram(RWX) : ORIGIN = 0x20000000, LENGTH = 28K
11 }
12
13 SECTIONS {
14     .text : {
15         *(.vectors*)
16         *(.text*)
17         *(.rodata)
18     } >flash
19
20     .data : {
21         *(.data)
22     } >flash
23
24     .bss : {
25         *(.bss)
26     } >sram
27
28 }
29
30
  
```

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.

The screenshot displays the compilation process and the resulting memory layout. The terminal window on the left shows the command prompt for MINGW64, where the user runs `arm-none-eabi-objdump.exe -h Map_file.map` and `arm-none-eabi-objdump.exe -h Lab_3.elf`. The output for `Lab_3.elf` shows the file format as `elf32-littlearm` and lists the sections with their sizes, VMA, LMA, file offsets, and alignments. The sections include `.text`, `.data`, `.debug_info`, `.debug_abbrev`, `.debug_loc`, `.debug_aranges`, `.debug_line`, `.debug_str`, `.comment`, `.ARM.attributes`, and `.debug_frame`.

The IDE window on the right shows the linker script `linker_script.ld` with the following content:

```
1  /*
2  @author: Mahmoud Mowafey
3  Date: 09_Oct_2024
4  File: linker_script.ld
5  */
6
7
8  /* Memory Layout Description */
9  MEMORY {
10     flash(RX) : ORIGIN = 0x08000000, LENGTH = 128k
11     sram(RWX) : ORIGIN = 0x20000000, LENGTH = 28K
12 }
13
14
15 SECTIONS {
16     .text : {
17         *(.vectors)
18         *(.text*)
19         *(.rodata)
20     } > flash
21
22     .data : {
23         *(.data)
24     } > flash
25
26     .bss : {
27         *(.bss)
28     } > sram
29
30 }
```

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

The screenshot displays the Keil IDE interface with three main components:

- linker_script.ld (Left Panel):**

```

1  /*
2  @author: Mahmoud Mowafey
3  Date: 09_Oct_2024
4  File: linker_script.ld
5  */
6
7  /* Memory Layout Description */
8
9  MEMORY {
10     flash(RX) : ORIGIN = 0x08000000, LENGTH = 1
11     sram(RWX) : ORIGIN = 0x20000000, LENGTH = 2
12 }
13
14 SECTIONS{
15     .text : {
16         *(.vectors*)
17         *(.text*)
18         *(.rodata)
19     }>flash
20
21
22     .data : {
23         *(.data)
24     }>flash
25
26     .bss : {
27         *(.bss)
28     }>sram
29
30 }
```
- startup.c (Middle Panel):**

```

1  /* startup_cortexM3.c
2  Mahmoud Mowafey
3  */
4  #include "stdint.h"
5
6  /* Define the Stack_top at 0x20001000 */
7  #define STACK_TOP 0x20001000
8
9  void Reset_Handler(void);
10 void NMI_Handler(void);
11 void Bus_Fault_Handler(void);
12
13 uint32_t vectors[] __attribute__((section(".vectors"))) = {
14     /* put the stack_top address as the first element */
15     (uint32_t)STACK_TOP,
16     /* write the Handlers according to the table */
17     (uint32_t)&Reset_Handler,
18     (uint32_t)&NMI_Handler,
19     (uint32_t)&Bus_Fault_Handler,
20 };
21
22 void Reset_Handler(void)
23 {
24     main();
25 }
26
27 void NMI_Handler(void)
28 {
29     Reset_Handler();
30 }
31
32 void Bus_Fault_Handler(void)
33 {
34     Reset_Handler();
35 }
```
- Terminal (Right Panel):**

```

-n, --numeric-sort      Sort symbols numerical
-o                       Same as -A
-p, --no-sort           Do not sort the symbols
-P, --portability       Same as --format=posix
-r, --reverse-sort      Reverse the sense of the sort
--plugin NAME           Load the specified plugin
-s, --print-size        Print size of defined symbols
-s, --print-arp         Include index for symbols
--size-sort             Sort symbols by size
--special-syms          Include special symbols
--synthetic             Display synthetic symbols
-t, --radix=RADIX       Use RADIX for printing
--target=BFDNAME        Specify the target object
-u, --undefined-only    Display only undefined symbols
--with-symbol-versions  Display versioned symbols
-X 32_64                (ignored)
@FILE                   Read options from FILE
-h, --help              Display this information
-V, --version            Display this program's version

C:\Program Files (x86)\GNU Arm Embedded Toolchain\bin\arm-none-eabi-nm.exe: supported targets: elf32-littlearm elf32-littlearm elf32-bigarm-fdpic elf32-little elf32-big srec sym
x plugin
Report bugs to <http://www.sourceware.org/bugzilla>

mahms@Mowafey MINGW64 /d/Embedded Diploma_K_S/Unit_3 Embedded C/Keil Embedded_C_Practice/Lesson_3 Labs/Lab_3
$ arm-none-eabi-nm.exe Lab_3.elf
080000a4 T Bus_Fault_Handler
08000010 T main
08000098 T NMI_Handler
080000b0 D R_ODR
0800008c T Reset_Handler
08000000 T vectors

mahms@Mowafey MINGW64 /d/Embedded Diploma_K_S/Unit_3 Embedded C/Keil Embedded_C_Practice/Lesson_3 Labs/Lab_3
$
```

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.

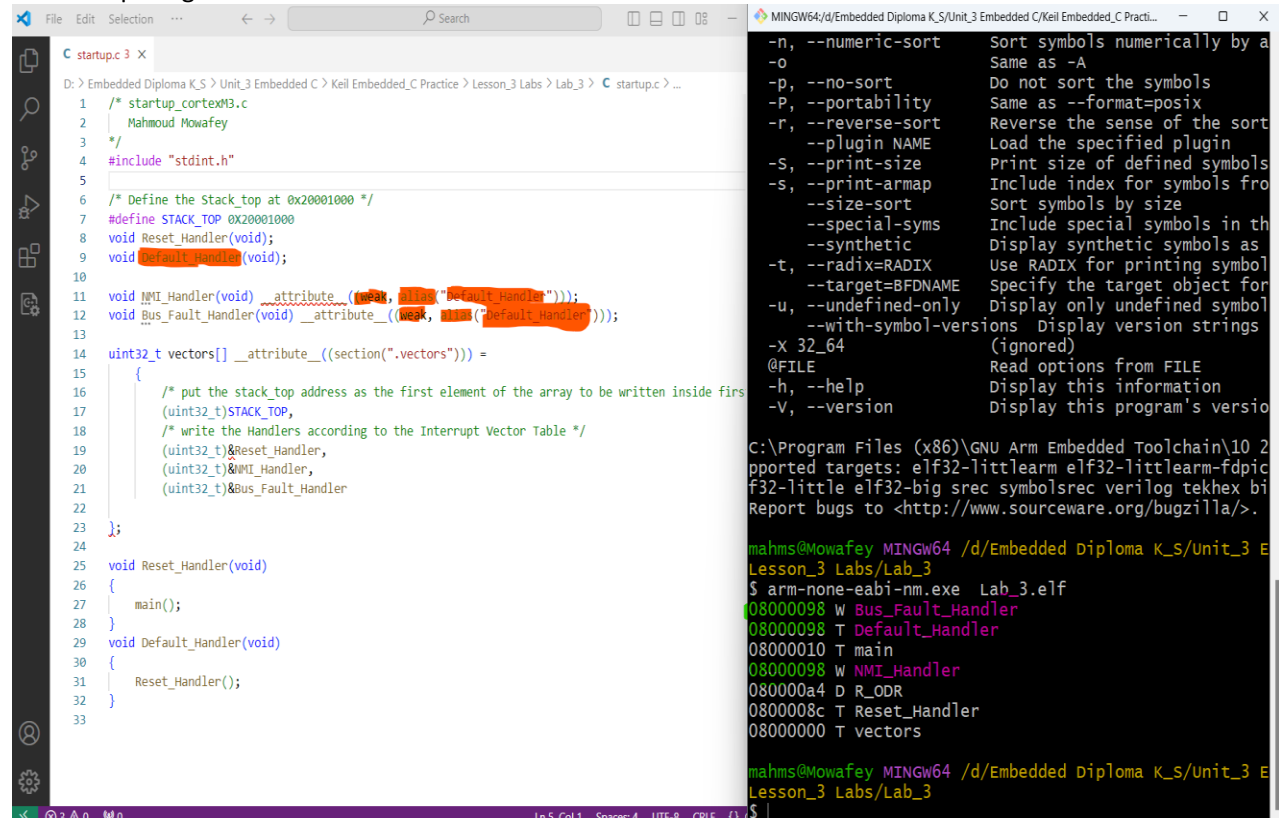
```
void Default_Handler(void);
void Bus_Fault_Handler(void) __attribute__((weak, alias("Default_Handler")));
void NMI_Handler(void) __attribute__((weak, alias("Default_Handler")));
```

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

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

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

Let's compile again.



```

1  /* startup_cortexM3.c
2  Mahmoud Mowafey
3  */
4  #include "stdint.h"
5
6  /* Define the Stack_top at 0x20001000 */
7  #define STACK_TOP 0x20001000
8  void Reset_Handler(void);
9  void Default_Handler(void);
10
11 void NMI_Handler(void) __attribute__((weak, alias("Default_Handler")));
12 void Bus_Fault_Handler(void) __attribute__((weak, alias("Default_Handler")));
13
14 uint32_t vectors[] __attribute__((section(".vectors"))) =
15 {
16     /* put the stack_top address as the first element of the array to be written inside flash */
17     (uint32_t)STACK_TOP,
18     /* write the Handlers according to the Interrupt Vector Table */
19     (uint32_t)&Reset_Handler,
20     (uint32_t)&NMI_Handler,
21     (uint32_t)&Bus_Fault_Handler
22 };
23
24 void Reset_Handler(void)
25 {
26     main();
27 }
28
29 void Default_Handler(void)
30 {
31     Reset_Handler();
32 }
33

```

```

-n, --numeric-sort      Sort symbols numerically by address
-o, --output FILE       Same as -A
-p, --no-sort           Do not sort the symbols
-P, --portability       Same as --format=posix
-r, --reverse-sort      Reverse the sense of the sort
--plugin NAME           Load the specified plugin
-S, --print-size        Print size of defined symbols
-s, --print-arpmap      Include index for symbols from archive
--size-sort             Sort symbols by size
--special-syms          Include special symbols in the sort
--synthetic             Display synthetic symbols as defined
-t, --radix=RADIX       Use RADIX for printing symbol names
--target=BFDNAME        Specify the target object for the linker
-u, --undefined-only    Display only undefined symbols
--with-symbol-versions  Display version strings
-X 32_64                (ignored)
@FILE                  Read options from FILE
-h, --help              Display this information
-V, --version           Display this program's version

```

```

C:\Program Files (x86)\GNU Arm Embedded Toolchain\10.2\bin\arm-none-eabi-nm.exe Lab3.elf
Supported targets: elf32-littlearm elf32-littlearm-fdpic
f32-little elf32-big srec symbolsrec verilog tekhex binary
Report bugs to <http://www.sourceware.org/bugzilla/>.

mahms@Mowafey MINGW64 /d/Embedded Diploma K_S/Unit_3 Embedded C/Keil Embedded_C Practice/Lesson_3 Labs/Lab_3
$ arm-none-eabi-nm.exe Lab3.elf
08000098 w Bus_Fault_Handler
08000098 T Default_Handler
08000010 T main
08000098 w NMI_Handler
080000a4 D R_ODR
0800008c T Reset_Handler
08000000 T vectors

```

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**.

```

1  /*
2  @author: Mahmoud Mowafey
3  Date: 09_Oct_2024
4  File: linker_script.ld
5  */
6
7
8  /* Memory Layout Description */
9  MEMORY {
10     flash(RX) : ORIGIN = 0X08000000, LENGTH = 128k
11     sram(RWX) : ORIGIN = 0X20000000, LENGTH = 28K
12 }
13
14
15 SECTIONS{
16     .text : {
17         *(.vectors*)
18         *(.text*)
19         *(.rodata)
20         _E_TEXT = ;
21     }>flash
22
23     .data : {
24         _S_DATA = ;
25         *(.data)
26         _E_DATA = ;
27     }>sram AT> flash
28
29     .bss : {
30         _S_BSS = .;
31         *(.bss)
32         _E_BSS = ALIGN(4);
33     }>sram
34
35 }
36

```


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.

The screenshot shows two files in the Keil IDE: linker_script.ld and Map_file.map.

linker_script.ld:

```

1  /*
2  @author: Mahmoud Mowafey
3  Date: 09_Oct_2024
4  File: linker_script.ld
5  */
6
7
8  /* Memory Layout Description */
9  MEMORY {
10     flash(RX) : ORIGIN = 0x08000000, LENGTH = 128K
11     sram(RWX) : ORIGIN = 0x20000000, LENGTH = 28K
12 }
13
14 SECTIONS{
15     .text : {
16         *(.vectors*)
17         *(.text*)
18         *(.rodata)
19         _E_TEXT = .;
20     }>flash
21
22     .data : {
23         _S_DATA = .;
24         *(.data)
25         _E_DATA = .;
26     }>sram AT> flash
27
28     .bss : {
29         _S_BSS = .;
30         *(.bss)
31         _E_BSS = .;
32     }>sram
33
34 }
35

```

Map_file.map:

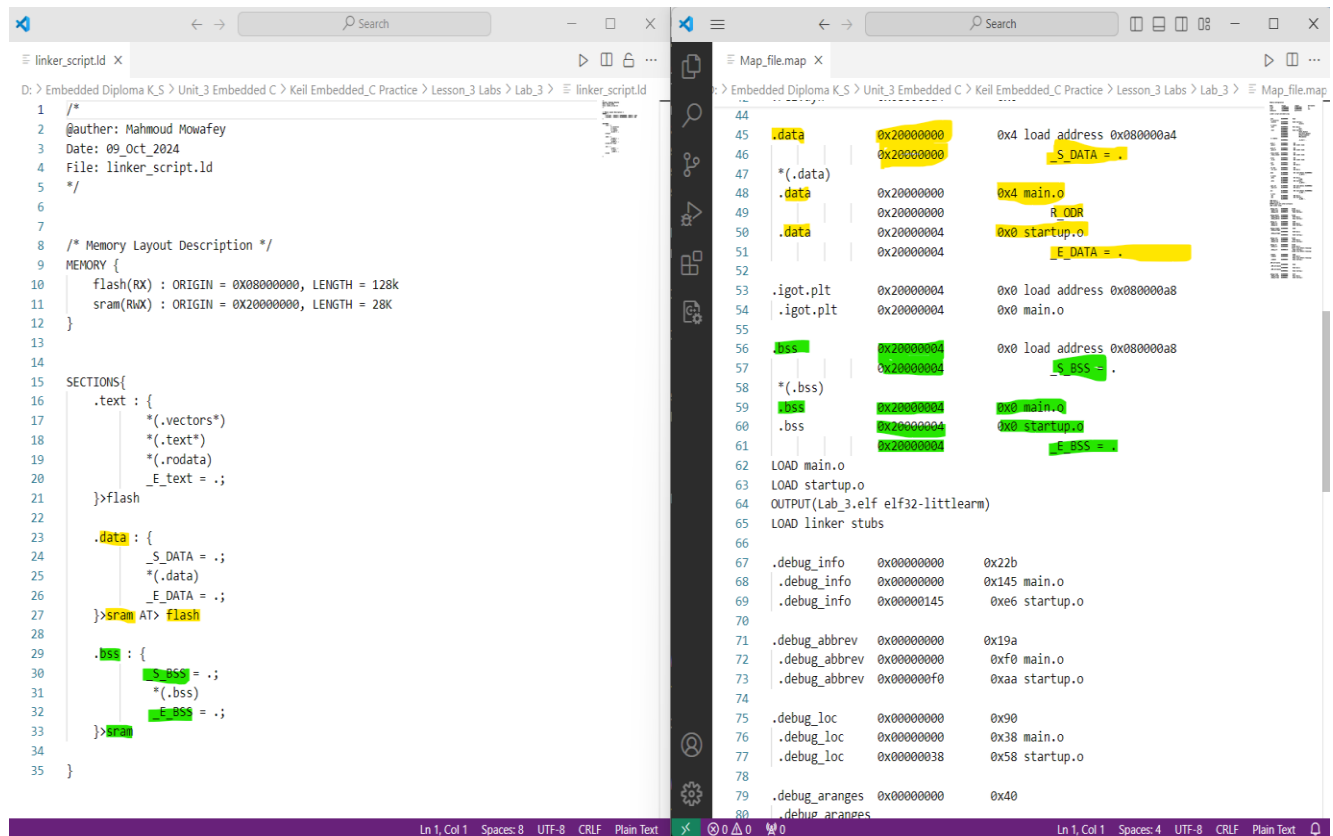
Name	Origin	Length	Attributes
flash	0x08000000	0x00020000	XR
sram	0x20000000	0x00007000	XRW
default	0x00000000	0xffffffff	

Linker script and memory map

Section	Start Address	End Address	Symbol
.text	0x08000000	0x08000004	
(.vectors)	0x08000000	0x08000004	
.vectors	0x08000000	0x08000004	0x10 startup.o
(.text)	0x08000000	0x08000004	vectors
.text	0x08000010	0x08000014	0x7c main.o
*(.rodata)	0x08000010	0x08000014	main
.text	0x08000018	0x0800001c	0x18 startup.o
*(.rodata)	0x08000018	0x0800001c	Reset_Handler
.text	0x0800001c	0x08000020	Bus_Fault_Handler
*(.rodata)	0x0800001c	0x08000020	Default_Handler
.text	0x08000020	0x08000024	NMI_Handler
*(.rodata)	0x08000020	0x08000024	E_TEXT =
.glue_7	0x080000a4	0x080000a4	0x0
.glue_7	0x080000a4	0x080000a4	0x0 linker stubs
.glue_7t	0x080000a4	0x080000a4	0x0
.glue_7t	0x080000a4	0x080000a4	0x0 linker stubs
.vfp11_veneer	0x080000a4	0x080000a4	0x0
.vfp11_veneer	0x080000a4	0x080000a4	0x0 linker stubs
.v4_bx	0x080000a4	0x080000a4	0x0
.v4_bx	0x080000a4	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 beginning 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 uninitialized 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.

The screenshot shows three windows from an IDE:

- linker_script.ld**: Contains memory layout and section definitions. The `SECTIONS` block defines `.text`, `.data`, `.bss`, and `.E_BSS` sections. `.bss` is at `0x20000004` and `.E_BSS` is at `0x20000007`.
- Map_file.map**: Shows the memory map. It lists sections and their addresses. `.bss` is at `0x20000004` and `.E_BSS` is at `0x20000007`. There is a gap between them.
- C main.c**: Contains C code for initializing variables and setting up the clock. It defines `RCC_IOPEN`, `GPIOA13`, and `RCC_APB2ENR`. It also defines `R_ODR_t` and `R_ODR_t` as a union of `uint32_t` and `uint32_t`.

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.

The screenshot shows the same three windows as before, but with modifications to the linker script and C code to fix the alignment issue:

- linker_script.ld**: The `SECTIONS` block now includes `.fill 0x00000001` after `.bss` to align the `_E_BSS` section. The `.bss` section is now at `0x20000004` and `.E_BSS` is at `0x20000008`.
- Map_file.map**: Shows the updated memory map. `.bss` is at `0x20000004` and `.E_BSS` is at `0x20000008`. The gap between them is now filled with zeros.
- C main.c**: The C code remains the same, but the `R_ODR_t` union is now defined as `uint32_t` and `uint32_t`.

4. Startup code in C Language

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

CM3 Source Code - U1

```

volatile R_ODR_t *R_ODR = (volatile R_ODR_t *) (GPIOA_BASE + GPIOA_ODR_OFFSET);
// unsigned char g_var[3] = {1, 2, 3};
// unsigned char const c_var[3] = {1, 2, 3};

int main(void)
{
    // init the clock for GPIOA
    // IOPAEN: IO port A clock enable
    // set and cleared by software.
    // 0: IO port A clock disabled
    // 1: IO port A clock enabled
    RCC_APB2ENR |= RCC_IOPAEN;

    GPIOA_CRH = 0x0FFFFFFF;
    GPIOA_CRH |= 0x00200000;
    // Loop Forever
    while (1)
    {
        // GPIOA_ODR |= (1 << 13);
        R_ODR->Pin_P13 = 1;
        for (int i = 0; i < 5000; i++)
        {
            // GPIOA_ODR &= ~(1 << 13);
            R_ODR->Pin_P13 = 0;
            for (int i = 0; i < 5000; i++)
            {
            }
        }
    }

    typedef volatile unsigned int vuint32_t;
    #include <stdint.h>
    #include <stdio.h>
    #include <stdlib.h>

    // register address
    #define GPIOA_BASE 0x40010800
    #define GPIOA_CRH_OFFSET 0x04
    #define GPIOA_CRH (*(vuint32_t *) (GPIOA_BASE + GPIOA_CRH_OFFSET))
    #define GPIOA_ODR (*(vuint32_t *) (GPIOA_BASE + GPIOA_ODR_OFFSET))

```

CM3 Variables - U1

Name	Address	Value
R_ODR	08000004	0x4001080C
g_var	4001080C	0x00 0...
c_var	4001080C	0x00 0...
Pin	4001080C	0x00 0...
BP12	R20000FD4	5000
BP16	R20000FD0	5000

CM3 Registers - U1

PC	08000008	Privileged
PR1	FE.FFFFFFFF	Mode Handler
R0	00000000	R7 20000FD0
R1	00000000	R8 00000000
R2	4001080C	R9 00000000
R3	00000000	R10 00000000
R4	00000000	R11 00000000
R5	00000000	R12 00000000
R6	00000000	LR 08000001
PCSPR	20000FD0	PSP 00000000
IRQ	3	EPSR 01000000
APSR	CNOVZ	10000
		ICIT 000.0000

CM3 Source Code - U1

```

volatile R_ODR_t *R_ODR = (volatile R_ODR_t *) (GPIOA_BASE + GPIOA_ODR_OFFSET);
// unsigned char g_var[3] = {1, 2, 3};
// unsigned char const c_var[3] = {1, 2, 3};

int main(void)
{
    // init the clock for GPIOA
    // IOPAEN: IO port A clock enable
    // set and cleared by software.
    // 0: IO port A clock disabled
    // 1: IO port A clock enabled
    RCC_APB2ENR |= RCC_IOPAEN;

    GPIOA_CRH = 0x0FFFFFFF;
    GPIOA_CRH |= 0x00200000;
    // Loop Forever
    while (1)
    {
        // GPIOA_ODR |= (1 << 13);
        R_ODR->Pin_P13 = 1;
        for (int i = 0; i < 5000; i++)
        {
            // GPIOA_ODR &= ~(1 << 13);
            R_ODR->Pin_P13 = 0;
            for (int i = 0; i < 5000; i++)
            {
            }
        }
    }

    typedef volatile unsigned int vuint32_t;
    #include <stdint.h>
    #include <stdio.h>
    #include <stdlib.h>

    // register address
    #define GPIOA_BASE 0x40010800
    #define GPIOA_CRH_OFFSET 0x04
    #define GPIOA_CRH (*(vuint32_t *) (GPIOA_BASE + GPIOA_CRH_OFFSET))
    #define GPIOA_ODR (*(vuint32_t *) (GPIOA_BASE + GPIOA_ODR_OFFSET))

```

CM3 Variables - U1

Name	Address	Value
R_ODR	08000004	0x4001080C
g_var	4001080C	0x00 0...
c_var	4001080C	0x00 0...
Pin	4001080C	0x00 0...
BP12	R20000FD4	5000
BP16	R20000FD0	5000

CM3 Registers - U1

PC	08000008	Privileged
PR1	FE.FFFFFFFF	Mode Handler
R0	00000000	R7 20000FD0
R1	00000000	R8 00000000
R2	4001080C	R9 00000000
R3	00000000	R10 00000000
R4	00000000	R11 00000000
R5	00000000	R12 00000000
R6	00000000	LR 08000001
PCSPR	20000FD0	PSP 00000000
IRQ	3	EPSR 01000000
APSR	CNOVZ	10000
		ICIT 000.0000

