

Bare metal embedded application for arm versatilepb board

Scope:

This document is considered with building bare metal embedded application from the ground up. The implemented application is a simple one which writes an arbitrary string on Uart and prints it on the terminal.

Tools and components:

Tools:

Editor: NotePad++ to write the application files

Toolchain: arm-non-eabi

Shell: Powershell to execute the gcc commands for windows

Simulation: Qemu

Target: Arm Versatilepb board

The application components are:

- 1- Startup.s: Arm assembly startup code for the application to start from and to assign a value for the stack pointer
- 2- linkerScript_Arm926ej_s.ld: the linker script to help in resolving and relocating symbols
- 3- Uart.c: the source code for Uart driver. Because of the used target has a very simple steps to make the uart driver ready for use, "Uart_Send" implemented in Uart.c does its job in just one line of code
- 4- Uart.h: the header file for Uart.c

- 5- main.c: contains a definition for char array to send and the main function which calls Uart_Send function

Building process:

Acronym: 'O' at the end of step means the step is optional for learning purposes

Note: screenshots in the coming steps are high resolution images. You can zoom in for a better readability

- Build each .c and .s file to produce .o relocatable objects. -c option is used to direct the gcc compiler to compile only (with no linking) and -I option is used to give the information of the header directory whereas -mcpu option specifies the name of the target processor

```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-gcc.exe -mcpu=arm926ej-s -c main.c -I ..\Header -o main.o
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-gcc.exe -mcpu=arm926ej-s -c Uart.c -I ..\Header -o Uart.o
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>
```

- Let's now investigate the content of these object files
Use the command in the picture below and let's take main.c as an example:

```

PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-objdump.exe -s main.o
main.o:      file format elf32-littlearm

Contents of section .text:
0000 00482de9 04b08de2 0c009fe5 feffffeb  .H-.....
0010 0030a0e3 0300a0e1 0088bde8 00000000  .0-.....
Contents of section .data:
0000 4c656172 6e2d496e 2d446570 74683a20  Learn-In-Depth:
0010 4d6f6861 6d656420 486f736e 7900      Mohamed Hosny.
Contents of section .comment:
0000 00474343 3a202847 4e552054 6f6f6c73  .GCC: (GNU Tools
0010 20666f72 2041726d 20456d62 65646465  for Arm Embedde
0020 64205072 6f636573 736f7273 20372d32  d Processors 7-2
0030 3031372d 71342d6d 616a6f72 2920372e 017-q4-major) 7.
0040 322e3120 32303137 30393034 20287265  2.1 20170904 (re
0050 6c656173 6529205b 41524d2f 656d6265  lease) [ARM/embe
0060 64646564 2d372d62 72616e63 68207265  dded-7-branch re
0070 76697369 6f6e2032 35353230 345d00    vision 255204].
Contents of section .ARM.attributes:
0000 41310000 00616561 62690001 27000000  A1...aeabi...'...
0010 0541524d 39323645 4a2d5300 06050801  .ARM926EJ-S.....
0020 09011204 14011501 17031801 19011a01  .....
0030 1e06      ..
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>

```

Here you can see the content of each section of the generated object file where the first column represents the line number and the other words (32 bit hex) represents arm instructions in machine code.

You can also find that the auto generated comment section contains additional information about gcc toolchain and the Arm.attributes section contains the information about the running processor but the most interesting part here is the .data section as you can see the string literal in the main.c was located in this section. Pay attention that the hex coded numbers here represents the ASCII code of the string literal. For example at the beginning of .data section '4c' byte represents the letter 'L' and the same apply for the other bytes but is there a better way for human to read the instructions in .text section? Here where the objdump utility comes in. if you used the command below, you will get the assembly version of main.o

Disassembly of section `.text`:

```
00000000 <main>:
 0: e92d4800 push    {fp, lr}
 4: e28db004 add fp, sp, #4
 8: e59f000c ldr r0, [pc, #12] ; 1c <main+0x1c>
 c: ebfffffe bl 0 <Uart_Send>
10: e3a03000 mov r3, #0
14: e1a00003 mov r0, r3
18: e8bd8800 pop {fp, pc}
1c: 00000000 andeq   r0, r0, r0
```

Disassembly of section `.data`:

```
00000000 <Str>:
 0: 7261654c | rsbvc   r6, r1, #76, 10 ; 0x13000000
 4: 6e492d6e cdpvs   13, 4, cr2, cr9, cr14, {3}
 8: 7065442d rsbvc   r4, r5, sp, lsr #8
 c: 203a6874 eorscs  r6, sl, r4, ror r8
10: 61686f4d cmnvs   r8, sp, asr #30
14: 2064656d rsbcs   r6, r4, sp, ror #10
18: 6e736f48 cdpvs   15, 7, cr6, cr3, cr8, {2}
1c: Address 0x0000001c is out of bounds.
```

Here you can see a more readable version of `main.o`. for example, the `.text` section have the assembly instructions `push`, `add`, .. etc.

Note that the hex code in the two files (`.o` and `.s`) are the same except that it's written in little endian in the disassembly version.

Note also that the first column represents the offset of each instruction and because of the Arm instructions are 4 byte wide, each offset is incremented by 4.

But why the offset `1c` has a full range of zeros? You surly aware that we didn't link object files together to get the final output file and the `main.o` file is just a locatable object file. This means that addresses assigned to each symbol in this type of files are not absolute (i.e are not in final version) and the linker task is to

resolve the symbols (i.e associate each symbol reference with exactly one symbol definition) and locate absolute memory addresses throughout the locator and linker script. The relocation process is a two steps process. In the first stage addresses are assigned for each section and symbol definition. In the second stage, references to each symbol are updated with the absolute addresses. Relocation tables convey a sense of the symbols references to be located. By using the command below we can watch out the relocation table

```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-objdump.exe -r .\main.o
.\main.o:      file format elf32-littlearm

RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE          VALUE
0000000c  R_ARM_CALL      Uart_Send
0000001c  R_ARM_ABS32      Str

PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>
```

You can find that the offset "1c" (filled with zeros in the object file) represents the "Str" symbol (the array of characters) and has a type of "R_ARM_ABS32" which indicates "statically located data symbol". This explains why this symbol passed to the Uart_Send function as an argument has an address of zeros because it's a reference to "Str" symbol that will be then allocated by the linker in the coming stage. Also since "Uart_Send" symbol in the main function has not an immediate definition, it will be relocated by the linker in the next stage.

Let's take a look to another important section in the relocated object file: symbol table

The compiler exports to the assembler the scope and type for each symbol. Based on these information, the symbol table which is the output of the assembler associated with an important information about each symbol that helps the linker in resolving symbols. During linking process, the static linker

includes only the object files referenced in source code files. It iterates on the relocatable object files and/or possibly objects archived in libraries and look for unresolved symbols (throughout the symbol tables) and their definition in the next object files (in the same order of the input files to the linker). At the end of this process, there will be no more unresolved symbols and the number of unresolved and defined symbols will not further changed. However if unresolved symbols counter still has a positive value, unresolved symbol error will be raised. To show the symbol table, type in the following command

```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-objdump.exe -t .\main.o
.\main.o:      file format elf32-littlearm

SYMBOL TABLE:
00000000 l  df *ABS*  00000000 main.c
00000000 l  d  .text  00000000 .text
00000000 l  d  .data  00000000 .data
00000000 l  d  .bss   00000000 .bss
00000000 l  d  .comment 00000000 .comment
00000000 l  d  .ARM.attributes 00000000 .ARM.attributes
00000000 g  O  .data  0000001e Str
00000000 g  F  .text  00000020 main
00000000      *UND*  00000000 Uart_Send

PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>
```

'l' and 'g' in the second column represents local and global symbols respectively from the linker prospective. Global means the symbol defined in the current module and can be referenced in another one where local means the symbol scope restricted on the current object. Local symbols defined with static keyword. In the picture above you can find also the type of each symbols, one worthwhile symbol is the last one denoted by "UND" means that the symbol is undefined in in this module and need to be resolved in the linking stage and this is true for the Uart_Send symbol which it's definition is located in another object file (Uart.o)

- Till now , there is still one important file to write and compile: startup.c
The startup file in general is responsible for

- 1- Initialize the stack
- 2- Initialize the stack pointer register to point the top of stack
- 3- Initialize heap memory (if needed)
- 4- Copy .data section from ROM to RAM
- 5- Create and initialize .bss section with zeros in RAM
- 6- Define the vector section for (exception handlers and interrupts)
- 7- Configure and turn on any external memory (if absolutely required)
- 8- Call main()

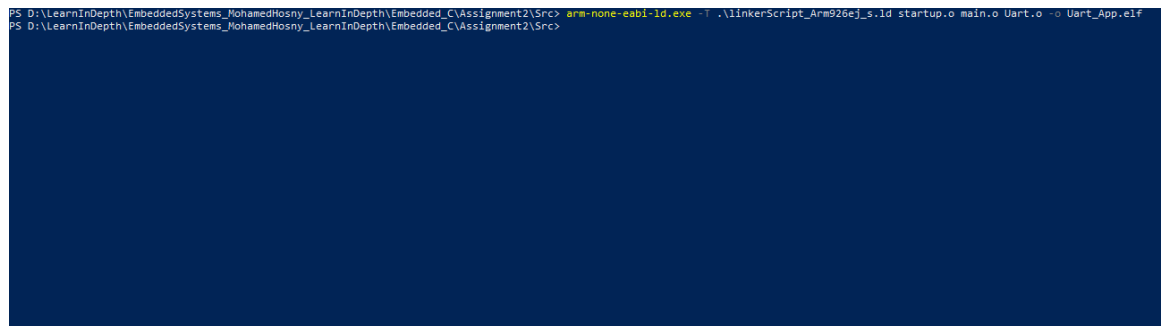
In this simple project we just create a reset section to start the execution from and initialize the stack pointer with the stack_top then branch to the main

You should compile startup.s assembly file either with the gcc command with `-c` option or directly using the gnu assembler "as".

- The last needed step in the compilation process is to invoke the linker to link all the relocatable object files and create one executable file that is ready to run on the target.

You will have to write a linker script in this step. The linker script is just a directive for the linker to locate sections and symbols to absolute addresses. In some linkers like the one in gnu toolchain, the linker and locator work as one unit, in other toolchains the linker and locator may be found as separated two units. However the main task for the linker is to resolve symbols statically or dynamically and for the locator to give absolute addresses for symbols. There are two main commands in gnu linker scripts, the MEMORY and SECTIONS commands. MEMORY command help in defining the memory layout so you can define each used memory, it's origin start address and it's size while the other command helps in aggregate different input sections (from relocatable object files) into output sections. These output sections will get absolute addresses and

will be loaded in the final executable image. Another important thing in SECTIONS command is that you can define LMA and VMA addresses for each section. The LMA is the load memory address when you burn or load the executable image to the memory while the VMA is a virtual memory address which represents the destination address for the section when copied to the RAM. It also assist in defining .data and .bss boundaries to be later used in startup code to copy .data and create .bss section in RAM (will be discussed in more details later in Bare metal application Part2). To link the object files, use the below command in the picture:

A screenshot of a terminal window with a dark blue background. The command prompt shows the path 'D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>' followed by the command 'arm-none-eabi-ld.exe -T linkerScript_Arm926ej_s.ld startup.o main.o Uart.o -o Uart_App.elf'. The command is split across two lines in the image.

```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-ld.exe -T linkerScript_Arm926ej_s.ld startup.o main.o Uart.o -o Uart_App.elf
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>
```

Note that in many cases the order of input object files to the linker is significant and start from left to right. The preferred usage is to write object files with unresolved symbols first then their definitions. For example if startup.o calls a function in main.o which calls another function in Uart.o then they should appear in the command line like the follow: arm-none-eabi-ld.exe linkerScript.ld startup.o main.o Uart.o

- Now to make sure that everything works as expected, it's important to generate the map file for the final executable object file. To do so:
-


```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-ld.exe -T .\linkerScript_Arm926ej_s.ld startup.o .\main.o .\Uart.o -o Uart_App.elf -Map=Uart_App.map
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src>
```

The map file output the actual memory layout, sections boundaries, sections and symbols absolute addresses.

You can also note unaligned sections (starting form odd address number) and start linking again after adding Align() command in the linker script to the unaligned section for better code performance

- In order to reduce the memory space of the executable file, you may extract binary only to burn it to your target board leaving any debug or common sections. Use the objcopy binary utility as shown below to do that:

```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> arm-none-eabi-objcopy.exe -O binary .\Uart_App.elf Uart_App.bin
```

- To simulate the target board, we use Qemu tool. After installing it, you can head to the binary location of the simulator tool and run the below command passing to it the processor name, the memory used and nographic option if you don't want to run graphics on this board and path to the kernel image (our simple executable binary in our case)

```
PS D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src> cd C:\Program Files (x86)\qemu
PS C:\Program Files (x86)\qemu> .\qemu-system-arm.exe -M versatilepb -m 128M -nographic -kernel D:\LearnInDepth\EmbeddedSystems_MohamedHosny_LearnInDepth\Embedded_C\Assignment2\Src\Uart_App.elf
Learn-In-Depth: Mohamed Hosny
```

Because the uart driver channel on the target board is directed to the terminal, the expected passed string to Uart_Send function is printed now on the screen