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 :

A computer screen shot of a code

Description automatically generated

Uart.c :

A computer code with blue text

Description automatically generated with medium confidence

Uart.h :

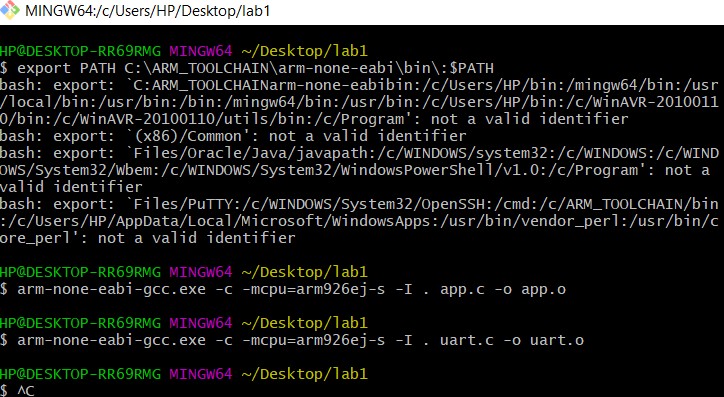
A close-up of a math formula

Description automatically generated

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

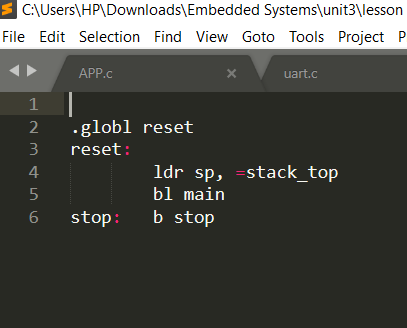
To generate app.o and uart.o with consideration of microcontroller architecture

.



Startup.s :

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

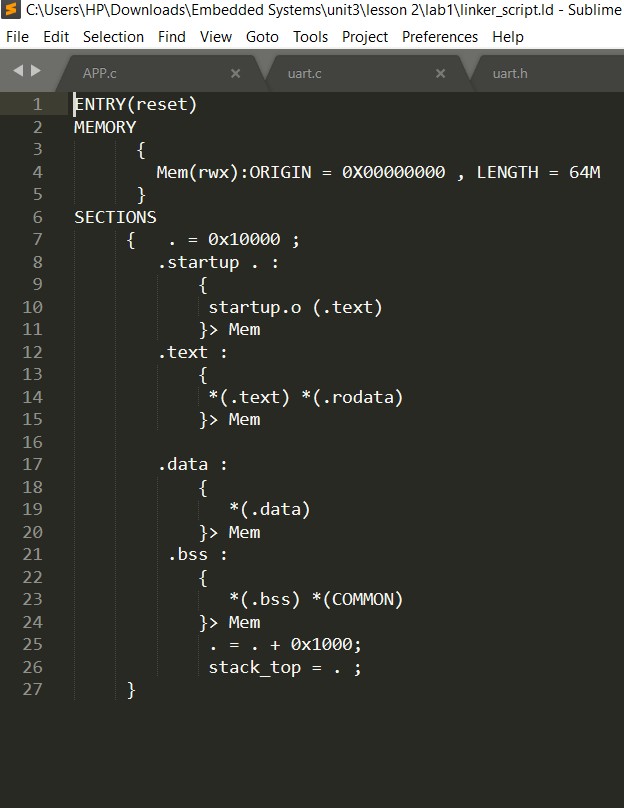


Then we will use commands to generate startup.o using assembler .

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

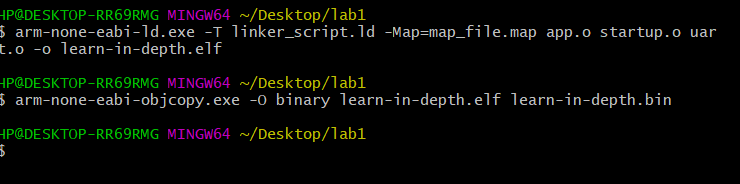
# Linker script :

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

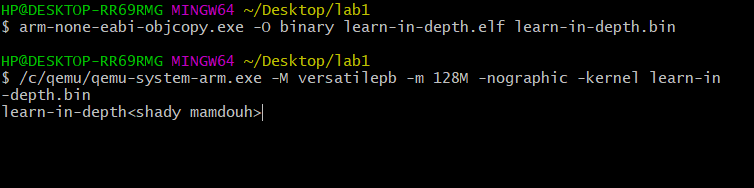


Then we will link all object files app.o, startup.o and uart.o with linker script using linker and generate our .elf file and .map file .

Then generate binary code that will be burnt on board.



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

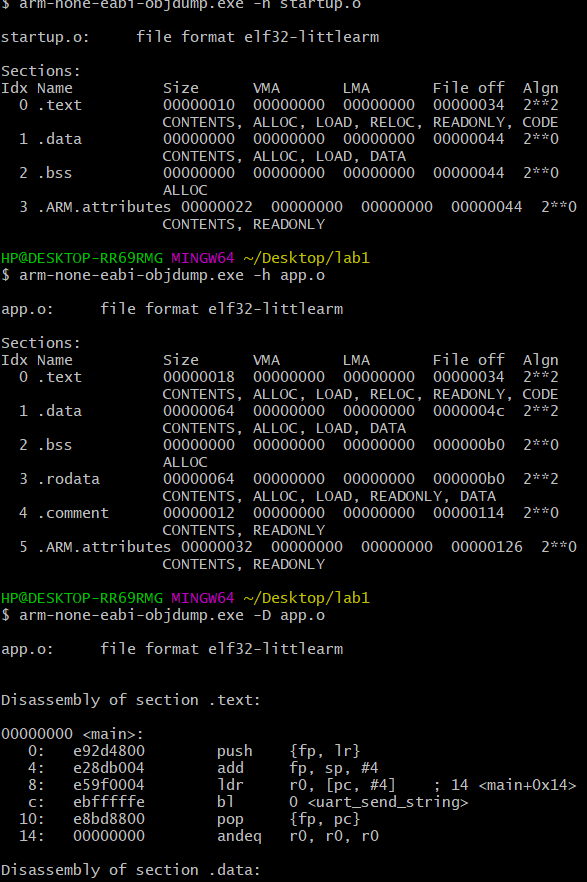


Lets use some binary utilities to differentiate between different stages of code :

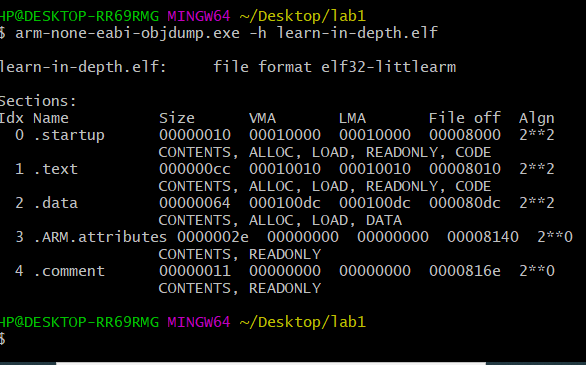
1- Objdump -h & -d for startup.o and app.o :

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

.



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 .



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 .

