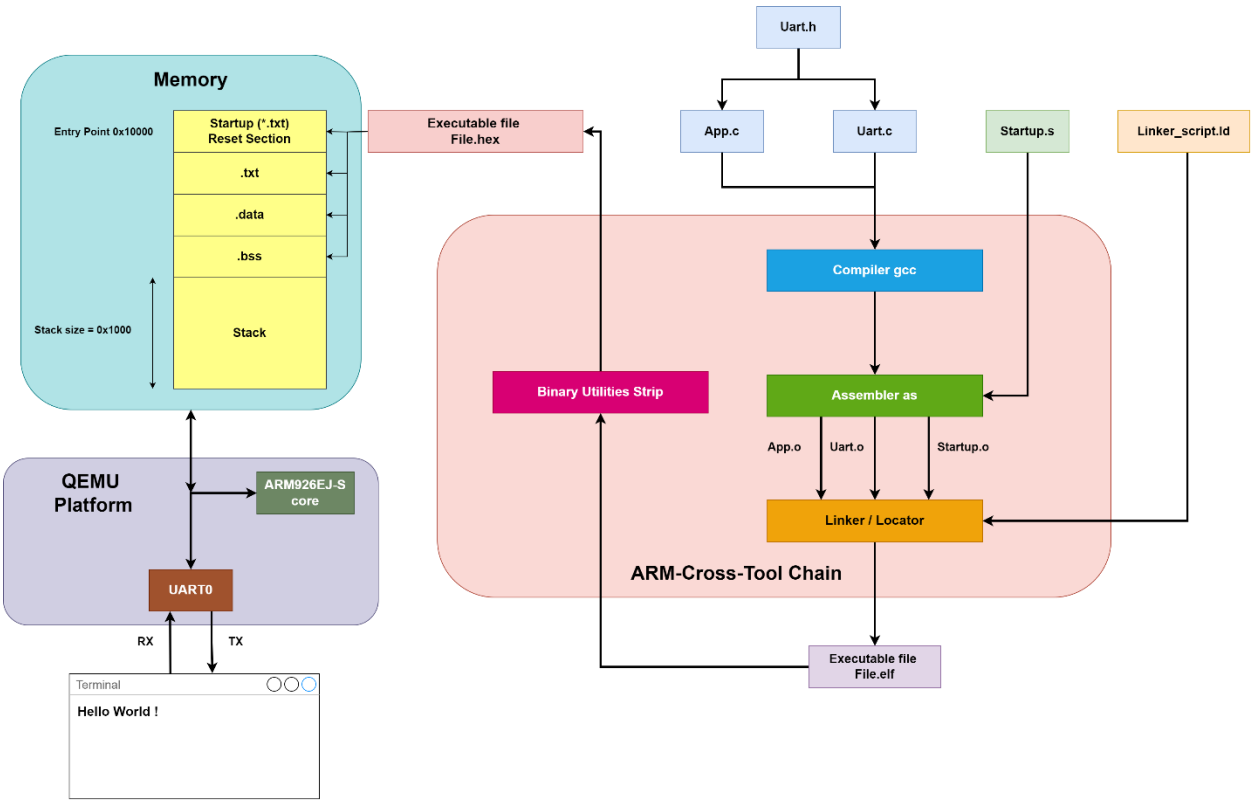
Lab1 – UART bare-metal code from scratch:

This lab covers creating a bare-metal Software to send a “Hello World!” using UART in ARM VersatilePB MCU.



Codes:

- Uart.h

```h
#ifndef UART_H
#define UART_H

void Uart_Send_String(unsigned char*);

#endif
```

- Uart.c

```c
#include "uart.h"

// UART register at address 0x101f1000 in ARM926EJ-S
// UART 0 data register is at offset 0x00
#define UART0_DR *((volatile unsigned int*)((unsigned int*)0x101f1000))

void Uart_Send_String(unsigned char* p_tx_string)
{
    // Check if end of string is reahced
    while (*p_tx_string != '\0')
    {
        // Transmit each character in string
        UART0_DR = (unsigned int)(*p_tx_string);
        p_tx_string++;
    }
}
```

- App.c

```c
#include "uart.h"

unsigned char str_buffer[100] = "Learn-in-depth: <Ahmed Hassan>";

void main(void)
{
    Uart_Send_String(str_buffer);
}
```

To compile the codes using git bash:

- Run the following command to include the ARM Toolchain path:
  $ export PATH=C:\ARM_TOOLCHAIN\bin:$PATH

- The following command will compile the code without linking. Note that we need to specify the target processor. We included the option to debug "-g" and include the files "-I":
  $ arm-none-eabi-gcc.exe -c -g -mcpu=arm926ej-s -I . app.c -o app.o

- In this stage the object codes generated are "Relocatable Binary" and to display the content of their section headers we run the following command:
  $ arm-none-eabi-objdump.exe -h app.o

Navigate the .obj files (relocatable images):

```
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 .debug_info   0000006c  00000000  00000000  000000b0  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  4 .debug_abbrev 0000005a  00000000  00000000  0000011c  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_loc    0000002c  00000000  00000000  00000176  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_aranges 00000020  00000000  00000000  000001a2  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  7 .debug_line   00000035  00000000  00000000  000001c2  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_str    000000b9  00000000  00000000  000001f7  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .comment      00000012  00000000  00000000  000002b0  2**0
                  CONTENTS, READONLY
 10 .ARM.attributes 00000032  00000000  00000000  000002c2  2**0
                  CONTENTS, READONLY
 11 .debug_frame  0000002c  00000000  00000000  000002f4  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
```

- The "**.text**" section contains the object file **instructions**.
- The "**.data**" section contains **static and initialized global variables**.
- The "**.bss**" section contains the **uninitialized global variables**.
- Lines 3-8 are the **debugging info's** included optional when we created the object file.
- **VMA (Virtual Memory Address of the output section)**: the address the object file will be copied to when transferring the code to another software.
- **LMA (Load Memory Address of the output section)**: the **physical address** used when burning the object code to the processor.
- Note that all the memory contents are **zeros** because these object codes are **relocatable images**.
- Note that there is no "**.rodata**" section because we did not define any **constant variables** in our codes and **stays in the flash ROM**.

Variable	Load location	Runtime location	Section
Global initialized or Global static initialized or Local static initialized	FLASH	RAM	.data Copied from flash to ram by startup code
Global uninitialized or Global static uninitialized or Local static uninitialized		RAM	.bss Startup code reserves space for it in ram and initialized it by zero
Local initialized or Local uninitialized or Local const		Stack (RAM)	In stack at run time
Global Const	FLASH		.rodata section

- The following command is used to get the disassembly of the object code:
  $ arm-none-eabi-objdump.exe -D app.o >> app.s

```
00000000 <str_buffer>:
   0: 7261654c    rsbvc r6, r1, #76, 10   ; 0x13000000
   4: 6e692d6e    cdpvs 13, 6, cr2, cr9, cr14, {3}
   8: 7065642d    rsbvc r6, r5, sp, lsr #8
   c: 203a6874    eorscs   r6, sl, r4, ror r8
  10: 6d68413c    stfvse   f4, [r8, #-240]!  ; 0xffffff10
  14: 48206465    stmdami  r0!, {r0, r2, r5, r6, sl, sp, lr}
  18: 61737361    cmnvs r3, r1, ror #6
  1c: 00003e6e    andeq r3, r0, lr, ror #28
```

  Note that the addresses provided in the assembly code are **virtual addresses** which will be linked to physical addresses by the **linker**.

C startup:

Startup code is the code that runs before the main.

- ❖ Note that the **startup needs to be written in assembly** because we can't run a C code **without preparing the stack**. **BUT ARM Cortex processors assign an address to the stack pointer before accessing the startup code which allows for the startup code to be written in C**. (Startup code located after the entry point address)

Startup code tasks:

- ➢ Disable all interrupts.
- ➢ Create a vector table for your microcontroller.
- ➢ Copy any initialized data from ROM to RAM.
- ➢ Zero the uninitialized data area.
- ➢ Allocate space for and initialize the stack.
- ➢ Initialize the processor's stack pointer.
- ➢ Create and initialize the heap
- ➢ Enable interrupts.
- ➢ Call main.

- Now we create and write the startup code in assembly:

```
.global reset           @Set reset section to global to be seen by linker

reset:
    ldr sp, =0x00011000 @Load the stack base address to SP
    bl main             @Branch to main function

stop:
    b stop              @Infinite loop
```

Linker Script:

Linker Script Commands:

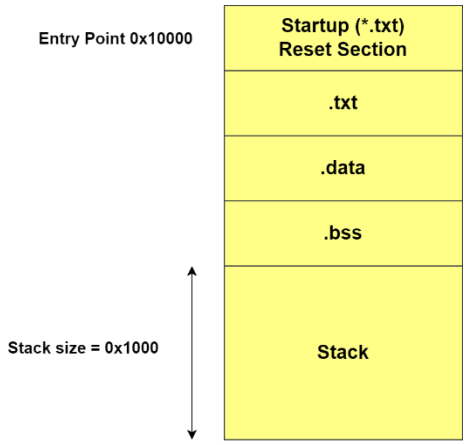
Command	Description
ENTRY	Defines the entry point of an application, which appears in the final ELF file header. Typically, the entry point is the reset handler, executed after a processor reset. This information helps the debugger (like **gdb**) identify the starting function. While not mandatory, it is necessary when debugging with **gdb**.
MEMORY	Defines the memory regions available on the target device. It specifies the **name**, **origin** (starting address), and **length** (size) of each memory block (e.g., FLASH, RAM). This helps the linker know where to place different sections (like code or data) in memory during program linking.
SECTIONS	Used to create different output sections in the final executable file
Location Counter '.'	Representing the current memory address during linking. It helps define memory layout boundaries and assign specific addresses to sections. It should be used only within the **SECTIONS** command.
>(vma)	Vma is specify relocatable section address in **run-time** located
AT>(lma)	Lmais specify relocatable section address in **load-time** located

Linker script (Symbols):

- Symbol is the name of an address.
- Symbol declaration is not equivalent to variable declaration.
- Each object has its own symbol table; the linker is resolving the symbols between all obj files.
- Symbol also is used to specify Memory layout boundaries.
- To display the symbols for an object file, run the following command:
  $ arm-none-eabi-nm.exe app.o

- Now we write the linker code:

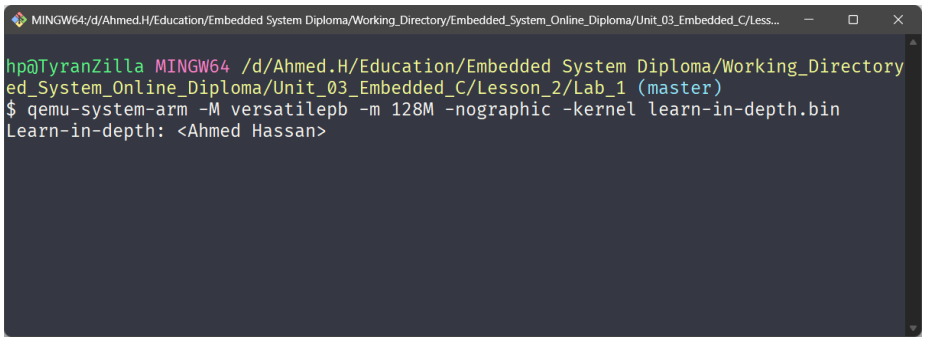
```
ENTRY(reset)

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

SECTIONS
{
    . = 0X10000;

    .startup . :
    {
        startup.o(.text)
    }> Mem

    .text :
    {
        *(.text)
    }> Mem

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

    .bss :
    {
        *(.bss) *(COMMON)
    }> Mem

    . = . + 0X1000;
    stack_top = .;
}
```



- Run the following command to run the linker script including also the file map to verify memory segmentations:
  $ arm-none-eabi-ld.exe -T linker_script.ld -Map=output.map app.o startup.o uart.o -o learn-in-depth.elf

- Run the following command to generate the binary code:
  $ arm-none-eabi-objcopy.exe -O binary learn-in-depth.elf learn-in-depth.bin

- Finaly we run the following command to run binary code on QEMU simulator:
  $ qemu-system-arm -M versatilepb -m 128M -nographic -kernel learn-in-depth.bin



Map File:

```
Memory Configuration

Name             Origin             Length             Attributes
Mem              0x00000000         0x04000000         xrw
*default*        0x00000000         0xffffffff

Linker script and memory map

                 0x00010000                 . = 0x10000

.startup         0x00010000       0x10
 startup.o(.text)
 .text           0x00010000       0x10 startup.o
                 0x00010000                 reset

.text            0x00010010       0x68
 *(.text)
 .text           0x00010010       0x18 app.o
                 0x00010010                 main
 .text           0x00010028       0x50 uart.o
                 0x00010028                 Uart_Send_String

.glue_7          0x00010078        0x0
 .glue_7         0x00000000        0x0 linker stubs

.glue_7t         0x00010078        0x0
 .glue_7t        0x00000000        0x0 linker stubs

.vfp11_veneer    0x00010078        0x0
 .vfp11_veneer   0x00000000        0x0 linker stubs

.v4_bx           0x00010078        0x0
 .v4_bx          0x00000000        0x0 linker stubs

.iplt            0x00010078        0x0
 .iplt           0x00000000        0x0 startup.o

.rel.dyn         0x00010078        0x0
 .rel.iplt       0x00000000        0x0 startup.o

.data            0x00010078       0x64
 *(.data)
 .data           0x00010078        0x0 startup.o
 .data           0x00010078       0x64 app.o
                 0x00010078                 str_buffer
 .data           0x000100dc        0x0 uart.o

.igot.plt        0x000100dc        0x0
 .igot.plt       0x00000000        0x0 startup.o

.bss             0x000100dc        0x0
 *(.bss)
 .bss            0x000100dc        0x0 startup.o
 .bss            0x000100dc        0x0 app.o
 .bss            0x000100dc        0x0 uart.o
 *(COMMON)
                 0x000110dc                 . = (. + 0x1000)
                 0x000110dc                 stack_top = .
LOAD app.o
LOAD startup.o
LOAD uart.o
OUTPUT(learn-in-depth.elf elf32-littlearm)

.ARM.attributes
                 0x00000000       0x2e
 .ARM.attributes
                 0x00000000       0x22 startup.o
 .ARM.attributes
                 0x00000022       0x32 app.o
 .ARM.attributes
                 0x00000054       0x32 uart.o

.comment         0x00000000       0x11
 .comment        0x00000000       0x11 app.o
                                  0x12 (size before relaxing)
 .comment        0x00000000       0x12 uart.o
```