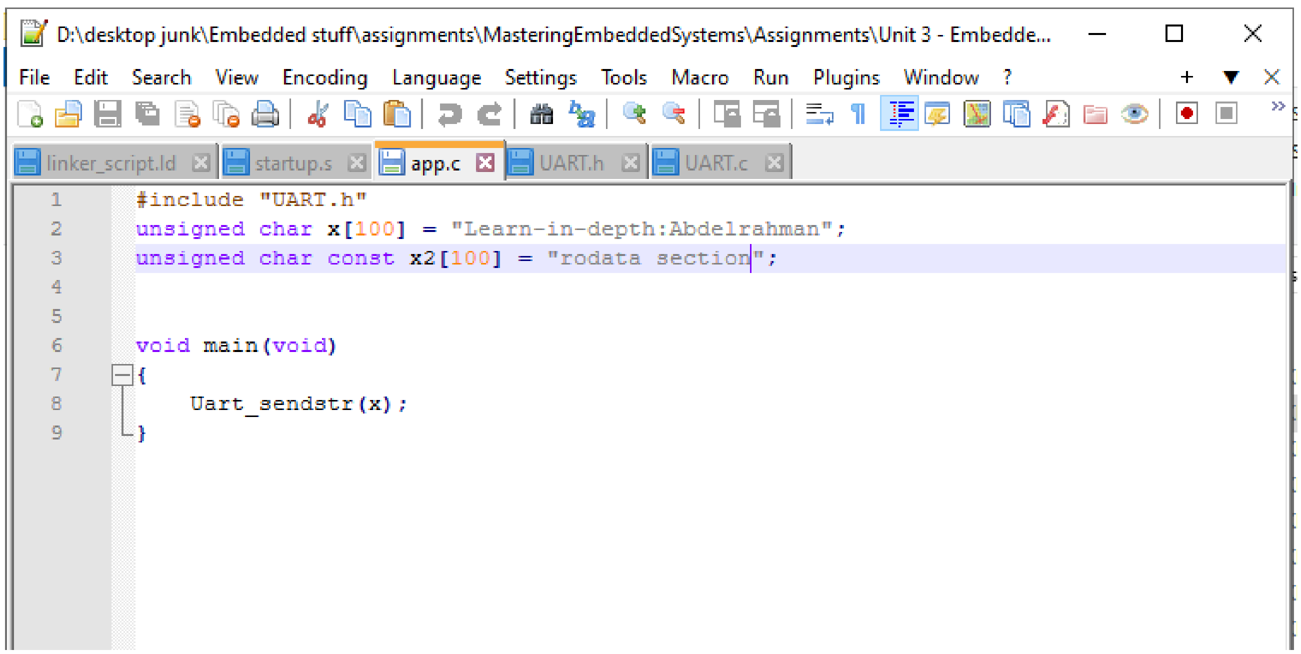
UNIT 3 LAB 1

In this lab we are going to write a very simple bare-metal application using ARM Cross-Toolchain & running it on QEMU (Quick emulator) where application sends a string to UART data register and print it out on a versatile physical board (ARM926EJ-S).

Step 1

Writing our C code for the app and the header files.

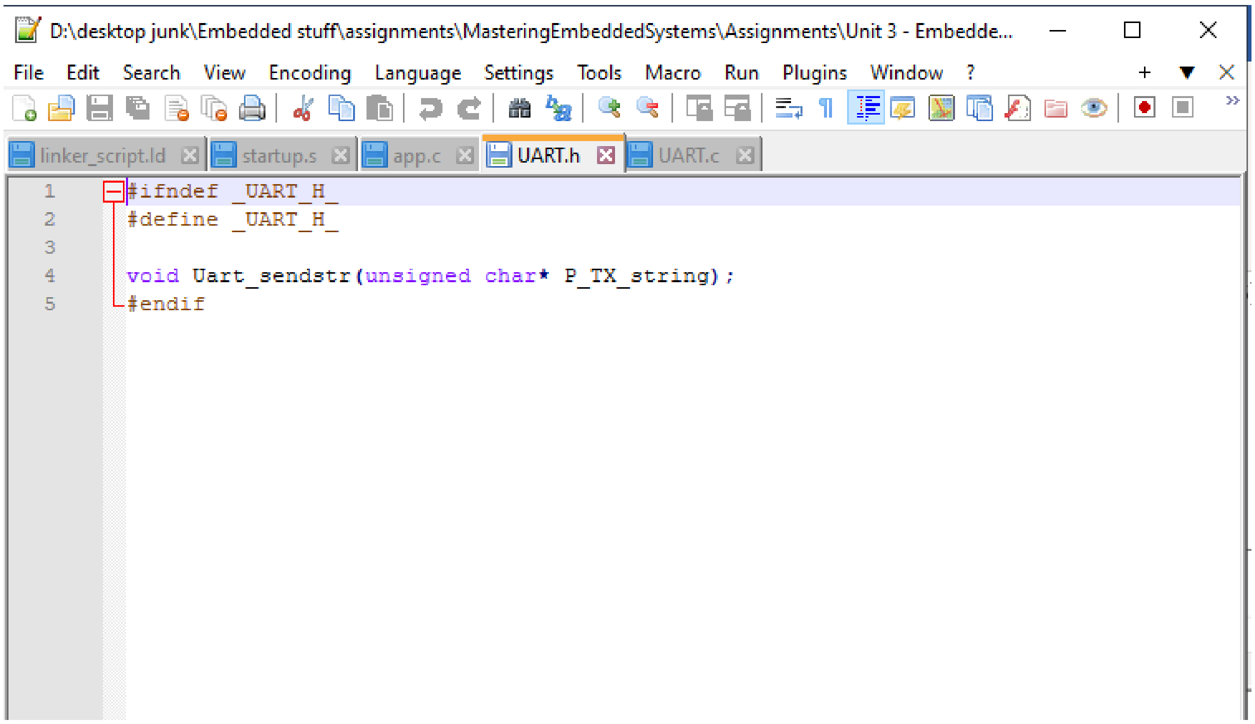
App.c:



UART.c:

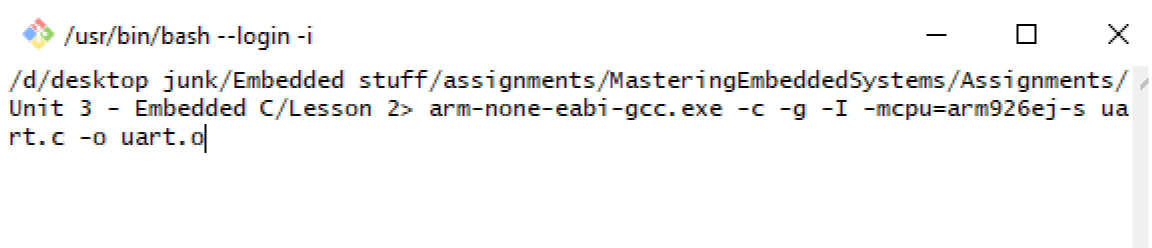
```c
#include "UART.h"
#define UART0DR     *((volatile unsigned int*)((unsigned int*)0x101f1000))

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

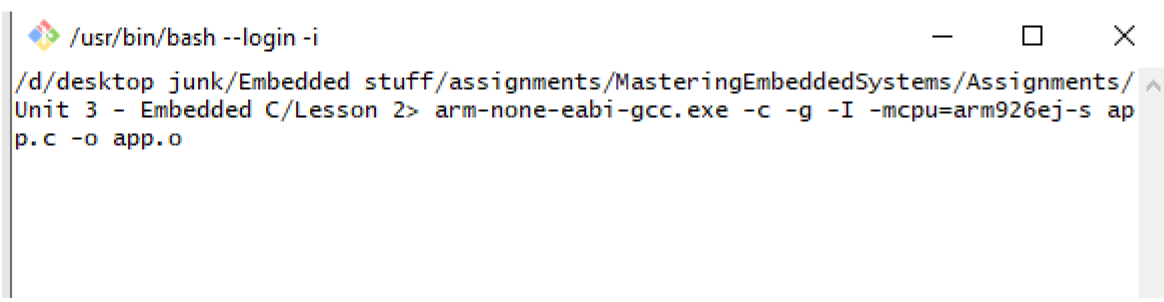
UART.h:



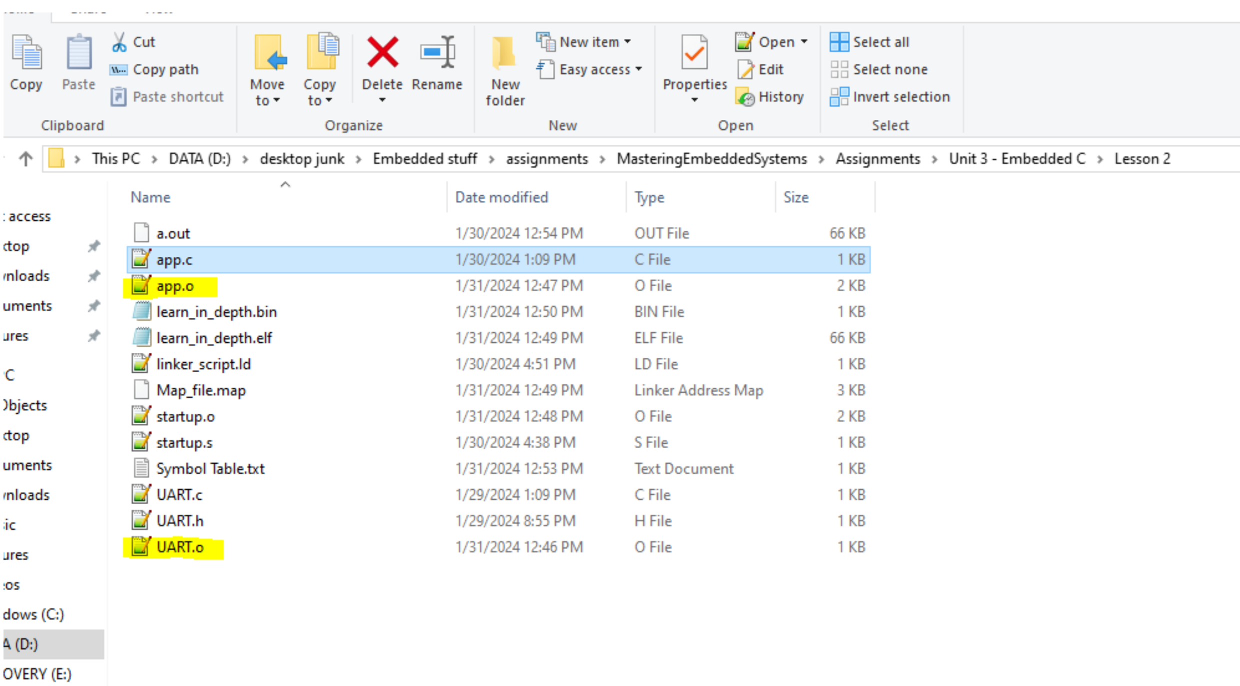
Step 2

Generating Object code for UART.c and app.c using arm-none-eabi-gcc.exe (ARM Toolchain).



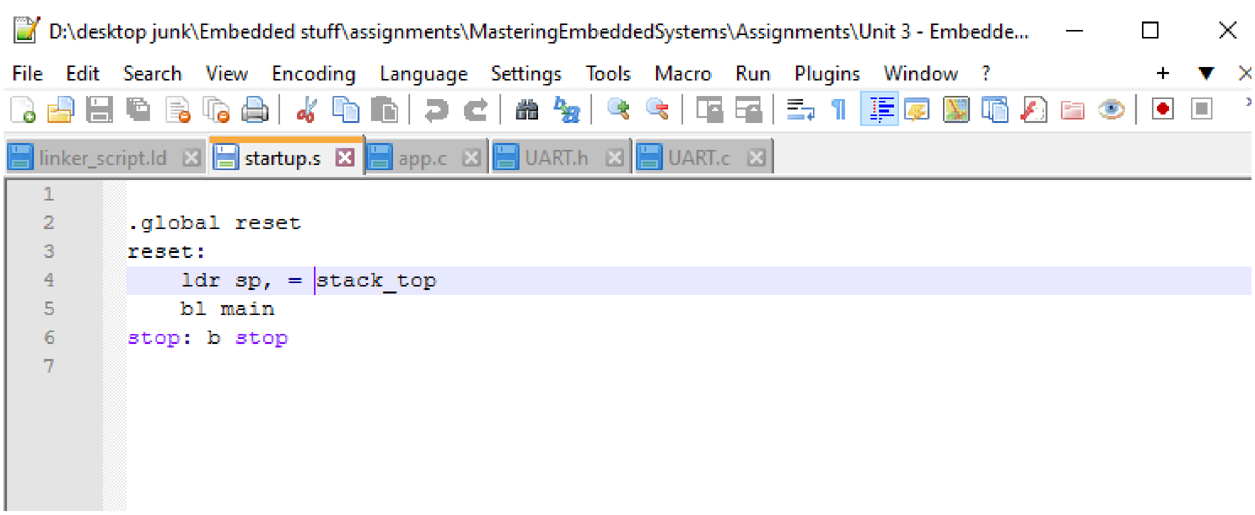


UART.o and app.o generated successfully:

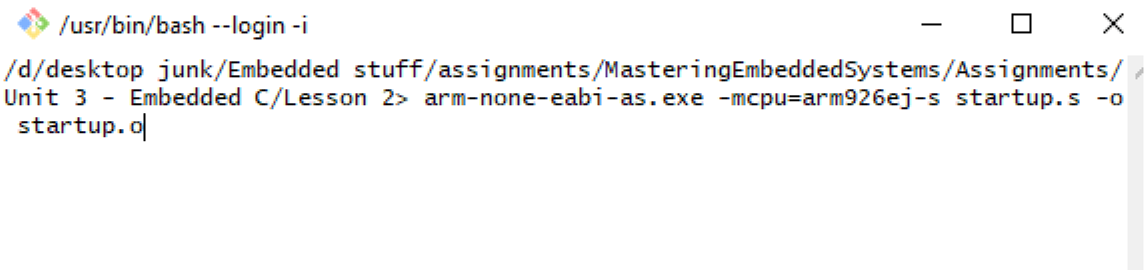


Step3

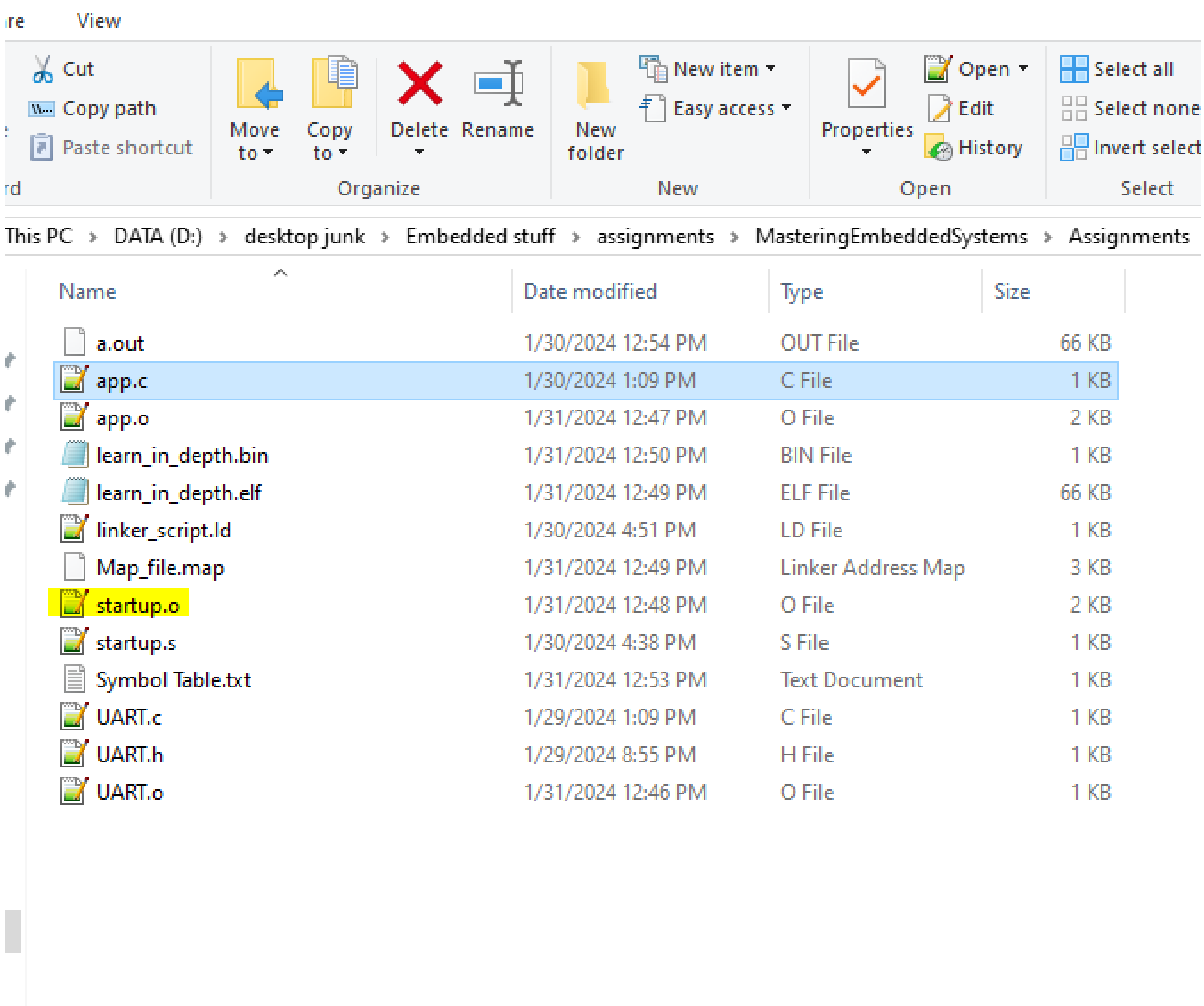
Writing startup.s code and then generating object file startup.o using arm-none-eabi-as.exe (ARM assembler).



Note: stack_top is later on defined in the linker script as last address for stack pointer (explanation in next step).



Startup.o generated successfully:



Step 4

Writing linker script to link all the object files together to generate executable file.

```
1  ENTRY(reset)
2
3  MEMORY
4  {
5      Mem (rwx) : ORIGIN = 0x00000000 , LENGTH = 64M
6  }
7
8  SECTIONS
9  {
10     . = 0x10000;
11     .reset . :
12     {
13         startup.o(.text)
14     }>Mem
15
16     .text :
17     {
18         *(.text) *(.rodata)
19     }>Mem
20
21     .data :
22     {
23         *(.data)
24     }>Mem
25
26     .bss :
27     {
28         *(.bss)
29     }>Mem
30     . = . + 0x1000;
31     stack_top = . ;
32 }
```

Explanation:

The entry point of the program should start from the address of the section “reset”, hence we write ENTRY(reset).

Linker scripts usually have more than one memory to serve the program, but for this simple example we are using only one.

Line 8 is for the code sections, where the linker organizes and links all object files together in their corresponding sections, so all code text across all files is organized in a certain memory range, all code data across all files is organized in memory range right after the code text section's final address, and so on.

Line 30 is for leaving an acceptable memory range after the last address of the last section for stack to execute code, then we give a symbol to that address as stack_top to write it down in the startup code.

Step 5

Linking all object codes together to generate .elf file and map file.

```
/usr/bin/bash --login -i
/d/desktop junk/Embedded stuff/assignments/MasteringEmbeddedSystems/Assignments/
Unit 3 - Embedded C/Lesson 2> arm-none-eabi-ld.exe -T linker_script.ld -Map=Map_
file.map app.o UART.o startup.o -o learn_in_depth.elf
```

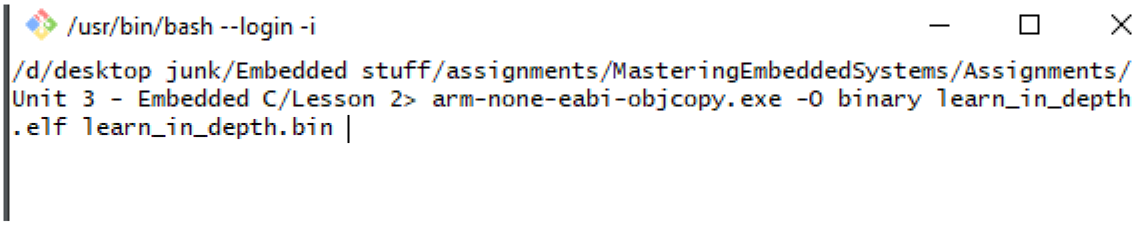
Executable file and .map file generated successfully:

C › DATA (D:) › desktop junk › Embedded stuff › assignments › MasteringEmbeddedSystems › Assignmen

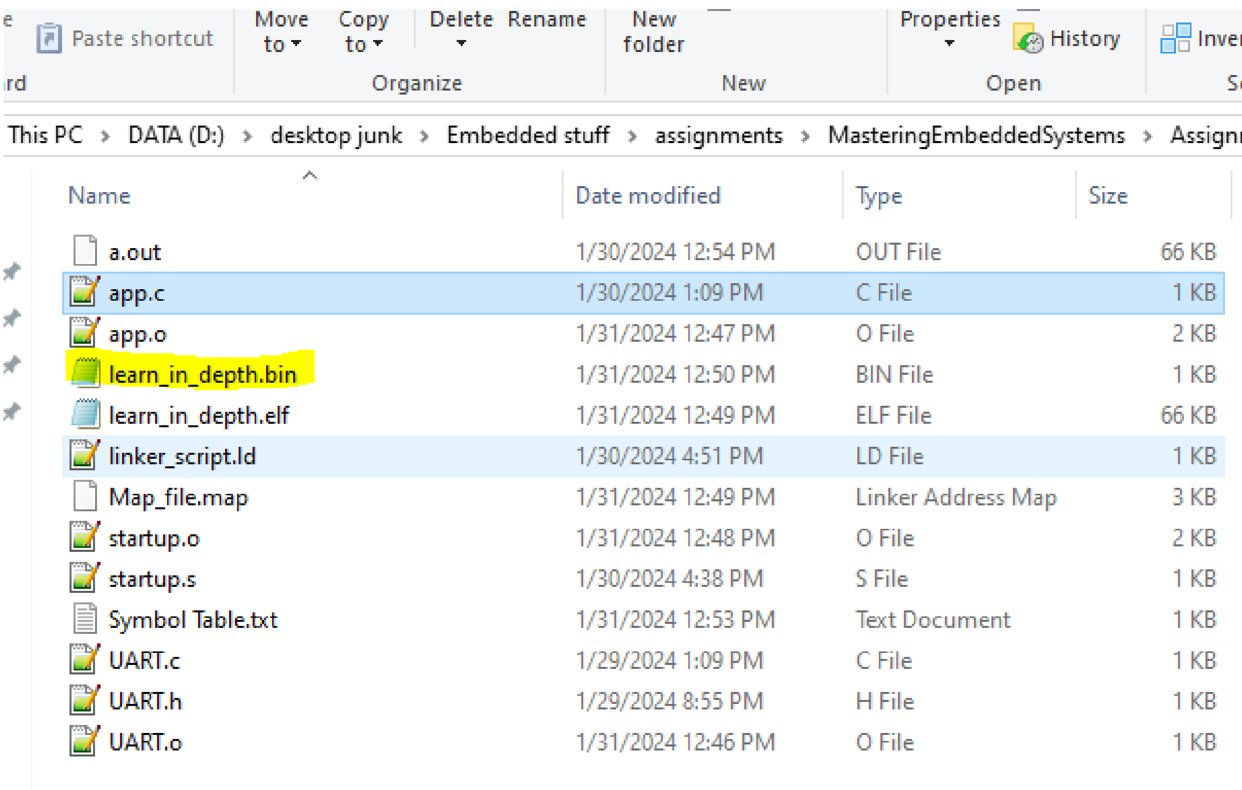
Name	Date modified	Type	Size
a.out	1/30/2024 12:54 PM	OUT File	66 KB
app.c	1/30/2024 1:09 PM	C File	1 KB
app.o	1/31/2024 12:47 PM	O File	2 KB
learn_in_depth.bin	1/31/2024 12:50 PM	BIN File	1 KB
learn_in_depth.elf	1/31/2024 12:49 PM	ELF File	66 KB
linker_script.ld	1/30/2024 4:51 PM	LD File	1 KB
Map_file.map	1/31/2024 12:49 PM	Linker Address Map	3 KB
startup.o	1/31/2024 12:48 PM	O File	2 KB
startup.s	1/30/2024 4:38 PM	S File	1 KB
Symbol Table.txt	1/31/2024 12:53 PM	Text Document	1 KB
UART.c	1/29/2024 1:09 PM	C File	1 KB
UART.h	1/29/2024 8:55 PM	H File	1 KB
UART.o	1/31/2024 12:46 PM	O File	1 KB

Step 6

Generating binary file to run on QEMU.



.bin file generated successfully:



Step 7

Running binary file on QEMU.