Microprocessor and Microcontroller – bare-metal programming: Development of lower-level system library for unrestricted programming

Microprocessor Lab

use C-language

Computer Science and Engineering, University of Dhaka,

Version 1.2

Submission Date: 25.04.2022

Demonstration: 26.04.2022

April 3, 2022

Contents

1 Objectives **1**
- 1.1 General Objectives 1
- 1.2 Working Objectives 2

2 Description **2**
- 2.1 Create Helper files and build system initialize files for the assignment 5
- 2.2 User Test Programming 5

3 Assignment Report **5**

4 What to Submit **6**

1 Objectives

1.1 General Objectives

This lab assignment aims to understand and have hands-on training to build a microcontroller driver for unrestricted/unlimited programming. This assignment envisions enriching students to have the boldness to develop deep system-level programming for the use of any developed micro-controller. In this case, developers do not need to wait for a new release of hardware abstraction layer like HAL (in CubeIde) or CMSIS of Keil. Besides, the assignment will unveil and give a solid knowledge of the linker, loader, and makefile components and concepts. In addition, the work will describe the memory mapping and address generation for area and sections for the machine code.

1.2 Working Objectives

First, the student will be able to configure the system clock and GPIO port as done in the microcontroller assignments. Next, the students will be able to recognize input and lightening and LED using a GPIO port as they did in the μC assignment. The program will be able to compile and run any program that uses a clock and GPIO. However, the student can extend their design or tools with the full support of SMT32xxx to the programmers. This assignment will be a tutorial for an initial library or driver development for the STM32 microcontroller.

2 Description

You must go through the following tutorial and prepare your report and develop a library with following specifications of STM32F446re

1. Bare metal embedded lecture-1 to 7
2. FLASH memory: Address starting from 0x08000000 and length is 512K
3. SRAM memory : Address starting from 0x2000000, length is 128K (combined SRAM1 and SRAM2)
4. Peripheral's address starting from (See datasheet table 12)
    - 0x4000 0000 – Address of APB1 (TIM2)
    - 0x4001 0000 – Address of APB2 (TIM1)
    - 0x4002 0000 – Address of AHB1 (GPIOA)
    - 0x5000 0000 – Address of AHB2 (USB OTG FS)
    - 0x6000 0000 – Address of AHB2 (FMC bank 1)

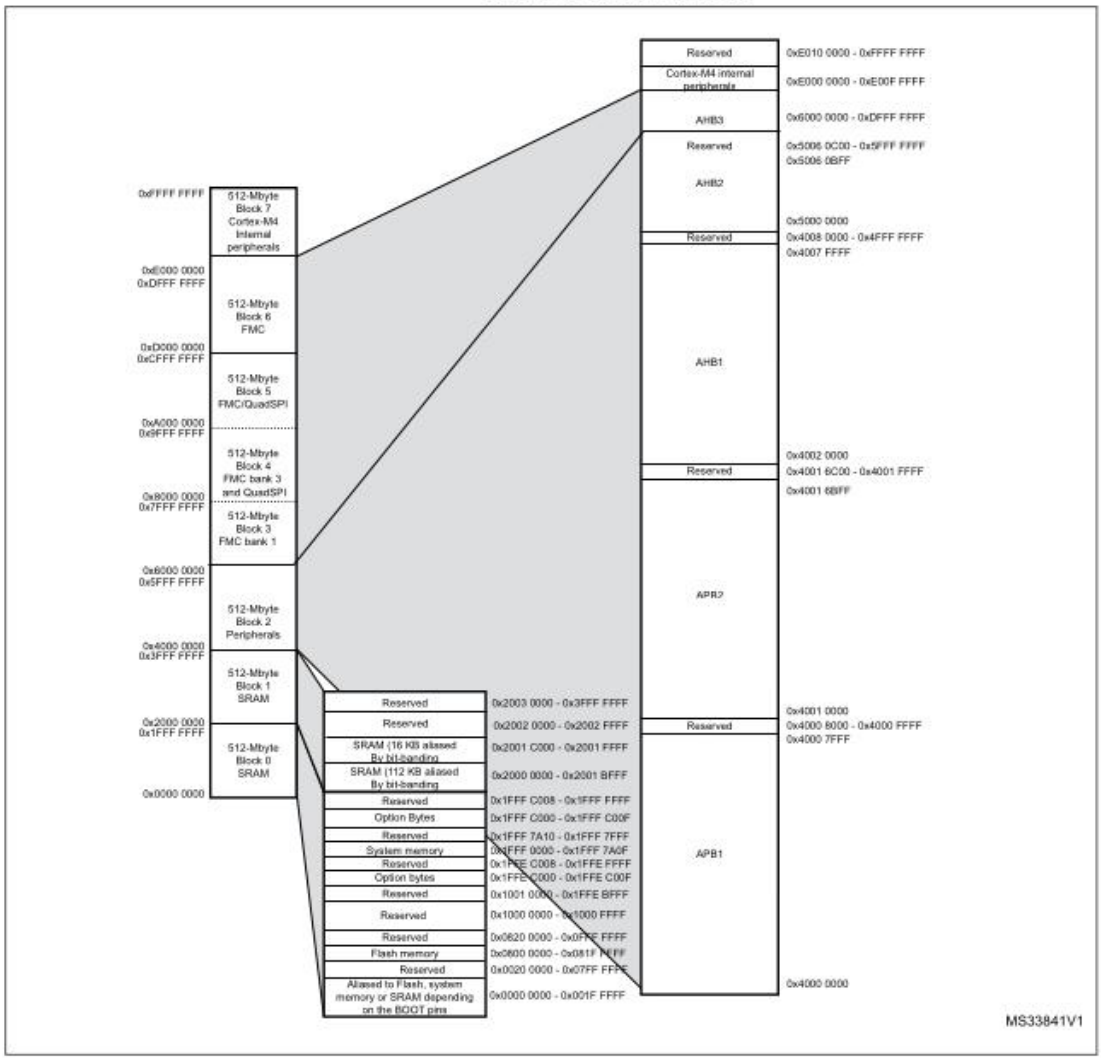

Figure 1: Stm32F4xx Memory Map

You must design a data structure for RCC registers and **GPIO ports** as following (Verify the address)

```
/*
* Data Structure for GPIO port
*/
typedef struct
{
uint32_t volatile MODER;         /* Offset: 0x00 (R/W) Mode Register                                    */
uint32_t volatile OTYPER;        /* Offset: 0x04 (R/W) Output Type Register                             */
uint32_t volatile OSPEEDR;       /* Offset: 0x08 (R/W) Output Speed Register                            */
uint32_t volatile PUPDR;         /* Offset: 0x0C (R/W) Pull-up/Pull-down Register                       */
uint32_t volatile IDR;           /* Offset: 0x10 (R/W) Input Data Register                              */
uint32_t volatile ODR;           /* Offset: 0x14 (R/W) Output Data Register                             */
uint32_t volatile BSRR;          /* Offset: 0x18 (R/W) Bit Set/Reset Register                           */
uint32_t volatile LCKR;          /* Offset: 0x1C (R/W) Configuration Lock Register                      */
uint32_t volatile AFRL;          /* Offset: 0x20 (R/W) Alternate Function Low Register                  */
uint32_t volatile AFRH;          /* Offset: 0x24 (R/W) Alternate Function High Register                 */
} GPIO_t;
#define GPIOA ((GPIO_t *)0x40020000)
#define GPIOB ((GPIO_t *)0x40020400)
#define GPIOC ((GPIO_t *)0x40020800)
#define GPIOD ((GPIO_t *)0x40020C00)
#define GPIOE ((GPIO_t *)0x40021000)
#define GPIOF ((GPIO_t *)0x40021400)
#define GPIOG ((GPIO_t *)0x40021800)
#define GPIOH ((GPIO_t *)0x40021C00)
```

and Data Structure for the RCC

```
/*
* Data Structure for Reset and Clock Control Registers (RCC), Address Range: 0x4002 3800 - 0x4002 3BFF */
typedef struct
{
uint32_t volatile CR;            /* Offset: 0x00 (R/W) Clock Control Register                                  */
uint32_t volatile PLLCFGR;       /* Offset: 0x04 (R/W) PLL Configuration Register                              */
uint32_t volatile CFGR;          /* Offset: 0x08 (R/W) Clock Configuration Register                            */
uint32_t volatile CIR;           /* Offset: 0x0C (R/W) Clock Interrupt Register                                */
uint32_t volatile AHB1RSTR;      /* Offset: 0x10 (R/W) AHB1 Peripheral Reset Register                          */
uint32_t volatile AHB2RSTR;      /* Offset: 0x14 (R/W) AHB2 Peripheral Reset Register                          */
uint32_t volatile AHB3RSTR;      /* Offset: 0x18 (R/W) AHB3 Peripheral Reset Register                          */
uint32_t volatile reserved0;
uint32_t volatile APB1RSTR;      /* Offset: 0x20 (R/W) APB1 Peripheral Reset Register                          */
uint32_t volatile APB2RSTR;      /* Offset: 0x24 (R/W) APB2 Peripheral Reset Register                          */
uint32_t reserved1[2];
uint32_t volatile AHB1ENR;       /* Offset: 0x30 (R/W) AHB1 Peripheral Clock Enable Register                   */
uint32_t volatile AHB2ENR;       /* Offset: 0x34 (R/W) AHB2 Peripheral Clock Enable Register                   */
uint32_t volatile AHB3ENR;       /* Offset: 0x38 (R/W) AHB3 Peripheral Clock Enable Register                   */
uint32_t reserved2;
uint32_t volatile APB1ENR;       /* Offset: 0x40 (R/W) APB1 Peripheral Clock Enable Register                   */
uint32_t volatile APB2ENR;       /* Offset: 0x44 (R/W) APB1 Peripheral Clock Enable Register                   */
uint32_t reserved3[2];
uint32_t volatile AHB1LPENR;     /* Offset: 0x50 (R/W) AHB1 Peripheral Clock Enable Lower Power Mode Register */
uint32_t volatile AHB2LPENR;     /* Offset: 0x54 (R/W) AHB2 Peripheral Clock Enable Lower Power Mode Register */
uint32_t volatile AHB3LPENR;     /* Offset: 0x58 (R/W) AHB3 Peripheral Clock Enable Lower Power Mode Register */
uint32_t reserved4;
uint32_t volatile APB1LPENR;     /* Offset: 0x60 (R/W) APB1 Peripheral Clock Enable Lower Power Mode Register */
uint32_t volatile APB2LPENR;     /* Offset: 0x64 (R/W) APB2 Peripheral Clock Enable Lower Power Mode Register */
uint32_t reserved5[2];
uint32_t volatile BDCR;          /* Offset: 0x70 (R/W) Backup Domain Control Register                          */
uint32_t volatile CSR;           /* Offset: 0x74 (R/W) Clock Control & Status Register                         */
uint32_t reserved6[2];
uint32_t volatile SSCGR;         /* Offset: 0x80 (R/W) Spread Spectrum Clock Generation Register               */
uint32_t volatile PLLI2SCFGR;    /* Offset: 0x84 (R/W) PLLI2S Configuration Register                           */
uint32_t volatile PLLSAICFGR;    /* Offset: 0x88 (R/W) PLLSAI Configuration Register                           */
uint32_t volatile DCKCFGR;       /* Offset: 0x8C (R/W) Dedicated Clocks Configuration Register                 */
uint32_t volatile CKGATENR;      /* Offset: 0x90 (R/W) Clocks Gated Enabled Register                           */
uint32_t volatile DCKCFGR2;      /* Offset: 0x94 (R/W) Dedicated Clocks Configuration Register 2               */
} RCC_t;
#define RCC ((RCC_t *)0x40023800)
```

2.1 Create Helper files and build system initialize files for the assignment

- Create and define **NVIC vector table and ISR** (IRQ handler) Ref: Table 38 of the reference manual.
  - Define array for the vector table with '.isr˙vector' section attributes
  - Define the Reset_Hadler for the reset IRQ. This ISR will start execution after resetting or powering on the device.
  - you must add 'main()' at the end of the Reset˙Handler to transfer control to the user-defined 'main()' function – the starting point of the program
- **Makefile** :
  - Cross Compilation for other processor like this one (arm) in the Intel environment
  - Compilation flags and conditional compilation.
  - Linker and loader definition and linking
  - Define target for object code and executable
  - Code cleaning and so on
- **Linker and loader** (e.g., stm32_ls.ld)
  - Memory: Flash memory, SRAM definition for the location where code and data are stored (FLASH Memory) and loaded (SRAM) for execution.
  - Section Definition
  - Define section boundary
  - Address alignment

2.2 User Test Programming

Write a program or main function to set the system clock at 180MHz and enable the GPIOA port. Use PA1 for input and PA4 for output. When you connect PA1 to Vcc, the LED will glow, and for GND, the LED will not emit any light. Your assignment will successfully compile the program and download it to the microcontroller board, and execute it without any error. You can use GDB to debug from Linux or Windows OS; however, we prefer Linux here. Follow the instruction given in the video lectures.

3 Assignment Report

The report must describe clearly the different gcc compilation and linker flags as

- -c, -mcpu, -mthumb, -Wall, -std=gnu11, and -O0
- -nostdlib -T stm32˙ls.ld -Wl,-Map

In addition the syntax and semantic of the linker (.ld) file

- ENTRY, SECTIONS, and memory mapping
- Explain Sections in the .ld files in respect to gcc

- Section .text, .data, .bss
- ALIGN, and location counter

Moreover, add explanation

- Memory mapping from FLASH to SRAM (Loader), absolute address generation
- Address of GPIOx and RCC, explain how to determine them.
- Explain 'week' definition of the functions and section attributes (.isr_vectior) for NVIC array

4 What to Submit

- Makefile, Linker file, system files for NVIC and Hadler definition, Data structure file for RCC and GPIO, and a Test program for input and output.
- Report