Lab1

-We will write a code from scratch to send string to uart0 and uart0 will display it on Board name : verastilepb (arm926ej-s) .

-This lab we will write c code, linker script and startup code and use all binary utilities such as objdump, objcopy, nm and readelf

-in this lab we will write the whole code using only arm-none-eabi tool chain without any IDE.

From specs we found out :

Entry point of processor is : 0x10000

To activate UART0 you just write on UART0DR register (32bit).

And its address is :0x101f1000

Codes :

App.c :

```c
#include "uart.h"


unsigned char string_buffer1[100]="learn-in-depth<Mostafa Rashed>";
unsigned char const string_buffer2[100]="learn-in-depth<Rashed Kamel>";


void main(void)
{
uart_send_string(string_buffer1);
}

```

Uart.c :

```c
#include "uart.h"

#define UART0DR *(( volatile unsigned int* const)((unsigned int*)0x101f1000))


void uart_send_string(unsigned char* p_tx )
{
    while(*p_tx !='\0')
    {
        UART0DR=(unsigned int)(*p_tx);
        p_tx++ ;
    }
}
```

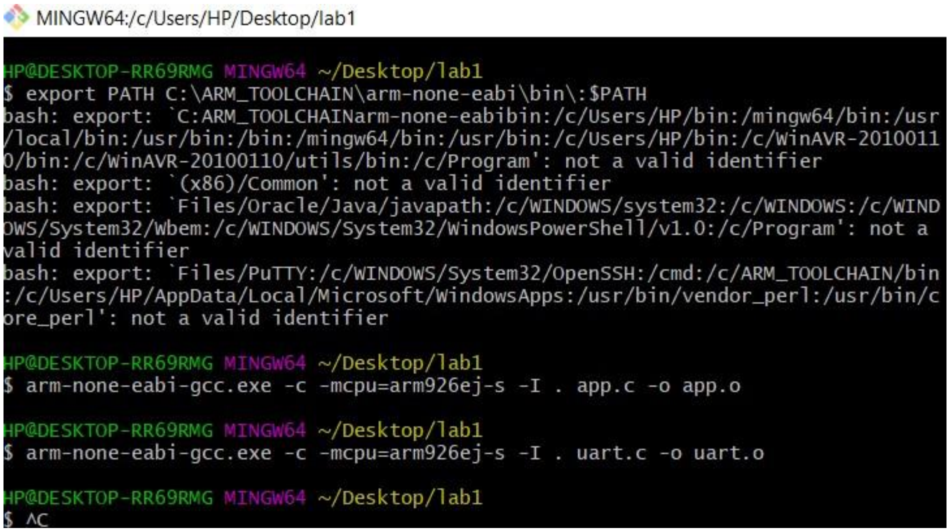
Uart.h :

```c

#ifndef _UART_H_

#define _UART_H_
 void uart_send_string(unsigned char* p_tx);
#endif
```

Then we will open terminal and export path of toolchain to our directory

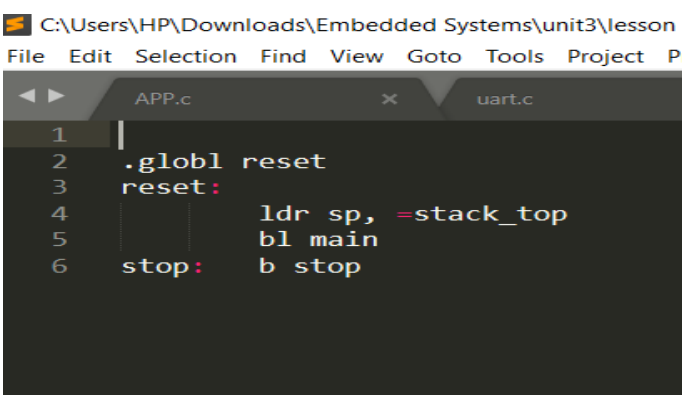
To generate app.o and uart.o with consideration of microcontroller architecture

.



Startup.s :

We defined reset section as a global to be seen by linker script to make it an entry point .

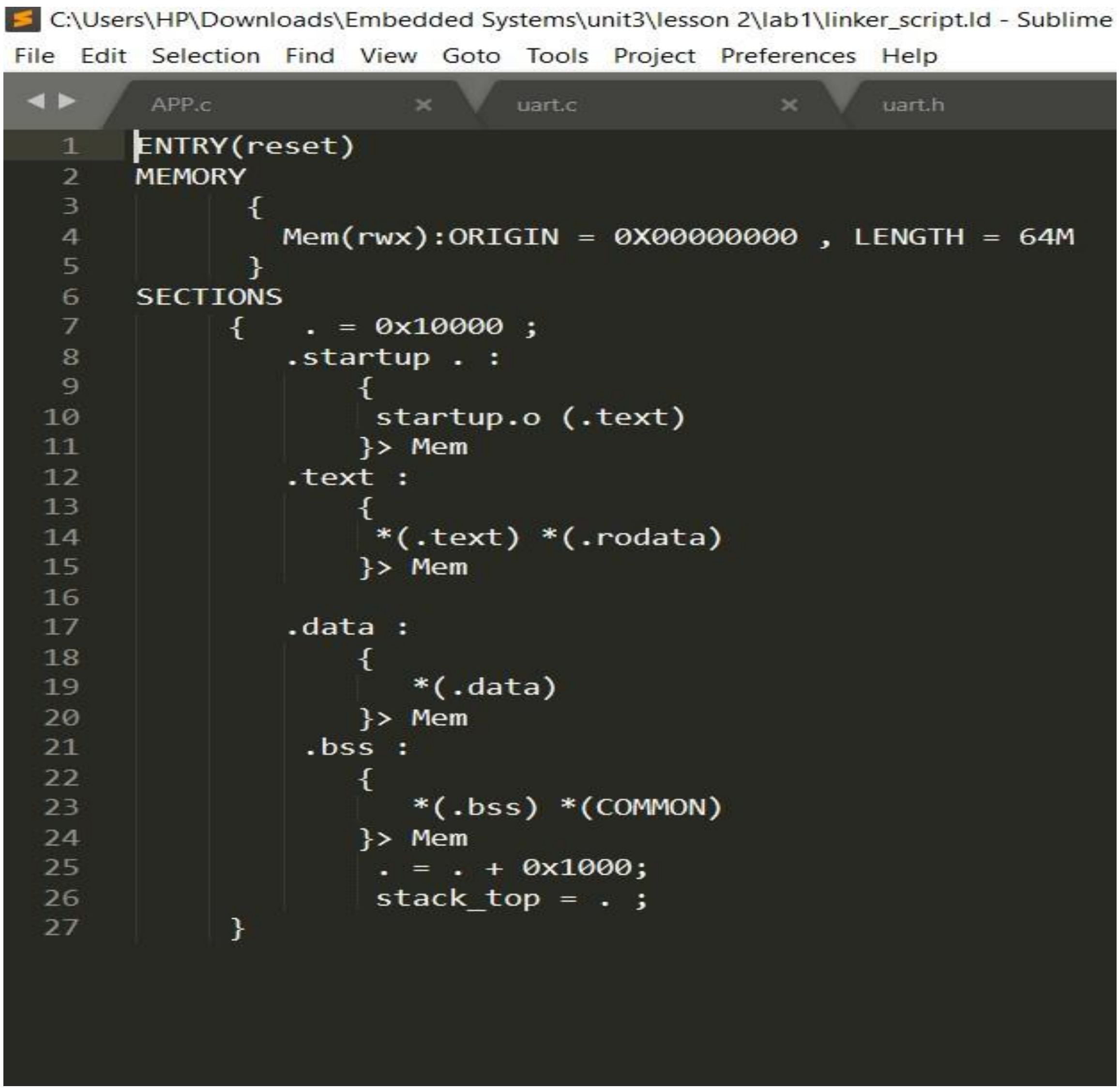


Then we will use commands to generate startup.o using assembler .

```
$ arm-none-eabi-as.exe  -mcpu=arm926ej-s startup.s -o startup.o
```

Linker script :

In this section we control all memory locations, memory sizes , starting Point of our program and stack size .




```
ENTRY(reset)
MEMORY
      {
        Mem(rwx):ORIGIN = 0X00000000 , LENGTH = 64M
      }
SECTIONS
     {    . = 0x10000 ;
         .startup . :
             {
              startup.o (.text)
             }> Mem
         .text :
             {
              *(.text) *(.rodata)
             }> Mem

         .data :
             {
                *(.data)
             }> Mem
          .bss :
             {
                *(.bss) *(COMMON)
             }> Mem
              . = . + 0x1000;
              stack_top = . ;
     }
```

Then we will link all object files app.o, startup.o and uart.o with linker script using linker and generate our .elf file and .map file .

Then generate binary code that will be burnt on board.

```
HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-ld.exe -T linker_script.ld -Map=map_file.map app.o startup.o uar
t.o -o learn-in-depth.elf

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-objcopy.exe -O binary learn-in-depth.elf learn-in-depth.bin

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$
```

Now we will call qemo emulator to run the code on the board and see the expected output : *learn-in-depth<shady Mamdouh>*

```
HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-objcopy.exe -O binary learn-in-depth.elf learn-in-depth.bin

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ /c/qemu/qemu-system-arm.exe -M versatilepb -m 128M -nographic -kernel learn-in
-depth.bin
learn-in-depth<shady mamdouh>
```

Lets use some binary utilities to differentiate between different stages of code :

1- Objdump -h & -d for startup.o and app.o :

We will find symbols that are not resolved and all addresses are not physical addresses because they are object files and as we know abject files are relocated image that will be resolved and allocated in linker with linker script .

```
$ arm-none-eabi-objdump.exe -h startup.o

startup.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000010  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000044  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000044  2**0
                  ALLOC
  3 .ARM.attributes 00000022  00000000  00000000  00000044  2**0
                  CONTENTS, READONLY

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-objdump.exe -h app.o

app.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000018  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000064  00000000  00000000  0000004c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000000b0  2**0
                  ALLOC
  3 .rodata       00000064  00000000  00000000  000000b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000012  00000000  00000000  00000114  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 00000032  00000000  00000000  00000126  2**0
                  CONTENTS, READONLY

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-objdump.exe -D app.o

app.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   e59f0004        ldr     r0, [pc, #4]    ; 14 <main+0x14>
   c:   ebfffffe        bl      0 <uart_send_string>
  10:   e8bd8800        pop     {fp, pc}
  14:   00000000        andeq   r0, r0, r0

Disassembly of section .data:
```

So if we use objdump utility after linking stage we will find that all symbols have been resolved and all sections allocated in the expected addresses in the memory according to specs and our linker script .

```
HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-objdump.exe -h learn-in-depth.elf

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

Sections:
Idx Name            Size      VMA       LMA       File off  Algn
  0 .startup        00000010  00010000  00010000  00008000  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text           000000cc  00010010  00010010  00008010  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data           00000064  000100dc  000100dc  000080dc  2**2
                    CONTENTS, ALLOC, LOAD, DATA
  3 .ARM.attributes 0000002e  00000000  00000000  00008140  2**0
                    CONTENTS, READONLY
  4 .comment        00000011  00000000  00000000  0000816e  2**0
                    CONTENTS, READONLY

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$
```

If we want to use more binary utilities such as nm :

In this picture we will see the symbols before and after linking :

The difference between app.o and learn-in-depth.elf appears in resolved symbols and real physical addresses .

```
HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$ arm-none-eabi-nm.exe app.o learn-in-depth.elf

app.o:
00000000 T main
00000000 D string_buffer1
00000000 R string_buffer2
         U uart_send_string

learn-in-depth.elf:
00010010 T main
00010000 T reset
00011140 D stack_top
00010008 t stop
000100dc D string_buffer1
00010078 T string_buffer2
00010028 T uart_send_string

HP@DESKTOP-RR69RMG MINGW64 ~/Desktop/lab1
$
```