Microprocessor and Assembly Language Lab 05

Submitted by:
Kazi Shadman Sakib, FH-97
3rd Year, CSE, University of Dhaka

Date: June 18, 2022

Description of different GCC Compilation and Linker flags:

GCC Compilation Flags:

1) **-c :** Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

   By default, the object file name for a source file is made by replacing the suffix ‘.c’, ‘.i’, ‘.s’, etc., with ‘.o’.

   Unrecognized input files, not requiring compilation or assembly, are ignored.

2) **-mcpu :** Specify the name of the target processor, optionally suffixed by one or more feature modifiers. In simple words, it chooses the ARM processor version. In this assignment Cortex-M4 is used.

   Format : -mcpu=name

   GCC uses the name to determine what kind of instructions it can emit when generating assembly code and to determine the target processor for which to tune for performance.

3) **-mthumb :** Generates code that executes only in Thumb state. This supports only the thumb instruction state.

4) **-Wall :** This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.

Note that some warning flags are not implied by '-Wall'. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning.

5) **-std=gnu11 :** "-std=" Determines the language standard. The compiler can accept several base standards, such as 'c90' or 'c++98', and GNU dialects of those standards, such as 'gnu90' or 'gnu++98'. When a base standard is specified, the compiler accepts all programs following that standard plus those using GNU extensions that do not contradict it.

   "gnu11" GNU dialect of ISO C11.

6) **-O0 :** Reduce compilation time and make debugging produce the expected results. This is the default. Simply, no optimization.

Compiler Code that was executed in the Makefile of the assignment :

arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o main.o main.c

arm-none-eabi-gcc -c -mcpu=cortex-m4 -mthumb -std=gnu11 -Wall -O0 -o stm32_startup.o stm32_startup.c

Linker Flags:

1) **-nostdlib :** Does not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify are passed to the linker, and options specifying linkage of the system libraries, such as '-static-libgcc' or '-shared-libgcc', are ignored.

2) **-T :** Use "-T script" as the linker script. This option is supported by most systems using the GNU linker. On some targets, such as bare-board targets without an operating system, the '-T' option may be required when linking to avoid references to undefined symbols.

3) **stm32_ls.ld :** This is the name of the linker script file, where the linker script is written.

4) **-Wl :** Passes option as an option to the linker. If an option contains commas, it is split into multiple options at the commas. You can use this syntax to pass an argument to the option. For example, '-Wl,-Map,final.map' passes '-Map final.map' to the linker. When using the GNU linker, you can also get the same effect with '-Wl,-Map=final.map'.

5) **-Map :** "-Map=final.map", this command maps all the memory addresses used in .text, .rodata, .data, .bss sections. In the "final.map" file memory configuration is shown.

Linker Code that was executed in the Makefile of the assignment :

arm-none-eabi-gcc -nostdlib -T stm32_ls.ld -Wl,-Map=final.map -o final.elf main.o stm32_startup.o

The syntax and semantic of the linker (.ld) file:

Linker script commands:

1) **ENTRY :** This command is used to set the "Entry point address" information in the header of the final elf file generated. In our case, "Reset_Handler" is the entry point into the application. This is the first piece of code that executes right after the processor resets. The debugger uses this information to locate the first function to execute.

   Syntax :

   ENTRY(_symbol_name_)
   ENTRY(Reset_Handler)

2) **MEMORY :** This command allows us to describe the different memories present in the target and their start address and size information. The linker uses information mentioned in this command to assign addresses to merged sections. The information given under this command also helps the linker to calculate total code and data memory consumed so far and throw an error message if data, code, heap or stack areas can not fit into available size. By using memory command, we can fine-tune various memories available in our target and allow different sections to occupy different memory. Typically one linker script has one memory command.

Syntax :

```
MEMORY
{
        name(attr):ORIGIN =origin,LENGTH =length
}
```

```
MEMORY
{
        FLASH(rx):ORIGIN =0x08000000,LENGTH =512K
        SRAM(rwx):ORIGIN =0x20000000,LENGTH =128K
}
```

3) **SECTIONS :** Sections command is used to create different output sections in the final elf executable generated. This is an important command by which we can instruct the linker how to merge the input sections to yield an output section. This command also controls the order in which different output sections appear in the elf file generated. By using this command, we also mention the placement of a memory region. For example, we can instruct the linker to place the .text section in the flash memory, which is described by the MEMORY command.

.text : It holds the code/instruction part of the program.
.rodata : It holds read only data part of the program
.data : It holds the data part of the program.
.bss : It holds all the uninitialized data part of the program.

Syntax :

```
SECTIONS
{

        .text :
        {
                *(.isr_vector)
                *(.text)
                *(.rodata)
                . = ALIGN(4);
                _etext = .;
        }> FLASH
```

```
        .data :
        {
                _sdata = .;
                *(.data)
                . = ALIGN(4);
                _edata = .;
        }> SRAM AT> FLASH

        .bss :
        {
                _sbss = .;
                *(.bss)
                . = ALIGN(4);
                _ebss = .;
        }> SRAM
}
```

4) **ALIGN :** This command is used for word/double alignment. This command aligns the location counter to the next word/double word boundary.

   Syntax :

   . = ALIGN(4); //takes previous location counter and does word alignment, saves back the new location counter

5) **Location counter :** This is a special linker symbol denoted by a dot “.” . This symbol is called “location counter” since linker automatically updates this symbol with location address information. We can use this symbol inside the linker script to track and define boundaries of various sections. We can also set the location counter to any specific value while writing a linker script. Location counter appears only inside the SECTIONS command. The location counter is then incremented by the size of the output section.

   Syntax :
   _etext = .; //gets the end location address of .text in the SECTIONS command.
   _sdata = .;//gets the start location address of .data in the SECTIONS command.
   _edata = .;//gets the end location address of .data in the SECTIONS command.
   _sbss = .;//gets the start location address of .bss in the SECTIONS command.
   _ebss = .;//gets the end location address of .bss in the SECTIONS command.

Memory mapping from FLASH to SRAM (Loader) : (Absolute address generated)

The memory mapping of FLASH starts from 0x08000000.
The memory mapping of SRAM starts from 0x20000000.
The memory mapping of .isr_vector (Vector table) starts from 0x08000000.
The memory mapping of .text of main.o starts from 0x080001c4.
The memory mapping of .text of stm32_startup starts from 0x08000228.
The memory mapping of .rodata of main.o starts from 0x080002b4.
The memory mapping of .data of main.o starts from 0x20000000.
The memory mapping of .data of stm32_startup starts from 0x20000004.
The memory mapping of .bss of main.o starts from 0x20000004.
The memory mapping of .bss of stm32_startup starts from 0x20000008.

Address of GPIOx and RCC:

We know,

Memory address = Base address + offset.

Thus, from the above statement we can determine all the addresses of GPIOx and RCC.

Example (GPIOx) :

GPIOA->OTYPER memory address is, 0x40020000 (Base address) + 0x04 (Offset)
That is, 0x40020004.

Example (RCC) :

RCC->AHB1ENR memory address is, 0x40023800 (Base address) + 0x30 (Offset)
That is, 0x40023830.

P.s- All the base addresses and offset values are given in the data structures of GPIOx and RCC under the filename “main.h”.

Definition of the “Weak” functions and section attributes (.isr_vector) for NVIC array:

Firstly we created a NVIC array named vectorTable that holds the MSP and handler addresses.

Code :

```
uint32_t vectorTable[] = {stored MSP and addresses of various handlers here};
```

We instructed the compiler not to include the above array in the .data section but in a different user defined section named **“.isr_vector”**.

Code :

```
uint32_t vectorTable[] __attribute__ ((section (".isr_vector"))) = {stored MSP and addresses of various handlers here};
```

section (“section-name”) : Normally, the compiler places the objects it generates in sections like data and bss. Sometimes, however, we need additional sections, or we need certain particular variables to appear in special sections, for example to map to special hardware. The section attribute specifies that a variable (or function) lives in a particular section. For example, in this assignment we created the NVIC vector table with the section named “.isr_vector”. The purpose of making a new section is that, we need to place the vector table at the start of the FLASH Memory address.

Function attribute (“weak”) : “weak” lets programmers override already defined weak function (dummy function) with the same function name.

Function attribute (“alias”) : “alias” lets programmers give an alias name for a function.

Example code :

```
void NMI_Handler(void) __attribute__ ((weak, alias ("Default_Handler")));
```