Table of Contents

1 Introduction:

Writing a bare-metal application from scratch is not that hard you think, in this report I will build a simple bare-metal application that sends a string message using UART protocol in VersatilePB micro-controller chip which is based on arm926ej-s micro-processor, I will build everything from scratch including startup, linker script and source codes, and compile them using arm-none-eabi cross tool chain for arm processors.

I will execute this simple bare metal application on a virtual board by using “qemu” tool which simulate many micro-controller chips.

2 Source code

2.1 main application code

```c
#include "uart.h"
char  uart_message[]="Learn-in-depth: Mohamed Mostafa";
int main()
{
    send_uart_message(uart_message);
    return 0;
}
```

fig 2.1 app source code

In the main application code, which I called “app.c” I defined an array of characters (string) and initialized it by the message which will be sent via UART, then pass the message to a function called “send_uart_data” which is defined in included header file “uart.h”.

2.2 uart code

```c
#define UART0DR *(volatile unsigned int*)((unsigned int*)0x101f1000)
void send_uart_message(char* message)
{

    while (*message != '\0')
    {
        UART0DR = (unsigned int) *message;
        message++;
    }
}
```

fig 2.2 uart source code

I will send my message via UART port 0 which is mapped in address 0x101f1000. From specs there is the register UART0DR which is used to transmit and receiving data to UART byte by byte at offset 0x0, so I need to read or write at the beginning of the memory allocated for the UART0. In uart.c there is a macro will be used to write data at the memory address of UART0DR to be transmitted to UART0, in the function “send_uart_message” there is a while loop which checks the end of message and while it is not the end of message then pass byte by byte to the register UART0DR to be transmitted to the UART0.

2.3 uart header file

uart.h

```c
#ifndef _UART_H_
#define _UART_H_

extern void send_uart_message(char* message);

#endif
```

fig 2.3 uart header file

It is a very simple header file which have first two lines and last line for header file protection to avoid multi including of the same file which will cause a compilation error of re-declaration function, then there is the prototype of the function “send_uart_message” which takes an argument of type pointer to char. The “extern” storage class because its definition is in another file and not in “app.c” file which is the main file of code.

2.4 startup code

startup.s

```asm
.global reset

reset :
	ldr sp, =stack_top
	bl main
	b stop
stop :
	b stop
```

fig 2.4 startup code in assembly

Startup code is written in assembly language because it is dependent on micro-processor architecture, in this simple startup code we just make a reset section which will be first section to execute before main as it will be burned at the processor entry point, in this section I just initialized the stack pointer with “stack_top” which is a symbol will be resolved while linking process from linker script, then just brunch to main function and after that loop in your position and don’t do anything.

2.5 linker script

linker_script.ld

```
ENTRY(reset)
MEMORY
{
    mem (rwx) : ORIGIN = 0x0, LENGTH = 64M
}

SECTIONS
{
    . = 0x10000;
    .reset . :
    {
        startup.o(.text)
    }>mem
    .texet :
    {
        *(.text)
    }>mem
    .data :
    {
        *(.data)
    }>mem
    . = . + 0x1000;
    stack_top = .;
}
```

fig 2.5 linker script

In this simple linker script, we define the entry point of my whole application which is the reset section in the startup code. Then we define memory boundaries and its attributes, here I have one memory I called it mem which have attributes (read, write and execute). The last section in linker script is that divide my whole code in all files to organized sections to be burned to the micro-controller, here I used the location counter and set it to the entry point of the processor (from specs) 0x10000 and I created a section will be loaded in this address contains the .text(instructions) of the startup code, after this section the remained .text of whole other file will be combined to .text section and will be loaded to the memory after .reset section, the last section is ,.data section which combines whole global data of whole files and will be loaded after .text section.

Now the location counter is having the address of the end of .data section so I will add 0x1000 which equals 4MB (my stack size) and make a symbol to be used in startup to initialize stack pointer register.

3 Symbols

Using nm binary utility, I can hack every binary file and see its symbols and which section every symbol belongs to and address of every symbol. In object files there is only virtual addresses, and every symbol will take a real load address after linking process in the elf image.

3.1 app.o symbols

this object file contains three symbols,

1. Main: which is in text section.
2. Send_uart_message: which is un-resolved and will be resolved when link this file to uart.o file.
3. Uart_message: which is in data section.

```
$ arm-none-eabi-nm.exe app.o
00000000 T main
         U send_uart_message
00000000 D uart_message
```

fig 3.1 app.o symbols

3.2 uart.o symbols

this object file contains only one symbol,

1. Send_uart_message: which is in text section.

```
$ arm-none-eabi-nm.exe uart.o
00000000 T send_uart_message
```

fig 3.2 uart.o symbols 1

3.3 resolving app.o symbols

```
moham@DESKTOP-3792IKQ MINGW64 /d/Mohamed/embedded/diploma/workspace/lab1_embedde
dC_L2/2nd time
$ arm-none-eabi-ld.exe app.o uart.o -o learn-in-depth.elf
D:\programs_instalation\7 2017-q4-major\bin\arm-none-eabi-ld.exe: warning: canno
t find entry symbol _start; defaulting to 00008000

moham@DESKTOP-3792IKQ MINGW64 /d/Mohamed/embedded/diploma/workspace/lab1_embedde
dC_L2/2nd time
$ arm-none-eabi-nm.exe learn-in-depth.elf
00018094 D __bss_end__
00018094 D __bss_start
00018094 D __bss_start__
00018074 D __data_start
00018094 D __end__
00018094 D _bss_end__
00018094 D _edata
00018094 D _end
00080000 D _stack
         U _start
00008000 T main
00008020 T send_uart_message
00018074 D uart_message
```

fig 3.3 resolved app.o symbols

The unresolved symbol in app.o is now resolved when linking app.o with uart.o files, the first symbols is generated while linking the to files together but we only care about symbols in app.o and uart.o files.

3.4 startup.o symbols

This object file contains four symbols,

1. Main: which is unresolved in this file and will be resolved when link this file with main.o file.
2. Reset: which is in text section.
3. Stack_top: which is unresolved in this file and will be resolved while linking process from linker script file.
4. Stop: which is in text section.

```
$ arm-none-eabi-nm.exe startup.o
         U main
00000000 T reset
         U stack_top
0000000c t stop
```

fig 3.4 startup.o symbols

3.5 elf image symbols

After final linking process, all symbols should be resolved, and take the load memory addresses.

```
$ arm-none-eabi-nm.exe learn-in-depth.elf
00010014 T main
00010000 T reset
00010034 T send_uart_message
000110a8 D stack_top
0001000c t stop
00010088 D uart_message
```

fig 3.5 elf image symbols

4 Sections headers

In this section we just care about .text, .data, .bss and .rodata sections, linker may add some other sections like .ARM.attributes and .comment but we don’t care about these sections as it will be excluded in the final executable file and wont be loaded the micro-controller.

4.1 app.o sections header

```
app.txt

app.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000020  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000020  00000000  00000000  00000054  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000074  2**0
                  ALLOC
  3 .comment      0000007f  00000000  00000000  00000074  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000032  00000000  00000000  000000f3  2**0
                  CONTENTS, READONLY
```

fig 4.1 app.o sections

Sections:

1. .text section: have a size of 0x20 (size of instructions of code).
2. .data section: have a size of 0x20 (size of the initialized global array).
3. .bss section: have a size of 0x0 (as there is no uninitialized global data).

All sections have equal LMA and VMA = 0x0 as this is an object file (relocatable image).

4.2 uart.o sections

```
uart.txt

uart.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000054  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000088  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000088  2**0
                  ALLOC
  3 .comment      0000007f  00000000  00000000  00000088  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000032  00000000  00000000  00000107  2**0
                  CONTENTS, READONLY
```

fig 4.2 uart.o sections

There is only .text section as there is no global data.
All sections have equal LMA and VMA = 0x0 as this is an object file (relocatable image).

4.3 startup.o sections

```
startup.txt

startup.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000014  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000048  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000048  2**0
                  ALLOC
  3 .ARM.attributes 00000022  00000000  00000000  00000048  2**0
                  CONTENTS, READONLY
```

fig 4.3 startup.o sections

There is only .text section as there is no global data.

All sections have equal LMA and VMA = 0x0 as this is an object file (relocatable image).

4.4 final elf image sections

```
learn-in-depth.txt

learn-in-depth.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .reset        00000014  00010000  00010000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .texet        00000074  00010014  00010014  00010014  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data         00000020  00010088  00010088  00010088  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .ARM.attributes 0000002e  00000000  00000000  000100a8  2**0
                  CONTENTS, READONLY
  4 .comment      0000007e  00000000  00000000  000100d6  2**0
                  CONTENTS, READONLY
```

fig 4.4 final elf image sections

Sections:

1. .reset section: have a size of 0x14 (size of instructions of startup code).
2. .text section: have a size of 0x74 (size of instructions of whole files code except start up code).
3. .data section: have a size of 0x20 (size of the initialized global array in app.o file).

All sections now have the real load memory addresses which is given to every section in linker script.

5 executing application using qemu tool

```
$  qemu-system-arm -M versatilepb -m 128M -nographic -kernel learn-in-depth.bin
Learn-in-depth: Mohamed Mostafa
```

fig 5.1 executing application

Here we consider the terminal as the UART port 0 output and can see the message was sent in the code