Lab 1

Build a Bare-metal application

from scratch using terminal.

- **Application:**

  Printing a string to UART of Arm versatilepb board by writing in the register UARTDT.

- **Board used:**

  Arm versatilepb.

- **Prossecor used:**

  Arm926ej-s.

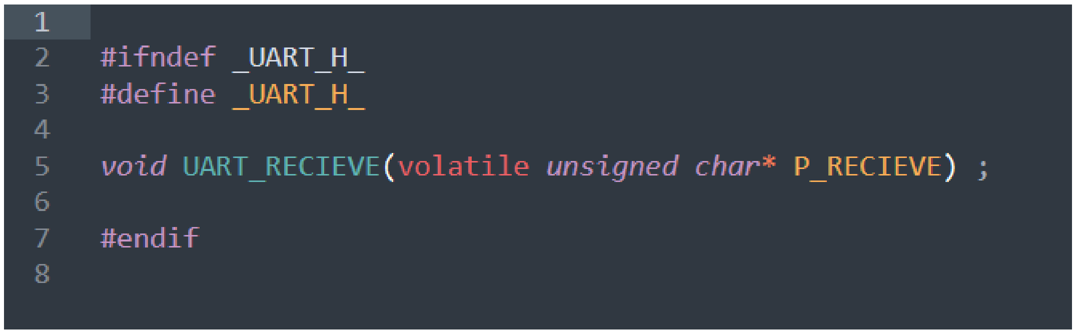
- **Tool Chain used:**
  Arm cross tool chain.

- **Steps:**
  1- Writing the C code.
  2- Writing Startup file.
  3- Writing linker script.
  4- Running the application using QEMU emulator.

1- **Writing C code:**

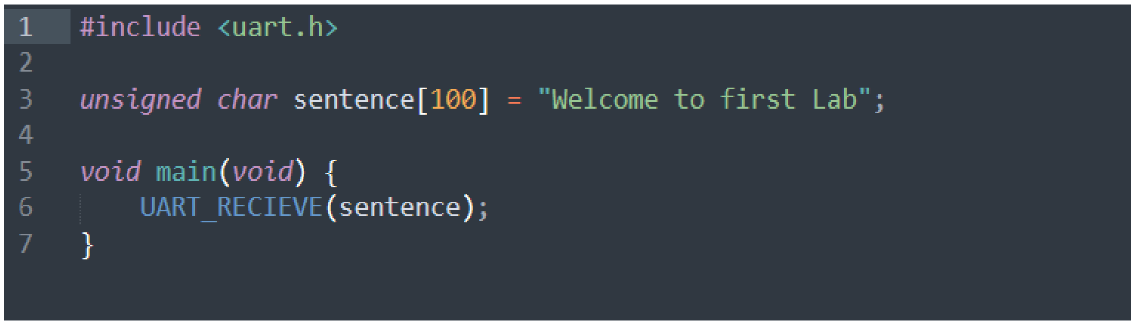
Versatilepb board has 4 UART terminals , we used UART0 .
The logic of c code is that we send the string which we want to print to UARTDT Register which prints anything once we write on it .
For doing this we used a pointer refers to our string then we stored the value of location pointed to by the pointer , in the UARTDT Register so the character gets printed directly , after this we incremented the value of the pointer to step to the next character and do the same process. Eventually the string will be printed.

uart.h :



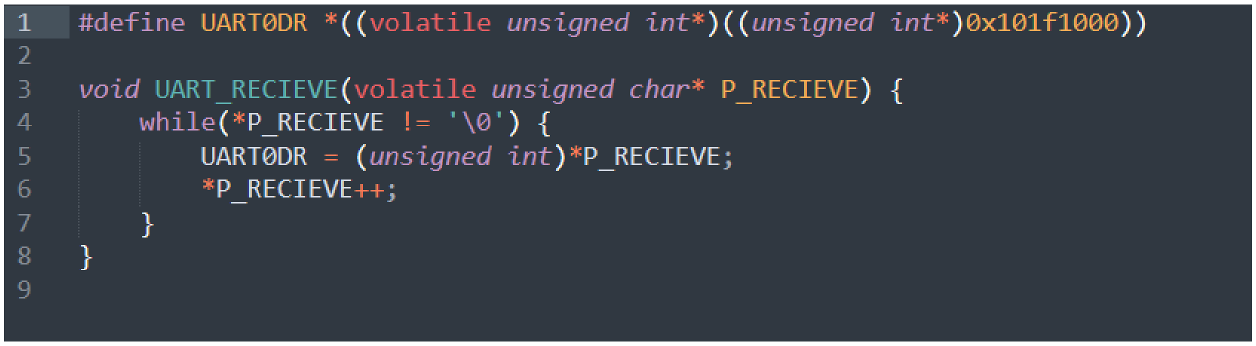
```c

#ifndef _UART_H_
#define _UART_H_

void UART_RECIEVE(volatile unsigned char* P_RECIEVE) ;

#endif

```

app.c:



```c
#include <uart.h>

unsigned char sentence[100] = "Welcome to first Lab";

void main(void) {
    UART_RECIEVE(sentence);
}
```

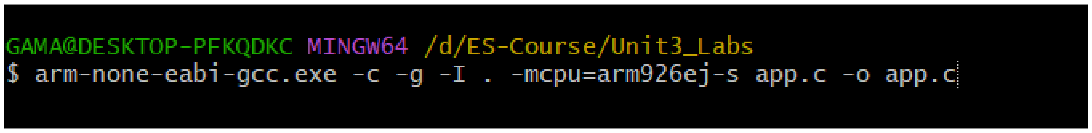
uart.c :



```c
#define UART0DR *((volatile unsigned int*)((unsigned int*)0x101f1000))

void UART_RECIEVE(volatile unsigned char* P_RECIEVE) {
    while(*P_RECIEVE != '\0') {
        UART0DR = (unsigned int)*P_RECIEVE;
        *P_RECIEVE++;
    }
}
```

- **Relocatable File (.o) :**
  This file is the second stage of the compiling process after preprocessing .
  It consists of sections , each section includes specific category (data section ,text section etc..).These sections are created by default with default names and default content.
  The most important thing about relocatable file is that these sections are set in virtual not real addresses.

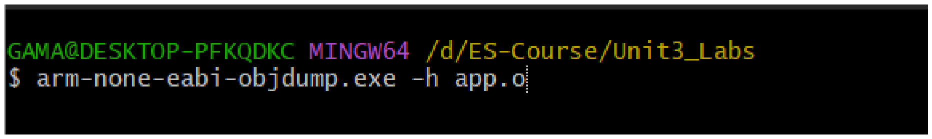
  The command used to get this file is :



```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ arm-none-eabi-gcc.exe -c -g -I . -mcpu=arm926ej-s app.c -o app.c
```

  -g option to include the debug information in the file.

  To see the output sections of any file we can use the binary utility “objdump” as shown in the following command:

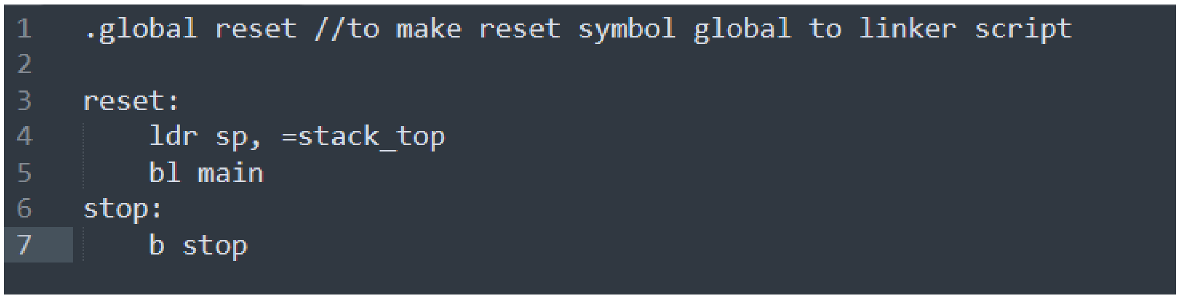


```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ arm-none-eabi-objdump.exe -h app.o
```

  We also can see disassembly of sections using objdump utility, and more other options . we can use --help to know these options.

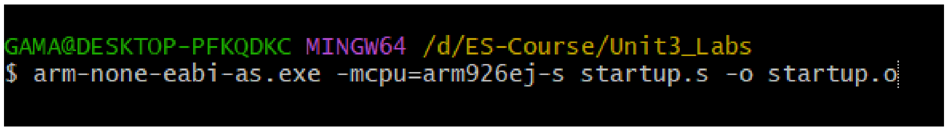
2- **Writing Startup file:**

Startup file does a group of tasks before executing the c code such as initializing some hardwares, setting SP Register, copying data section from Rom to Ram ,and other tasks. The following one is a simple startup.
We fisrt set the Stack Pointer(SP) Register to initialize the top of our stack which we hold for the application. Then we branched to the start of our c code which in most cases is the main function (This depends on the symbol name which we branch to in the startup which we can change).
The startup file in most cases is assembly code, but in some processors (Cortex-M Family) is written as a c code.



```
.global reset //to make reset symbol global to linker script

reset:
    ldr sp, =stack_top
    bl main
stop:
    b stop
```

Command used to get startup.o in case it's assembly:

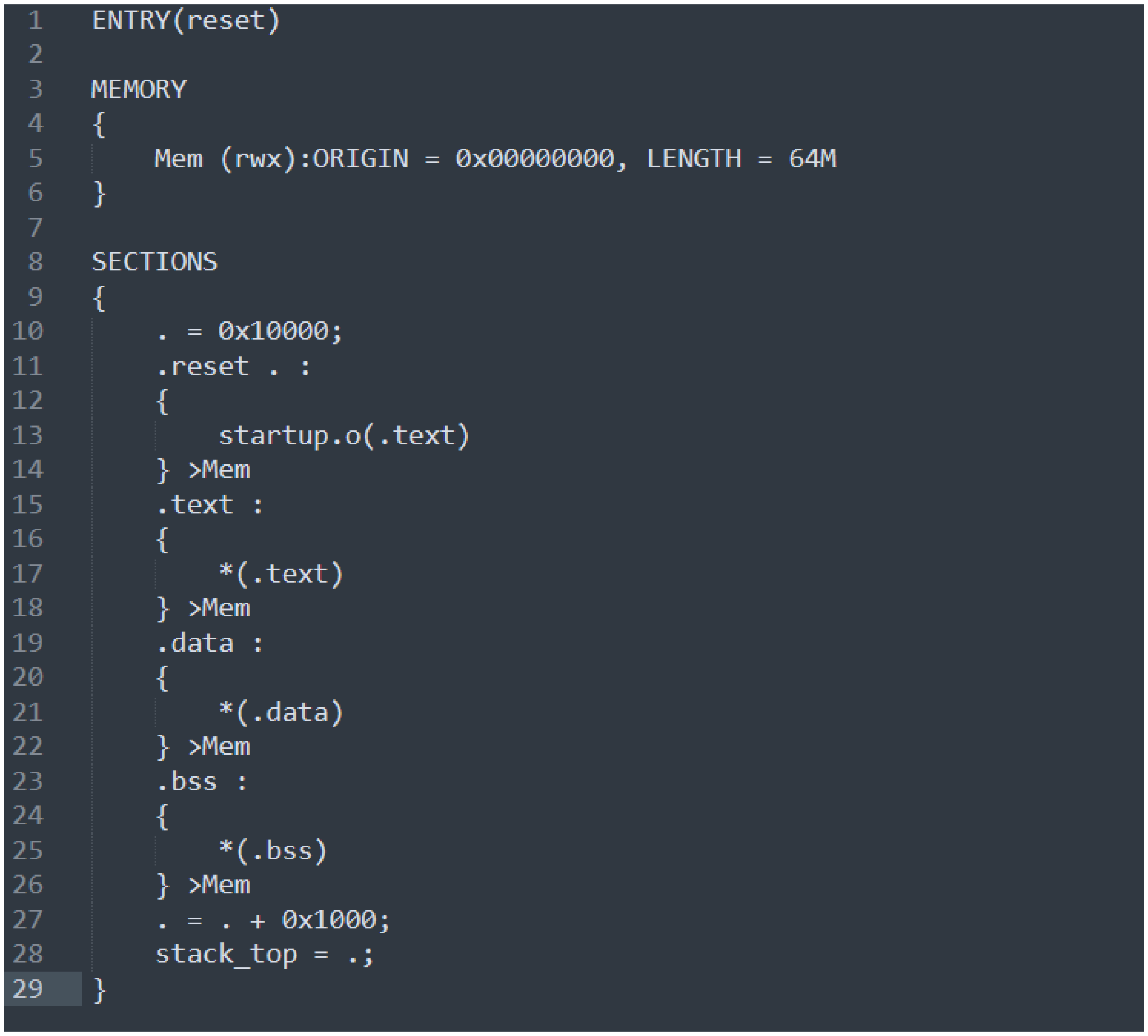


```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ arm-none-eabi-as.exe -mcpu=arm926ej-s startup.s -o startup.o
```

3- **Writing Linker Script:**

Linker Script links the output .o files with startup.o and libraries used. There are commands used in the linker script in order to arrange The output executable file (specify the memory used and location for each section, the sections of the output file, the entry symbol for debug, and more other things).
The linker script basicly does two main actions:

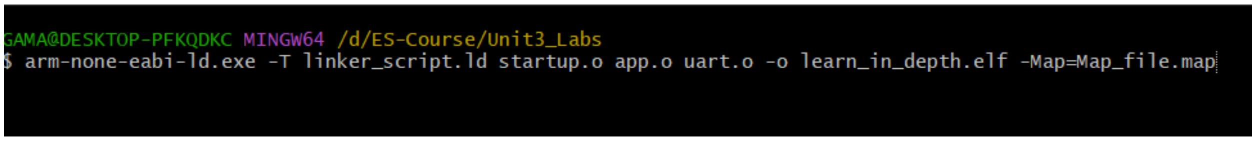
- It merges different sections from different .o files and put them in one big section according to our arrangement using SECTION command.
- Mapping the addresses from their previous virtual addresses into real physical addresses.



```
ENTRY(reset)

MEMORY
{
    Mem (rwx):ORIGIN = 0x00000000, LENGTH = 64M
}

SECTIONS
{
    . = 0x10000;
    .reset . :
    {
        startup.o(.text)
    } >Mem
    .text :
    {
        *(.text)
    } >Mem
    .data :
    {
        *(.data)
    } >Mem
    .bss :
    {
        *(.bss)
    } >Mem
    . = . + 0x1000;
    stack_top = .;
}
```

Here we put the instructions of startup in which we set SP then branched to main, in the first section in order to be executed first.

Command used to link files using linker script is:

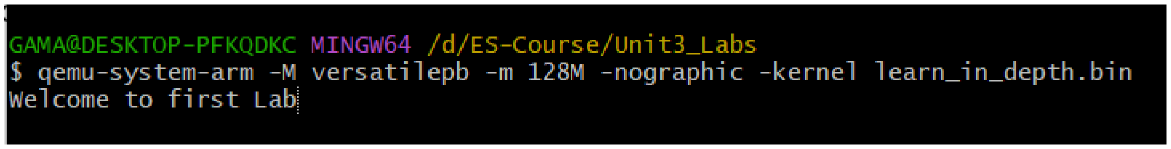


```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ arm-none-eabi-ld.exe -T linker_script.ld startup.o app.o uart.o -o learn_in_depth.elf -Map=Map_file.map
```

-Map=Map_file.map is an option used to create a file shows different sections and addresses in the memory.

4- Running the application using QEMU emulator:

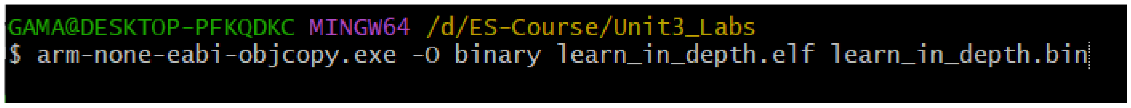
The last step is to emulate and run the application using QEMU:



```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ qemu-system-arm -M versatilepb -m 128M -nographic -kernel learn_in_depth.bin
Welcome to first Lab
```

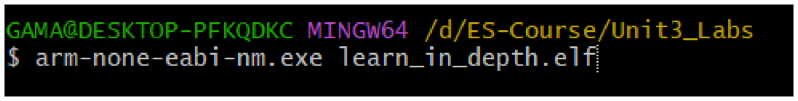
- **Some useful commands:**

1- Extracting the binary only from output executable file without debug information , we use this file to burn it on the board as we don't need debug information.



```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ arm-none-eabi-objcopy.exe -O binary learn_in_depth.elf learn_in_depth.bin
```

2- Show the symbols of any executable of relocatable file:



```
GAMA@DESKTOP-PFKQDKC MINGW64 /d/ES-Course/Unit3_Labs
$ arm-none-eabi-nm.exe learn_in_depth.elf
```