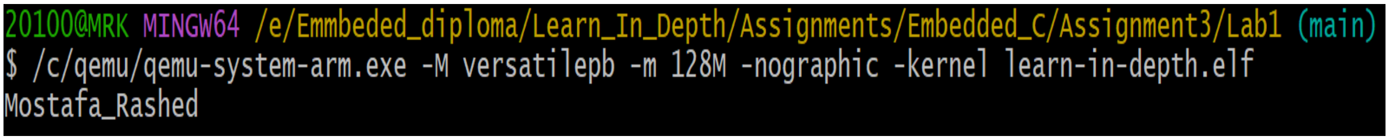
Report

Lab1(gdb&Makefile)

Lab2(Startup.s&Startup.c)

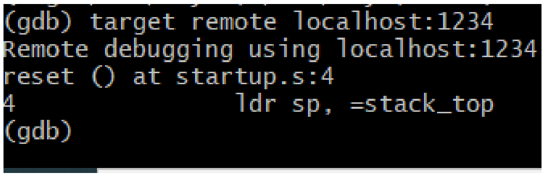
Lab1(gdb&Makefile) :

- First we will open gdb circuit in qemo tool for board that we debug on called versatilepb using this command :



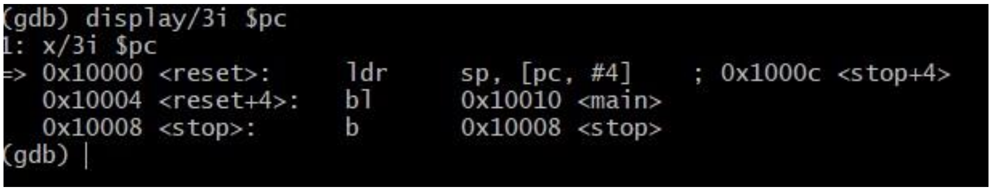
```
20100@MRK MINGW64 /e/Emmbeded_diploma/Learn_In_Depth/Assignments/Embedded_C/Assignment3/Lab1 (main)
$ /c/qemu/qemu-system-arm.exe -M versatilepb -m 128M -nographic -kernel learn-in-depth.elf
Mostafa_Rashed
```

- As we know to connect to gdb server on board you must have IP address and port number
- In our case we use qemu tool to virtually debug our code so the IP address will be our localhost address and port number is :1234



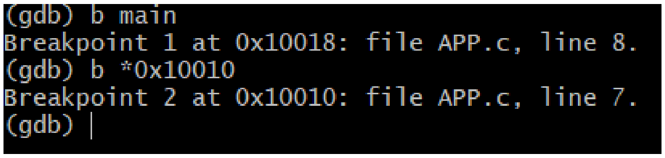
```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
reset () at startup.s:4
4               ldr sp, =stack_top
(gdb)
```

- There is command show us 3 assembly instructions starting with line we stand , the arrow points to reset symbol in startup.s file :



```
(gdb) display/3i $pc
1: x/3i $pc
=> 0x10000 <reset>:     ldr     sp, [pc, #4]    ; 0x1000c <stop+4>
   0x10004 <reset+4>:   bl      0x10010 <main>
   0x10008 <stop>:      b       0x10008 <stop>
(gdb) |
```

If we want to make breaking point at main
The main function at address 0x10010 :

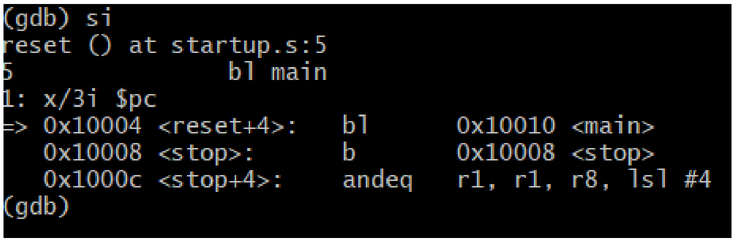


```
(gdb) b main
Breakpoint 1 at 0x10018: file APP.c, line 8.
(gdb) b *0x10010
Breakpoint 2 at 0x10010: file APP.c, line 7.
(gdb) |
```

We found out that real address of main symbol is at 0x10018

Notice : the address of 0x10010 is related with context instructions it is about creating stack and store PC in lR

- If we want to step one instruction in assembly we can use “si” command but if we debug in C level we can use “s” command that step one C line that may contains many assembly instructions :

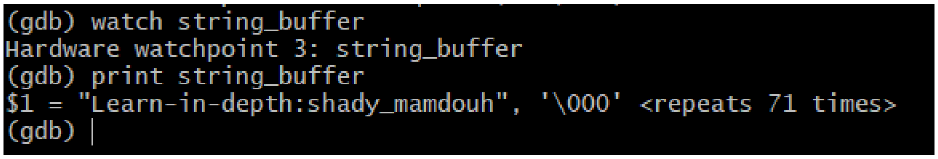


```
(gdb) si
reset () at startup.s:5
5               bl main
1: x/3i $pc
=> 0x10004 <reset+4>:   bl      0x10010 <main>
   0x10008 <stop>:      b       0x10008 <stop>
   0x1000c <stop+4>:    andeq   r1, r1, r8, lsl #4
(gdb)
```

-if we want to print a specific variable we can use

“ print var_name “ .

-If we want to watch a specific variable that debugger will stand if their value has been changed , we can use command “watch var_name” :



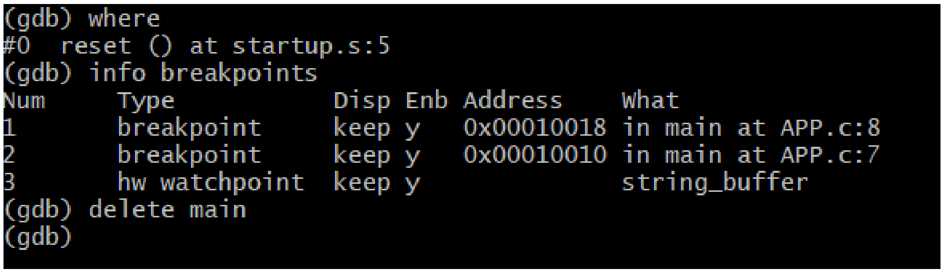
```
(gdb) watch string_buffer
Hardware watchpoint 3: string_buffer
(gdb) print string_buffer
$1 = "Learn-in-depth:shady_mamdouh", '\000' <repeats 71 times>
(gdb) |
```

-If we want to know where are we , we can use this command “where”

-if we want to know information about breaking points and their number we use command “info breakpoints”

-if we want to delete some breakpoint we can use

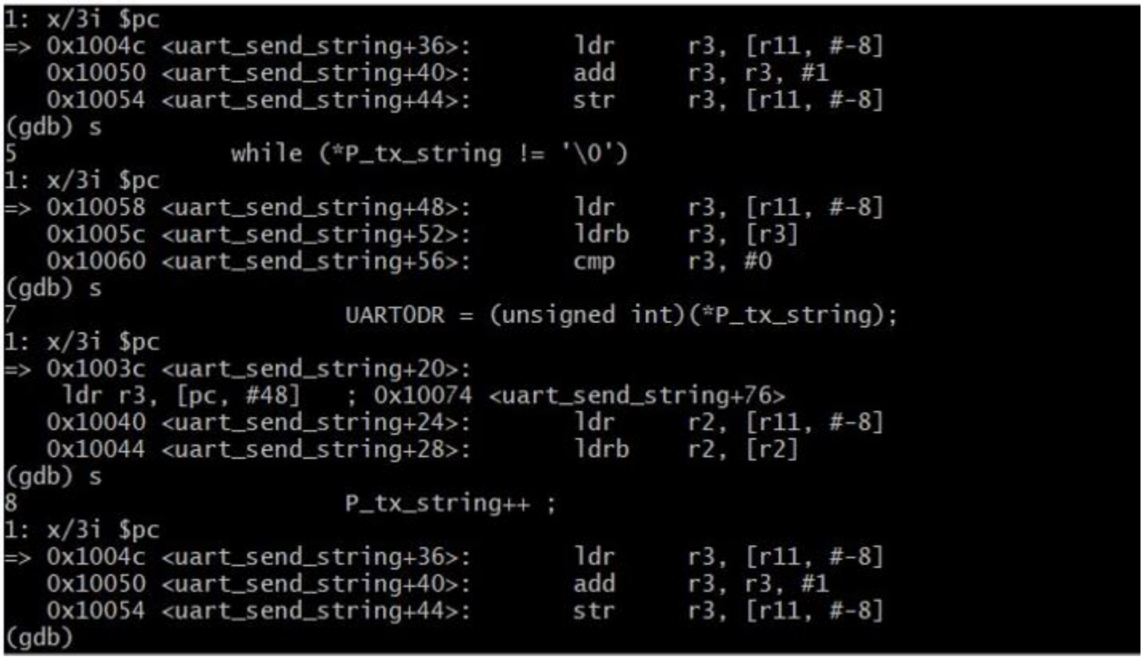
“delete b_name” :



```
(gdb) where
#0  reset () at startup.s:5
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010018 in main at APP.c:8
2       breakpoint     keep y   0x00010010 in main at APP.c:7
3       hw watchpoint  keep y              string_buffer
(gdb) delete main
(gdb)
```

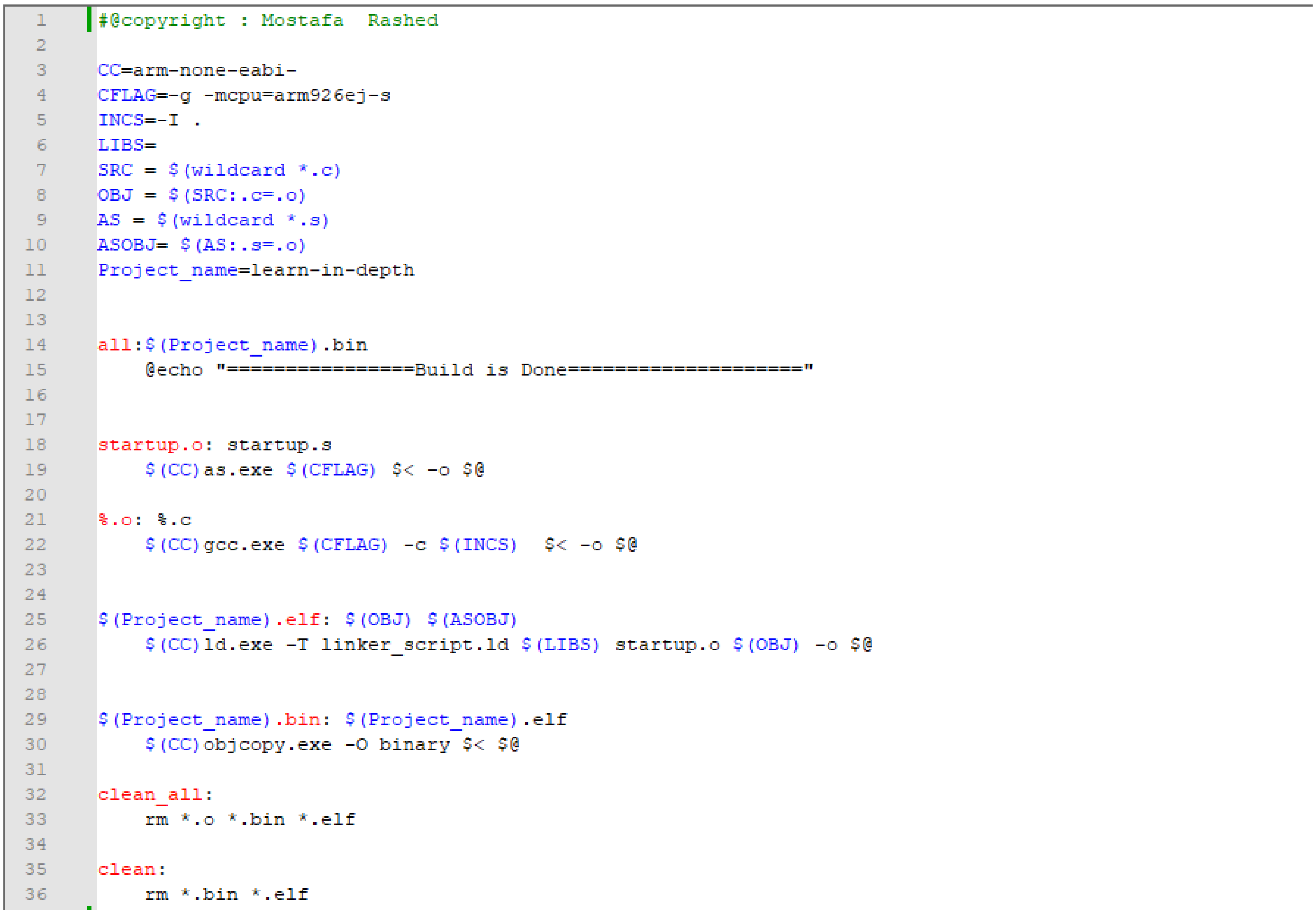
- If we want to tell gdb to continue till closest breaking point We can use command “c”

- We will step in C until uart.c and we will find that the string will printed character by character on the qemo terminal :



```
1: x/3i $pc
=> 0x1004c <uart_send_string+36>:       ldr     r3, [r11, #-8]
   0x10050 <uart_send_string+40>:       add     r3, r3, #1
   0x10054 <uart_send_string+44>:       str     r3, [r11, #-8]
(gdb) s
5               while (*P_tx_string != '\0')
1: x/3i $pc
=> 0x10058 <uart_send_string+48>:       ldr     r3, [r11, #-8]
   0x1005c <uart_send_string+52>:       ldrb    r3, [r3]
   0x10060 <uart_send_string+56>:       cmp     r3, #0
(gdb) s
7                       UART0DR = (unsigned int)(*P_tx_string);
1: x/3i $pc
=> 0x1003c <uart_send_string+20>:
    ldr r3, [pc, #48]   ; 0x10074 <uart_send_string+76>
   0x10040 <uart_send_string+24>:       ldr     r2, [r11, #-8]
   0x10044 <uart_send_string+28>:       ldrb    r2, [r2]
(gdb) s
8                       P_tx_string++ ;
1: x/3i $pc
=> 0x1004c <uart_send_string+36>:       ldr     r3, [r11, #-8]
   0x10050 <uart_send_string+40>:       add     r3, r3, #1
   0x10054 <uart_send_string+44>:       str     r3, [r11, #-8]
(gdb)
```

Makefile of lab 1 :



```
#@copyright : Mostafa  Rashed

CC=arm-none-eabi-
CFLAG=-g -mcpu=arm926ej-s
INCS=-I .
LIBS=
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
AS = $(wildcard *.s)
ASOBJ= $(AS:.s=.o)
Project_name=learn-in-depth


all:$(Project_name).bin
    @echo "================Build is Done===================="


startup.o: startup.s
    $(CC)as.exe $(CFLAG) $< -o $@

%.o: %.c
    $(CC)gcc.exe $(CFLAG) -c $(INCS)  $< -o $@


$(Project_name).elf: $(OBJ) $(ASOBJ)
    $(CC)ld.exe -T linker_script.ld $(LIBS) startup.o $(OBJ) -o $@


$(Project_name).bin: $(Project_name).elf
    $(CC)objcopy.exe -O binary $< $@

clean_all:
    rm *.o *.bin *.elf

clean:
    rm *.bin *.elf
```

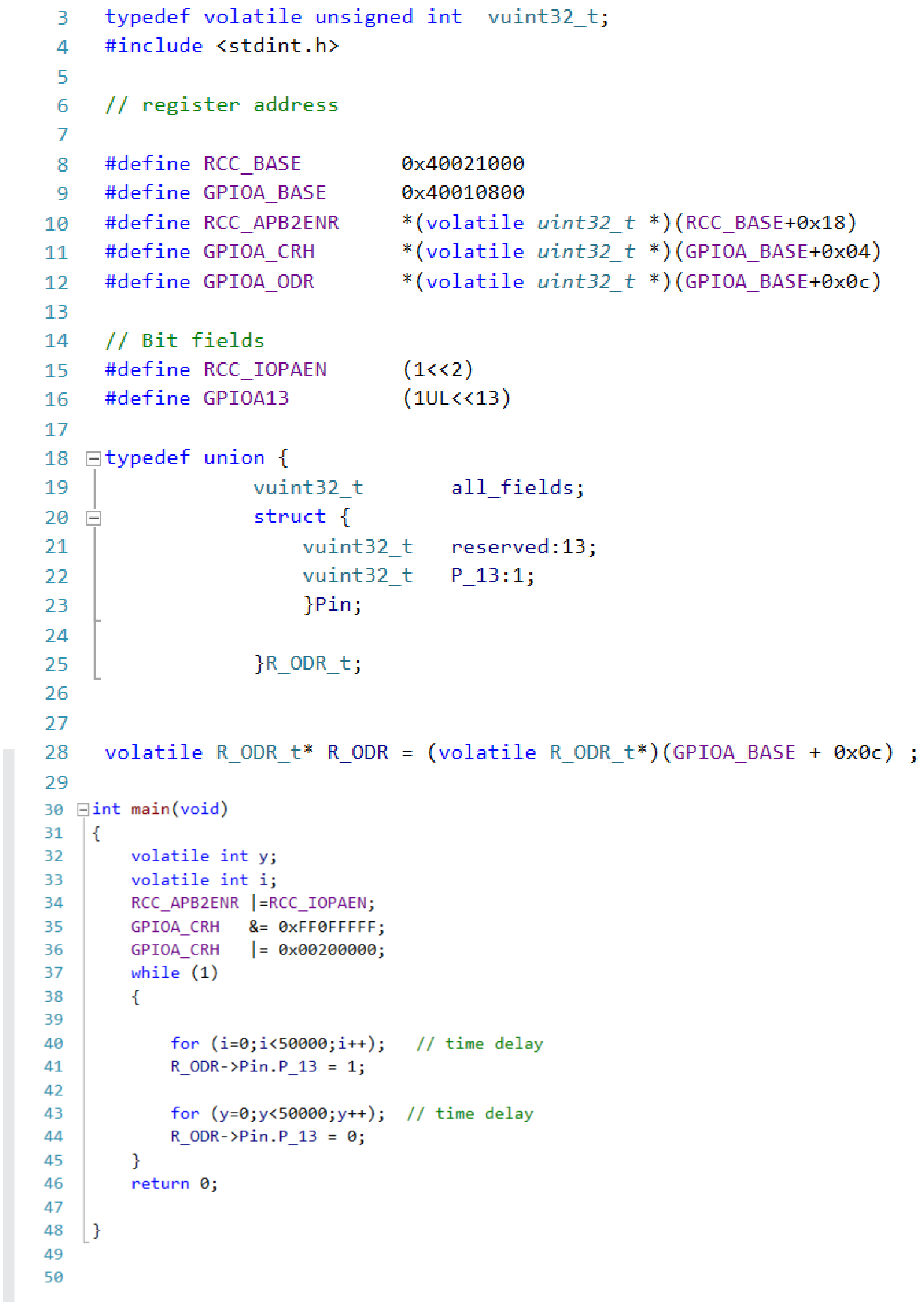
Lab2(Startup.s&Startup.c)

Startup.s

Board name : STM32f103c8t6

Notice: Entry point of this cortex-m3 based is 0x0800000

It must contain SP value of address that points to in sram main.c :



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

// register address

#define RCC_BASE        0x40021000
#define GPIOA_BASE      0x40010800
#define RCC_APB2ENR     *(volatile uint32_t *)(RCC_BASE+0x18)
#define GPIOA_CRH       *(volatile uint32_t *)(GPIOA_BASE+0x04)
#define GPIOA_ODR       *(volatile uint32_t *)(GPIOA_BASE+0x0c)

// Bit fields
#define RCC_IOPAEN      (1<<2)
#define GPIOA13         (1UL<<13)

typedef union {
            vuint32_t       all_fields;
            struct {
                vuint32_t   reserved:13;
                vuint32_t   P_13:1;
                }Pin;

            }R_ODR_t;


volatile R_ODR_t* R_ODR = (volatile R_ODR_t*)(GPIOA_BASE + 0x0c) ;

int main(void)
{
    volatile int y;
    volatile int i;
    RCC_APB2ENR |=RCC_IOPAEN;
    GPIOA_CRH   &= 0xFF0FFFFF;
    GPIOA_CRH   |= 0x00200000;
    while (1)
    {

        for (i=0;i<50000;i++);   // time delay
        R_ODR->Pin.P_13 = 1;

        for (y=0;y<50000;y++);  // time delay
        R_ODR->Pin.P_13 = 0;
    }
    return 0;

}
```

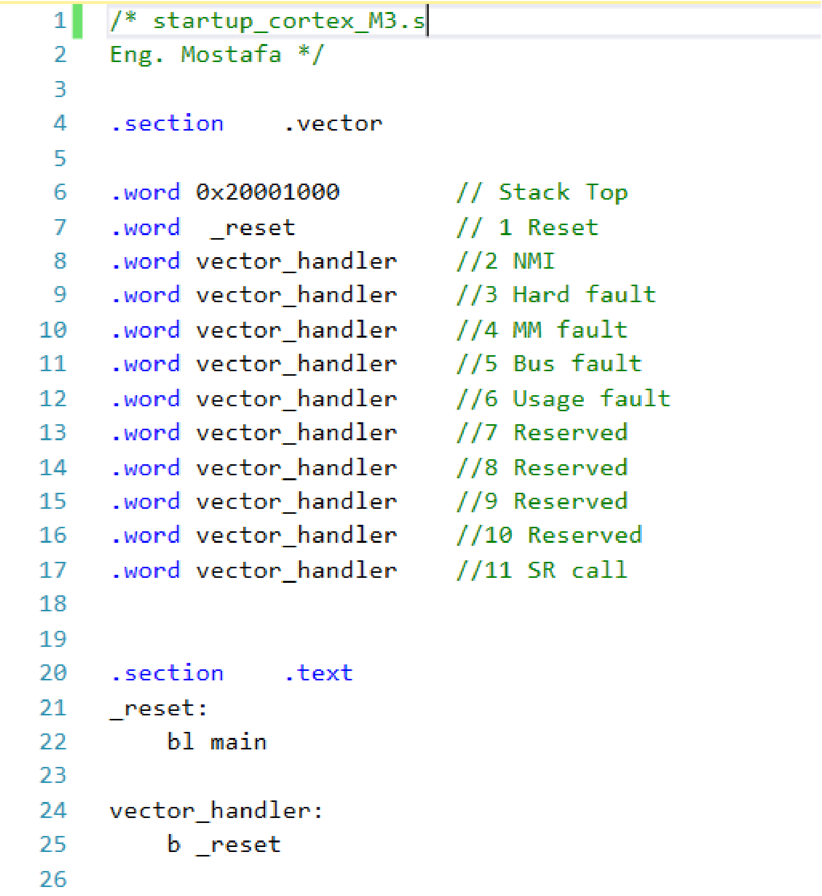
Startup.s :

We gave command to assembler to make section called vectors

And we defined first word as a value of SP is 0x20001000

Within range of sram

According to specs the interrupt vector table must start after SP assigning , so we make vector_handler to handle any interrupt



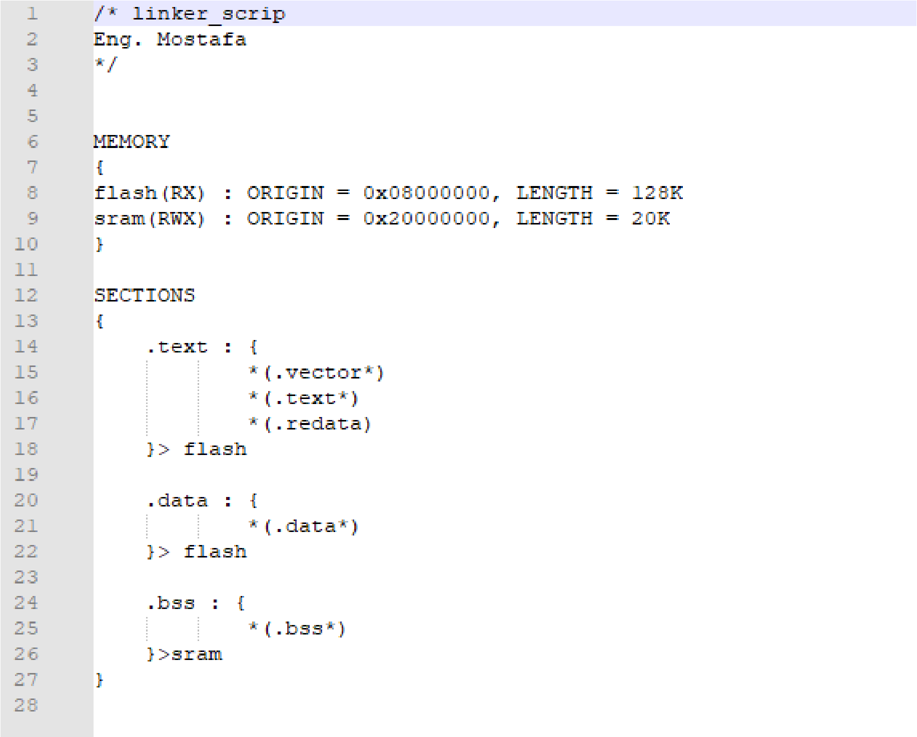
```
/* startup_cortex_M3.s
Eng. Mostafa */

.section    .vector

.word 0x20001000        // Stack Top
.word  _reset           // 1 Reset
.word vector_handler    //2 NMI
.word vector_handler    //3 Hard fault
.word vector_handler    //4 MM fault
.word vector_handler    //5 Bus fault
.word vector_handler    //6 Usage fault
.word vector_handler    //7 Reserved
.word vector_handler    //8 Reserved
.word vector_handler    //9 Reserved
.word vector_handler    //10 Reserved
.word vector_handler    //11 SR call


.section    .text
_reset:
    bl main

vector_handler:
    b _reset

```

Linker script :

According to specs flash memory starts with 0x08000000

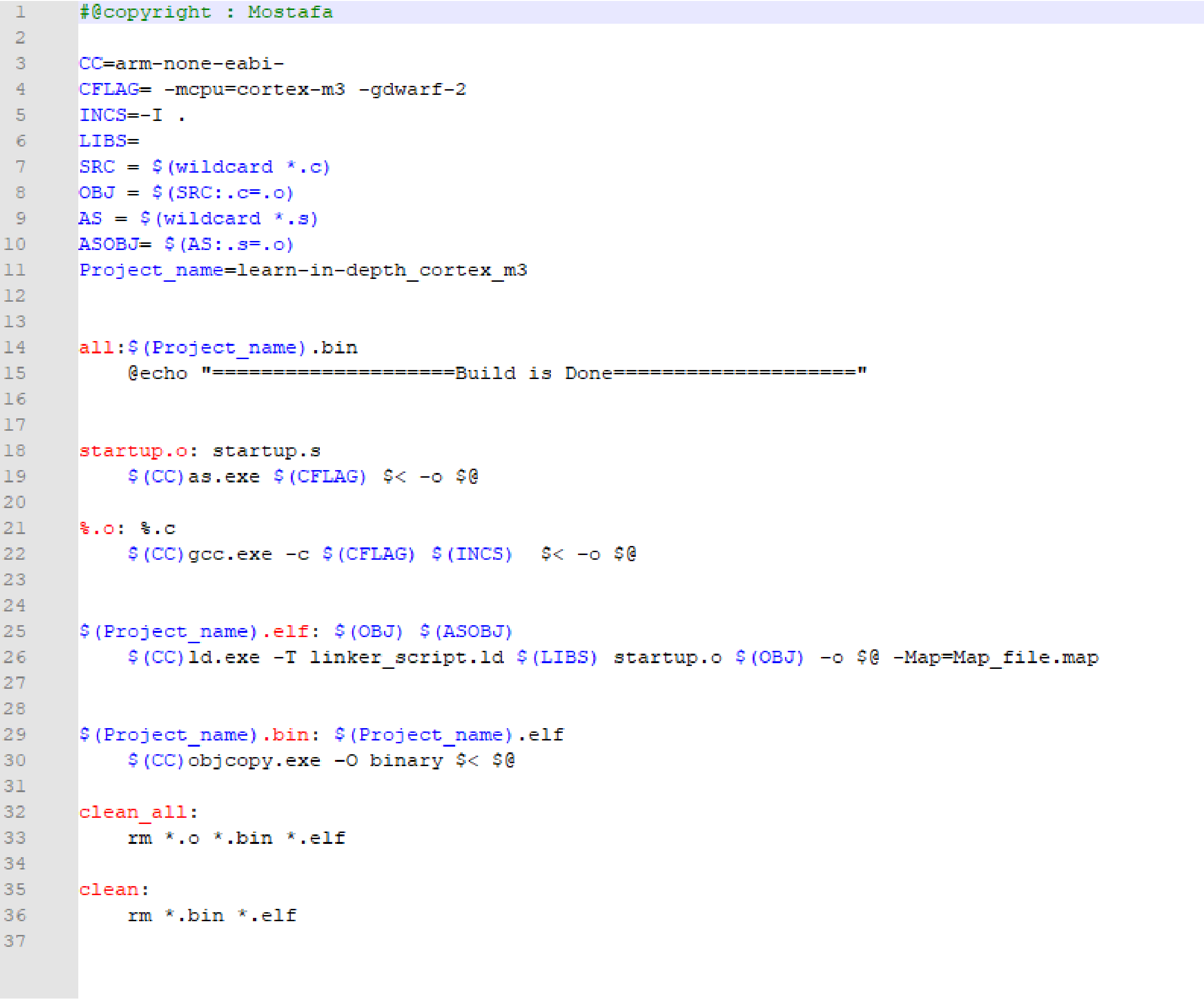
And sram starts with 0x20000000

-we make vector section at the start of sections to be located at the start of flash memory



```
/* linker_scrip
Eng. Mostafa
*/


MEMORY
{
flash(RX) : ORIGIN = 0x08000000, LENGTH = 128K
sram(RWX) : ORIGIN = 0x20000000, LENGTH = 20K
}

SECTIONS
{
    .text : {
            *(.vector*)
            *(.text*)
            *(.redata)
    }> flash

    .data : {
            *(.data*)
    }> flash

    .bss : {
            *(.bss*)
    }>sram
}

```

Make file : somethings will be edited compared with lab1 such as project name and board name :



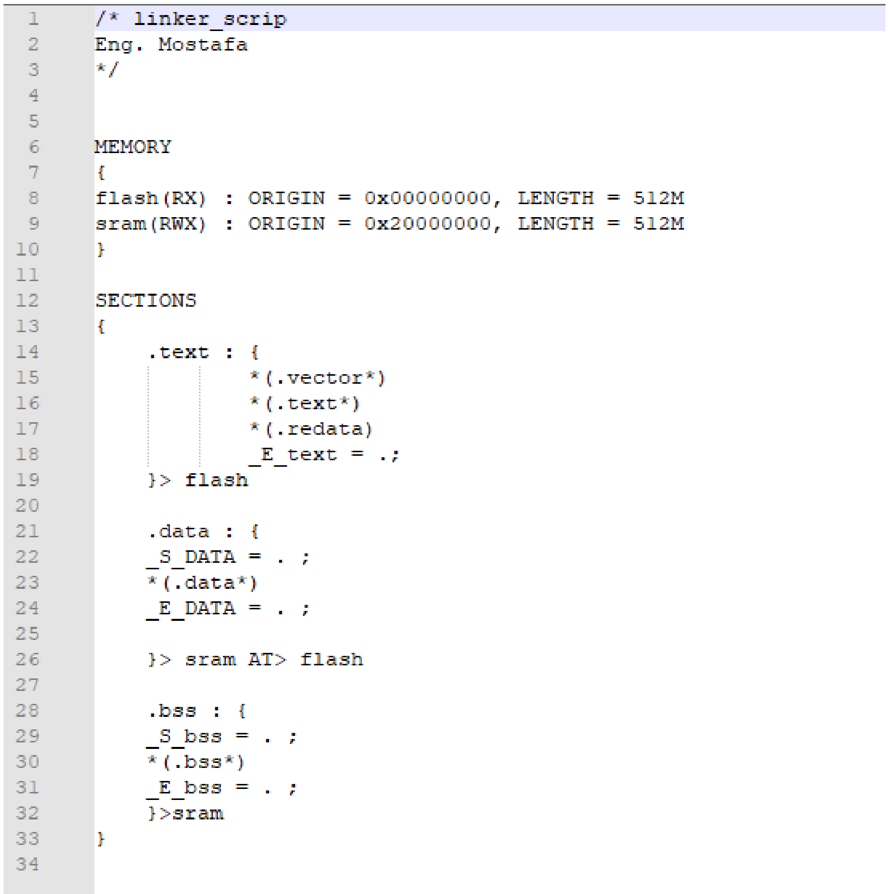
```
#@copyright : Mostafa

CC=arm-none-eabi-
CFLAG= -mcpu=cortex-m3 -gdwarf-2
INCS=-I .
LIBS=
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
AS = $(wildcard *.s)
ASOBJ= $(AS:.s=.o)
Project_name=learn-in-depth_cortex_m3


all:$(Project_name).bin
    @echo "====================Build is Done===================="


startup.o: startup.s
    $(CC)as.exe $(CFLAG) $< -o $@

%.o: %.c
    $(CC)gcc.exe -c $(CFLAG) $(INCS)  $< -o $@


$(Project_name).elf: $(OBJ) $(ASOBJ)
    $(CC)ld.exe -T linker_script.ld $(LIBS) startup.o $(OBJ) -o $@ -Map=Map_file.map


$(Project_name).bin: $(Project_name).elf
    $(CC)objcopy.exe -O binary $< $@

clean_all:
    rm *.o *.bin *.elf

clean:
    rm *.bin *.elf

```

Lab2,part2

Startup.c

- As we mentioned before the reason that stop you from coding Startup.c is initializing stack because c codes use stack , so some boards have a feature allow you to initialize stack with just write the address that you want SP to point in the entry point of processor
- Board name : STM32f103c8t6 arm-cortex-m3 based .
- Flash starts with 0x08000000
- Sram starts with 0x20000000
- We want to make . text section starts with start of flash And contains . vectors section as a first section then other .text sections from all files
- .vectors section will contain SP and interrupt vector table So the first symbol in .vectors will be relative to the start of flash memory as we target .
- We want to copy .data section from flash to sram and initialize .bss section in sram.
- In linker script we will define some variables to make memory boundary at start and end of each section to help us to calculate the size of sections and to copy .data and create .bss in sram

Linker script :



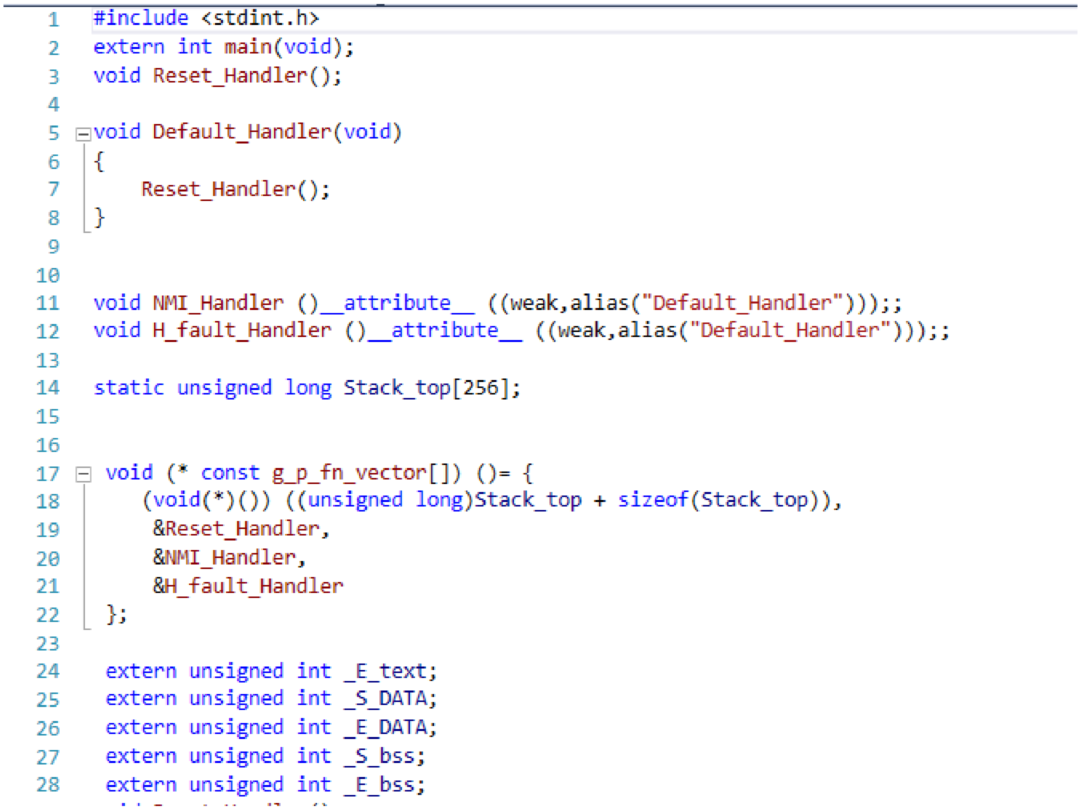
```
/* linker_scrip
Eng. Mostafa
*/


MEMORY
{
flash(RX) : ORIGIN = 0x00000000, LENGTH = 512M
sram(RWX) : ORIGIN = 0x20000000, LENGTH = 512M
}

SECTIONS
{
    .text : {
            *(.vector*)
            *(.text*)
            *(.redata)
            _E_text = .;
    }> flash

    .data : {
    _S_DATA = . ;
    *(.data*)
    _E_DATA = . ;

    }> sram AT> flash

    .bss : {
    _S_bss = . ;
    *(.bss*)
    _E_bss = . ;
    }>sram
}
```

- We made padding by 0x1000 memory locations in sram between .bss and stack top that will be used to create function stacks to avoid any crash .

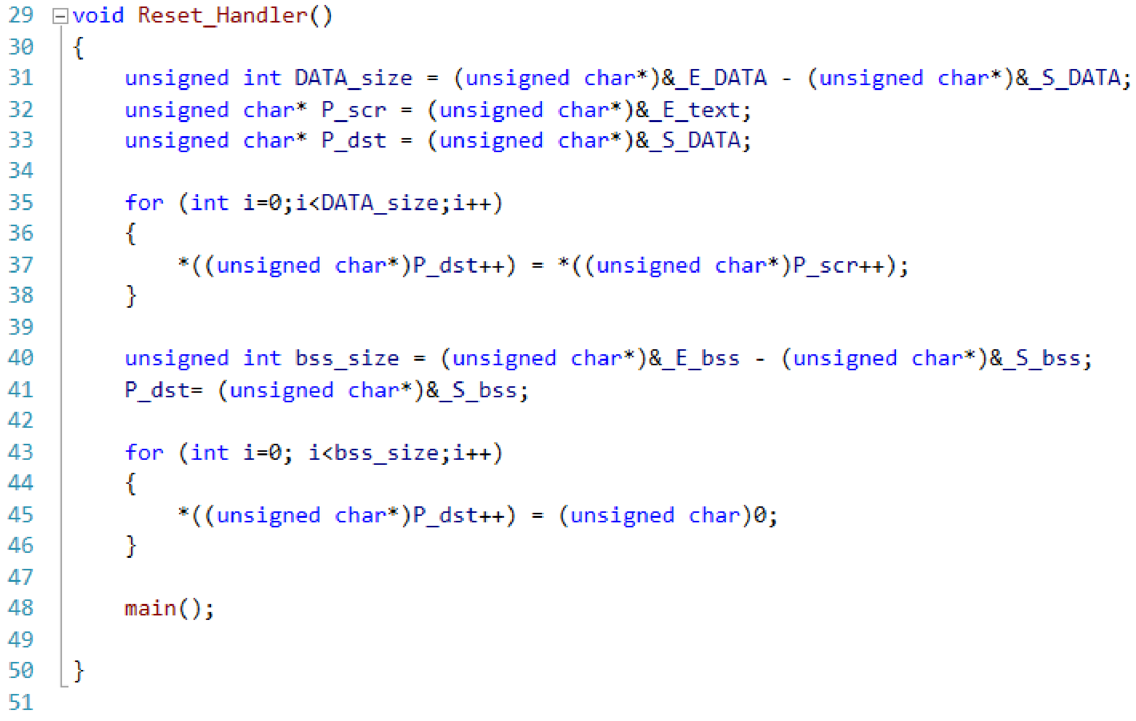
Starup.c :

- We use attribute to pass commands to compiler to create section called .vectors and we make array of addresses that we want to be in this section
  This addresses represent SP and all interrupts vector table

- We use attribute of weak and alias vector handler to make all vectors point to default symbol and allow user to override with his own handler



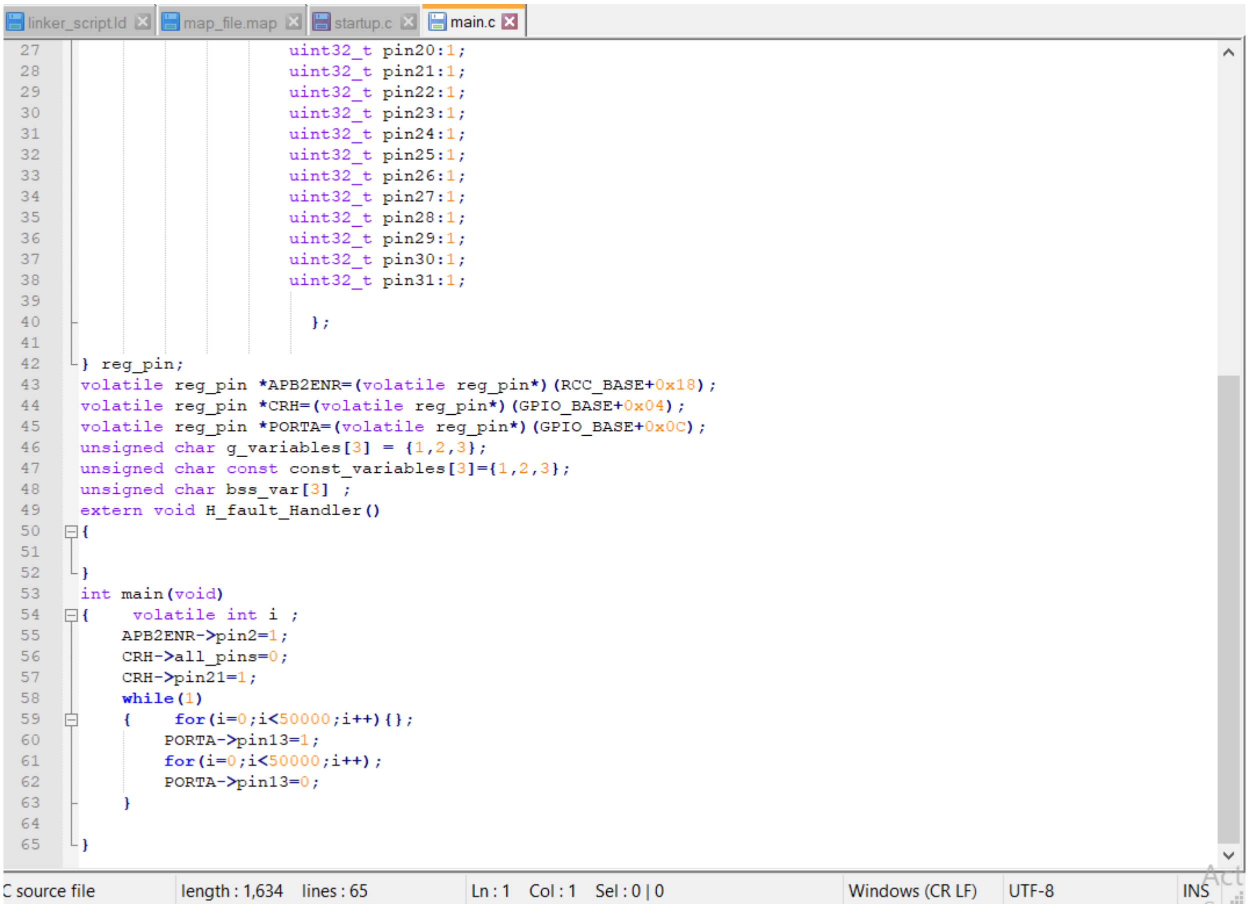
```c
#include <stdint.h>
extern int main(void);
void Reset_Handler();

void Default_Handler(void)
{
    Reset_Handler();
}


void NMI_Handler ()__attribute__ ((weak,alias("Default_Handler")));;
void H_fault_Handler ()__attribute__ ((weak,alias("Default_Handler")));;

static unsigned long Stack_top[256];


 void (* const g_p_fn_vector[]) ()= {
    (void(*)()) ((unsigned long)Stack_top + sizeof(Stack_top)),
     &Reset_Handler,
     &NMI_Handler,
     &H_fault_Handler
 };

 extern unsigned int _E_text;
 extern unsigned int _S_DATA;
 extern unsigned int _E_DATA;
 extern unsigned int _S_bss;
 extern unsigned int _E_bss;
```



```c
void Reset_Handler()
{
    unsigned int DATA_size = (unsigned char*)&_E_DATA - (unsigned char*)&_S_DATA;
    unsigned char* P_scr = (unsigned char*)&_E_text;
    unsigned char* P_dst = (unsigned char*)&_S_DATA;

    for (int i=0;i<DATA_size;i++)
    {
        *((unsigned char*)P_dst++) = *((unsigned char*)P_scr++);
    }

    unsigned int bss_size = (unsigned char*)&_E_bss - (unsigned char*)&_S_bss;
    P_dst= (unsigned char*)&_S_bss;

    for (int i=0; i<bss_size;i++)
    {
        *((unsigned char*)P_dst++) = (unsigned char)0;
    }

    main();

}
```

Main.c :

- In main we defined H_fault_handler() to prove concept of overriding the default symbol and change the symbol address
- We defined uninitialized global variable to represent .bss section



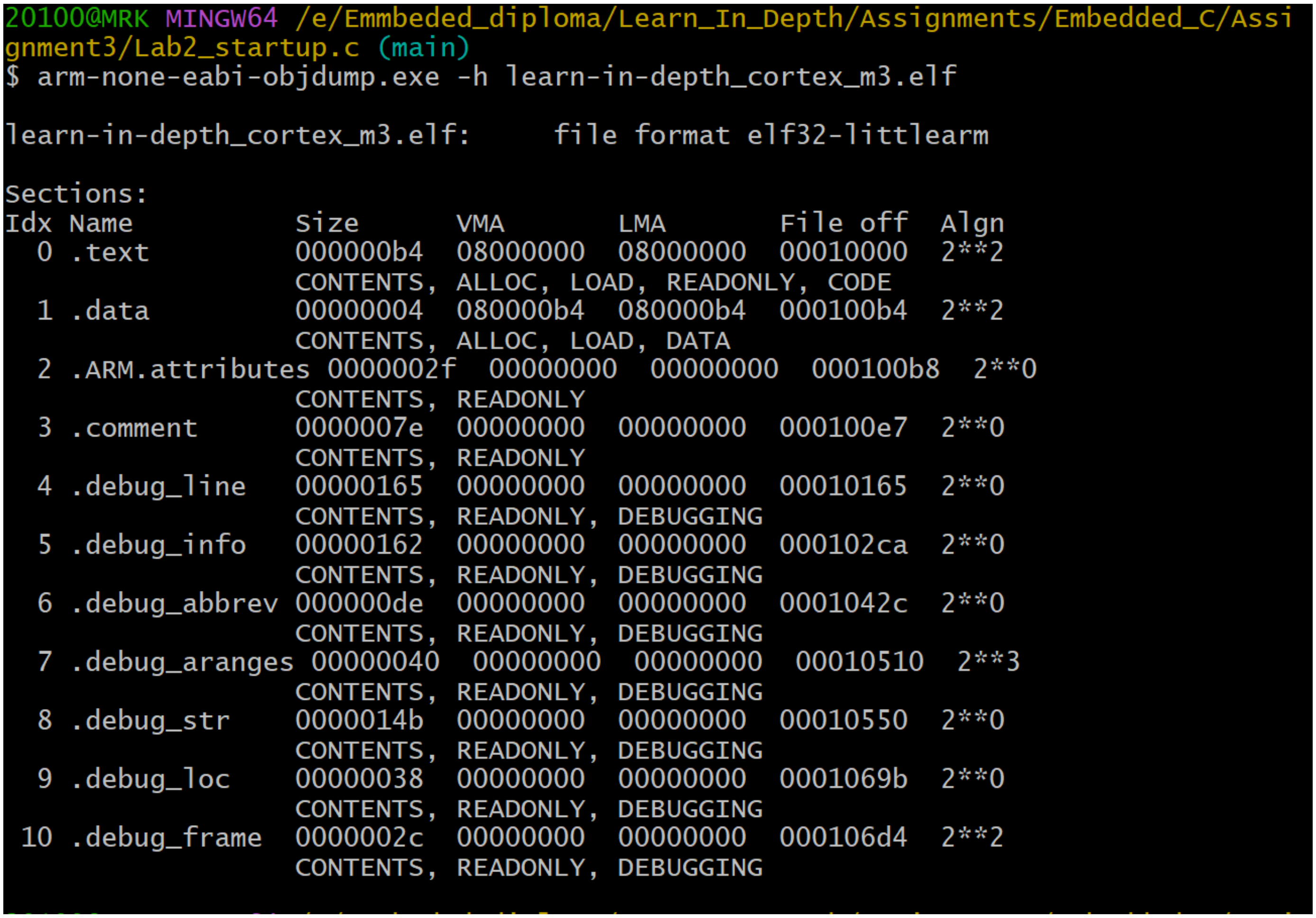
```c
                        uint32_t pin20:1;
                        uint32_t pin21:1;
                        uint32_t pin22:1;
                        uint32_t pin23:1;
                        uint32_t pin24:1;
                        uint32_t pin25:1;
                        uint32_t pin26:1;
                        uint32_t pin27:1;
                        uint32_t pin28:1;
                        uint32_t pin29:1;
                        uint32_t pin30:1;
                        uint32_t pin31:1;

                          };

} reg_pin;
volatile reg_pin *APB2ENR=(volatile reg_pin*)(RCC_BASE+0x18);
volatile reg_pin *CRH=(volatile reg_pin*)(GPIO_BASE+0x04);
volatile reg_pin *PORTA=(volatile reg_pin*)(GPIO_BASE+0x0C);
unsigned char g_variables[3] = {1,2,3};
unsigned char const const_variables[3]={1,2,3};
unsigned char bss_var[3] ;
extern void H_fault_Handler()
{

}
int main(void)
{     volatile int i ;
    APB2ENR->pin2=1;
    CRH->all_pins=0;
    CRH->pin21=1;
    while(1)
    {     for(i=0;i<50000;i++){};
        PORTA->pin13=1;
        for(i=0;i<50000;i++);
        PORTA->pin13=0;
    }

}
```

C source file
length : 1,634   lines : 65
Ln : 1   Col : 1   Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

- lets make sure that everything is correct
  .text section has LMA equal VMA starts with 0x08000000
  As we want
  Because it hasn't been copied from flash to ram

- .data section has LMA within flash range and it will be copied to sram so it has VMA within start of sram as we want

- .bss section has VMA within sram range .

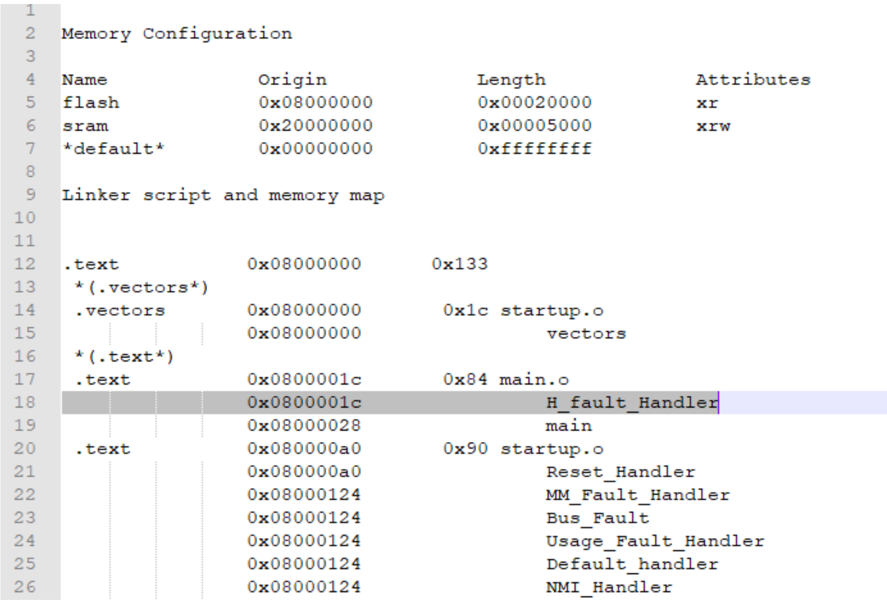


```
20100@MRK MINGW64 /e/Emmbeded_diploma/Learn_In_Depth/Assignments/Embedded_C/Assi
gnment3/Lab2_startup.c (main)
$ arm-none-eabi-objdump.exe -h learn-in-depth_cortex_m3.elf

learn-in-depth_cortex_m3.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000b4  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  080000b4  080000b4  000100b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .ARM.attributes 0000002f  00000000  00000000  000100b8  2**0
                  CONTENTS, READONLY
  3 .comment      0000007e  00000000  00000000  000100e7  2**0
                  CONTENTS, READONLY
  4 .debug_line   00000165  00000000  00000000  00010165  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_info   00000162  00000000  00000000  000102ca  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_abbrev 000000de  00000000  00000000  0001042c  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_aranges 00000040  00000000  00000000  00010510  2**3
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_str    0000014b  00000000  00000000  00010550  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_loc    00000038  00000000  00000000  0001069b  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_frame  0000002c  00000000  00000000  000106d4  2**2
                  CONTENTS, READONLY, DEBUGGING
```

Lets see map file to get more details :

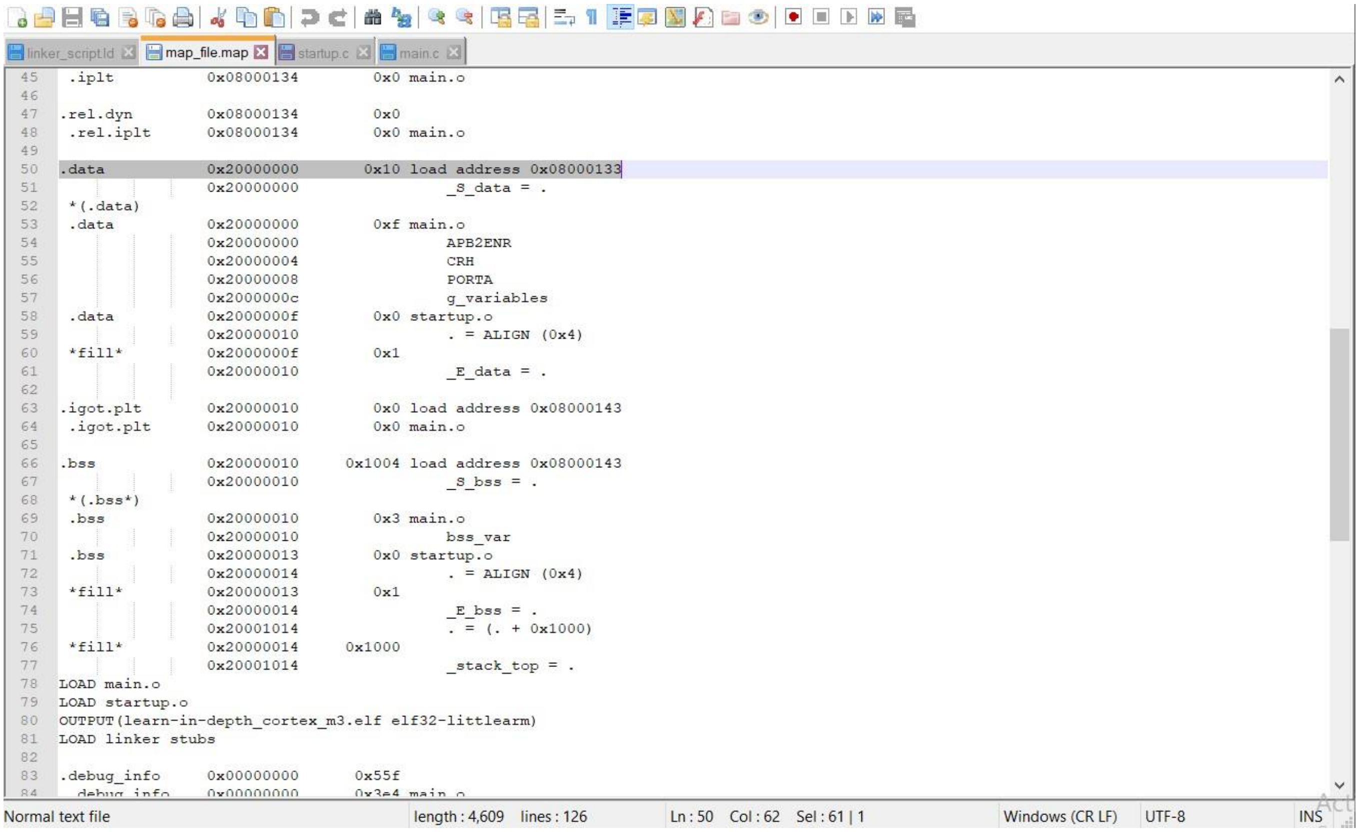
- H_fault_handler has address of 0x0800001c
  That is different from the default address of other handlers 0x08000124 to prove concept of overriding

- .vectors section at the start of flash



```

Memory Configuration

Name             Origin             Length             Attributes
flash            0x08000000         0x00020000         xr
sram             0x20000000         0x00005000         xrw
*default*        0x00000000         0xffffffff

Linker script and memory map


.text           0x08000000       0x133
 *(.vectors*)
 .vectors       0x08000000        0x1c startup.o
                0x08000000                 vectors
 *(.text*)
 .text          0x0800001c        0x84 main.o
                0x0800001c                 H_fault_Handler
                0x08000028                 main
 .text          0x080000a0        0x90 startup.o
                0x080000a0                 Reset_Handler
                0x08000124                 MM_Fault_Handler
                0x08000124                 Bus_Fault
                0x08000124                 Usage_Fault_Handler
                0x08000124                 Default_handler
                0x08000124                 NMI_Handler
```

- .data section has load address of 0x08000133 in flash and 0x20000000 at the start of sram as we want
- .bss section starts with 0x20000010 and end at 0x20000013 And there is memory aligning occurred with 1 byte



```
 .iplt          0x08000134        0x0 main.o

.rel.dyn        0x08000134        0x0
 .rel.iplt      0x08000134        0x0 main.o

.data           0x20000000       0x10 load address 0x08000133
                0x20000000                _S_data = .
 *(.data)
 .data          0x20000000        0xf main.o
                0x20000000                APB2ENR
                0x20000004                CRH
                0x20000008                PORTA
                0x2000000c                g_variables
 .data          0x2000000f        0x0 startup.o
                0x20000010                . = ALIGN (0x4)
 *fill*         0x2000000f        0x1
                0x20000010                _E_data = .

.igot.plt       0x20000010        0x0 load address 0x08000143
 .igot.plt      0x20000010        0x0 main.o

.bss            0x20000010     0x1004 load address 0x08000143
                0x20000010                _S_bss = .
 *(.bss*)
 .bss           0x20000010        0x3 main.o
                0x20000010                bss_var
 .bss           0x20000013        0x0 startup.o
                0x20000014                . = ALIGN (0x4)
 *fill*         0x20000013        0x1
                0x20000014                _E_bss = .
                0x20001014                . = (. + 0x1000)
 *fill*         0x20000014     0x1000
                0x20001014                _stack_top = .
LOAD main.o
LOAD startup.o
OUTPUT(learn-in-depth_cortex_m3.elf elf32-littlearm)
LOAD linker stubs

.debug_info     0x00000000      0x55f
 .debug_info    0x00000000      0x3e4 main.o
```