Embedded C

Prepared by: Ahmed Hassan Ibrahim

LinkedIn Link: https://www.linkedin.com/in/ahmedhassanese/

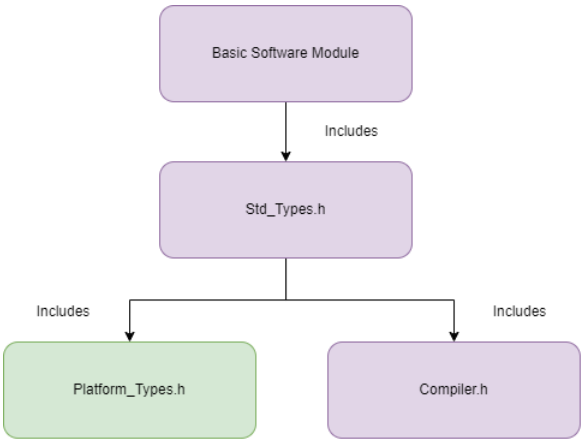
Table of Contents

Typedef

Notes:

- When writing typedef, use **all letters capital**. EX: NAME_VAR.
- Put "**_t**" after typedef name.
- Put **S capital if structure** and **U if union and e if enum**.

Typdef in AUTOSAR:



Platform_Types.h header file includes all the type definitions used in the board.

- Basic typedefs are included in header file "**stdint.h**"

Header Protection:

Header protection in C refers to the use of *include guards* or *#pragma once* to prevent **multiple inclusion** of the same header file in a program, which can cause **redefinition errors**.

```c
#ifndef HEADER_FILE_NAME_H
#define HEADER_FILE_NAME_H

// Declarations go here

#endif
```

Optimization:

Optimization in C refers to techniques used to make a program **run faster**, **use less memory**, or **perform more efficiently** without changing its functionality.

Optimization level O0

- No optimization.
- Not recommended for productions if you have limited code and ram space.
- Has fastest compilation time
- This is debugging friendly and used during development
- A code which works with O0 may **not work with O0+ optimization levels**.
    - Code needs to be verified again when applying higher optimization levels

Optimization level O+

- **Optimization level O1**
    - Moderate optimization to decrease memory access and code space
- **Optimization level O2**
    - Full optimization
    - Slow compilation time
    - Not debugging friendly
- **Optimization level O3**
    - Full optimization of O2 + some more aggressive steps will be taken by the compiler.
    - Slowest compilation time
    - May cause bugs in the program
    - Not debugging friendly

Volatile Type Qualifier:

volatile type qualifier in C tells the compiler **not to optimize** a variable because its value may change **unexpectedly**, such as by hardware or another thread.

Proper Use of C's volatile Keyword:

- Memory-mapped peripheral registers
  (e.g., when a pointer points to a hardware register and the value at that address may change independently of the program, such as by hardware events)

```c
int main()
{
    uint8_t* pReg = (uint8_t*) 0x1234
    //Wait for register to become non-zero
    while(*pReg == 0){}
    return 0;
}
```

```
    mov ptr, #0x1234
    mov a, @ptr
loop:
    bz loop
```

To force the compiler to do what we want, we modify the declaration to:

```
int main()
{
    uint8_t volatile *pReg = (uint8_t volatile*) 0x1234
    //Wait for register to become non-zero
    while(*pReg == 0){}
    return 0;
}
```

```
    mov ptr, #0x1234
loop:
    mov a, @ptr
    bz loop
```

- Global variables modified by an interrupt service routine

```
int etx_rcvd = FALSE;
int main()
{
    ...
    while(!etx_rcvd)
    {
        ...
    }
    ...
}
//Interrupt
void rx_isr(void)
{
    if(ETX == rx_char)
        etx_rcvd = TRUE
}
```

Optimization will cause issues because it will assume that etx-rcvd **will always be FALSE** and thus will **remove the while**. To fix it we define the **global variable as volatile**.

- Global variables accessed by multiple tasks within a multi-threaded application

```
int cntr;
void task1(void)
{
    cntr = 0;
    while(cntr == 0)
        sleep(1);
}
void task2(void)
{
    cntr++;
    sleep(0);
}
```

This code will likely fail once the compiler's optimizer is enabled. Declaring cntr to be volatile is the proper way to solve the problem. **The compiler does not know there is a timer that triggers context switching**.

Writing on Registers:

Write 0xFFFFFFFF on SIU register which have absolute address 0x30610000

- Solution 1

```c
//Decalre a pointer
volatile int *p;
//Assign the address of the I/O memory location to the pointer
p = (volatile int*) 0x30610000;
//Output a 32-bit value
*p = 0xFFFFFFFF;
```

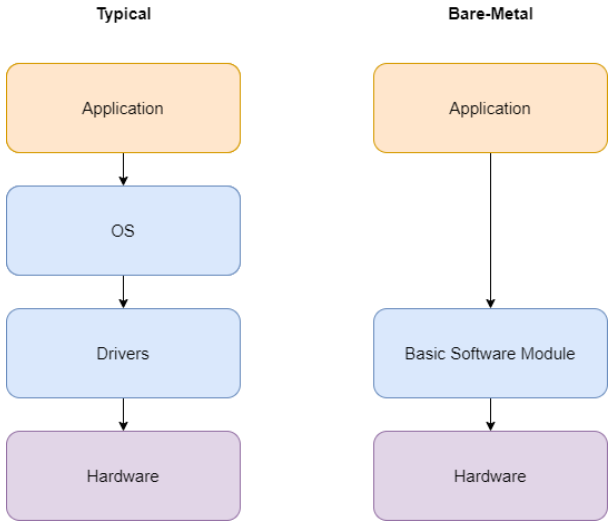
- Solution 2

```c
#define REGISTER *((volatile unsigned long*)(0x306100))
REGISTER = 0xFFFFFFFF;
```

- Solution 3

```c
typdef union
{
    vuint32_t ALL_ports;
    struct
    {
        vuint32_t PORTA:8;
        vuint32_t PORTB:8;
        vuint32_t PORTC:8;
        vuint32_t PORTD:8;
    }SIU_fields;
}SIU_R;

volatile SIU_R* PORTS = (volatile SIU_R*) 0x306100;
PORTS->ALL_PORTS = 0xFFFFFFFF;
PORTS->SIU_fields.PORTA = 0xFF;
```

Bare metal Embedded SW:



Bare-Metal Embedded Software: Software that runs directly on the hardware without any operating system.

- No OS, no abstraction layers.
- Direct control over peripherals and memory.
- Uses startup code, main loop, and interrupt service routines.
- Requires manual handling of scheduling and timing.

Typical Software: Software that runs on an embedded operating system or real-time operating system (RTOS).

- Uses threads, tasks, and schedulers.
- May include middleware (e.g., file systems, networking stacks).
- Offers services like inter-task communication, time management, etc.

Cross-compiling Toolchains:

Native Toolchain: A set of development tools (compiler, linker, assembler, etc.) that runs on and generates code for the **same architecture and platform (host and target)**.

Cross Toolchain: A toolchain that runs on one platform (the **host**) but generates executable code for a **different platform or architecture** (the **target**).

Command	Action
gcc	Compiling C/C++ (and other languages) code into object files Also it **calls a linker internally** to link object files into an executable
ld	**Links** object files into an executable (called internally by gcc)
make	Reads the **Makefile** to manage the **building process**
ar	**Archives** multiple Object files into **a static Library**
readelf	Reads the **contents** of an ELF File **(object file or executable)**

objdump	Reads the internals of an ELF file **including assembly** code
nm	Reads the **symbols** inside an object file or and executable
strings	Reads **text strings** inside a binary file
strip	**Strips** the **binary file** from **some optional sections**
addr2line	**Convert** an **address** in the binary to **a source file name and line number**
size	Display the **ELF file section sizes** and **total size**
gdb	Debugger

Components – GCC:

- To perform complete compilation process of the main.c source
  - **$ gcc main.c**
- To change the name of the generated executable
  - **$ gcc main.c-o test**
- To specify the Include path (path for header files)
  - **$ gcc main.c-I/usr/share/include -I. -I./inc/ -I../../inc**
- To enable all warning during compilation process
  - **$ gcc-Wall main.c-o test**
- To convert warnings into errors
  - **$ gcc-Wall -Werror main.c-o test**
- To pass options to gcc in a file (instead of in the command)
  - **$ gcc main.c @options-file**

GCC Partial Compilation:

- **$gcc -E main.c > main.i**
  This will only generate main.i file which is a pre processed source file
- **$gcc -S main.c > main.s**
  This will generate an assembly file main.s
- **$gcc -C main.c > main.c**
  This will generate a relocatable object file main.o
- **$gcc -save-temps main.c -o test**
  This will save:
  • File after pre-processing (main.i)
  • File after compilation (main.s)
  • File after assembly (main.o)
  • Executable (test)

Creating a Static Library:

A **static library** is a file that contains compiled code (object files) which can be reused in multiple programs.

- It is **linked** into the final executable **at compile time**, becoming part of the binary.

static libraries can be used to enhance security in embedded systems by hiding the source code.

- To create a static library file
  - **$ ar rcs libmylib.a file1.o file2.o**
- Adding another object file to the library
  - **$ ar rlibmylib.a file3.o**
- To remove an object file from the library
  - **$ ar dlibmylib.a file3.o**
- To view the object files in the library
  - **$ ar tlibmylib.a**
- To extract object files from the library
  - **$ ar xlibmylib.a**

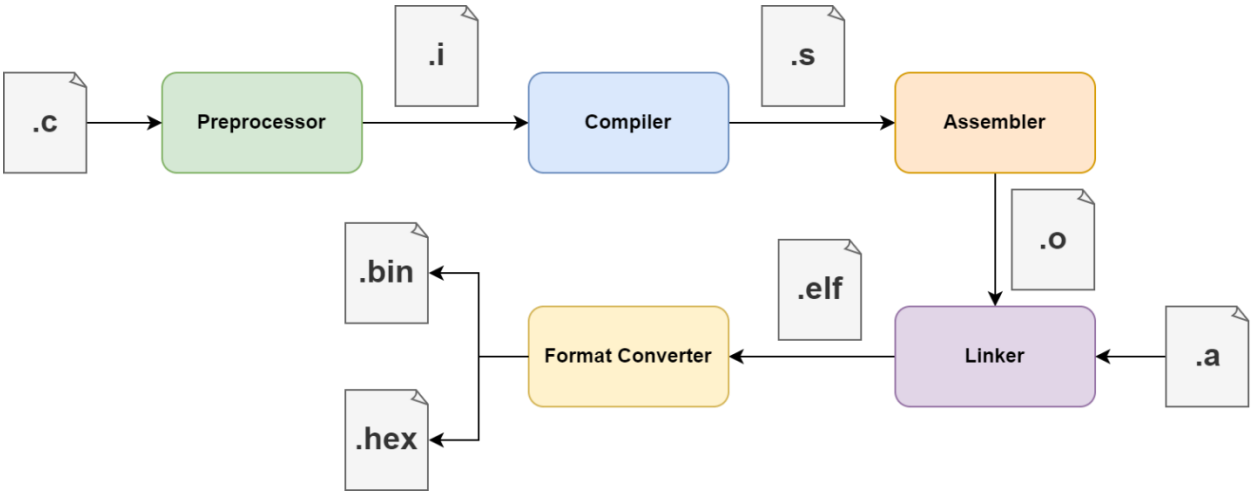
ARM Cross-toolchain:

An **ARM cross toolchain** is a collection of development tools (compiler, assembler, linker, etc.) that **runs on one platform (the host)** — usually your PC — and **generates executable code for an ARM-based target system** like a microcontroller or ARM SoC.

arm-none-eabi commands used for bare-metal software (ARM board directly).

arm-linux-eabi commands used for Linux based architecture (NOT ARM board directly).

Compilation Process:



- **Preprocessing (Preprocessor)**
  - It processes include files, conditional compilation instructions and macros.
  - Command: $cpp hello.c hello.i
  - hello.c is source program, hello.i is ASCII intermediate code.
- **Compiling (Compiler)**
  - It takes the output of the preprocessor and generates assembler source code.
  - Command: $cc hello.i–o hello.s
- **Assembly (Assembler)**
  - It takes the assembly source code and produces an assembly listing with offsets.
  - The assembler output is stored in an object file.
  - Command:$as –o hello.o hello.s
- **Linking (Linker)**
  - It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file.
  - Linux command for linker is ld.

❖ **Note regarding linker file ".ld"**

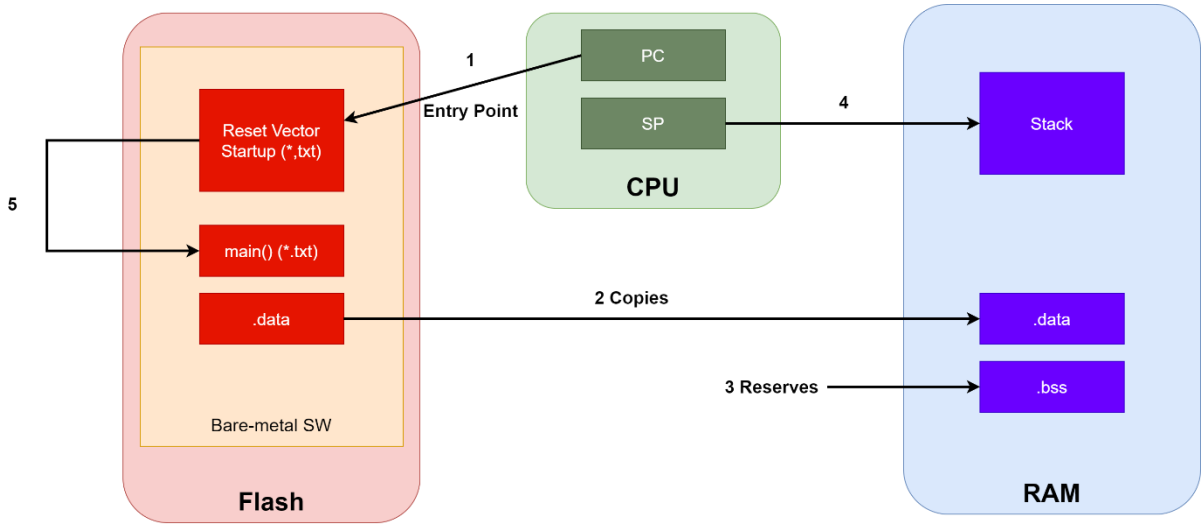
- Specifies **how** and **where** in memory different sections of the program (code, data, stack, etc.) should be placed.
- Controls **memory layout**, **section placement**, and **alignment**.
- Often used in **embedded systems** to map sections into flash, RAM, peripherals, etc.
- **Does not resolve dependencies itself.**

Boot Sequence:

When a processor is powered on or reset, it automatically begins execution from a **predefined default address (entry point)**. Hardware designers use this information to configure the Flash memory layout and address range. **This ensures that the CPU fetches the initial code (such as a bootloader or bare-metal software) from a known location, allowing software control to be established reliably.**

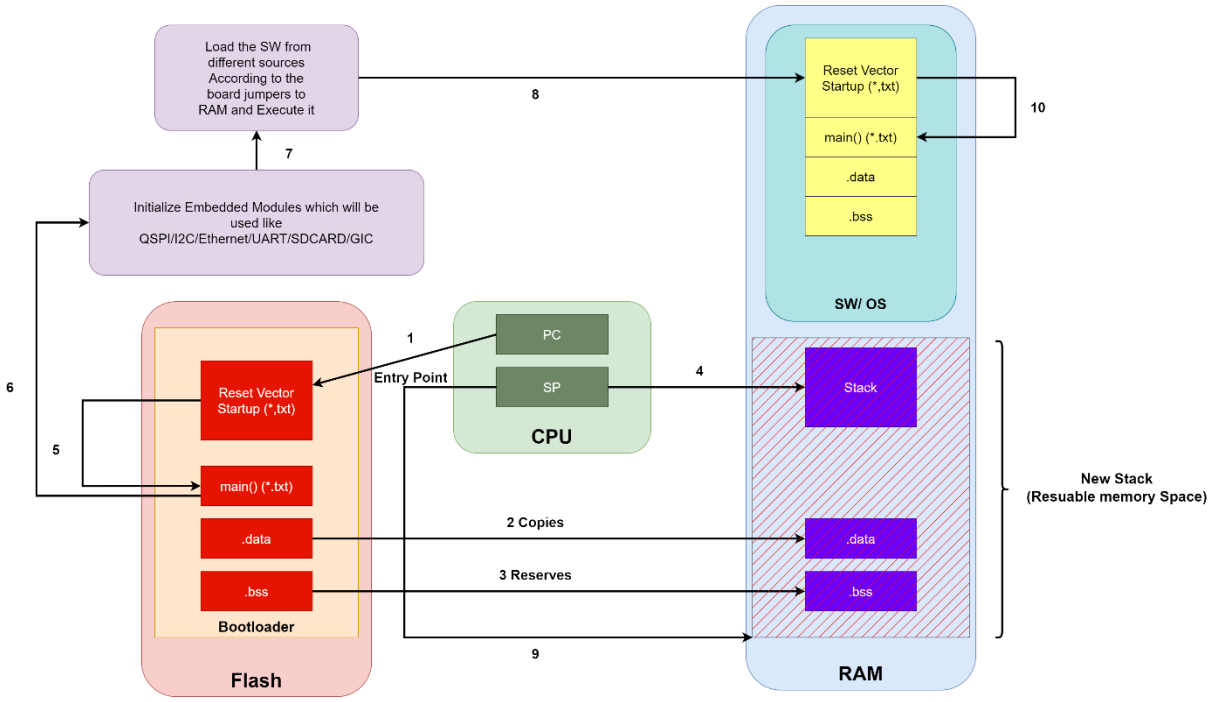
- **Bootloader:** A "startup assistant" that prepares the system and may load another program (like bare-metal SW or an OS).
- **Bare-metal SW:** Your main program running directly on hardware without an operating system.

Case 1:



1. **PC** is pointing to the **entry point** in which the programmer stored in the **reset** section of the code in the **startup address**. The **Reset Vector initializes the processor**.
2. The startup transferred the **".data"** section (**containing all static and initialized global variables**) to the **RAM**.
3. **Reserve a ".bss" space in the RAM** to store the **uninitialized global variables with zero**.
4. The **SP** register points to the **starting address of the stack**.
5. Now the startup has finished all its tasks and will **call the main function**.

Case 2:



1. **PC** is pointing to the **entry point** in which the programmer stored in the **reset** section of the code in the **startup address**. The **Reset Vector initializes the processor**.
2. The startup transferred the **".data"** section (**containing all static and initialized global variables**) to the **RAM**.
3. **Reserve a ".bss" space in the RAM** to store the **uninitialized global variables with zero**.
4. The **SP** register points to the **starting address of the stack**.
5. Now the startup will **call the main function**.
6. The **bootloader main function** will **initialize Embedded Modules** which will be used.
7. Also, the bootloader code will **load the SW from different sources** according to the **board jumpers** to RAM and Execute it.
8. Now the **PC** points to the **reset vector of the SW loaded by the bootloader**. Now the bootloader job is complete.
9. Now that the bootloader job is complete, the bootloader's stack, .data, and .bss sections in the RAM **won't be needed** and **they will be used as an extra memory space serving the loaded SW** in a **stack format**. The **starting address of that stack** will now be pointed at by **SP register**.
10. Now the startup has finished all its tasks and will **call the main function of the loaded SW**.

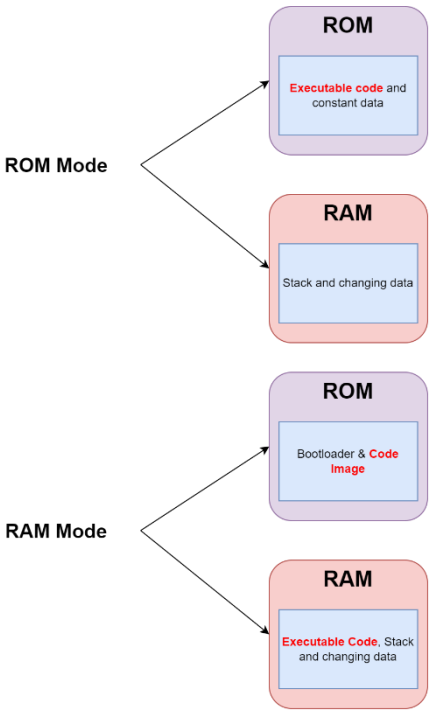
Running Mode:

ROM Mode (Case 1):

- Simple
- Require smaller memory
- Fixed code address
- Relatively Small Code

RAM Mode (Case 2):

- Complex
- Re-locatable code
- Faster (RAM faster than ROM)
- Large Code (SDRAM)



Bootloader Tasks:

- Basic hardware initialization
- Loading of an application binary, usually an operating system kernel:
  - From flash storage
  - From the network
  - From SDCard or from another type of non-volatile storage
  - From USB client
- Possibly decompression of the application binary
- Execution of the application
- most bootloaders provide a shell with various commands implementing different operations.
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc

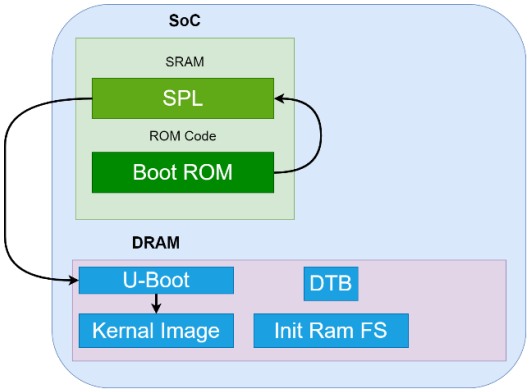
3 phases boot sequence:

Phase1 – (ROM Code):

- Runs immediately after reset or power-on and is stored on-chip in the SoC.
- **Permanently loaded** during manufacturing; **cannot be replaced or updated**.
- Typically, **it does not initialize the memory controller due to device-specific DRAM configurations; uses on-chip SRAM instead**.
- SoCs include small on-chip SRAM (**4 KB to several hundred KB**).
- ROM code loads a small piece of code into SRAM from predefined locations.
- If **SRAM is too small for a full bootloader** (e.g., U-Boot), a secondary program loader (**SPL**) is used as an intermediate step.

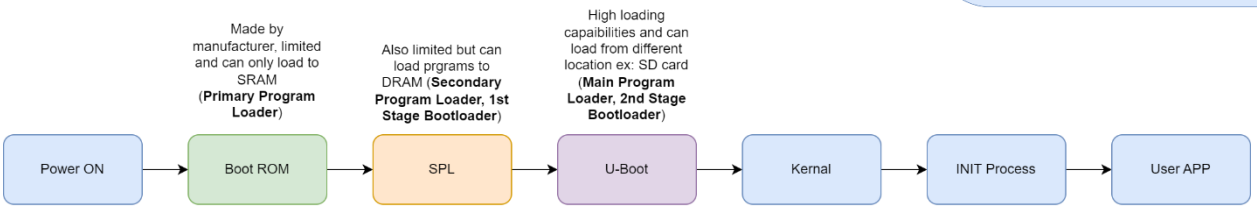
Phase 2 – secondary program loader:

- Initializes the memory controller and essential system components to prepare for loading U-Boot into DRAM.
- Runs from limited on-chip SRAM, so its functionality is minimal.
- It can read programs from various storage devices using built-in drivers.
- Capable of locating files like u-boot.img on disk partitions.



Phase 3 – U-BOOT:

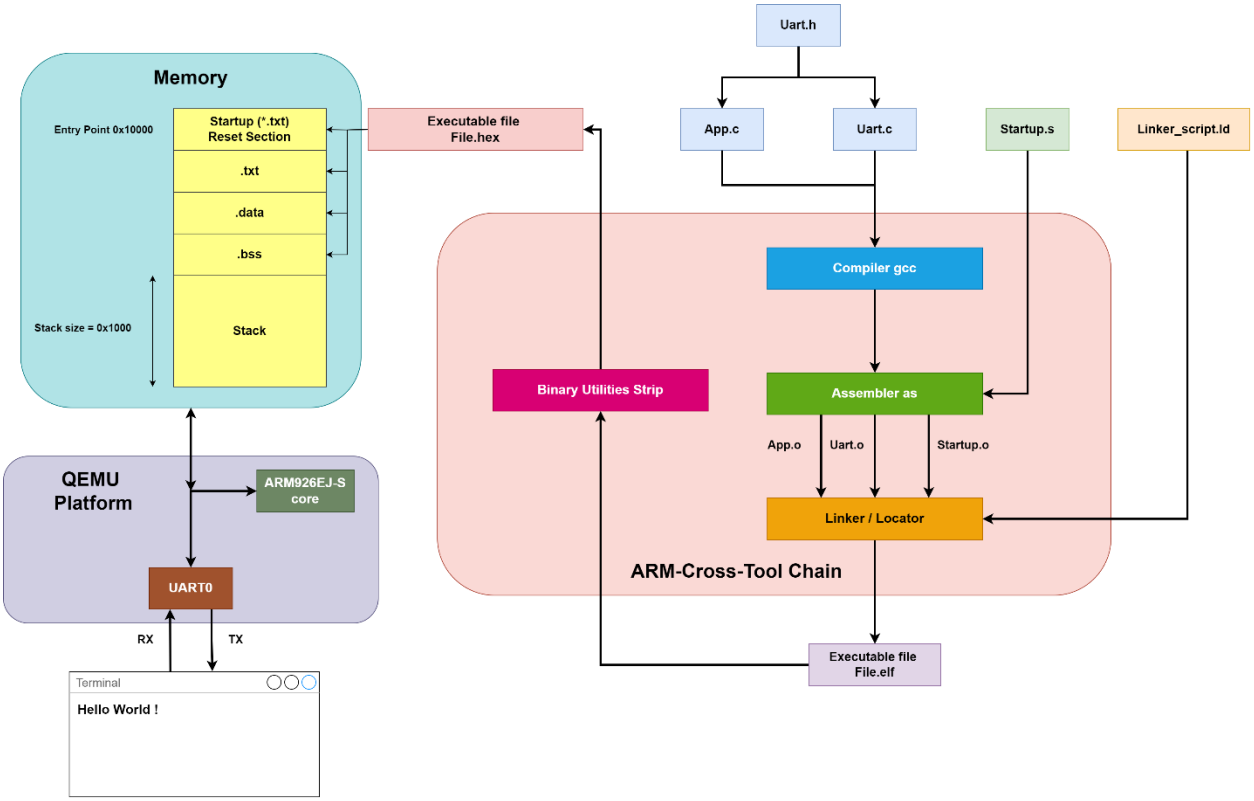
- A full-featured bootloader is now running.
- Provides a command-line interface for maintenance tasks, such as loading new boot or kernel images into RAM.



- **Note:** We don't need SPL in case the SRAM can run the U-Boot.

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
```

- We runt the following command to create object file for startup:
  arm-none-eabi-as.exe -mcpu=arm926ej-s -g startup.s -o startup.o

Linker Script:

Linker Script Commands:

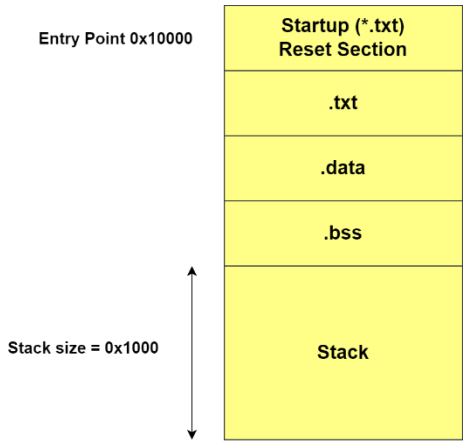
Command	Description
ENTRY	Defines the entry point of an application, which appears in the final ELF file header. Typically, the entry point is the reset handler, executed after a processor reset. This information helps the debugger (like **gdb**) identify the starting function. While not mandatory, it 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

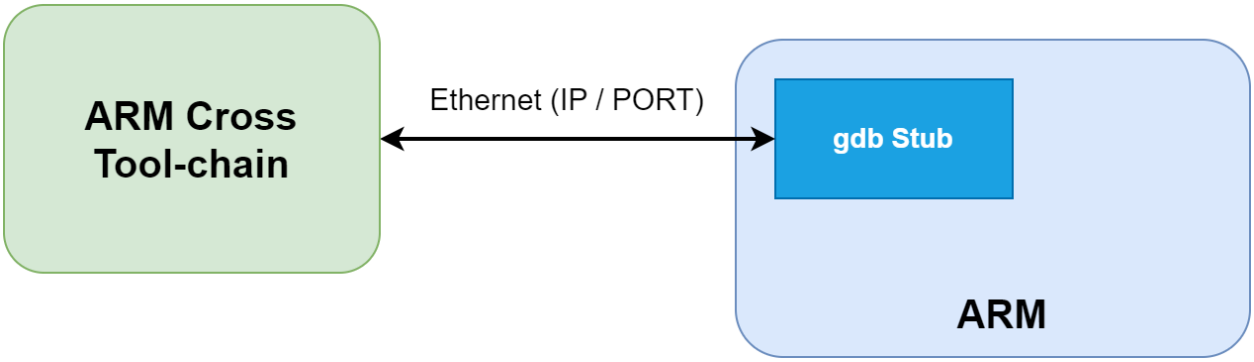
```
MINGW64:/d/Ahmed.H/Education/Embedded System Diploma/Working_Directory/Embedded_System_Online_Diploma/Unit_03_Embedded_C/Less...
hp@TyranZilla MINGW64 /d/Ahmed.H/Education/Embedded System Diploma/Working_Directory
ed_System_Online_Diploma/Unit_03_Embedded_C/Lesson_2/Lab_1 (master)
$ qemu-system-arm -M versatilepb -m 128M -nographic -kernel learn-in-depth.bin
Learn-in-depth: <Ahmed Hassan>
```

GDB Debugger Commands:

GDB (GNU Debugger) is a powerful debugging tool for C and C++ programs on UNIX systems. It allows developers to **control program execution**, **inspect variable values**, and **diagnose problems**. GDB can help identify where a program crashes (e.g., from a core dump), trace function calls and parameters during errors, examine variable values at specific execution points, and evaluate expressions during runtime.

- To enable built-in debugging info, include "**-g**" when creating object files using the command line.

To debug the code:



To debug using the ARM toolchain and GDB stub over Ethernet, the target device must run a **GDB-compatible stub** or server that listens for connections over the network. The host machine runs arm-none-eabi-gdb, which connects to the target using its **IP address** and **port number**. This setup allows the developer to remotely control the target for setting breakpoints, stepping through code, and inspecting memory or variables—all over **Ethernet**.

We will use 2 terminals, one for tool-chain and the other for simulated ARM MCU.

- We start the simulation using the following commands, also include "-s -S" to start debugging:
  qemu-system-arm -M versatilepb -m 128M -nographic -s -S -kernel learn-in-depth.elf

- We run the following command to start debug from tool-chain:
  arm-none-eabi-gdb.exe learn-in-depth.elf

- Now we need to start connections using local host IP address + "1234":
  target remote localhost:1234

GDB cheatsheet - page 1

Running

`# gdb <program> [core dump]`
Start GDB (with optional core dump).

`# gdb --args <program> <args…>`
Start GDB and pass arguments

`# gdb --pid <pid>`
Start GDB and attach to process.

`set args <args...>`
Set arguments to pass to program to be debugged.

`run`
Run the program to be debugged.

`kill`
Kill the running program.

Breakpoints

`break <where>`
Set a new breakpoint.

`delete <breakpoint#>`
Remove a breakpoint.

`clear`
Delete all breakpoints.

`enable <breakpoint#>`
Enable a disabled breakpoint.

`disable <breakpoint#>`
Disable a breakpoint.

Watchpoints

`watch <where>`
Set a new watchpoint.

`delete/enable/disable <watchpoint#>`
Like breakpoints.

<where>

`function_name`
Break/watch the named function.

`line_number`
Break/watch the line number in the current source file.

`file:line_number`
Break/watch the line number in the named source file.

Conditions

`break/watch <where> if <condition>`
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

`condition <breakpoint#> <condition>`
Set/change the condition of an existing break- or watchpoint.

Examining the stack

`backtrace`
`where`
Show call stack.

`backtrace full`
`where full`
Show call stack, also print the local variables in each frame.

`frame <frame#>`
Select the stack frame to operate on.

Stepping

`step`
Go to next instruction (source line), diving into function.

`next`
Go to next instruction (source line) but don't dive into functions.

`finish`
Continue until the current function returns.

`continue`
Continue normal execution.

Variables and memory

`print/format <what>`
Print content of variable/memory location/register.

`display/format <what>`
Like „print", but print the information after each stepping instruction.

`undisplay <display#>`
Remove the „display" with the given number.

`enable display <display#>`
`disable display <display#>`
En- or disable the „display" with the given number.

`x/nfu <address>`
Print memory.
n: How many units to print (default 1).
f: Format character (like „print").
u: Unit.

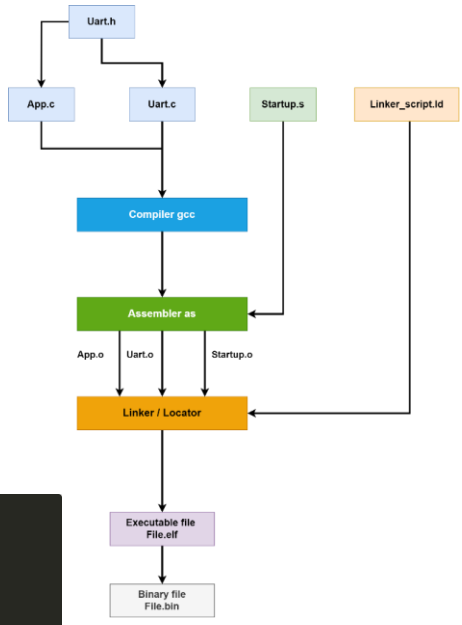
Unit is one of:

b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

Makefile:

Makefiles are a simple way to **organize code compilation**. **It only compiles the updated files and leaves the rest**.

- Now we create the Makefile:
  Note that:
  - **$< : dependencies**
  - **$@ : target**
  - **(Generic) Rules: %**



- In the terminal we run the following command:
  make

```
#@arm-none-eabi-#@Copyright: Ahmed Hassan

CC=arm-none-eabi-
CFLAGS=-g -mcpu=arm926ej-s
INCS=-I .
LIBS=
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)
As=$(wildcard *.s)
AsOBJ=$(As:.s=.o)
Project_name=learn_in_depth

all: $(Project_name).bin
    @echo "<<============= Build Complete =============>>"

startup.o: startup.s
    $(CC)as.exe $(CFLAGS) $< -o $@

%.o: %.c
    $(CC)gcc.exe -c $(CFLAGS) $(INCS) $< -o $@

$(Project_name).elf: $(OBJ) $(AsOBJ)
    $(CC)ld.exe -T linker_script.ld $(LIBS) -Map=output.map $(OBJ) $(AsOBJ) -o $@

$(Project_name).bin: $(Project_name).elf
    $(CC)objcopy.exe -O binary $< $@

clean_all:
    rm *.o *.elf *.bin

clean:
    rm *.elf *.bin
```

CMake vs. GMake:

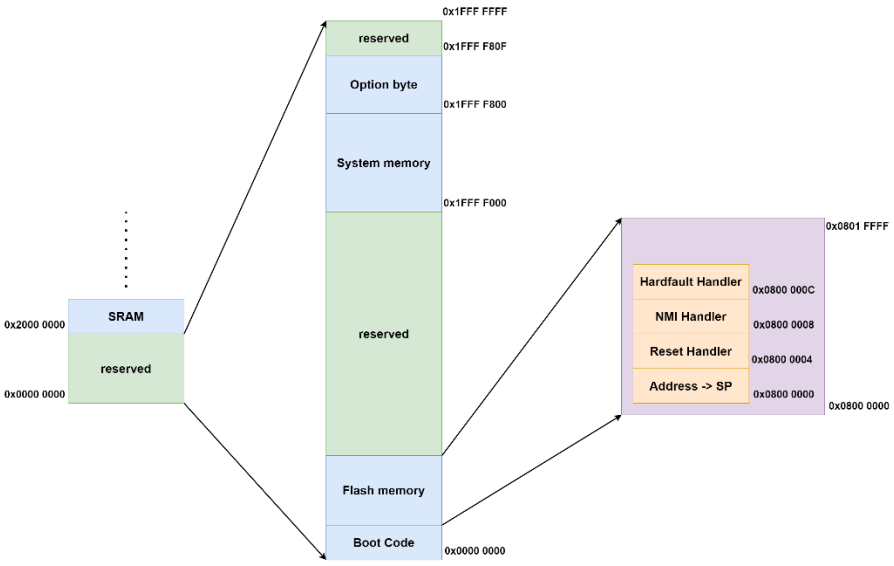
- Make or Gmake uses Makefiles to **describe how to build your code**
  - A Makefile contains **targets** and **recipes**.
  - A target can be an **object file** or **executable** and the recipe is how to produce that target.
- Cmake is a higher-level language with the same purpose but with a **higher level of abstraction**.
  - With cmake you can generate ordinary Makefiles or files for other **build systems**.
  - It generates the **projects** for programs like (VScode, CodeBlocks, Eclipse....)

Lab2 – Bare-metal toggle led on STM32:

This lab covers creating a bare-metal Software to blink a LED in ARM STM32 MCU (Cortex M3).

1. Define **Interrupt vectors** Section
2. Copy Data from **ROM** → **RAM**
3. Initialize **Data Area**
4. Initialize **Stack**
5. Create a **reset section** and call main( ).

We need first to determine where the flash address we will place the startup code at:



Makefile:

```
#@arm-none-eabi-#@Copyright: Ahmed Hassan

CC=arm-none-eabi-
CFLAGS=-mcpu=cortex-m3 -gdwarf-2 #included debugger for proteus
INCS=-I .
LIBS=
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)
As=$(wildcard *.s)
AsOBJ=$(As:.s=.o)
Project_name=Learn_in_depth_cortex_m3

all: $(Project_name).bin
    @echo "<<============== Build Complete ==============>>"

startup.o: startup.s
    $(CC)as.exe $(CFLAGS) $< -o $@

%.o: %.c
    $(CC)gcc.exe -c $(CFLAGS) $(INCS) $< -o $@

$(Project_name).elf: $(OBJ) $(AsOBJ)
    $(CC)ld.exe -T linker_script.ld $(LIBS) -Map=output.map $(OBJ) $(AsOBJ) -o $@

$(Project_name).bin: $(Project_name).elf
    $(CC)objcopy.exe -O binary $< $@

clean_all:
    rm *.o *.elf *.bin

clean:
    rm *.elf *.bin
```

Startup Code:

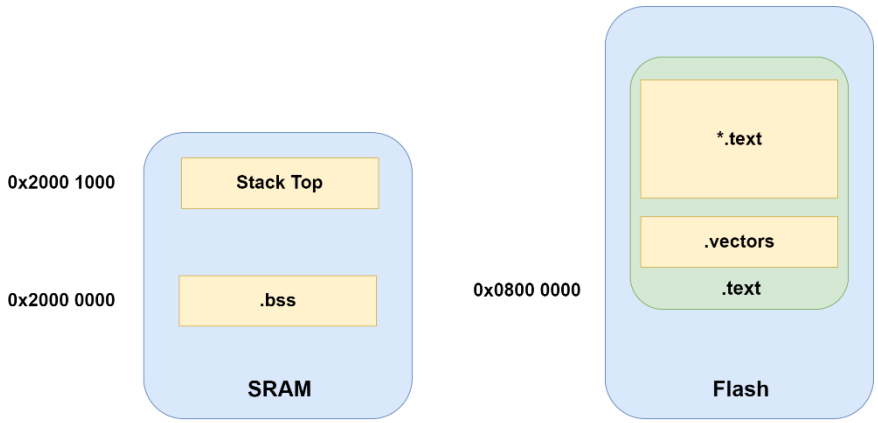
```
/* startup_cortexM3.s
Eng.Ahmed Hassan
*/

/* ================SRAM 0x2000 0000================ */
.section .vectors       /* Assembler command to set object section */

.word 0x20001000        /* For Stack */
.word _reset            /* For Vector Handler */
.word Vector_handler    /* 2 NMI */
.word Vector_handler    /* 3 Hard Fault */
.word Vector_handler    /* 4 MM Fault */
.word Vector_handler    /* 5 Bus Fault */
.word Vector_handler    /* 6 Usage Fault */
.word Vector_handler    /* 7 RESERVED */
.word Vector_handler    /* 8 RESERVED */
.word Vector_handler    /* 9 RESERVED */
.word Vector_handler    /* 10 RESERVED */
.word Vector_handler    /* 11 SV Call */
.word Vector_handler    /* 12 Debug Resurved */
.word Vector_handler    /* 13 RESERVED */
.word Vector_handler    /* 14 PendSV */
.word Vector_handler    /* 15 SysTick */
.word Vector_handler    /* 16 IRQ0 */
.word Vector_handler    /* 17 IRQ1 */
.word Vector_handler    /* 18 IRQ2 */
.word Vector_handler    /* 19 .... */
                        /* On until IRQ67 */

.section .text

_reset:
    bl main
    b .         /* Branch to yourself */

.thumb_func     /* Enable thumb mode */

Vector_handler:
    b _reset
```

Table 61. Vector table for connectivity line devices

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000_0000
-	-3	fixed	Reset	Reset	0x0000_0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
-	-1	fixed	HardFault	All class of fault	0x0000_000C
-	0	settable	MemManage	Memory management	0x0000_0010
-	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
-	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV	Pendable request for system service	0x0000_0038
-	6	settable	SysTick	System tick timer	0x0000_003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper interrupt	0x0000_0048
3	10	settable	RTC	RTC global interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000_0060

Linker Script:

We want to mimic the following memory segments:



```
/* Linker Script CortexM3
Eng. Ahmed Hassan
*/

MEMORY
{
    flash(RX) : ORIGIN = 0x08000000, LENGTH = 128k
    SRAM (RWX) : ORIGIN = 0x20000000, LENGTH = 20k
}

SECTIONS
{
    .text :
    {
        *(.vectors*)    /* Get .vectors from any object files */
        *(.text*)       /* Get .text from any object files */
        *(.rodata)      /* Get .rodata from any object files */
    }> flash

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

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

Startup using C code:

As (CortexM3) can initialize the **SP with the first 4 bytes**, so we can write **startup by C code**.

Functional Attribute: weak and alias in embedded c:

```
$ arm-none-eabi-nm.exe Learn_in_depth_cortex_m3.elf
080000c8 T Bus_Fault
080000e0 T const_variables
080000e8 D g_variables
080000b0 T H_fault_Handler
0800001c T main
080000bc T MM_Fault_Handler
080000a4 T NMI_Handler
080000e4 D ODR_A
08000098 T Reset_Handler
080000d4 T Usage_Fault_Handler
08000000 T vectors
```

Managing **many ISRs (Interrupt Service Routines)** manually becomes **tedious**. This is where GCC attributes like **__attribute__((weak))** and **__attribute__((alias("...")))** become extremely helpful.

If you write a handler for every possible exception and interrupt, even if you don't use most of them, you'll end up with lots of **empty** or **duplicate** code just to avoid linker errors.

- **__attribute__((weak)) (Weak Linking):**
  - Allows a function to be overridden by a strong (non-weak) definition elsewhere.
  - Provides a default implementation that can be replaced by the user.

- **__attribute__((alias)) (Function Aliasing)**
  - Creates an alternative name for a function.
  - Useful for backward compatibility or generic function redirection.

Now we can see that **handlers with the pragma commands** have the **same symbol** as the **Default_Handler**:

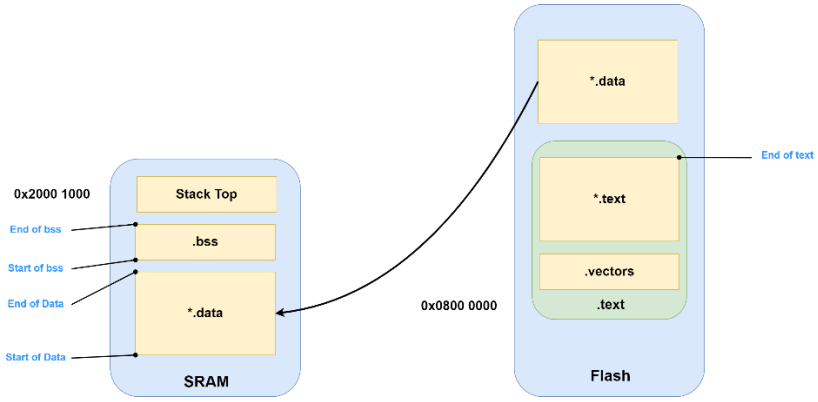
```
$ arm-none-eabi-nm.exe Learn_in_depth_cortex_m3.elf
08000098 W Bus_Fault
080000b0 T const_variables
08000098 T Default_Handler
080000b8 D g_variables
08000098 W H_fault_Handler
0800001c T main
08000098 W MM_Fault_Handler
08000098 W NMI_Handler
080000b4 D ODR_A
080000a4 T Reset_Handler
08000098 W Usage_Fault_Handler
08000000 T vectors
```

Startup.c code:

```c
//Startup.c
//Eng.Ahmed Hassan

#include <stdint.h>

extern int main (void);

void Default_Handler();
void Reset_Handler();
void NMI_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void H_fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void MM_Fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void Bus_Fault() __attribute__ ((weak, alias ("Default_Handler")));
void Usage_Fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));

//Pragma to create a section with the name ".vectors" so the linker can assign to correct address
uint32_t vectors[] __attribute__ ((section(".vectors"))) = {

(uint32_t)  0x20001000,         //The vector handler start at 0x20001000
(uint32_t)  &Reset_Handler,
(uint32_t)  &NMI_Handler,
(uint32_t)  &H_fault_Handler,
(uint32_t)  &MM_Fault_Handler,
(uint32_t)  &Bus_Fault,
(uint32_t)  &Usage_Fault_Handler

};

void Default_Handler()
{
    Reset_Handler();
}

void Reset_Handler()
{
    main();
}
```

How to copy (data and create .bss sections):

Now we want to copy the .data from flash to SRAM and create .bss in SRAM.



As a start, we can use the same address value for "**End of text**" = "**Start of Data**". We will also use **virtual memory** to calculate the **size** of data and bss sections.

Modified Linker script:

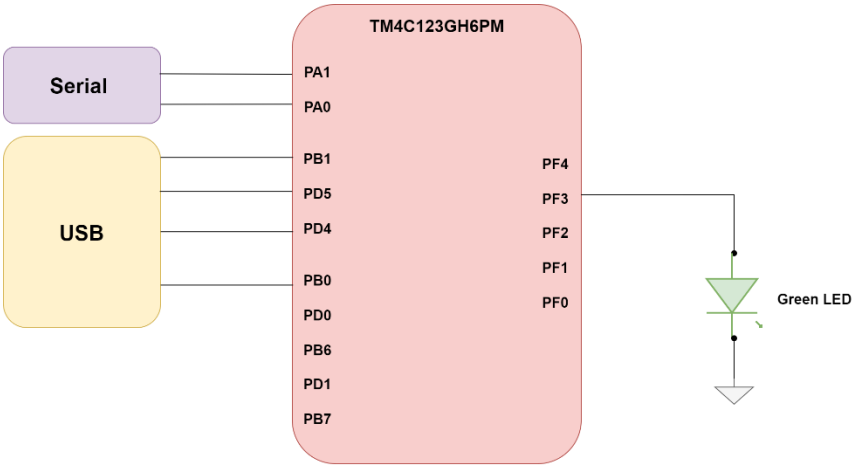
```
/* Linker Script CortexM3
Eng. Ahmed Hassan
*/

MEMORY
{
    flash(RX) : ORIGIN = 0x08000000, LENGTH = 128k
    SRAM (RWX) : ORIGIN = 0x20000000, LENGTH = 20k
}

SECTIONS
{
    .text :
    {
        *(.vectors*)        /* Get .vectors from any object files */
        *(.text*)           /* Get .text from any object files */
        *(.rodata)          /* Get .rodata from any object files */
        _E_text = . ;        /* End of text */
    }> flash                /* Both VM and LM in flash*/

    .data :
    {
        _S_DATA = . ;        /* Start of data right after .text */
        *(.data)
        . = ALIGN(4) ;       /* Enable memory allignment */
        _E_DATA = . ;        /* End of data */
    }> SRAM AT> flash       /* Virtual address at SRAM and at burning start in flash */

    .bss :
    {
        _S_bss = . ;         /* Start of bss */
        *(.bss)
        _E_bss = . ;         /* End of bss */
        . = ALIGN(4) ;
        . = . + 0x1000 ;    /* Make sure to not overwrite the stack top */
        _stack_top = . ;
    }> SRAM                 /* Both VM and LM in SRAM*/
}
```

Modified Startup.c:

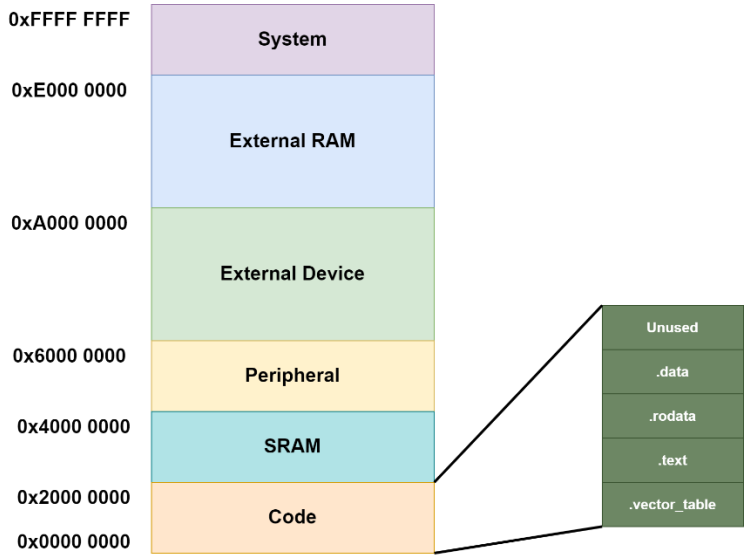
```c
//Startup.c
//Eng.Ahmed Hassan

#include <stdint.h>

extern int main (void);

void Default_Handler();
void Reset_Handler();
void NMI_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void H_fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void MM_Fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void Bus_Fault() __attribute__ ((weak, alias ("Default_Handler")));
void Usage_Fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));

extern uint32_t _stack_top;

//Pragma to create a section with the name ".vectors" so the linker can assign to correct address
uint32_t vectors[] __attribute__ ((section(".vectors"))) = {

(uint32_t)  &_stack_top,         //The vector handler start at 0x20001000
(uint32_t)  &Reset_Handler,
(uint32_t)  &NMI_Handler,
(uint32_t)  &H_fault_Handler,
(uint32_t)  &MM_Fault_Handler,
(uint32_t)  &Bus_Fault,
(uint32_t)  &Usage_Fault_Handler

};

//Data and bss sections from the linker script
extern uint32_t _E_text;
extern uint32_t _S_DATA;
extern uint32_t _E_DATA;
extern uint32_t _S_bss;
extern uint32_t _E_bss;

void Reset_Handler()
{
    //These are not variables but symbols so we act as if they are addresses
    //In case data is not alligned, we pass byte by byte while casting
    uint32_t DATA_size = (uint8_t*)&_E_DATA - (uint8_t*)&_S_DATA;
    uint8_t* P_src = (uint8_t*)&_E_text;
    uint8_t* P_dst = (uint8_t*)&_S_DATA;

    //Copy data section from flash to SRAM
    for (int i = 0; i < DATA_size; i++)
    {
        *((uint8_t*)P_dst++) = *((uint8_t*)P_src++);
    }

    //init .bss section in SRAM = 0
    uint32_t bss_size = (uint8_t*)&_E_bss - (uint8_t*)&_S_bss;
    P_dst = (uint8_t*)&_S_bss;

    for (int i = 0; i < bss_size; i++)
    {
        *((uint8_t*)P_dst++) = (uint8_t)0;
    }

    //Jump to main()
    main();
}

void Default_Handler()
{
    Reset_Handler();
}
```

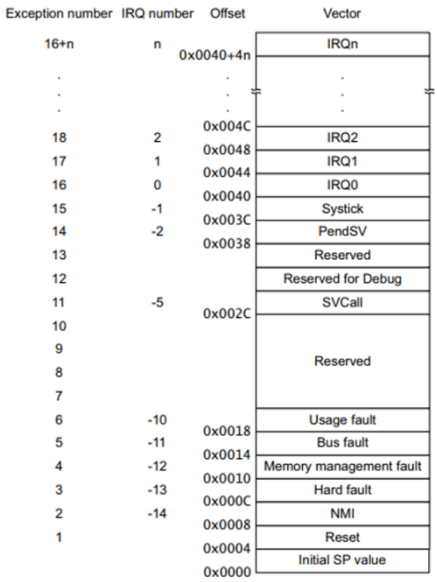
Lab 03 Bare-metal software on TM4C123 ARM CORTEXM4:

We will write a bare-metal SW to toggle **PF3** which relates to green LED.



CortexM4 Memory map:





Main.c code:

```c
//Eng.Ahmed Hassan

#define SYSCTL_RCGC2_R      (*((volatile unsigned int*)0x400FE108))
#define GPIO_PORTF_DIR_R    (*((volatile unsigned int*)0x40025400))
#define GPIO_PORTF_DEN_R    (*((volatile unsigned int*)0x4002551C))
#define GPIO_PORTF_DATA_R   (*((volatile unsigned int*)0x400253FC))

int main()
{
    SYSCTL_RCGC2_R = 0x20;          //Enable GPIO Port

    //Delay to make sure that GPIOF is up & running
    unsigned int volatile delay_count = 0;
    for (delay_count = 0; delay_count < 200; delay_count++);

    GPIO_PORTF_DIR_R |= (1 << 3);   //Set PF3 as output
    GPIO_PORTF_DEN_R |= (1 << 3);   //Enable Pin 3

    //Toggle Green LED
    while (1)
    {
        GPIO_PORTF_DATA_R ^= (1 << 3);
        for (delay_count = 0; delay_count < 50000; delay_count++);
    }

    return 0;
}
```

Makefile code:

```
#@arm-none-eabi-#@Copyright: Ahmed Hassan

CC=arm-none-eabi-
CFLAGS=-mcpu=cortex-m4 -gdwarf-2 -g #included debugger for proteus
INCS=-I .
LIBS=
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)
As=$(wildcard *.s)
AsOBJ=$(As:.s=.o)
Project_name=Unit3_Lab4_CortexM4

all: $(Project_name).bin
    @echo "<<============= Build Complete =============>>"

#startup.o: startup.s
#   $(CC)as.exe $(CFLAGS) $< -o $@

%.o: %.c
    $(CC)gcc.exe -c $(CFLAGS) $(INCS) $< -o $@

#Create .axf file extension to debug on KeiluVision
$(Project_name).elf: $(OBJ) $(AsOBJ)
    $(CC)ld.exe -T linker_script.ld $(LIBS) -Map=output.map $(OBJ) $(AsOBJ) -o $@
    cp $(Project_name).elf $(Project_name).axf

$(Project_name).bin: $(Project_name).elf
    $(CC)objcopy.exe -O binary $< $@

clean_all:
    rm *.o *.elf *.bin

clean:
    rm *.elf *.bin
```

Startup C code:

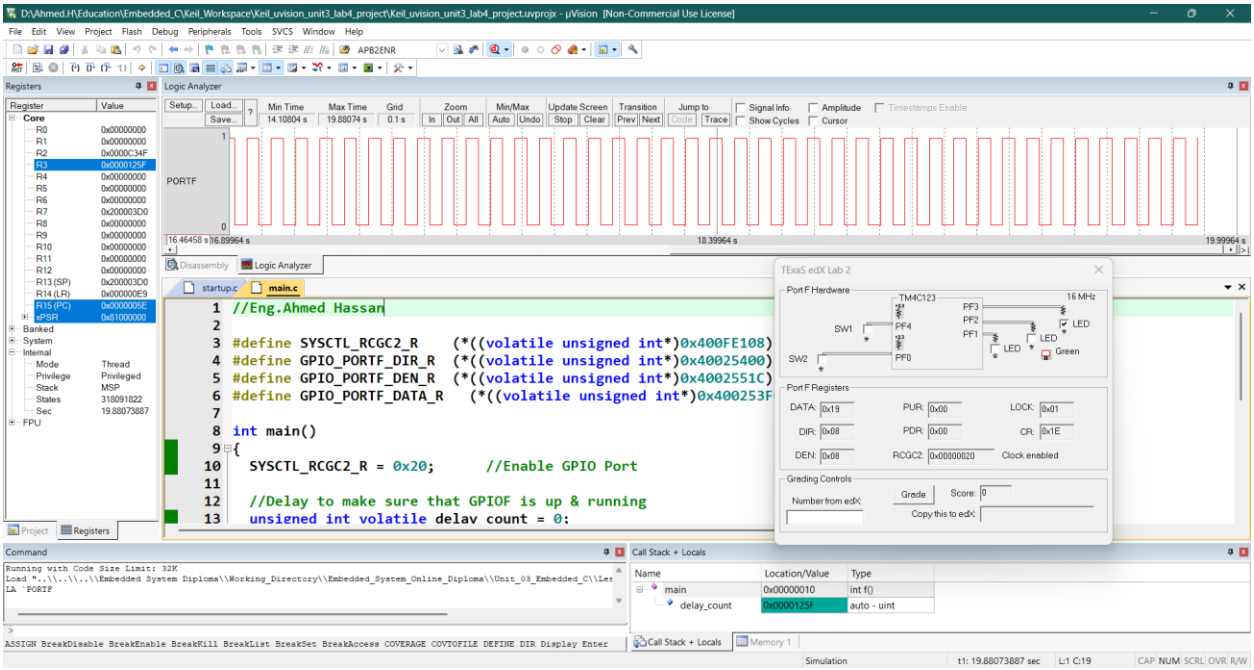
- **Interview Question: Modify the startup code so it does not use the extern symbol of the stack_top from the linker script and the stack_top is located 1024 bytes after .bss section in the SRAM.**
  - Solution: Create an **global uninitialized array** of 256 elements (1024/4) and let **SP = (array[0] + arr_size)**
  - Explanation: The **uninitialized array** will be stored in the **.bss** section by default.

```c
//Startup.c
//Eng.Ahmed Hassan

#include <stdint.h>

extern int main (void);

void Default_Handler();
void Reset_Handler();
void NMI_Handler() __attribute__ ((weak, alias ("Default_Handler")));
void H_fault_Handler() __attribute__ ((weak, alias ("Default_Handler")));

//Booking 1024 B located by .bss through unitialized array of int 256 elements (256*4=1024)
static uint32_t stack_top[256];

//Array of pointers to void functions
//Element size is 32bit beacuse of the pointer size
void (* const g_ptr_fun_Vectors[]) () __attribute__ ((section(".vectors"))) = {

(void (*)())  ((uint32_t)stack_top + sizeof(stack_top)),  //SP address assigned
//The following functions are already defined as void finctions, no need to cast
&Reset_Handler,
&NMI_Handler,
&H_fault_Handler,

};

//Data and bss sections from the linker script
extern uint32_t _E_text;
extern uint32_t _S_DATA;
extern uint32_t _E_DATA;
extern uint32_t _S_bss;
extern uint32_t _E_bss;

void Reset_Handler()
{
    //These are not variables but symbols so we act as if they are addresses
    //In case data is not alligned, we pass byte by byte while casting
    uint32_t DATA_size = (uint8_t*)&_E_DATA - (uint8_t*)&_S_DATA;
    uint8_t* P_src = (uint8_t*)&_E_text;
    uint8_t* P_dst = (uint8_t*)&_S_DATA;

    //Copy data section from flash to SRAM
    for (int i = 0; i < DATA_size; i++)
    {
        *((uint8_t*)P_dst++) = *((uint8_t*)P_src++);
    }

    //init .bss section in SRAM = 0
    uint32_t bss_size = (uint8_t*)&_E_bss - (uint8_t*)&_S_bss;
    P_dst = (uint8_t*)&_S_bss;

    for (int i = 0; i < bss_size; i++)
    {
        *((uint8_t*)P_dst++) = (uint8_t)0;
    }

    //Jump to main()
    main();
}

void Default_Handler()
{
    Reset_Handler();
}
```

Linker Script Code:

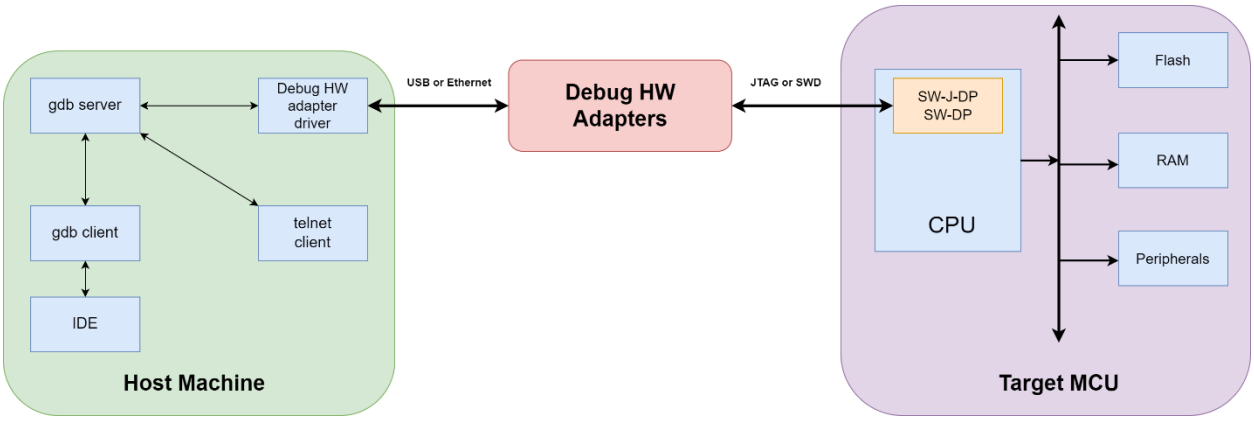
```
/* Linker Script CortexM3
Eng. Ahmed Hassan
*/

MEMORY
{
    flash(RX) : ORIGIN = 0x00000000, LENGTH = 512M
    SRAM (RWX) : ORIGIN = 0x20000000, LENGTH = 512M
}

SECTIONS
{
    .text :
    {
        *(.vectors*)        /* Get .vectors from any object files */
        *(.text*)           /* Get .text from any object files */
        *(.rodata)          /* Get .rodata from any object files */
        _E_text = . ;        /* End of text */
    }> flash                /* Both VM and LM in flash*/

    .data :
    {
        _S_DATA = . ;        /* Start of data right after .text */
        *(.data)
        . = ALIGN(4) ;       /* Enable memory allignment */
        _E_DATA = . ;        /* End of data */
    }> SRAM AT> flash       /* Virtual address at SRAM and at burning start in flash */

    .bss :
    {
        _S_bss = . ;         /* Start of bss */
        *(.bss)
        _E_bss = . ;         /* End of bss */
    }> SRAM                 /* Both VM and LM in SRAM*/
}
```

Keil uVision Debug:



Debugging Mechanism through Debug circuit

In embedded systems, **debugging mechanisms** are essential for identifying and fixing errors in software and hardware. If we want to **debug the MCU directly (not debugging through OS)**, we will need a **debugging circuit**. The debugging circuit may be **external or included in the MCU kit**.



OpenOCD Basics:

GDB client communicates with OpenOCD as a remote target and provides a familiar debugging environment for programmers.

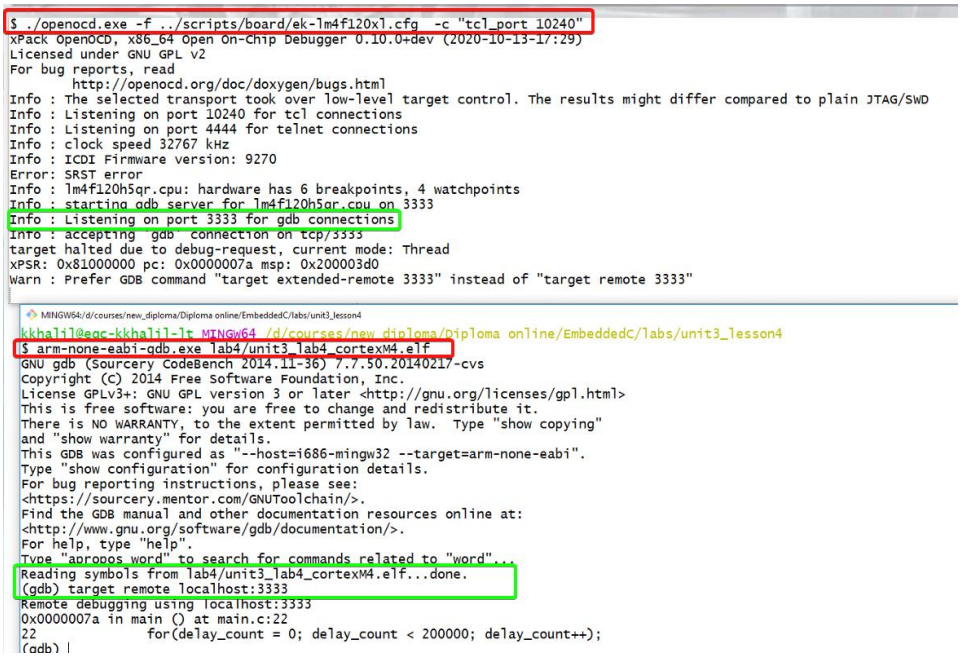
- OpenOCD requires a **configuration file to start up**.
- The configuration files specify the chips details (jtag tap) and the details of the interface we are using.
- Once started OpenOCD will start listening on 2 TCP ports
  - **4444** - OpenOCD telnet interface
  - **3333** - GDB target interface

Telnet to OpenOCD:

- telnet localhost 4444
- # Initialize the init script
  - ***init***
- # Perform a reset
  - ***reset***
- requesting target halt and executing a soft reset
- # Halt the cpu
  - ***halt***
- # Load an image (the image location is relative to the working directory of openocd)
  - ***load_image <elf image>***
- # Finish up
  - ***Exit***

GDB client:

- # Issue any OpenOCD commands by prefixing with monitor
  - ***monitor init***
  - ***monitor reset***
- # Loading an image
  - ***Load***



Memory Allocation:

The blocks of information in a memory system is called memory allocation.

- **STATIC MEMORY ALLOCATION:** In static memory allocation, size of the memory may be required for that **must be defined before loading and executing the program**.

- **DYNAMIC MEMORY ALLOCATION:** In dynamic memory allocation, the **memory is allocated to a variable or program at the run time**.
  The only way to access this dynamically allocated memory is through **pointer**.

Static Allocation	Dynamic Allocation
Performed at static or **compile time**	Performed at dynamic or **run time**
Assigned to **stack**	Assigned to **heap**
Size must be **known** at compile time	Size may be **unknown** at compile time
First in last out	**No order** of assignment
It is best if required **size of memory known in advance**.	It is best if we **don't have any idea about how much memory is required**.

Memory allocation is **not included in standard C**. To use the API, you need to include "**stdlib.h**" header file.

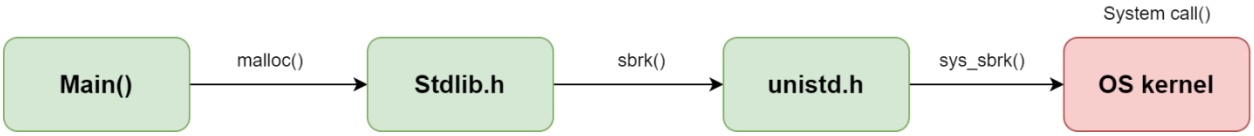
Function	Use of function
malloc(size_t size)	Allocates a block of memory of the given size (in bytes). The content is **uninitialized**. **Returns a pointer to the beginning of the block** or NULL if allocation fails.
calloc(size_t num, size_t size)	Allocates memory for an array of num elements, each of size bytes. **Initializes all bits to zero**. Returns a pointer to the memory or NULL if allocation fails.
realloc(void *ptr, size_t new_size)	**Resizes the memory block** pointed to by ptr to new_size bytes. Contents are preserved up to the lesser of the new and old sizes. Returns a new pointer or NULL if resizing fails.
free(void *ptr)	**Frees the memory previously allocated** by malloc, calloc, or realloc. Passing NULL has no effect.

Function to dynamically fill an array:

```c
int main()
{
    int n = 0, sum = 0, i = 0;
    int *ptr;

    while (1)
    {
        printf("Enter number of elements: ");
        scanf("%d", &n);

        ptr = (int *)malloc(n * sizeof(int));
        if (ptr == NULL)
        {
            printf("Error while locating memory!\n");
            exit(0);
        }

        printf("Fill the elements: \n");
        for(i = 0; i < n; i++)
        {
            scanf("%d", ptr+i);
            sum += *(ptr+i);
        }

        printf("Sum = %d\n", sum);
        free(ptr);
    }
    return 0;
}
```

- **Note** when using the dynamic location in embedded C, you will face a **linker error** in which it mentions "**undefined reference to the dynamic allocation functions**". This is caused because the **linker cannot link the "stdlib" object file**. To overcome this error, you will need to **include the stdlib object file path to the linker**:
  arm-none-eabi-ld.exe -T linker_script.ld -L<"stdlib path">.stdlib.o main.o startup.o -o learn-in-depth.elf -Map=output.map.

- Even if you run that command you will get the following error: "**undefined reference to _sbrk**". Why does that happen?



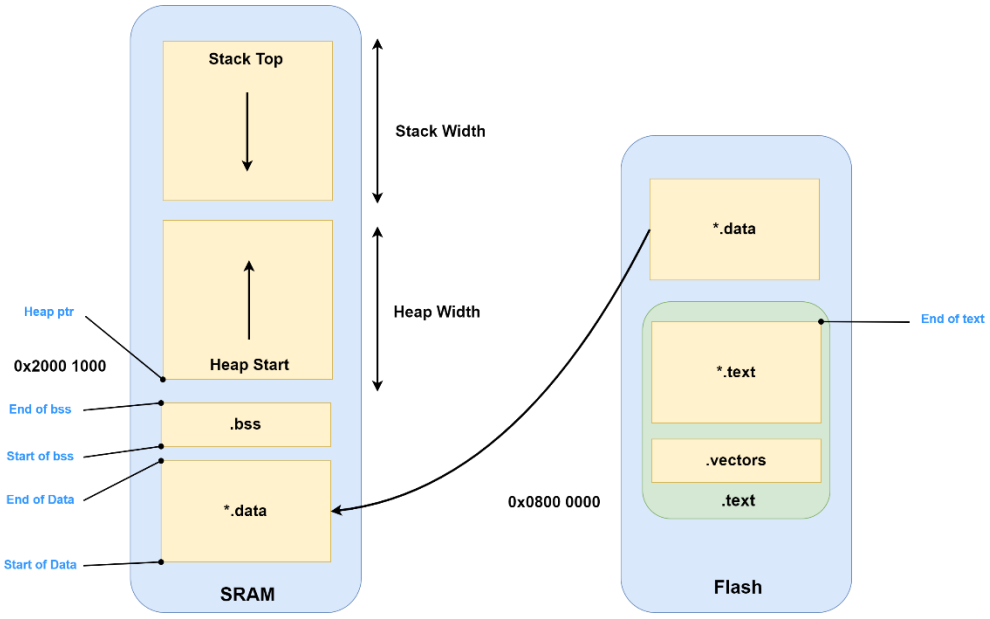
  - The C language provides **no built-in facilities** for performing such common operations as input/output, memory management, string manipulation, and the like (such as fopen, malloc, printf, exit etc).

  - The Implementations of the specific standard library are **platform dependent**.
    - **Linux:** GNU C Library (glibc)
    - **Windows:** Microsoft C run-time library (msvcrt)

  - In **MMU-less** embedded systems (embedded system not having OS), there are several implementations of the standard library, such as **Newlib**, **µClibc** etc.

  - By default, the arm-none-eabi-gcc is configured to link against **Newlib**.

Implement _sbrk to support malloc in embedded C:



Linker Script after adding heap section:

```
.bss :
{
    _S_bss = . ;          /* Start of bss */
    *(.bss)
    _E_bss = . ;          /* End of bss */
    . = ALIGN(4) ;
    . = . + 0x1000 ;     /* Make sure to not overwrite the Heap start */
    _heap_End = . ;
    . = ALIGN(4) ;
    . = . + 0x1000 ;     /* Make sure to not overwrite the stack top */
    _stack_top = . ;
}> SRAM
```

Spark function implementation in main.c:

```
void* _sbrk(int incr)
{
    static uint8_t* heap_ptr = NULL;
    uint8_t* prev_heap_ptr = NULL;
    extern uint32_t _E_bss;          //End of bss section symbol
    extern uint32_t _heap_End;       //End of heap symbol

    //First time intialized
    if (heap_ptr == NULL)
        heap_ptr = (uint8_t*)&_E_bss;

    prev_heap_ptr = heap_ptr;

    //Protect the stack
    if ((heap_ptr + incr) > &_heap_End)
        return (void*) NULL;

    //Booking the incremented size
    heap_ptr += incr;

    return (void*) prev_heap_ptr;
}
```

Now we run the following script to generate the elf file:
arm-none-eabi-gcc -mcpu=cortex-m3 -nostartfiles -T linker_script.ld startup.c main.c -o unit3_lesson5.elf

❖ **Note** if the library still includes the other **unimplemented functions for the printf()** for example, add the following simple implementations in the main code:

```c
#include <errno.h>
int _close(int file) {
    return -1;
}

int _lseek(int file, int ptr, int dir) {
    return 0;
}

int _read(int file, char *ptr, int len) {
    return 0;
}

int _write(int file, char *ptr, int len) {
    return len;
}

void _exit(int status) {
    while(1);
}

int _kill(int pid, int sig) {
    errno = EINVAL;
    return -1;
}

int _getpid(void) {
    return 1;
}
```

Because of how crirtical the dynamic memory allocation is and because the hassle it causes to implement in the embedded C code, it is avoided in embedded C as mentioned in MISRA C guidelines.