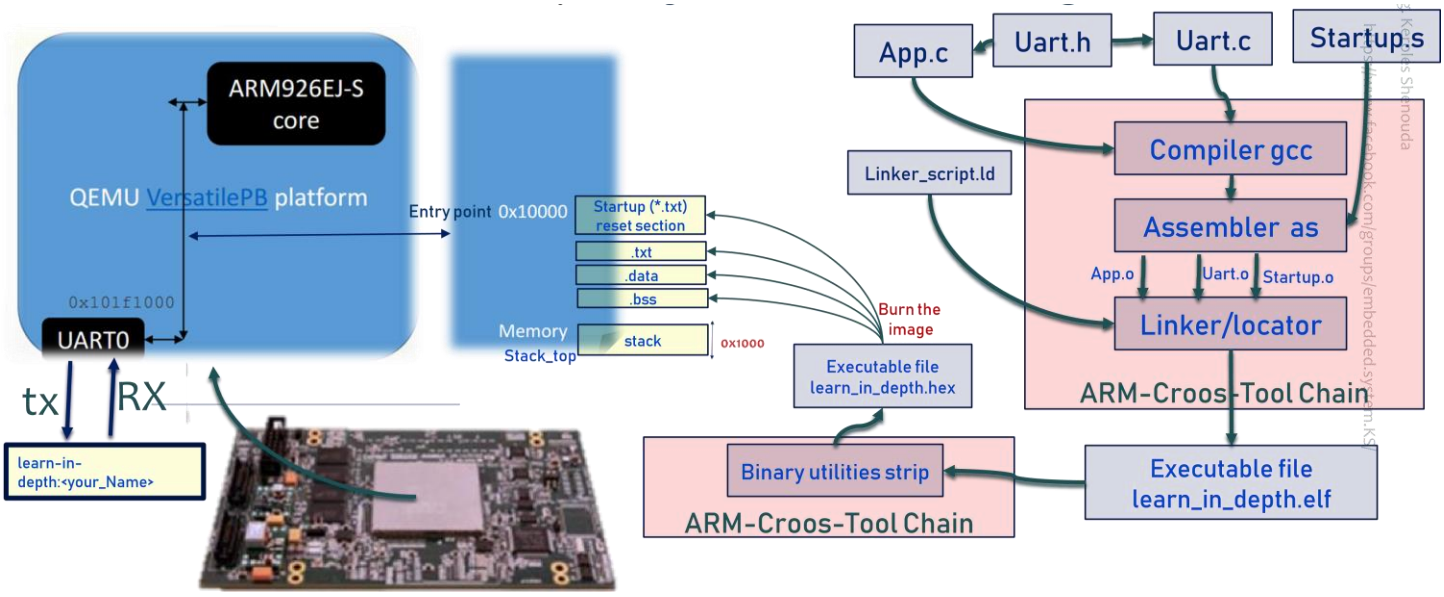
In this lab1: you have to create a baremetal Software to send a "learn-in-depth:<your_Name" using UART



The Most Important Flags Used with The Objdump

Flag	Description	Example Usage
`-d`	Disassemble executable sections of the file.	`objdump -d app.elf`
`-s`	Display the full contents of all sections.	`objdump -s app.elf`
`-h`	Display the section headers.	`objdump -h app.elf`
`-t`	Display the symbol table.	`objdump -t app.elf`
`-x`	Display all available information, including headers, symbols, and sections.	`objdump -x app.elf`
`-f`	Display the file header information, such as architecture and entry point address.	`objdump -f app.elf`
`-g`	Display debugging information, including line numbers.	`objdump -g app.elf`
`-p`	Display the program headers, which include information about how the file is loaded into memory.	`objdump -p app.elf`
`-j <section>`	Display the specified section(s) of the file.	`objdump -j .data -j .text app.elf`
`-a`	Display all the information, similar to combining `-x` and `-d`.	`objdump -a app.elf`

c code files

the address where the UART0 is mapped: 0x101f1000

app.c

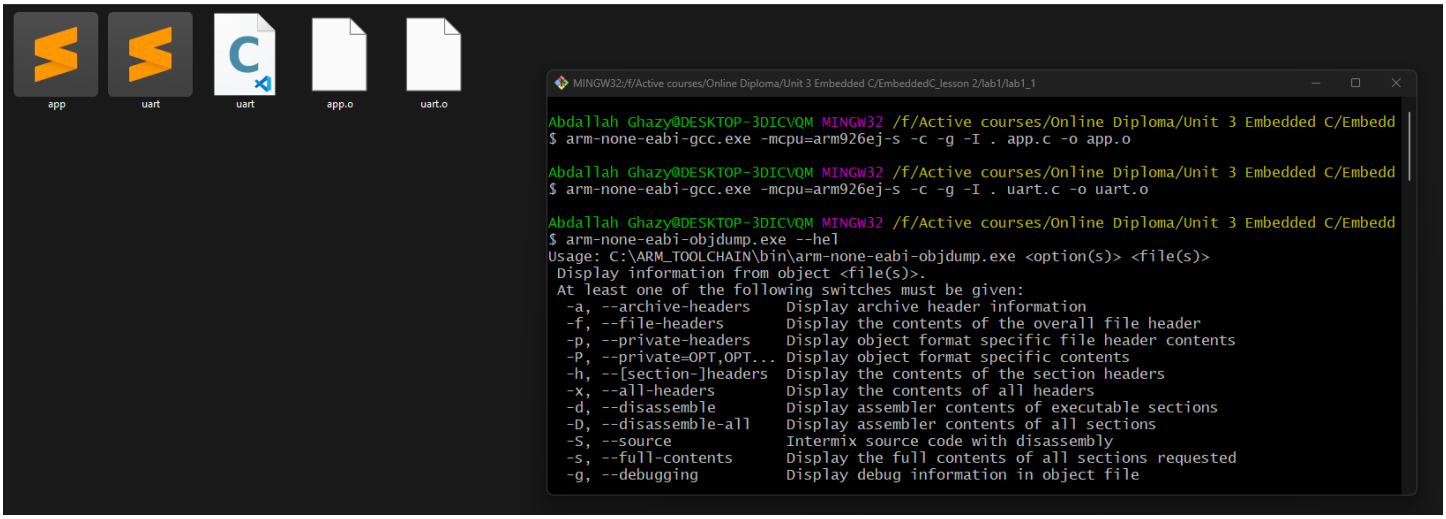
```c
#include "uart.h"
unsigned char string_buffer[100] = "Learn-in-depth:Abdallah Ghazy";
void main(void){
    uart_send_string(string_buffer);

}
```

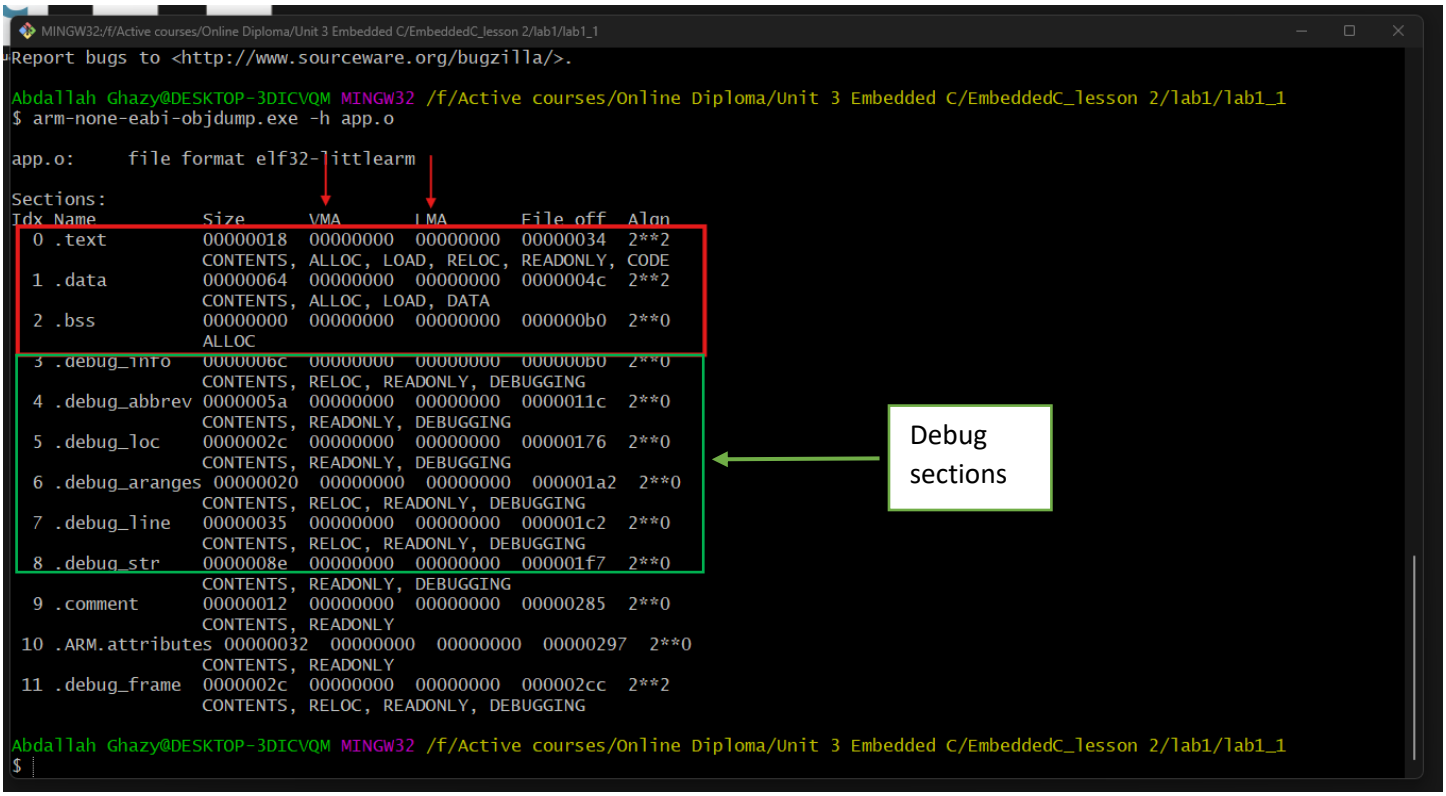
uart.c

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

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

```

uart.h

```c
#ifndef _UART_H_
#define _UART_H_
void uart_send_string(unsigned char* P_tx_string);

#endif
```





LMA (Load Memory Address)

- **Definition**: The Load Memory Address is the address at which a section of code or data is loaded into memory during the linking or loading phase. It represents the address where the program is initially placed in memory before execution.

Example: For a section of code in a linker script, the LMA might specify that the code is to be loaded at address 0x08000000 in flash memory.

VMA (Virtual Memory Address)

- **Definition**: The Virtual Memory Address is the address used by a program or process within its virtual address space. It is the address that the program uses when it references memory. This address is abstracted from the physical memory addresses by the operating system's memory management unit (MMU).

Example: In a memory-mapped file or a dynamically allocated memory region, the VMA is the address that the application code uses to access the memory.

Executable file sections (.data, .bss and rodata)

When C code is compiled, the compiler places initialized global variables in the .data section. So just as with the assembly, the .data has to be copied from Flash to RAM

The C language guarantees that all uninitialized global variables will be initialized to zero.

When C programs are compiled, a separate section called .bss is used for uninitialized variables. Since the value of these variables are all zeroes to start with, they do not have to be stored in Flash. Before transferring control to C code, the memory locations corresponding to these variables have to be initialized to zero.

.bss (block started by symbol)

Interview trick

bss is not in flash as it is not have a value, we just reserve a section for it in ram by knowing its size and initialize this section by zero.

Since those variables do not have any intial values, they are not required to be stored in .data section (.data section is stored in Flash)

All uninitialized (global/static)variables are stored in .bss

Read-only Data

GCC places global variables marked as const in a separate section, called .rodata. The .rodata is also used for storing string constants.

Since contents of .rodata section will not be modified, they can be placed in Flash. The linker script has to modified to accomodate this

Load Location (LMA - Load Memory Address)

Definition:

- The Load Location is the address where the program or data is loaded into memory during the initialization or loading phase. This is the physical memory address where the code or data is placed before the program starts executing.

Usage:

- **Embedded Systems**: In embedded systems, the Load Location is where the firmware is loaded into memory from non-volatile storage (e.g., flash memory) into RAM or another execution space.
- **Linker Scripts**: The Load Location is often specified in linker scripts to define where various sections of the code or data should be placed in memory.

Runtime Location (VMA - Virtual Memory Address)

- **Definition**:
  - The Runtime Location refers to the address that the program uses during execution to access its code or data. This is often an address within the program's virtual memory space.
- **Usage**:
  - **Virtual Memory**: In systems with virtual memory, the Runtime Location is the address as seen by the program. The operating system's memory management unit (MMU) maps this virtual address to a physical memory address.
  - **Debugging**: During debugging, you might refer to the Runtime Location to understand where variables and functions are accessed during program execution.
- **Example**:
  - When a program is running, it may access a variable at a virtual address 0x20001000. This is its Runtime Location, which is mapped to a physical address by the operating system.

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		

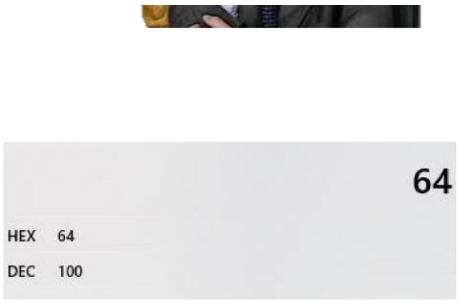
Where is a .rodata section we didn't find it, why and could you generate it now ?

```
$ 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         0000001c  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000064  00000000  00000000  00000050  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000000b4  2**0
                  ALLOC
  3 .comment      0000007f  00000000  00000000  000000b4  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000032  00000000  00000000  00000133  2**0
                  CONTENTS, READONLY
```

- GCC places global variables marked as const in a separate section, called .rodata. The .rodata is also used for storing string constants.

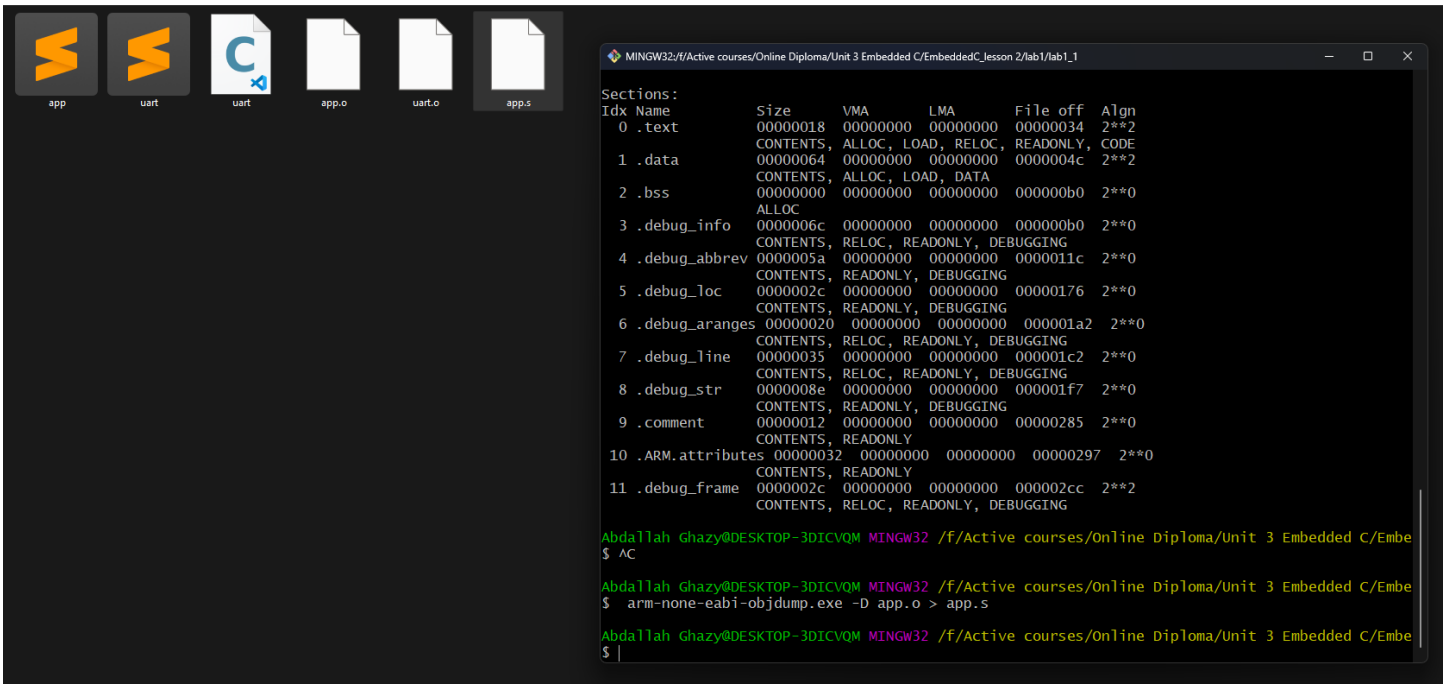
```
1 //@ learn-in-depth.com
2 //Mastering Embedded System online Diploma
3 // End.Keroles Shenouda
4 #include "uart.h"
5 unsigned char string_buffer[100] = "learn-in-depth:<keroles>" ;
6 unsigned char const string_buffer_2[100] = "to create a rodata section" ;
7
8 void main(void)
9 {
10 //VersatilePB physical Board
11 Uart_Send_string(string_buffer);
12 }
```

```
$ 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         0000001c  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000064  00000000  00000000  00000050  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000000b4  2**0
                  ALLOC
  3 .rodata       00000064  00000000  00000000  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000007f  00000000  00000000  00000118  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 00000032  00000000  00000000  00000197  2**0
                  CONTENTS, READONLY
```



You can see that the rodata section size = 100 Bytes as expected

Let us generate the disassembly file from the bin




```
1
2  app.o:     file format elf32-littlearm
3
4
5  Disassembly of section .text:
6
7  00000000 <main>:
8     0: e92d4800    push  {fp, lr}
9     4: e28db004    add   fp, sp, #4
10    8: e59f0004    ldr   r0, [pc, #4]   ; 14 <main+0x14>
11    c: ebfffffe    bl 0 <uart_send_string>
12   10: e8bd8800    pop   {fp, pc}
13   14: 00000000    andeq r0, r0, r0
14
15 Disassembly of section .data:
16
17 00000000 <string_buffer>:
18    0: 7261654c    rsbvc r6, r1, #76, 10   ; 0x13000000
19    4: 6e692d6e    cdpvs 13, 6, cr2, cr9, cr14, {3}
20    8: 7065642d    rsbvc r6, r5, sp, lsr #8
21    c: 413a6874    teqmi sl, r4, ror r8
22   10: 6c616462    cfstrdvs mvd6, [r1], #-392 ; 0xfffffe78
23   14: 2068616c    rsbcs r6, r8, ip, ror #2
24   18: 7a616847    bvc   185a13c <main+0x185a13c>
25   1c: 00000079    andeq r0, r0, r9, ror r0
26    ...
```

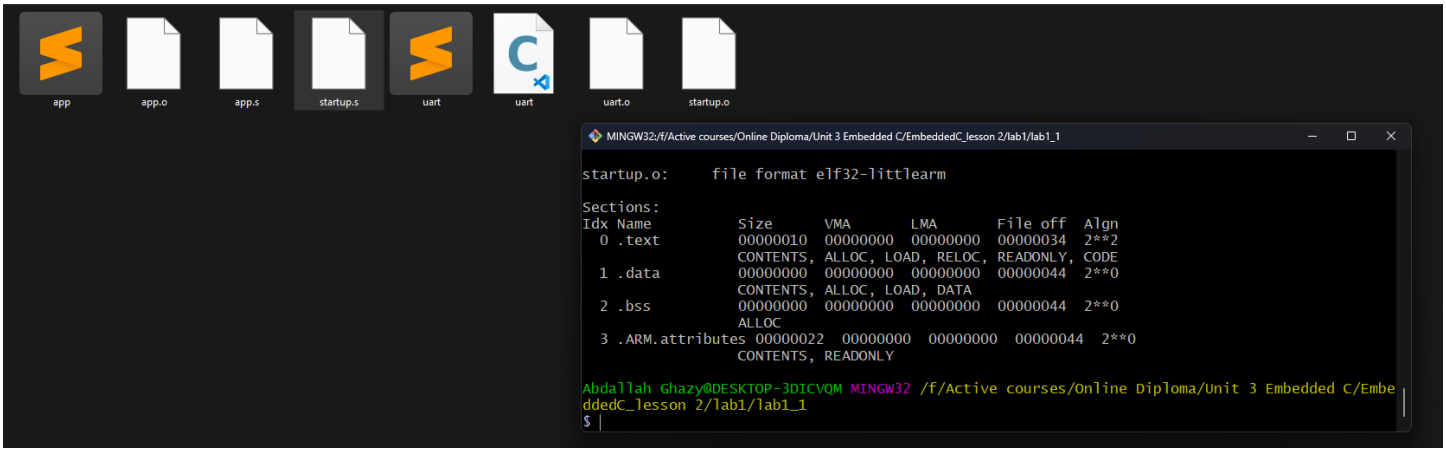
In Lab1: We will write a simple startup:

1. Create a reset section and Call main( ).

2. Initialize Stack

app.i | app.o | app.s | startup.s | uart.h | uart.c | app.c

```
1 .global reset
2 reset:
3     ldr sp, =stack_top
4     bl main
5 stop: b stop
```



The Linker Script and Locator (Linker Locator)

Locator (Linker Locator)

Definition

A **locator** (often referred to as a linker locator or memory locator) is not a separate tool but rather refers to the functionality or components within the linker that handle the placement and address calculation of code and data sections in memory.

Functions

1. **Address Calculation**:
    - Calculates and assigns addresses to code and data sections based on the linker script.
2. **Memory Management**:
    - Manages how sections are placed in specific memory regions and ensures they meet alignment requirements.
3. **Section Handling**:
    - Ensures that sections are located according to the directives in the linker script.

Linker Script

Definition

A **linker script** is a text file used to control the behavior of the linker during the linking process. It specifies how different sections of the code and data should be arranged in memory and how they should be combined.

Functions

1. **Memory Layout**:
    - Defines memory regions and their properties (e.g., FLASH, RAM).
    - Specifies the starting addresses and sizes of different memory regions.
2. **Section Placement**:
    - Controls where different sections of the program (e.g., .text, .data, .bss) should be placed in memory.
    - Determines how sections from various object files are combined.
3. **Symbol Definitions**:
    - Can define symbols or variables that the linker will use.
4. **Alignment**:
    - Specifies alignment requirements for sections.

reflect exactly the memory resources and memory map of the target microcontroller

GNU linker script has the file extension *.ld

You have to use the linker script at the linking phase by pass to the liker option –T

Common Linker Script Commands

SECTIONS

- **Purpose**: Defines the layout of the sections in memory.
- **Usage**: Specifies how different sections (like .text, .data, .bss) are mapped to memory.

- Example:

```
ld
SECTIONS
{
    .text : {
        *(.text)
    } > FLASH

    .data : {
        *(.data)
    } > RAM AT > FLASH

    .bss : {
        *(.bss)
    } > RAM
}
```

ENTRY

- **Purpose**: Specifies the entry point of the program.
- **Usage**: Defines where the execution starts.

- Example:

```
ld
ENTRY(_start)
```

OUTPUT_FORMAT

- **Purpose**: Sets the format of the output file.
- **Usage**: Specifies the format for the generated executable or object file.

  - Example:

```
ld                                                    نسخ الكود

OUTPUT_FORMAT("elf32-littlearm")
```

MEMORY

- **Purpose**: Defines the memory regions available for use.
- **Usage**: Specifies the start, end, and size of different memory regions.

- Example:

```
ld                                                    نسخ الكود

MEMORY
{
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}
```

PHDRS

- **Purpose**: Defines program headers for executables.
- **Usage**: Specifies the type of segments for the executable file.

- Example:

```
ld                                                    نسخ الكود

PHDRS
{
    text PT_LOAD;
    data PT_LOAD;
}
```

AT

- **Purpose**: Specifies the load address of a section.
- **Usage**: Indicates where a section should be loaded in memory.

- Example:

```
ld
.data : {
    *(.data)
} > RAM AT > FLASH
```

STARTUP

- **Purpose**: Specifies the startup file.
- **Usage**: Defines the initialization routine or startup code to be linked.

  - Example:

```
ld
ENTRY(_start)
```

PROVIDE

- **Purpose**: Defines a symbol with a default value.
- **Usage**: Useful for defining default values or placeholders.

- Example:

```
ld
PROVIDE(_end = .);
```

GROUP

- **Purpose**: Groups multiple files or sections together.
- **Usage**: Ensures that all files or sections in the group are included in the output.

- Example:

```ld
GROUP(file1.o, file2.o)
```

INCLUDE

- **Purpose**: Includes other linker script files.
- **Usage**: Modularizes linker scripts by including additional scripts.

- Example:

```ld
INCLUDE "common.ld"
```

Summary

- **SECTIONS**: Specifies the memory layout.
- **ENTRY**: Defines the entry point of the program.
- **OUTPUT_FORMAT**: Sets the output file format.
- **MEMORY**: Defines memory regions.
- **PHDRS**: Specifies program headers.
- **AT**: Indicates the load address of a section.
- **STARTUP**: Specifies startup code.
- **PROVIDE**: Defines default values for symbols.
- **GROUP**: Groups files or sections together.
- **INCLUDE**: Includes additional linker scripts.

Linker script command's location counter

Linker symbol '.' Dot

This dot is called "location counter" it is automatically address calculated by each section size

We can use it to track and define the memory layout boundaries υ

Also we can use it to specify specific address for specific section

Location counter should use only in sections command

Linker script commands using >(vma) AT>(lma)

vma is specify relocatable section address in run-time located

Lma is specify relocatable section address in load-time located

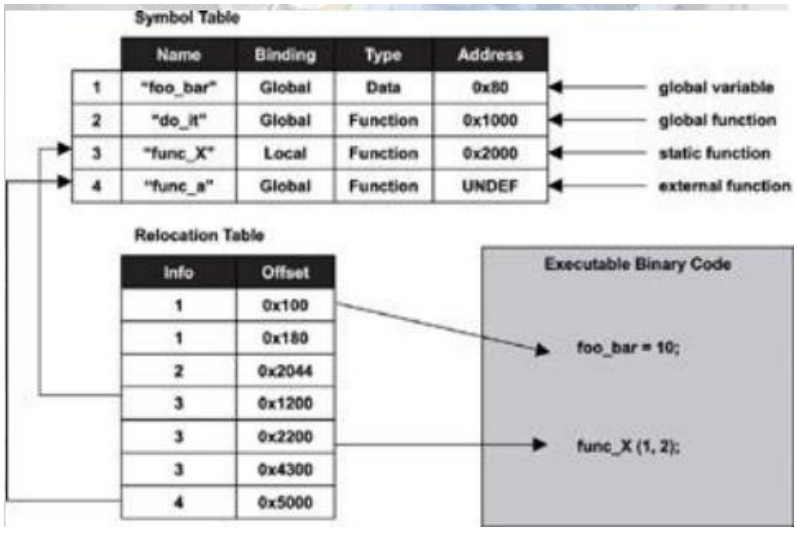
Linker script (Symbols)

Symbol is the name of an address

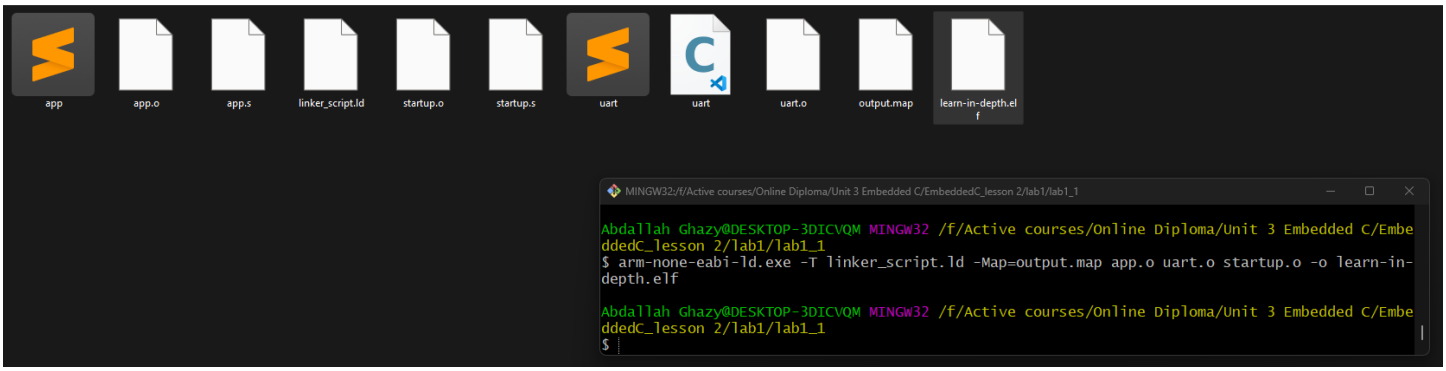
Symbol declaration is not equivalent to variable declaration.

Each object have its own symbol table, the linker is resolving the symbols between all obj files.

Symbol also is used to specify Memory layout boundaries

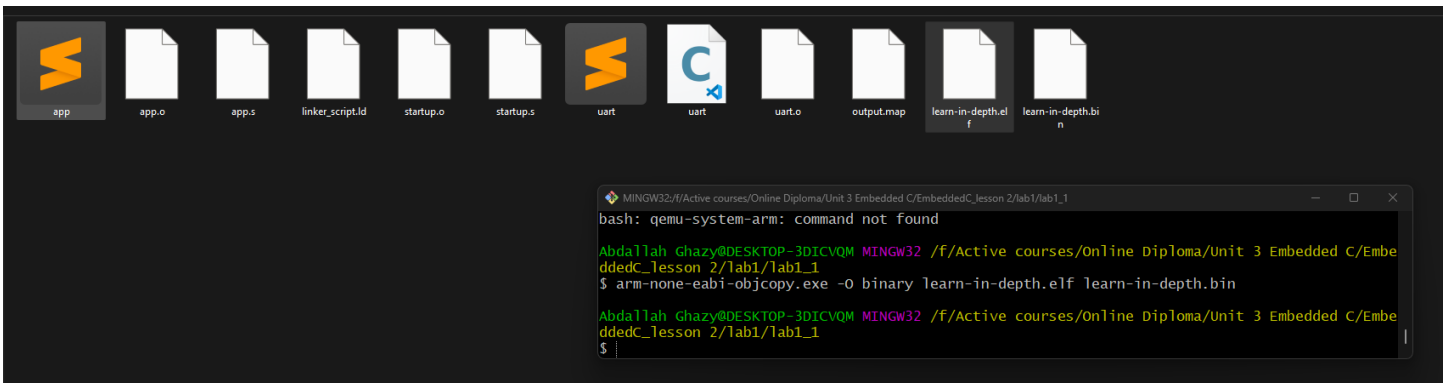


```
MINGW32:/f/Active courses/Online Diploma/Unit 3 Embedded C/EmbeddedC_lesson 2/lab1/lab1_1
  1 .data         00000000  00000000  00000000  00000044  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000044  2**0
                  ALLOC
  3 .ARM.attributes 00000022  00000000  00000000  00000044  2**0
                  CONTENTS, READONLY

Abdallah Ghazy@DESKTOP-3DICVQM MINGW32 /f/Active courses/Online Diploma/Unit 3 Embedded C/Embe
ddedC_lesson 2/lab1/lab1_1
$ arm-none-eabi-nm.exe app.o
00000000 T main
00000000 D string_buffer
         U uart_send_string

Abdallah Ghazy@DESKTOP-3DICVQM MINGW32 /f/Active courses/Online Diploma/Unit 3 Embedded C/Embe
ddedC_lesson 2/lab1/lab1_1
$
```



The .map file gives a complete listing of all code and data addresses for the final software image. It provides information similar to the contents of the linker script described earlier. However, these are results rather than instructions and therefore include the actual lengths of the sections and the names and locations of the public symbols found in the relocatable program

Generate binary file



To run the program in the QEMU Simulator ("VersatilePB physical Board")