BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

EEE 416 – Microprocessor and Embedded Systems Laboratory
January 2022

Week 2: Experiment 02

Arm Cortex M: General Purpose Input Output (GPIO): Interfacing LED, Push-Button, and Stepper Motor

Evaluation Form:

IMPORTANT! You must complete this experiment during your scheduled lab period. All work for this experiment must be demonstrated to and verified by your lab instructor *before the end* of your scheduled lab period.

STEP	DESCRIPTION	MAX	SCORE
1	Blinking an LED (Section 3)	10	
2	Toggle LED with Pushbutton (Section 4)	10	
3	Run Stepper Motor in Full and Half Step (Section 5)	15	
4	Complete Lab Exercise 6.1 (Section 6)	20	
5	Complete Lab Exercise 6.2 (Section 6)	30	
6	Answer all questions	15	
	TOTAL	100	

Signature of Evaluator: ____

Academic Honesty Statement:

IMPORTANT! Please carefully read and sign the Academic Honesty Statement below. *You will not receive credit for this lab experiment unless this statement is signed in the presence of your lab instructor.*

"In signing this statement, I hereby certify that the work on this experiment is my own and that I have not copied the work of any other student (past or present) while completing this experiment. I understand that if I fail to honor this agreement, I will receive a score of ZERO for this experiment and be subject to possible disciplinary action."

Last Name (Printed): __________________________ **Lab Group:** ____ **Date:** __________

E-mail: ____________________@eee.buet.ac.bd **Signature:** __________________________

Contents

Evaluation Form:1

Academic Honesty Statement:1

1 Introduction1

1.1 Lab overview1

2 Pre-lab Study1

2.1 STM32 Nucleo Board2

2.2 Memory Mapped I/O3

2.3 Initializing a Port6

2.4 Stepper Motor8

3 Lighting up the LED9

4 Interfacing Push Button13

5 Stepper Motor Exercise14

6 Lab Exercises15

7 Question and Answer15

7.1 LED15

7.2 Push Button16

7.3 Stepper Motor18

Appendix A: Microcontroller STM32L47619

Appendix B: Microcontroller STM32F44620

Appendix C: Pin Connections on Nucleo-64 board21

Appendix D: Clock Configuration22

Appendix E: General Purpose I/O26

Appendix F: Install Keil MDKl..27

Appendix G: Creating a Base Project in Keil 29

Appendix H: Keil Debugger Tutorial 35

Appendix I: Code Comments and Documentation27

Appendix J: “Target not found” error28

Appendix K: Acknowledgement and References29

1 Introduction

1.1 Lab overview

At the end of this lab, you will be able to:

- Become familiar with the software development tool (Keil uVision)
- Create a C program that will run on the board (STM32 NUCLEO-L446RE)
- Learn basics of GPIO configuration
- Use GPIO for input (reading input from push button)
- Use GPIO for output (lighting up a LED)
- Understand polling I/O (busy waiting) and its inefficiency
- Understand software debounce techniques
- Understand the limits of GPIO output current
- Learn to use Darlington transistor arrays to drive devices requiring higher current loads
- Understand full and half stepping to control the speed and position of a stepper motor

2 Pre-lab Study

Before attempting this lab, please do the following:

1. Install **Windows** operation system if you do have it on your computer. Our development software, Keil uVision, can only run in Windows. If your computer runs Linux or Mac OS, you can install virtual machines running Windows.
2. Download free **Keil uVision 5**, or called Microcontroller Development Kit (MDK) Arm, https://www.keil.arm.com/mdk-community/) and install it. The Community Edition is free to use, but cannot compile ARMv7a or ARMv8a codes. For lab, we will be using ARMv8M code that uses Thumb encoding, and it should not be a problem.
3. Study:
    - Read [Zhu] Textbook **Chapter 4.6** to review bit-wise operations.
    - Read [Zhu] Textbook **Chapter 14 GPIO.**
    - Watch YouTube Tutorials (http://web.eece.maine.edu/~zhu/book/tutorials.php)
        (1) Lecture 5: Memory-mapped I/O (8 minutes)
        (2) Lecture 6: GPIO Output (11 minutes)
        (3) Lecture 7: GPIO Input (12 minutes)
    - Read Textbook Chapter 16 Stepper Motor
    - For background how stepper motors work, you can watch the following videos: *How the stepper motors are made and how they operate* (by pcbheaven)
        (1) Part 1 (5 minutes): http://www.youtube.com/watch?v=MHdz3c6KLrg
        (2) Part 2 (8 minutes): http://www.youtube.com/watch?v=t-3VnLadIbc

2.1 STM32 Nucleo Board

For the lab we use a STM32 Nucleo Board, with STM32F446RE processor. The STM32 Nucleo development boards allow developers to try out new ideas and quickly create prototypes with any STM32 MCU. Thanks to their connectors, STM32 Nucleo boards can extend their capabilities using multiple application-related hardware add-ons. The board used in the lab, Nucleo-64 boards feature Arduino Uno rev3 and ST morpho connectors. STM32 Nucleo boards integrate an ST-LINK debugger/programmer, eliminating the need for a separate probe. The development boards come with a comprehensive STM32 software HAL library and many software examples, and seamlessly work with a wide range of development environments, including IAR EWARM, Keil MDK-ARM and GCC/LLVM-based IDEs.

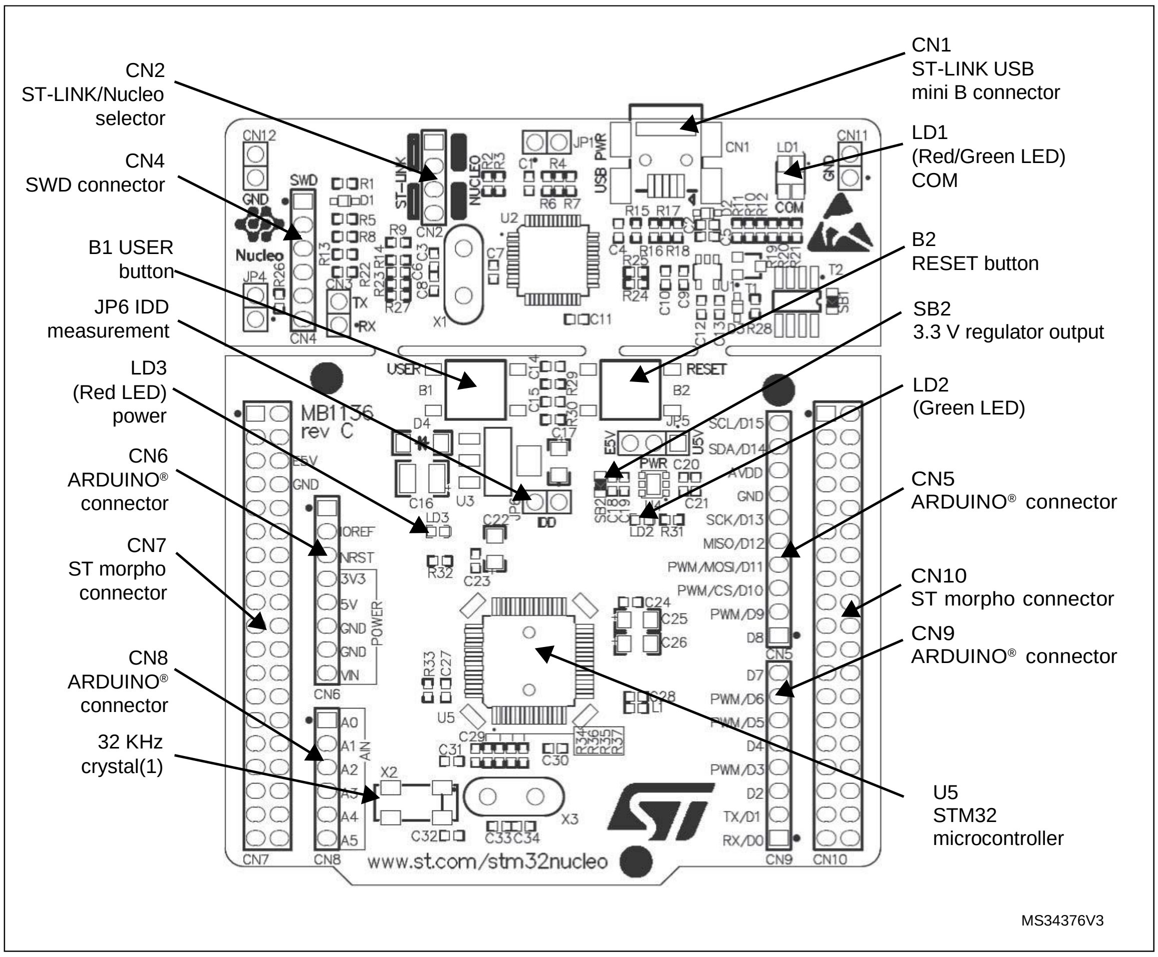


Figure 1: Component Layout of STM32-Nucleo Board Used in lab

The following onboard features will be used in this lab:
User LD2: the green LED is a user LED connected to ARDUINO® signal D13 corresponding to STM32 I/O PA5 (pin 21) or PB13 (pin 34) depending on the STM32 target.
B1 USER: the user button is connected to the I/O PC13 (pin 2) of the STM32 microcontroller.
B2 RESET: this push-button is connected to NRST, and is used to RESET the STM32 microcontroller.

2.2 Memory Mapped I/O

Arm uses Memory-Mapped I/O. Memory-mapped I/O uses the same address space to address both memory and I/O devices. The memory and registers of the I/O devices are mapped to (associated with) address values. So a memory address may refer to either a portion of physical RAM, or instead to memory and registers of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and must not be available for normal physical memory.

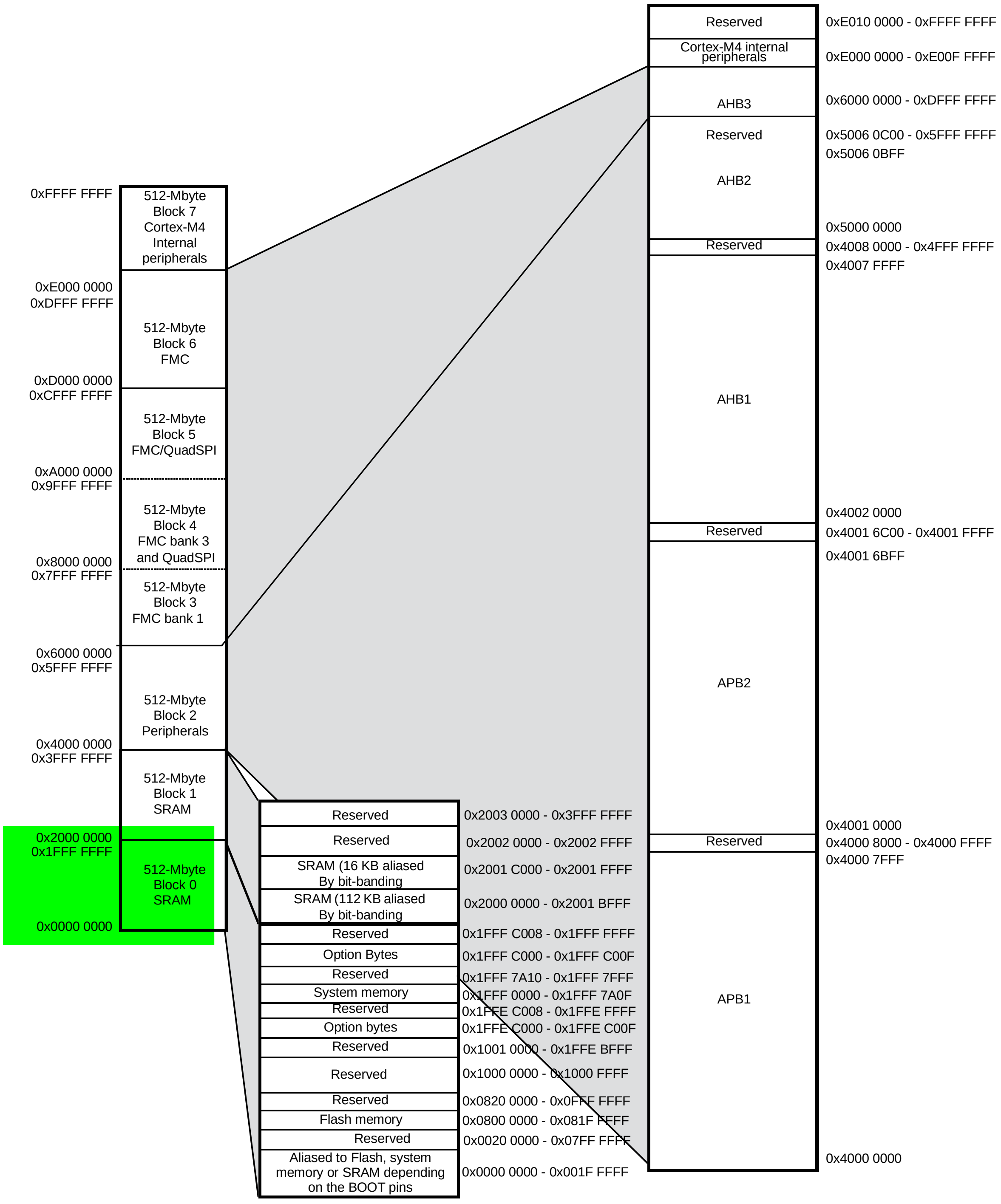

0xFFFF FFFF
512-Mbyte Block 7 Cortex-M4 Internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte Block 6 FMC
0xD000 0000
0xCFFF FFFF
512-Mbyte Block 5 FMC/QuadSPI
0xA000 0000
0x9FFF FFFF
512-Mbyte Block 4 FMC bank 3 and QuadSPI
0x8000 0000
0x7FFF FFFF
512-Mbyte Block 3 FMC bank 1
0x6000 0000
0x5FFF FFFF
512-Mbyte Block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte Block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte Block 0 SRAM
0x0000 0000
Reserved 0xE010 0000 - 0xFFFF FFFF
Cortex-M4 internal peripherals 0xE000 0000 - 0xE00F FFFF
AHB3 0x6000 0000 - 0xDFFF FFFF
Reserved 0x5006 0C00 - 0x5FFF FFFF
0x5006 0BFF
AHB2
0x5000 0000
Reserved 0x4008 0000 - 0x4FFF FFFF
0x4007 FFFF
AHB1
0x4002 0000
Reserved 0x4001 6C00 - 0x4001 FFFF
0x4001 6BFF
APB2
0x4001 0000
Reserved 0x4000 8000 - 0x4000 FFFF
0x4000 7FFF
APB1
0x4000 0000
Reserved 0x2003 0000 - 0x3FFF FFFF
Reserved 0x2002 0000 - 0x2002 FFFF
SRAM (16 KB aliased By bit-banding 0x2001 C000 - 0x2001 FFFF
SRAM (112 KB aliased By bit-banding 0x2000 0000 - 0x2001 BFFF
Reserved 0x1FFF C008 - 0x1FFF FFFF
Option Bytes 0x1FFF C000 - 0x1FFF C00F
Reserved 0x1FFF 7A10 - 0x1FFF 7FFF
System memory 0x1FFF 0000 - 0x1FFF 7A0F
Reserved 0x1FFE C008 - 0x1FFE FFFF
Option bytes 0x1FFE C000 - 0x1FFE C00F
Reserved 0x1001 0000 - 0x1FFE BFFF
Reserved 0x1000 0000 - 0x1000 FFFF
Reserved 0x0820 0000 - 0x0FFF FFFF
Flash memory 0x0800 0000 - 0x081F FFFF
Reserved 0x0020 0000 - 0x07FF FFFF
Aliased to Flash, system memory or SRAM depending on the BOOT pins 0x0000 0000 - 0x001F FFFF

Figure 2. Memory map of STM32F446

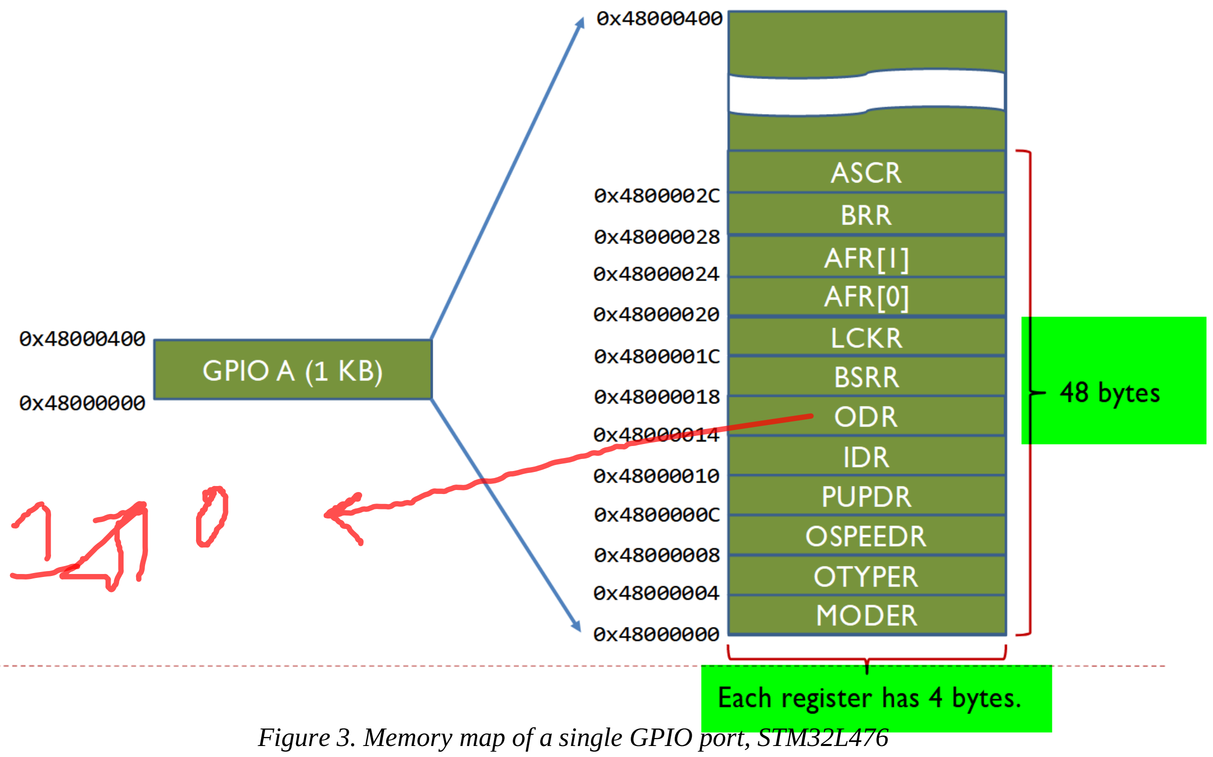
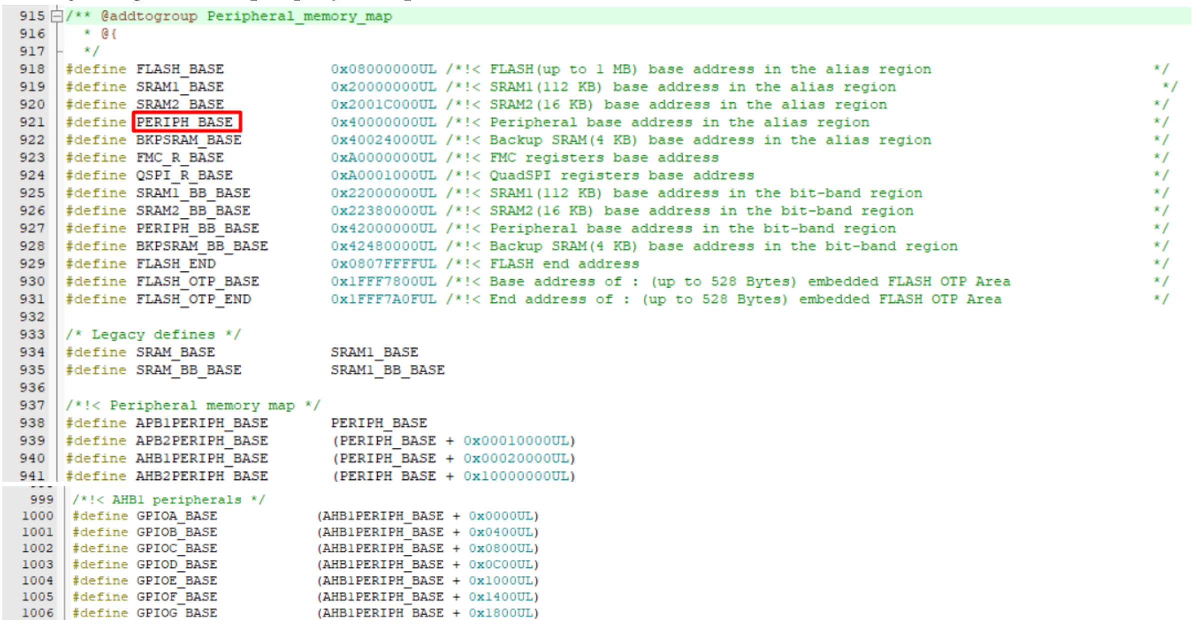


Figure 3. Memory map of a single GPIO port, STM32L476

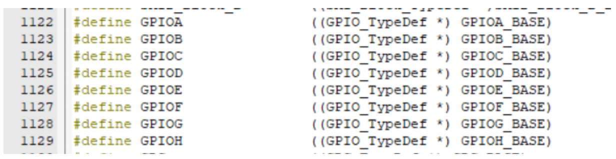
Depending on specific processor, the address of a particular output register can vary. For example, STM32L476 (As used in Zhu's book) has the address of GPIOB as 0x4800 0400 - 0x4800 07FF, where as STM32F446 (As used in Lab) has the address of GPIOB as 0x4002 0400 - 0x4002 07FF.

It is inconvenient to memorize exact address of the ports for writing program, so manufacturers typically give header library files with the addresses. **For the lab course, we will always try to use definitions from header file for writing code.**

With your given sample project, open the stm32f446xx.h header file.



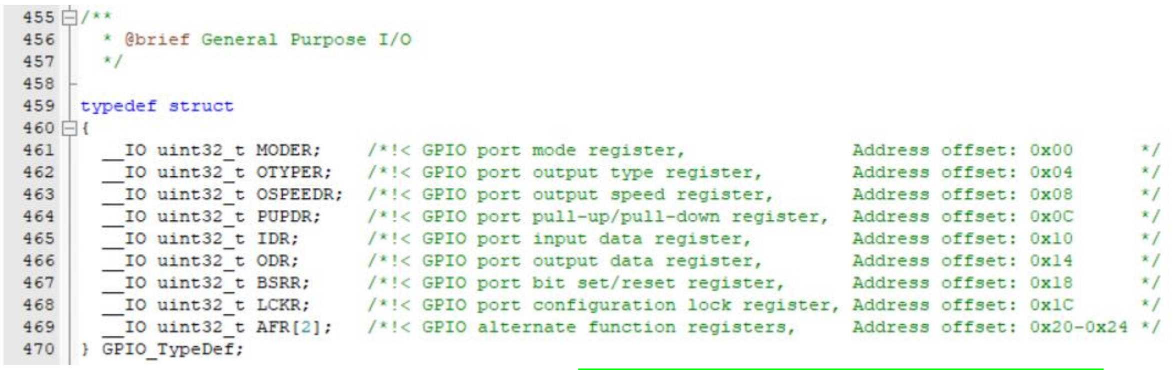
```
/** @addtogroup Peripheral_memory_map
  * @{
  */
#define FLASH_BASE            0x08000000UL /*!< FLASH(up to 1 MB) base address in the alias region                         */
#define SRAM1_BASE            0x20000000UL /*!< SRAM1(112 KB) base address in the alias region                              */
#define SRAM2_BASE            0x2001C000UL /*!< SRAM2(16 KB) base address in the alias region                              */
#define PERIPH_BASE           0x40000000UL /*!< Peripheral base address in the alias region                                */
#define BKPSRAM_BASE          0x40024000UL /*!< Backup SRAM(4 KB) base address in the alias region                         */
#define FMC_R_BASE            0xA0000000UL /*!< FMC registers base address                                                 */
#define QSPI_R_BASE           0xA0001000UL /*!< QuadSPI registers base address                                             */
#define SRAM1_BB_BASE         0x22000000UL /*!< SRAM1(112 KB) base address in the bit-band region                          */
#define SRAM2_BB_BASE         0x22380000UL /*!< SRAM2(16 KB) base address in the bit-band region                           */
#define PERIPH_BB_BASE        0x42000000UL /*!< Peripheral base address in the bit-band region                             */
#define BKPSRAM_BB_BASE       0x42480000UL /*!< Backup SRAM(4 KB) base address in the bit-band region                      */
#define FLASH_END             0x0807FFFFUL /*!< FLASH end address                                                          */
#define FLASH_OTP_BASE        0x1FFF7800UL /*!< Base address of : (up to 528 Bytes) embedded FLASH OTP Area                */
#define FLASH_OTP_END         0x1FFF7A0FUL /*!< End address of : (up to 528 Bytes) embedded FLASH OTP Area                 */

/* Legacy defines */
#define SRAM_BASE             SRAM1_BASE
#define SRAM_BB_BASE          SRAM1_BB_BASE

/*!< Peripheral memory map */
#define APB1PERIPH_BASE       PERIPH_BASE
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x00010000UL)
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000UL)
#define AHB2PERIPH_BASE       (PERIPH_BASE + 0x10000000UL)

/*!< AHB1 peripherals */
#define GPIOA_BASE            (AHB1PERIPH_BASE + 0x0000UL)
#define GPIOB_BASE            (AHB1PERIPH_BASE + 0x0400UL)
#define GPIOC_BASE            (AHB1PERIPH_BASE + 0x0800UL)
#define GPIOD_BASE            (AHB1PERIPH_BASE + 0x0C00UL)
#define GPIOE_BASE            (AHB1PERIPH_BASE + 0x1000UL)
#define GPIOF_BASE            (AHB1PERIPH_BASE + 0x1400UL)
#define GPIOG_BASE            (AHB1PERIPH_BASE + 0x1800UL)
```



```
#define GPIOA               ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB               ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC               ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD               ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE               ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF               ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG               ((GPIO_TypeDef *) GPIOG_BASE)
#define GPIOH               ((GPIO_TypeDef *) GPIOH_BASE)
```

Observe that all ports addresses are defined with hash. Recall from your C programming, these "precompilation directives" are automatically translated to their respective integer values before compilation. We can simply use these definitions to configure the particular ports. The ports are type cast into 'pointers' (GPIO_TypeDef *) to ensure that whenever they are used in the C code, they refer to a memory location, and the value at that particular location can be updated.

The GPIO_TypeDef is defined as the following structure:



```
/**
  * @brief General Purpose I/O
  */

typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;     /*!< GPIO port bit set/reset register,      Address offset: 0x18      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

So it contains several register definitions inside it. To access the output data register of PortB, we can use PORTB->ODR, that would point to the memory location of that register.

If we want to set the output of GPIO port B pin 6 to high, we can use the following C statement. lUL is an unsigned long integer with a value of 1. Note the pins are numbered 0 - 15, instead of 1 - 16.

```
        GPIOB->ODR |= 1UL<<6; // Set bit 6
```

Setting a pin to 0 can be done with the following statement

```
        GPIOB->ODR &= ~(1UL<<6); // Clear bit 6
```

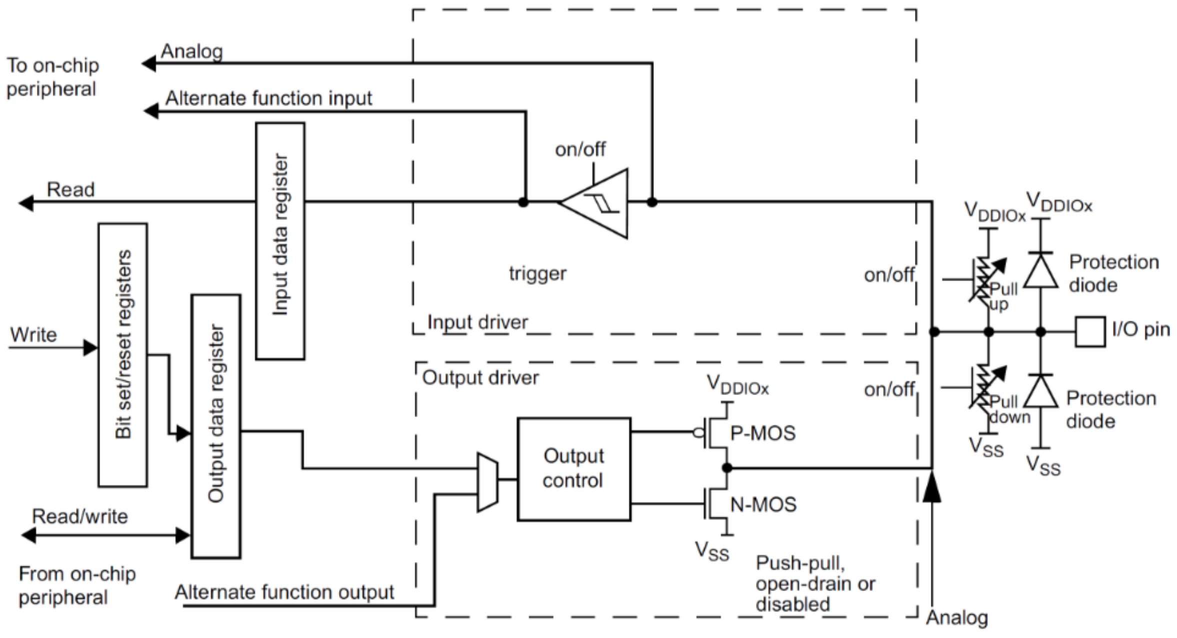
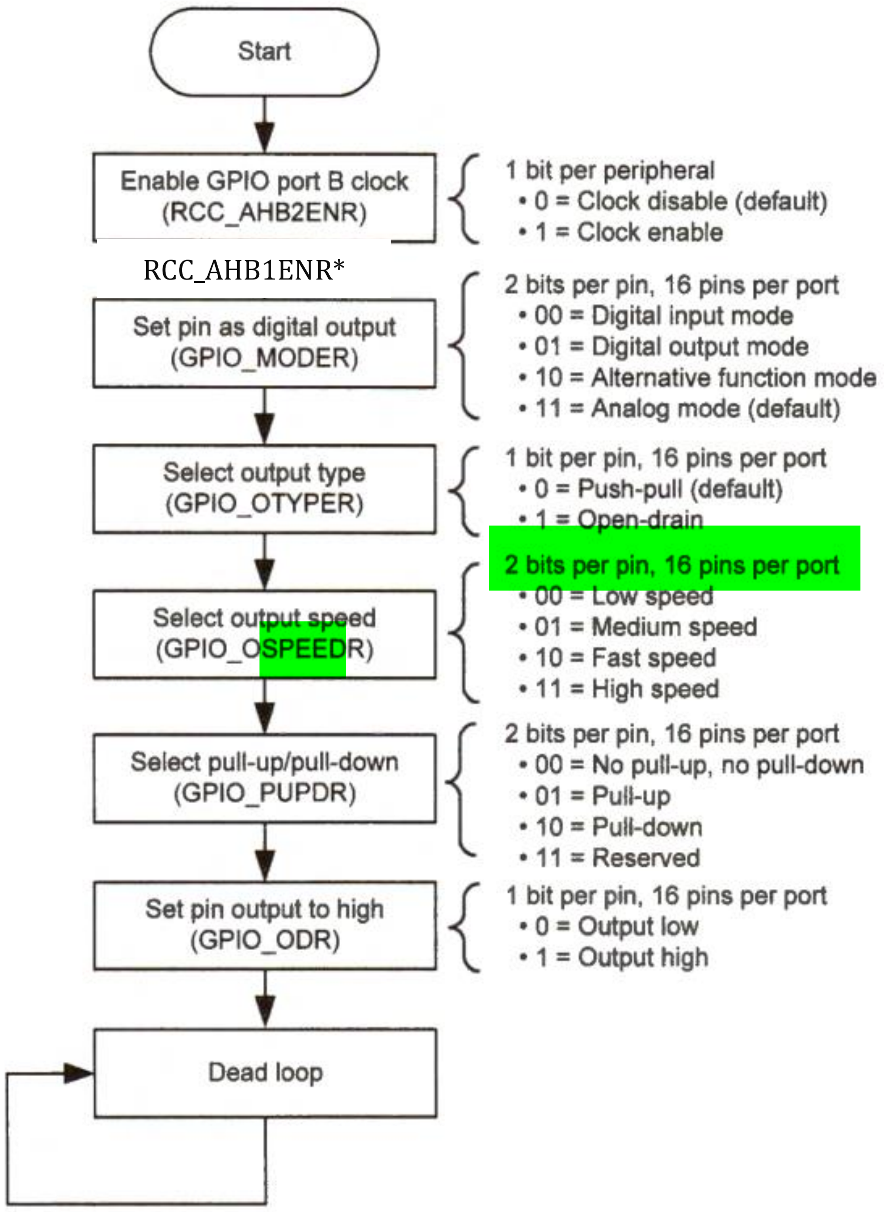


Figure 4. Basic Structure of an I/O Port Bit, Input and Output

But what does changing the value of a register mean? Each of the 16 pins of a single GPIO port has a lot of options integrated into its hardware, as shown in Fig. 4. above. We are basically enabling different parts of this diagram by putting certain value in the corresponding registers. [Details in **Appendix E**]

2.3 Initializing a Port

The following flowchart shows GPIO initialization steps:



Start
Enable GPIO port B clock (RCC_AHB2ENR)
RCC_AHB1ENR*
1 bit per peripheral
• 0 = Clock disable (default)
• 1 = Clock enable
Set pin as digital output (GPIO_MODER)
2 bits per pin, 16 pins per port
• 00 = Digital input mode
• 01 = Digital output mode
• 10 = Alternative function mode
• 11 = Analog mode (default)
Select output type (GPIO_OTYPER)
1 bit per pin, 16 pins per port
• 0 = Push-pull (default)
• 1 = Open-drain
Select output speed (GPIO_OSPEEDR)
2 bits per pin, 16 pins per port
• 00 = Low speed
• 01 = Medium speed
• 10 = Fast speed
• 11 = High speed
Select pull-up/pull-down (GPIO_PUPDR)
2 bits per pin, 16 pins per port
• 00 = No pull-up, no pull-down
• 01 = Pull-up
• 10 = Pull-down
• 11 = Reserved
Set pin output to high (GPIO_ODR)
1 bit per pin, 16 pins per port
• 0 = Output low
• 1 = Output high
Dead loop

Figure 14-18. Flowchart of GPIO initialization

(*Note, for STM32L4 architecture RCC_AHB2ENR register has GPIO clock, for STM32F4 architecture, RCC_AHB1ENR has the GPIO clock.)

For port A, for setting pin5 as output, I/O initialization, this can be done with the following code:

```
#define LED_PIN    5
static void configure_LED_pin(){
 // Enable the clock to GPIO Port A
 RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

        // GPIO Mode: Input(00), Output(01),
        // AlterFunc(10), Analog(11, reset)
        GPIOA->MODER &= ~(3UL<<(2*LED_PIN));
        GPIOA->MODER |=   1UL<<(2*LED_PIN);     // Output(01)

        // GPIO Speed: Low speed (00), Medium speed (01),
        // Fast speed (10), High speed (11)
        GPIOA->OSPEEDR &= ~(3U<<(2*LED_PIN));
        GPIOA->OSPEEDR |=   2U<<(2*LED_PIN);  // Fast speed

        // GPIO Output Type: Output push-pull (0, reset),
        // Output open drain (1)
        GPIOA->OTYPER &= ~(1U<<LED_PIN);      // Push-pull

        // GPIO Push-Pull:
        // No pull-up, pull-down (00),
        // Pull-up (01), Pull-down (10), Reserved (11)
        GPIOA->PUPDR  &= ~(3U<<(2*LED_PIN));
        // No pull-up, no pull-down
}
```

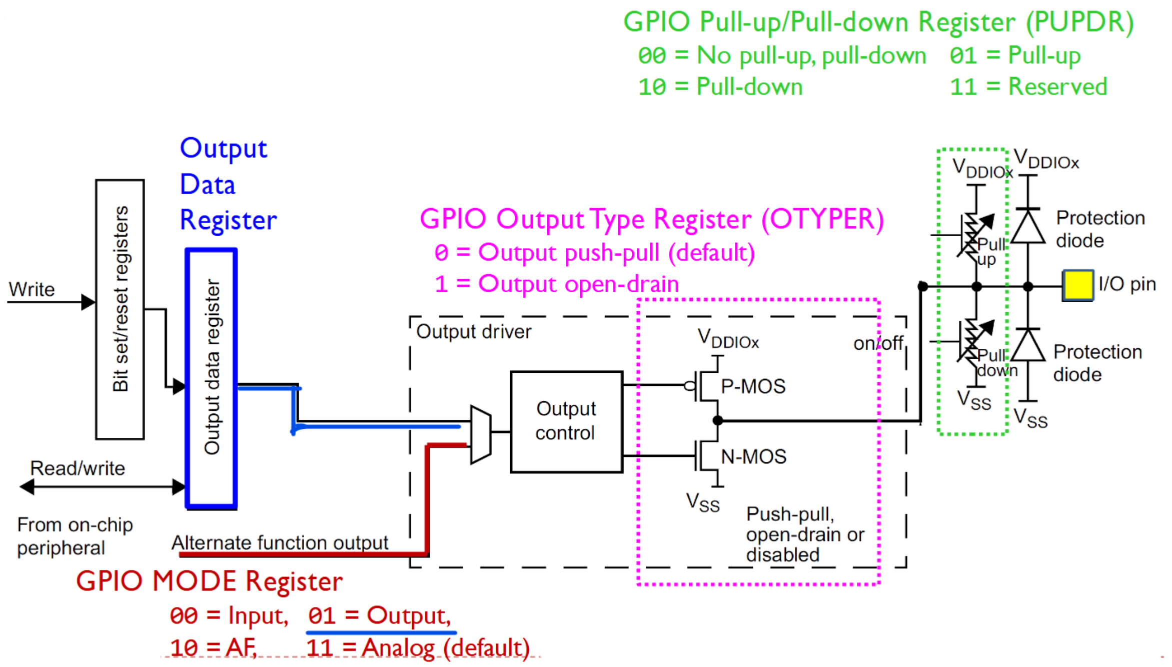
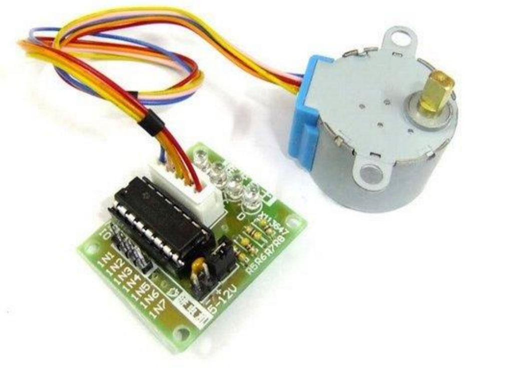

GPIO Pull-up/Pull-down Register (PUPDR)
00 = No pull-up, pull-down   01 = Pull-up
10 = Pull-down   11 = Reserved
Output Data Register
Bit set/reset registers
Write
Output data register
Read/write
From on-chip peripheral
Alternate function output
GPIO Output Type Register (OTYPER)
0 = Output push-pull (default)
1 = Output open-drain
Output driver
Output control
V_{DDIOx}
P-MOS
N-MOS
V_{SS}
on/off
Push-pull, open-drain or disabled
Pull up
Pull down
Protection diode
I/O pin
GPIO MODE Register
00 = Input, 01 = Output,
10 = AF, 11 = Analog (default)

Figure 5. Basic Sctucture of a GPIO pin in output mode

2.4 Stepper Motor



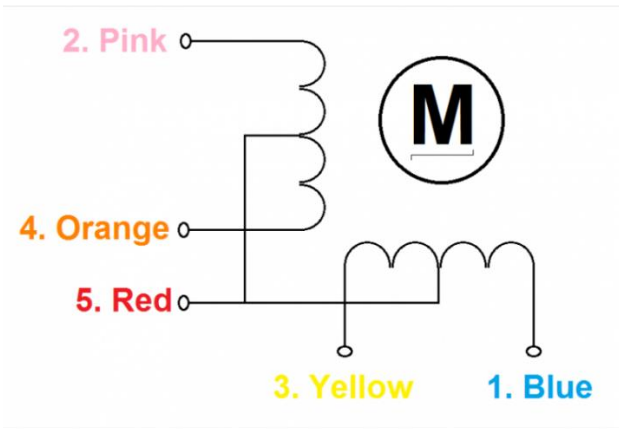


Figure 4. Stepper motor and control board (Model: 28BYJ-48).

A stepper motor and control board, as shown in Figure , will be provided. The control board has a ULN2003 Darlington Array. The following is the hardware information.

Motor model	**28BYJ-48**	Number of phases	2
Rated voltage	5V DC	Geared reduction ratio	1/64
DC resistance per phase	50Ω±7%(25℃)	Pull in torque	>300gf.cm / 5VDC 100pp

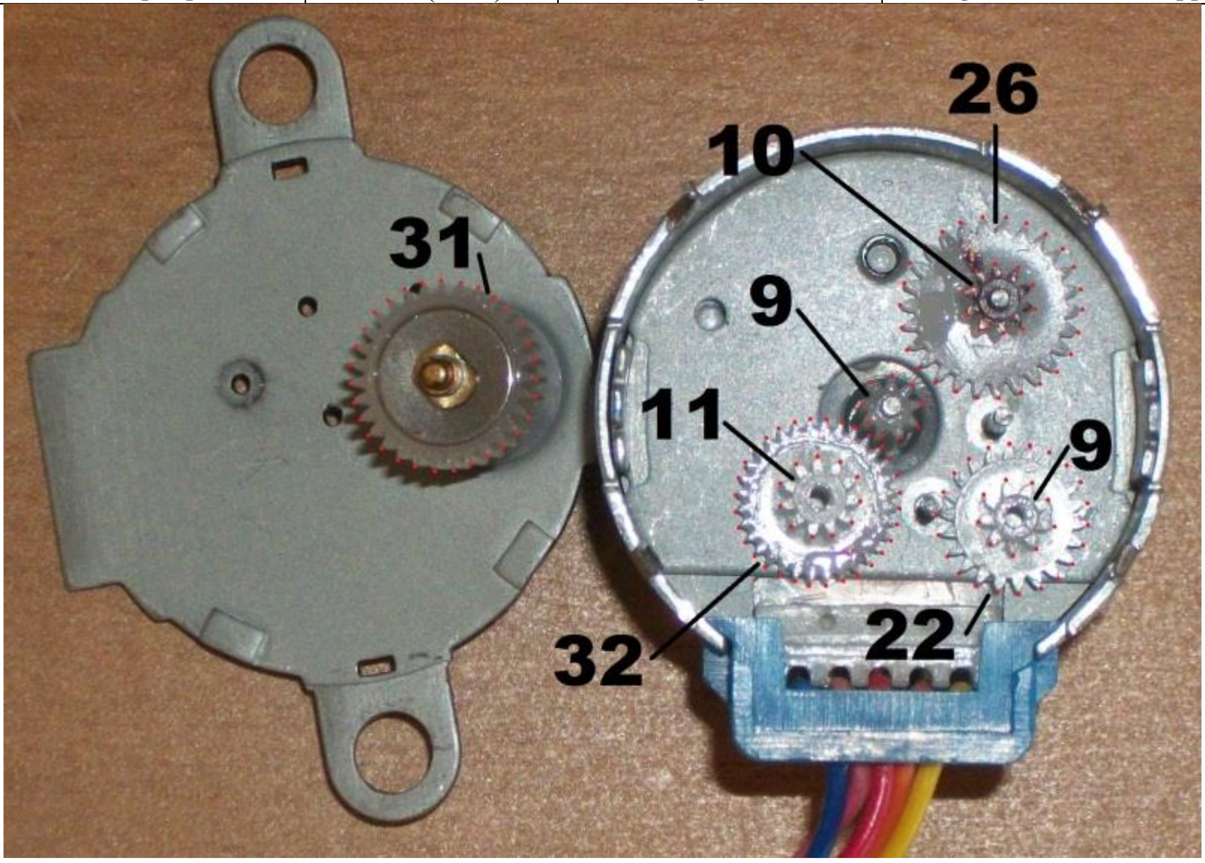


Figure 5. Internal gears of the stepper motor (image from forum.arduino.cc)

The gear ratio is:

$$\frac{31 \times 32 \times 26 \times 22}{11 \times 10 \times 9 \times 9} = 63.68395$$

If the output shaft rotates 1 resolution (gear with 31 teeth in the figure), the internal shaft (gear with 9 teeth in the middle) must rotate approximately 64 resolutions.

Full-stepping

- Internal motor: 32 steps per revolution
- Great reduction ratio: 1/63.68395, approximately 1/64
- Thus, it takes $32 \times 64 = 2048$ steps, or 512 repeats of the 4-step pattern, for the output shaft to complete a rotation

Half-stepping

- Internal motor: 64 steps per revolution
- Great reduction ratio: 1/63.68395 ≈ 1/64
- Thus, it takes $64 \times 64 = 4096$ steps, or 512 repeats of the 8-step pattern, for the output shaft to complete a rotation

To rotate a specific degree, your program should do the math to convert the value in degrees to a number of steps, and then do the rotation.

3 Lighting up the LED

- Do something cool. The following gives a few examples, but you are not limited to this. ***Creative ideas are always encouraged.***

There are one user-controllable green LED (light-emitting diodes) on the **STM32 NUCLEO-F446RE** board. The LED one is connected to the pin GPIO **PA5** (Port A Pin 5), as shown in the figure below.

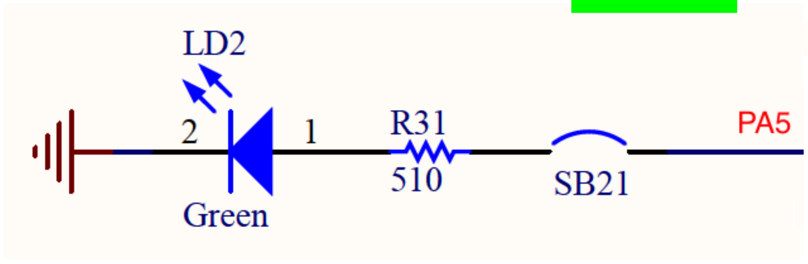


Figure 6: LED Circuit Diagram in the Nucleo Board

To turn on a LED, software must at least perform the following five operations:

1. **Enable the clock** of the corresponding GPIO port. By default, the clock to all peripherals, including GPIO ports, are turned off to save the energy.

2. Set the ***mode*** of the corresponding GPIO pin must be set as ***output.*** By default, the mode of all GPIO pin is analog.

3. Set the push-pull/open-drain setting for the GPIO pins to **push-pull**

4. Set the pull-up/pull-down setting for the GPIO pins to **no pull-up and no pull-down**.

5. Finally, set the output of the GPIO pin to have a value of 1 (corresponding to 3.3V)

For this section, we will program in C.

1. Copy the everything in the folder ***Project_C_Template_Simple*** to the folder ***lab_01***.

2. See if you can build the template code. It should build, but not do anything yet.

(1) Click on the **uvproj** file, and open the project from Keil.
(2) Click on the "**build**" button to build. This is near the top left, and looks like a box with an arrow going into it.
(3) Now edit the ***main.c*** file in the editor. Modify the C as described below.

3. Modify ***main.c*** to enable the registers you need to turn on the LED. You should have already calculated the values you need to write in the Pre-lab. If you want a reference for what you are programming, see *RM0351 Reference manual: STM32L4x5 and STM32L4x6 advanced Arm-based 32-bit MCUs*.

You will use bitwise operations to set/clear the registers. Some of this might seem repetitive. Feel free to use function calls or other more advanced C methods if you feel like it helps.

(1) Program the **AHB1ENR** register to enable the clock of **GPIO Port A**.

```
RCC->AHB1ENR
```

The provided ***stm32f476xx.h*** header file provides some helpers to make this easier. The memory-mapped I/O has been set up to point to the various structures with a volatile pointer. For example, to set the **GPIOCEN** bit (GPIO **Port C** enable) in the **AHB1ENR** register, you would do something like the following:

```
RCC->AHB1ENR |= (1<<2);
```

The header file ***stm32f446xx.h*** provides:

```
#define  RCC_AHB1ENR_GPIOCEN  ((uint32_t)0x00000004)
#define RCC_AHB1ENR_GPIOCEN_Pos         (2U)
#define RCC_AHB1ENR_GPIOCEN_Msk  (0x1UL << RCC_AHB1ENR_GPIOCEN_Pos)
#define RCC_AHB1ENR_GPIOCEN   RCC_AHB1ENR_GPIOCEN_Msk
```

Thus, the following is recommended to make your code more readable and easier to debug and maintain.

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
```

Note that the following statement is incorrect because it enables the clock of port C but at the time it turn off all the other clock controlled by this register. The code should enable the port C clock, without impacting the clock settings of the other ports.

```
RCC->AHB1ENR = RCC_AHB1ENR_GPIOCEN;
```

(2) Program the port A mode register (**MODER**) to set Pin 5 to be output.

```
GPIOA->MODER
```

(3) Program the port A output type register (**OTYPER**) to set Pin 5 as push-pull.

```
GPIOA->OTYPER
```

(4) Program the port A pull-up/pull-down register (**PUPDR**) to set Pin 5 as no-pull-up no pull-down.

```
GPIOA->PUPDR
```

(5) Finally, program the port A output data register (**ODR**) to set the output of Pin 5 to 1 or 0, which enable or disable the LED, respectively.

```
GPIOA->ODR
```

To output high (3.3V) on pin 5, just set bit 5 of **ODR** to 1.

(6) At the end of your code put an infinite loop to keep the code from executing off the end of your program. This would look something like

```
  while(1);
```

4. While doing this, be sure to comment your code appropriately!

5. Now compile/build your code

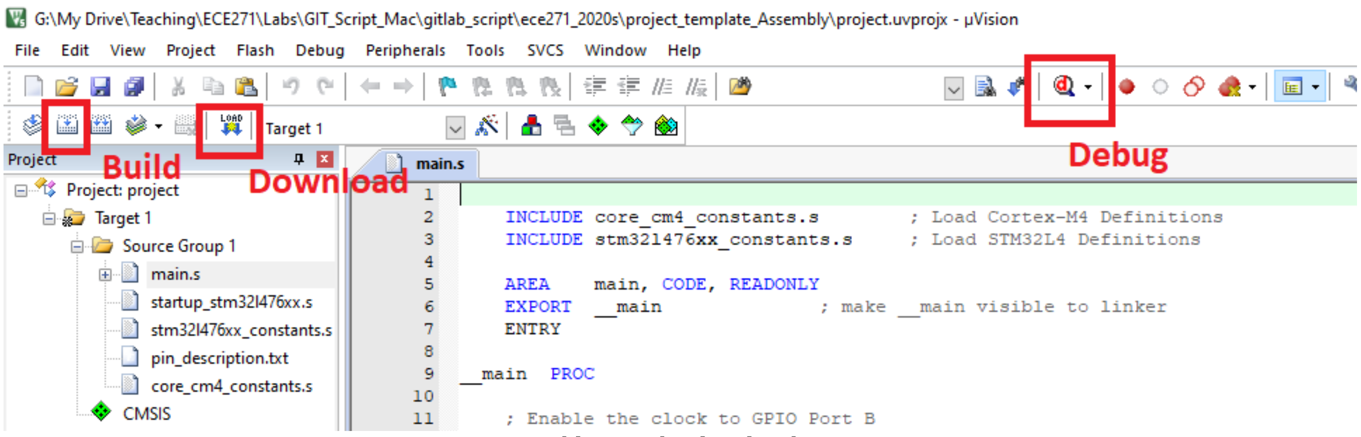
(1) Press the **build** button. Check the window at the bottom for warnings and especially for errors. The IDE will show little error icons by your code too if it detects any.

(2) Then, plug the board into your laptop with the USB cable if you haven't already.

(3) Now download your code to the board by pressing the **download** button.

(4) Push the reset button (black) on the board to run your program

It is possible that your code is buggy so that Keil can't program the board anymore. In this case, you can use ST-Link utility to erase the flash memory. See this YouTube tutorial: https://www.youtube.com/watch?v=OiwwB0AvIBI


G:\My Drive\Teaching\ECE271\Labs\GIT_Script_Mac\gitlab_script\ece271_2020s\project_template_Assembly\project.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Build
Download
Debug
main.s
Project: project
Target 1
Source Group 1
main.s
startup_stm32l476xx.s
stm32l476xx_constants.s
pin_description.txt
core_cm4_constants.s
CMSIS

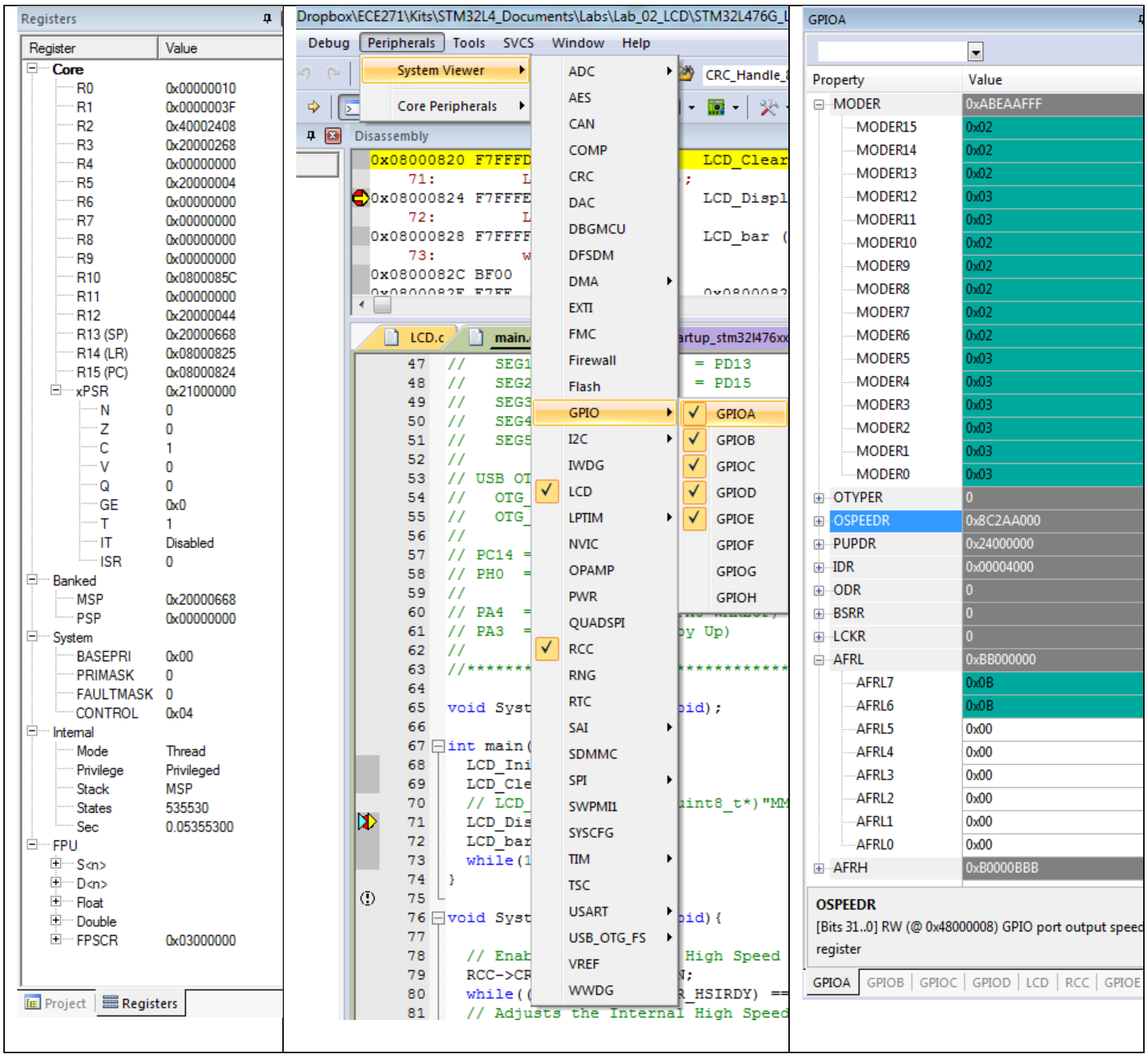
```

    INCLUDE core_cm4_constants.s        ; Load Cortex-M4 Definitions
    INCLUDE stm32l476xx_constants.s     ; Load STM32L4 Definitions

    AREA    main, CODE, READONLY
    EXPORT  __main              ; make __main visible to linker
    ENTRY

__main  PROC

    ; Enable the clock to GPIO Port B
```

Figure 7: Build, Download and Debug

If the code doesn’t work, you will have to debug your code to find out what is wrong.

Review this debug tutorial: https://youtu.be/w4gPcYRk9o8

Keil allows us to view the values of Cortex-M registers and all peripheral registers in real-time. Click the following: Peripherals, System Viewers, GPIO, and GPIOC



Register	Value
Core	
R0	0x00000010
R1	0x0000003F
R2	0x40002408
R3	0x20000268
R4	0x00000000
R5	0x20000004
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x0800085C
R11	0x00000000
R12	0x20000044
R13 (SP)	0x20000668
R14 (LR)	0x08000825
R15 (PC)	0x08000824
xPSR	0x21000000
N	0
Z	0
C	1
V	0
Q	0
GE	0x0
T	1
IT	Disabled
ISR	0
Banked	
MSP	0x20000668
PSP	0x00000000
System	
BASEPRI	0x00
PRIMASK	0
FAULTMASK	0
CONTROL	0x04
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	535530
Sec	0.05355300
FPU	
S<n>	
D<n>	
Float	
Double	
FPSCR	0x03000000

```
0x08000820 F7FFFD          LCD_Clear
      71:                L
0x08000824 F7FFFE          LCD_Displ
      72:                L
0x08000828 F7FFFF          LCD_bar (
      73:                w
0x0800082C BF00
0x0800082E F7FF            0x0800082
```

```
//    SEG1           = PD13
//    SEG2           = PD15
//    SEG3
//    SEG4
//    SEG5
//
// USB OT
//    OTG_
//    OTG_
//
// PC14 =
// PH0  =
//
// PA4  =
// PA3  =               by Up)
//
//*******               ************

void Syst               oid);

int main(
  LCD_Ini
  LCD_Cle
  // LCD_               uint8_t*)"MM
  LCD_Dis
  LCD_bar
  while(1
}

void Syst               oid){

  // Enab               High Speed
  RCC->CR               N;
  while((               R_HSIRDY) ==
  // Adjusts the Internal High Speed
```

Property	Value
MODER	0xABEAAFFF
MODER15	0x02
MODER14	0x02
MODER13	0x02
MODER12	0x03
MODER11	0x03
MODER10	0x02
MODER9	0x02
MODER8	0x02
MODER7	0x02
MODER6	0x02
MODER5	0x03
MODER4	0x03
MODER3	0x03
MODER2	0x03
MODER1	0x03
MODER0	0x03
OTYPER	0
OSPEEDR	0x8C2AA000
PUPDR	0x24000000
IDR	0x00004000
ODR	0
BSRR	0
LCKR	0
AFRL	0xBB000000
AFRL7	0x0B
AFRL6	0x0B
AFRL5	0x00
AFRL4	0x00
AFRL3	0x00
AFRL2	0x00
AFRL1	0x00
AFRL0	0x00
AFRH	0xB0000BBB

OSPEEDR
[Bits 31..0] RW (@ 0x48000008) GPIO port output speed register

Figure 8: Different Debug Components

4 Interfacing Push Button

There is a blue user button the board. The button is connected to the microcontroller's **GPIO Port C Pin 13 (PC 13)**, as shown on the right.

- The pin is pulled up to VDD via a resistor (R30).
- When the button is not pressed, the voltage on PC 13 is VDD.
- When the button is pressed down, the voltage on PC 13 is 0.
- The circuit performs ***hardware debouncing*** using a resistor (R29) and a capacitor (C15). It forms a simple RC filter (resistive capacitive filter), which debounces the pushbutton switch.

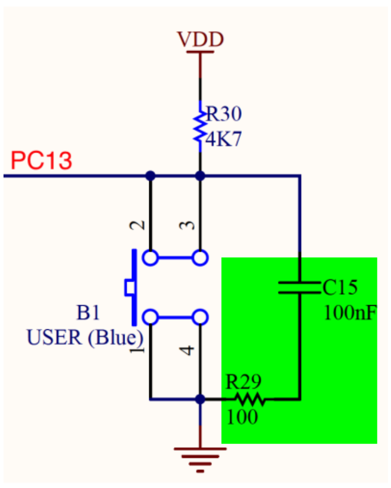


Figure 9: Push Button Connection

You will do the above in C.

1. Program **`RCC->AHB1ENR`** and enabled the clock to GPIO Port C (by default they are disabled).
2. Program **`GPIOC->MODER`** and set the mode of Pin PC 13 to be **input** (by default they are analog).
3. Program **`GPIOC->PUPDR`** and set the pull-up/pull-down setting of Pin PC 13 to have a **no pull-up no pull-down**.
4. Modify your code to have an infinite loop. Each time through the loop, read the status the USER pushbutton. Each time the button is pressed, the LED is toggled.

   (1) Read **`GPIOC->IDR`** and use proper bitwise operation to check whether bit 13 is 0. As hardware diagram shows, the voltage on pin PC 13 is high if the user button is not pressed. It is strongly recommended to constants defined in the provided ***stm32f446xx.h*** header file to make your code more readable. For example, the bit mask of bit 13 is defined in the header file, which is equivalent to `1<<13`.

```
// Provided in stm32F446xx.h and no need to redefine it.
 #define GPIO_IDR_IDR_13       GPIO_IDR_ID13

 #define GPIO_IDR_IDR_13      ((uint32_t)0x00002000)

 #define GPIO_IDR_ID13_Pos     (13U)

 #define GPIO_IDR_ID13_Msk     (0x1UL << GPIO_IDR_ID13_Pos)
     /*!< 0x00002000 */

 #define GPIO_IDR_ID13         GPIO_IDR_ID13_Msk
```

   The following statement waits until the button is pressed.

```
While(GPIOC->IDR & GPIO_IDR_IDR_13);
```

   (2) **Toggle the LED**. Review the bitwise operations regarding how to toggle a bit in a register.

5 Stepper Motor Exercise

Interfacing the stepper motor requires four pins, excluding +5V and ground. We will be using GPIO **PC 5**, **PC 6**, **PC 8**, and **PC 9**.

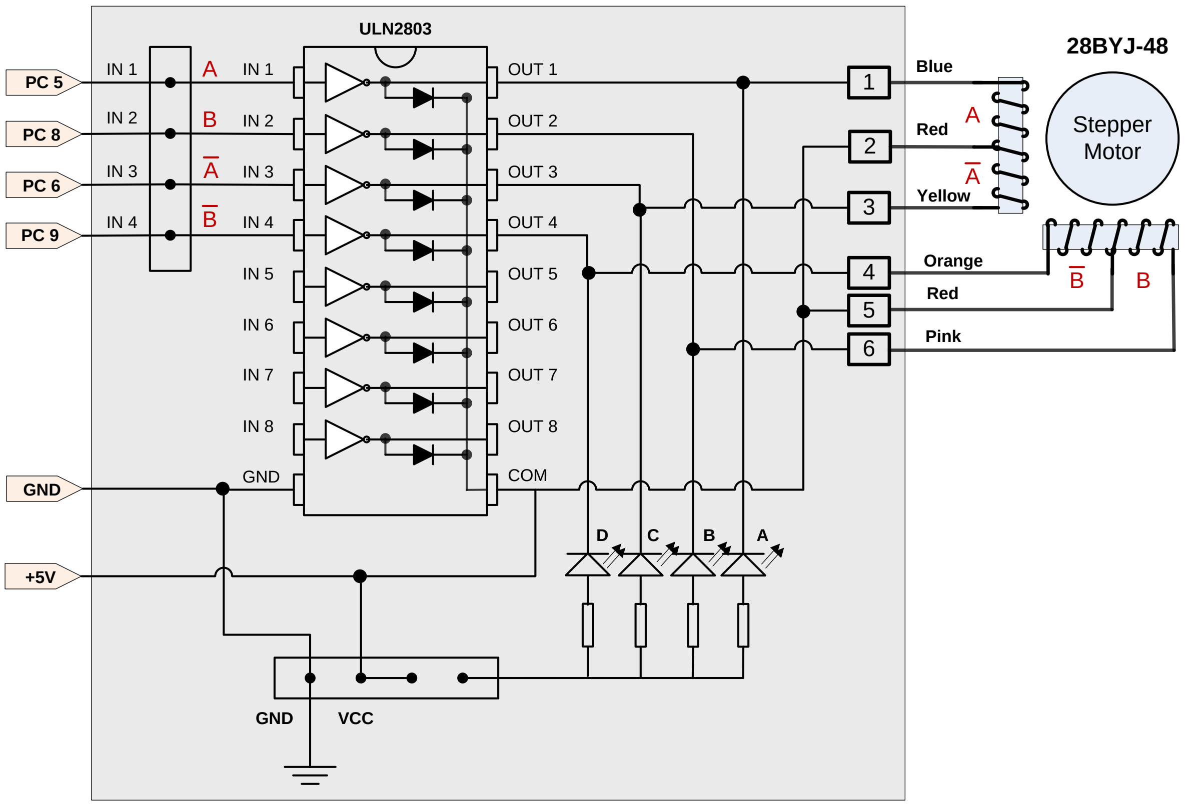
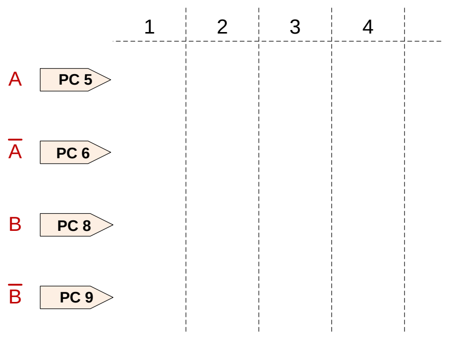
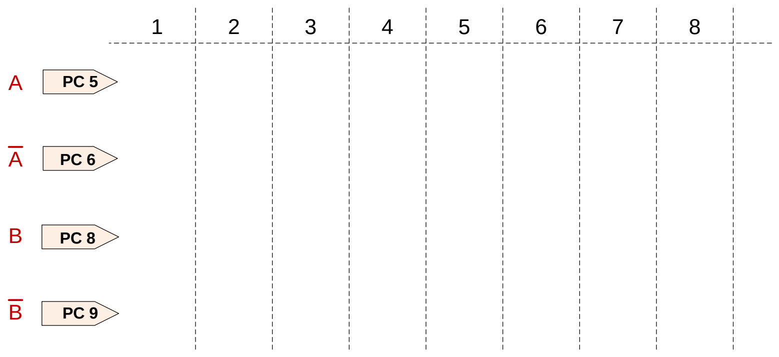


Figure 10. Connection diagram of the motor driver board (Model: 28BYJ-48).

Refer to Figure 16-10 and 16-12 of textbook, and sketch out below what the waveform should be on the pins for a full-stepping sequence and a half-stepping sequence.

Full stepping sequence **Half stepping sequence**





WARNING!
The motor constantly draws electrical currents. The motor will be overheated if you leave the power on for an extended period. **Disconnect the 5V power (Vcc) to the Darlington array if you are not actively debugging/testing it.**

a. How would you change the rotation speed of a stepper motor?

b. How would you reverse the rotation direction?

6 Lab Exercises

6.1 Problem 1: Using GPIO a LED to send out SOS in Morse code (· · · – – – · · ·) if the user button is pressed. DOT, DOT, DOT, DASH, DASH, DASH, DOT, DOT, DOT. DOT is on for ¼ second and DASH is on for ½ second, with ¼ second between these light-ons. Write the main program below. Demonstrate working code during lab time

6.2 Write a program that would do the following tasks. Demonstrate the code during lab time. Write the code below:

- When push button is not pressed, stepper motor will run counter clock-wise in full-step
- When push button is pressed, LED starts blinking AND stepper motor runs clock wise

```
static void main(void) {}
```

7 Question and Answer

Read the relevant chapter of the textbook and answer the following questions:

7.1 LED

a. Why did we configure the pin PA 5 that drives the LED as push-pull instead of open-drain?

b. What is GPIO output speed? What is the default speed? Did you notice any difference of you choose different speeds in this lab assignment?

c. Each GPIO pin has four programmable output speeds: low, medium, fast, and high. Specifically, for STM32F4, the slew rate of the high-speed output can be up to 80 MHz. Why the "low speed" configuration is recommended for controlling LED? (Hints: energy and electromagnetic interference)

d. Why did we configure the pin that drive the LED as push-pull instead of open-drain?

7.2 Push Button

e. The pushbutton on the board has a ***hardware debouncing*** circuit. Explain briefly how the hardware debouncing circuit works.

f. Debouncing can also be done in software. Explain how this could be done in software

g. Why do we configure the pins that drive the pushbutton as no pull-up no pull-down? Can we configure it as pull-down?

7.3 Stepper Motor

h. The Darlington array can only provide 500-mA of current. If you need a larger current, what could you use instead of the Darlington array.

i. Is it possible to rotate the motor less than 1/2 step? (Hints: micro-stepping, see Textbook Chapter 16.6)

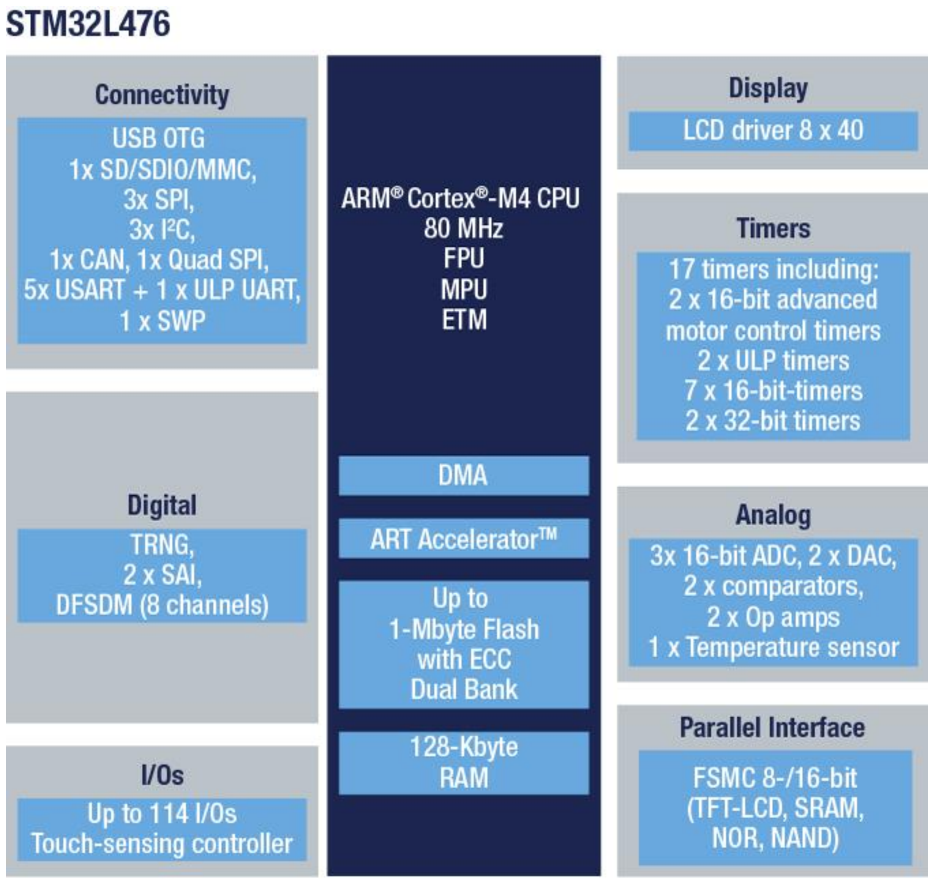
j. What is GPIO output speed? What is the default speed? Did you notice any difference of you choose different speeds in this lab assignment?

Appendix A: Microcontroller STM32L476

The NUCLEO-L476RG board has a 32-bit Arm Cortex-M4F microcontroller. This is used in our text book.

- Part Number: **STM32L476RGT6**
- Total number of pins: 64
- Total number of I/O pins: 51
- Maximum clock frequency: 80MHz
- Program memory size: 1 MB
- Operating supply voltage: 1.71V to 3.6V
- I/O voltage: 3.3V
- Analog supply voltage: 3.3V

The following figure summarizes the I/O connection supported and internal peripherals.



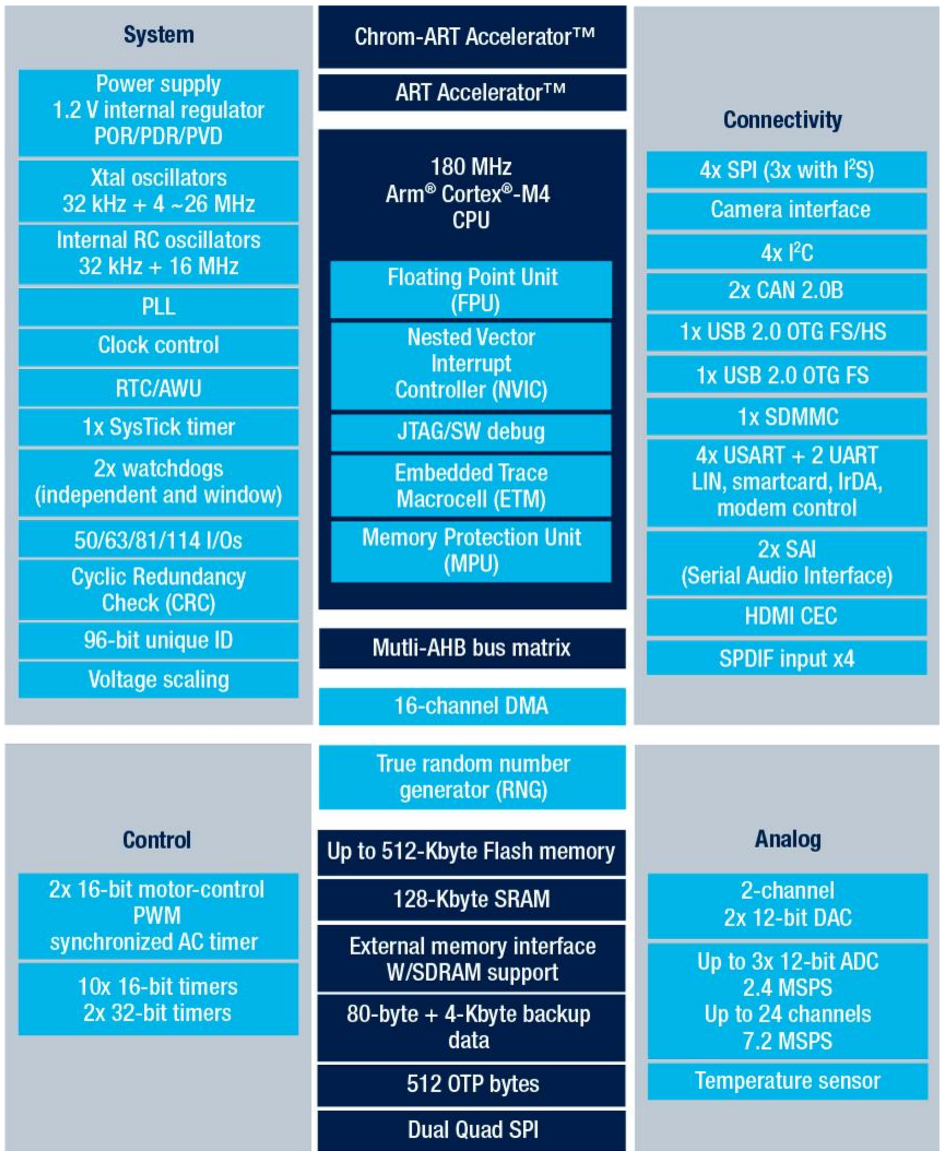
Appendix B: Microcontroller STM32F446

The NUCLEO-L446RE board has a 32-bit Arm Cortex-M4F microcontroller.
This board is used in BUET lab.

- Part Number: **STM32F446RE**
- Total number of pins: 64
- Total number of I/O pins: 51
- Maximum clock frequency: **180MHz**
- Program memory size: **512 kB**
- Operating supply voltage: 1.71V to 3.6V
- I/O voltage: **1.7-3.6V**
- Analog supply voltage: **1.7-3.6V**

The following figure summarizes the I/O connection supported and internal peripherals.

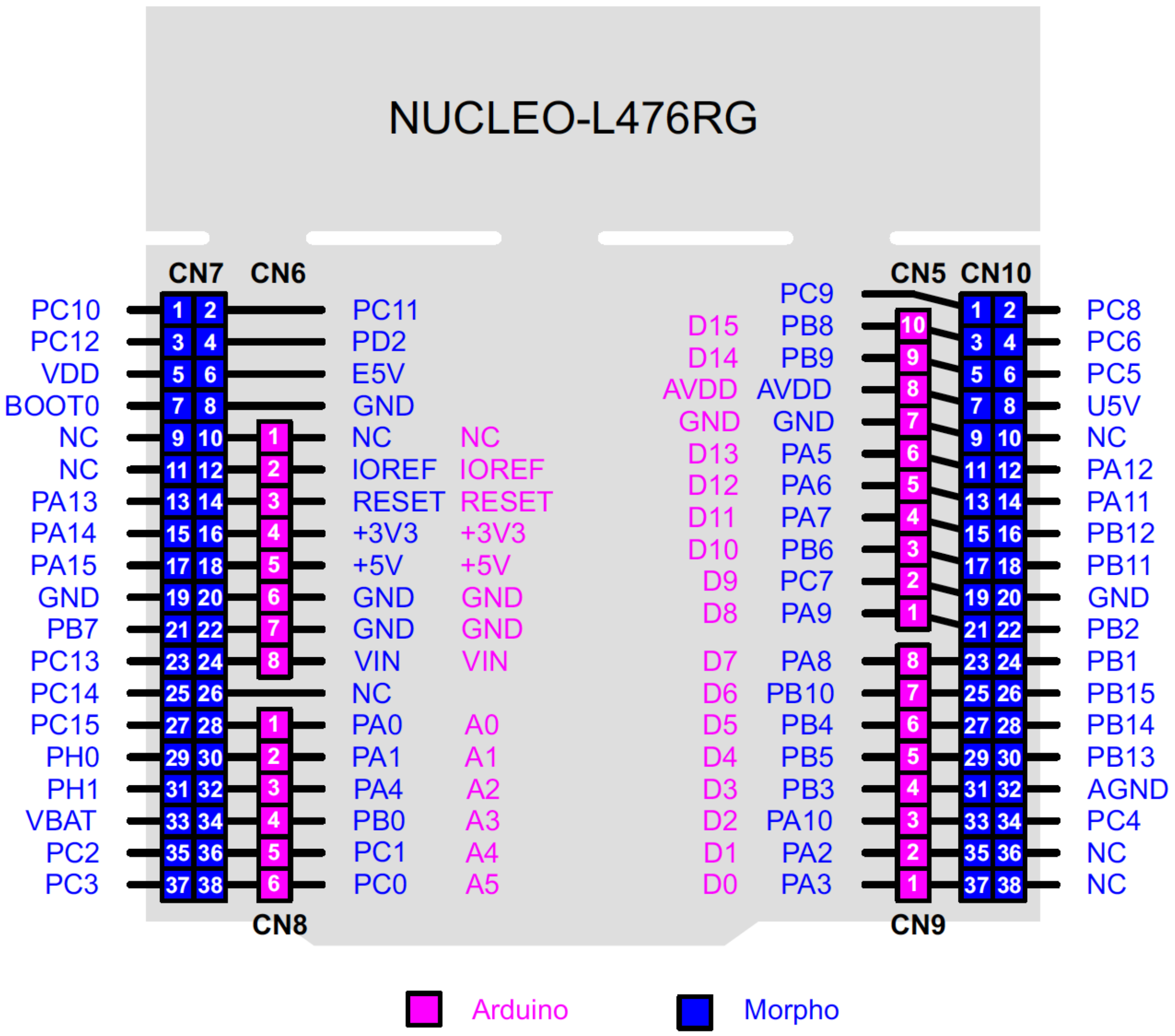
STM32F446


System
Power supply 1.2 V internal regulator POR/PDR/PVD
Xtal oscillators 32 kHz + 4 ~26 MHz
Internal RC oscillators 32 kHz + 16 MHz
PLL
Clock control
RTC/AWU
1x SysTick timer
2x watchdogs (independent and window)
50/63/81/114 I/Os
Cyclic Redundancy Check (CRC)
96-bit unique ID
Voltage scaling
Chrom-ART Accelerator™
ART Accelerator™
180 MHz Arm® Cortex®-M4 CPU
Floating Point Unit (FPU)
Nested Vector Interrupt Controller (NVIC)
JTAG/SW debug
Embedded Trace Macrocell (ETM)
Memory Protection Unit (MPU)
Mutli-AHB bus matrix
16-channel DMA
Connectivity
4x SPI (3x with I²S)
Camera interface
4x I²C
2x CAN 2.0B
1x USB 2.0 OTG FS/HS
1x USB 2.0 OTG FS
1x SDMMC
4x USART + 2 UART LIN, smartcard, IrDA, modem control
2x SAI (Serial Audio Interface)
HDMI CEC
SPDIF input x4
Control
2x 16-bit motor-control PWM synchronized AC timer
10x 16-bit timers 2x 32-bit timers
True random number generator (RNG)
Up to 512-Kbyte Flash memory
128-Kbyte SRAM
External memory interface W/SDRAM support
80-byte + 4-Kbyte backup data
512 OTP bytes
Dual Quad SPI
Analog
2-channel 2x 12-bit DAC
Up to 3x 12-bit ADC 2.4 MSPS Up to 24 channels 7.2 MSPS
Temperature sensor

Appendix C: Pin Connections on Nucleo-64 board

This diagram is common for both Nucleo-L476RG and Nucleo-L446RE

Board Component	Microcontroller Pin	Comment
Green LED	PA 5	SB42 closed and SB29 open by default
Blue user button	PC 13	Pulled up externally
Black reset button	NRST	Connect to ground to reset
ST-Link UART TX	PA 2	STLK_TX
ST-Link UART RX	PA 3	STLK_RX
ST-Link SWO/TDO	PB 3	Trace output pin/Test Data Out pin
ST-Link SWDIO/TMS	PA 13	Data I/O pin/Test Mode State pin
ST-Link SWDCLK/TCK	PA 14	Clock pin/Test Clock pin



Appendix D: Clock Configuration

There are two major types of clocks: **system clock** and **peripheral clock**. A video tutorial is given here: https://youtu.be/o6ZWD0PAoJk

- ***System Clock***: To meet the requirement of performance and energy-efficiency for different applications, the processor core can be driven by four different clock sources, including ***HSI*** (high-speed internal) oscillator clock, ***HSE*** (high-speed external) oscillator clock, **PLL** clock, and **LSI (**low-speed internal and LSE (low-speed external) oscillator clock. A faster clock provides better performance but usually consumes more power, which is not appropriate for battery-powered systems.

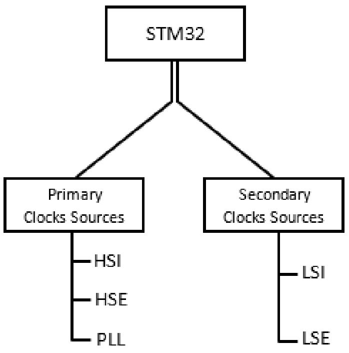


Figure-2: STM32F4xx Clock sources

- ***Peripheral Clock***: All peripherals require to be clocked to function. However, ***clocks of all peripherals are turned off by default to reduce power consumption***.

Overall the primary clock sources on STM32F4xx selection and distribution is shown in bellow figure.

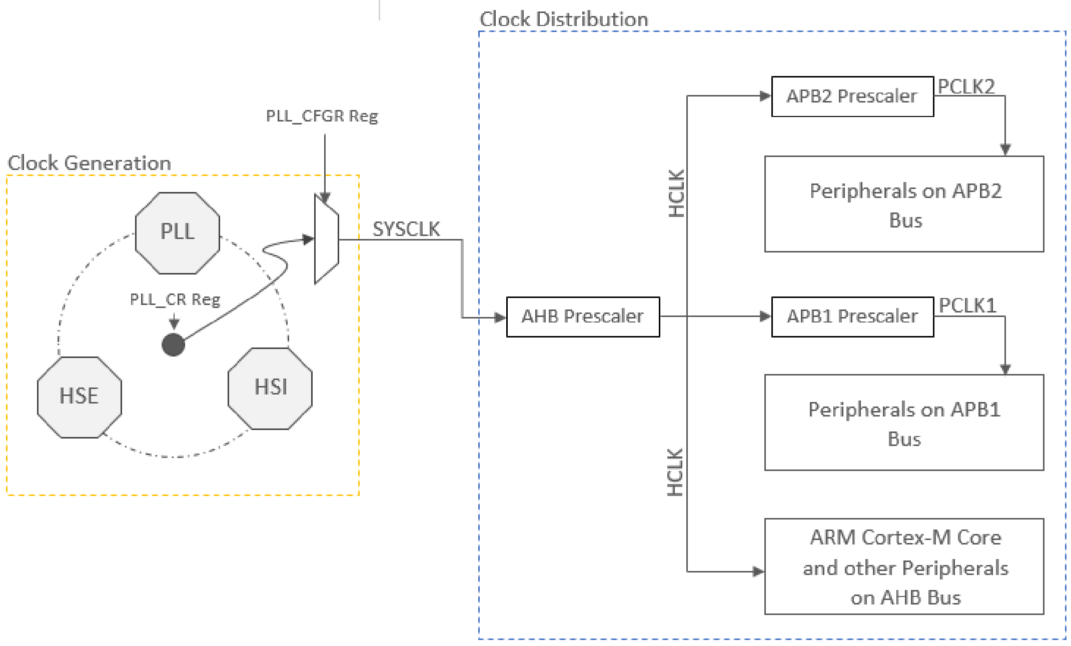
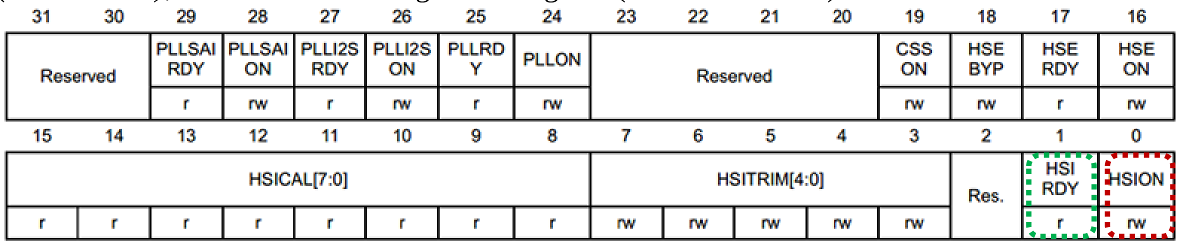


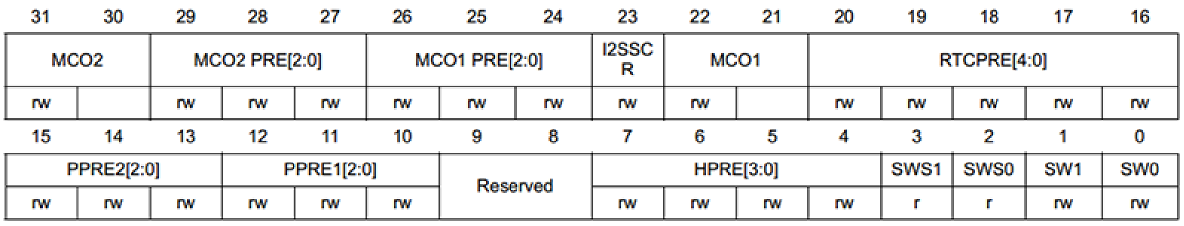
Figure-D1: STM32F4xx Primary clock selection and distribution

In STM32 any of three primary clock sources can be selected/deselected and switched via three RCC Registers i.e. *RCC clock control register (RCC_CR)* – Figure-4, *RCC clock configuration register (RCC_CFGR)*, and RCC PLL configuration register (RCC_PLLCFGR).



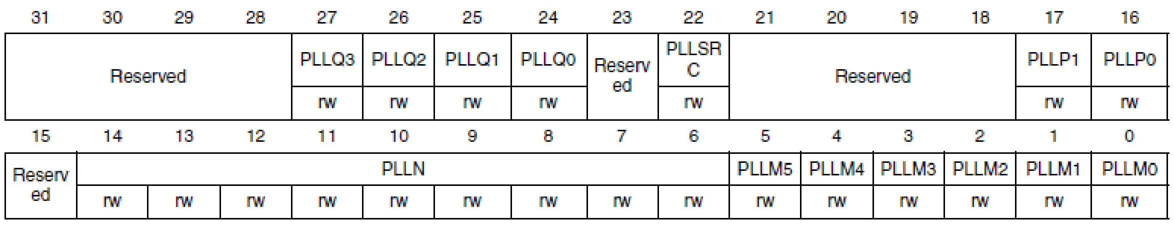
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		PLLSAI RDY	PLLSAI ON	PLLI2S RDY	PLLI2S ON	PLLRD Y	PLLON	Reserved				CSS ON	HSE BYP	HSE RDY	HSE ON
		r	rw	r	rw	r	rw					rw	rw	r	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
HSICAL[7:0]								HSITRIM[4:0]					Res.	HSI RDY	HSION
r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw		r	rw

Figure-D2: STM32 RCC clock control register (RCC_CR)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MCO2		MCO2 PRE[2:0]			MCO1 PRE[2:0]			I2SSC R	MCO1		RTCPRE[4:0]				
rw		rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PPRE2[2:0]			PPRE1[2:0]			Reserved		HPRE[3:0]				SWS1	SWS0	SW1	SW0
rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	r	r	rw	rw

Figure-D3: STM32 RCC clock configuration register (RCC_CFGR)

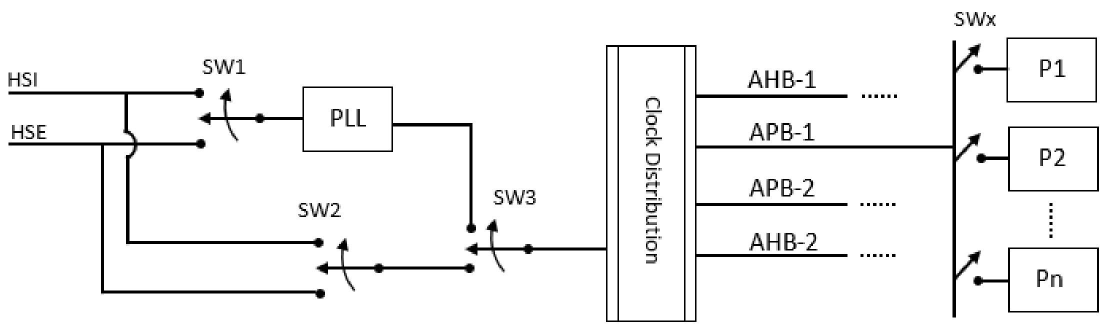


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLLQ3	PLLQ2	PLLQ1	PLLQ0	Reserv ed	PLLSR C	Reserved				PLLP1	PLLP0
				rw	rw	rw	rw		rw					rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserv ed	PLLN									PLLM5	PLLM4	PLLM3	PLLM2	PLLM1	PLLM0
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure-D4: STM32 RCC PLL configuration register (RCC_PLLCFGR)

Following are the steps to select *High Speed Internal (HSI)* as System Clock.

1. Turn ON HSI Oscillator (bit-0, RCC-CR, **HSION**)
2. Wait until the clock get stable (bit-1, RCC-CR, **HSIRDY**)
3. Switch System clock to HSI (bit-0,1 in RCC_CFGR; see Figure-7 SW1,SW2)



- SWx = Switch-X
- Pn = Peripheral-n

Figure-D5: STM32F4xx RCC functional diagram

PLL Configuration

STM32F uses PLLs to take the input clock and adjust/multiply it and emit on an output pin. It acts as a input frequency amplifier.

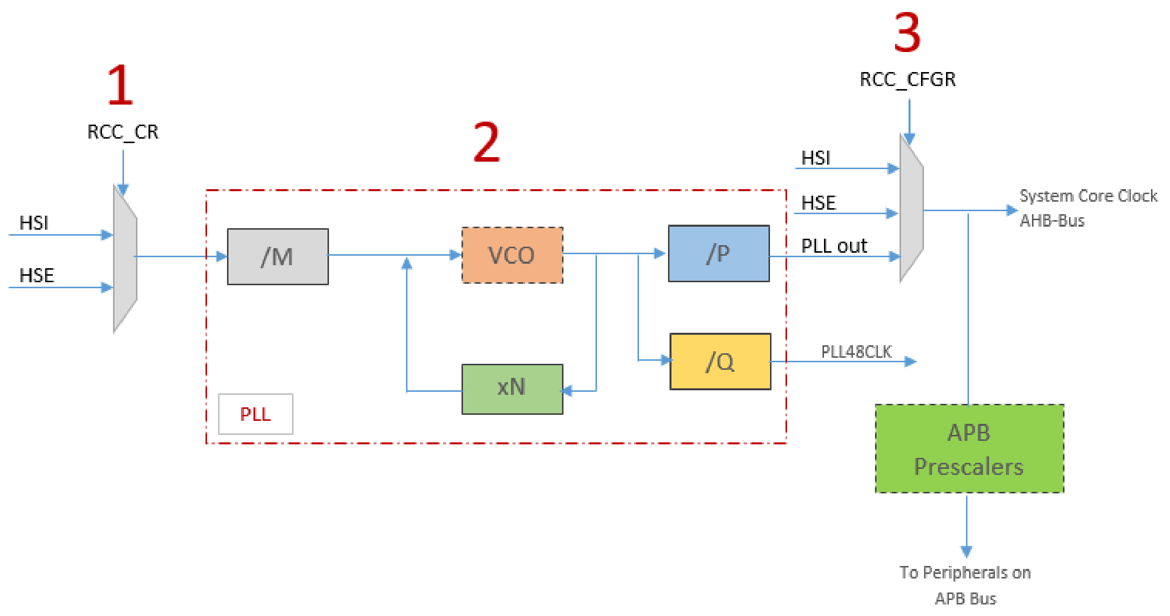


Figure-D6: STM32F4xx main PLL functional diagram

1. PLL input Clock Source: As mentioned earlier, PLL is not a clock source itself, it takes input clock from a clock source, adjust/multiply it, and emit it on output pin. It acts like an amplifier that amplifies input frequency. So before we can use PLL, we must feed it from some clock source – Figure-8.

2. Configuring PLL: The next step is to configure PLL main Logic as shown in Figure-8. The PLL output frequency is calculated as:

$$f_{(\text{VCO clock})} = \frac{f_{(\text{PLL clock input})} \times \text{PLLN}}{\text{PLLM}}$$

$$f_{(\text{PLL clock output})} = \frac{f_{(\text{VCO clock})}}{\text{PLLP}}$$

$$f_{(\text{USB OTG FS, SDIO, RNG Clock output})} = \frac{f_{(\text{VCO clock})}}{\text{PLLQ}}$$

While configuring PLL, following conditions need to meet otherwise PLL may not lock.
Reserved bits must be kept at their default values in PLL registers.

PLL bits can be written only when PLL is disabled.

VCO input frequency ranges from 1 to 2 MHz. It is recommended to select a frequency of 2 MHz to limit PLL jitter.

VCO output frequency must be in range 192 to 432 MHz.

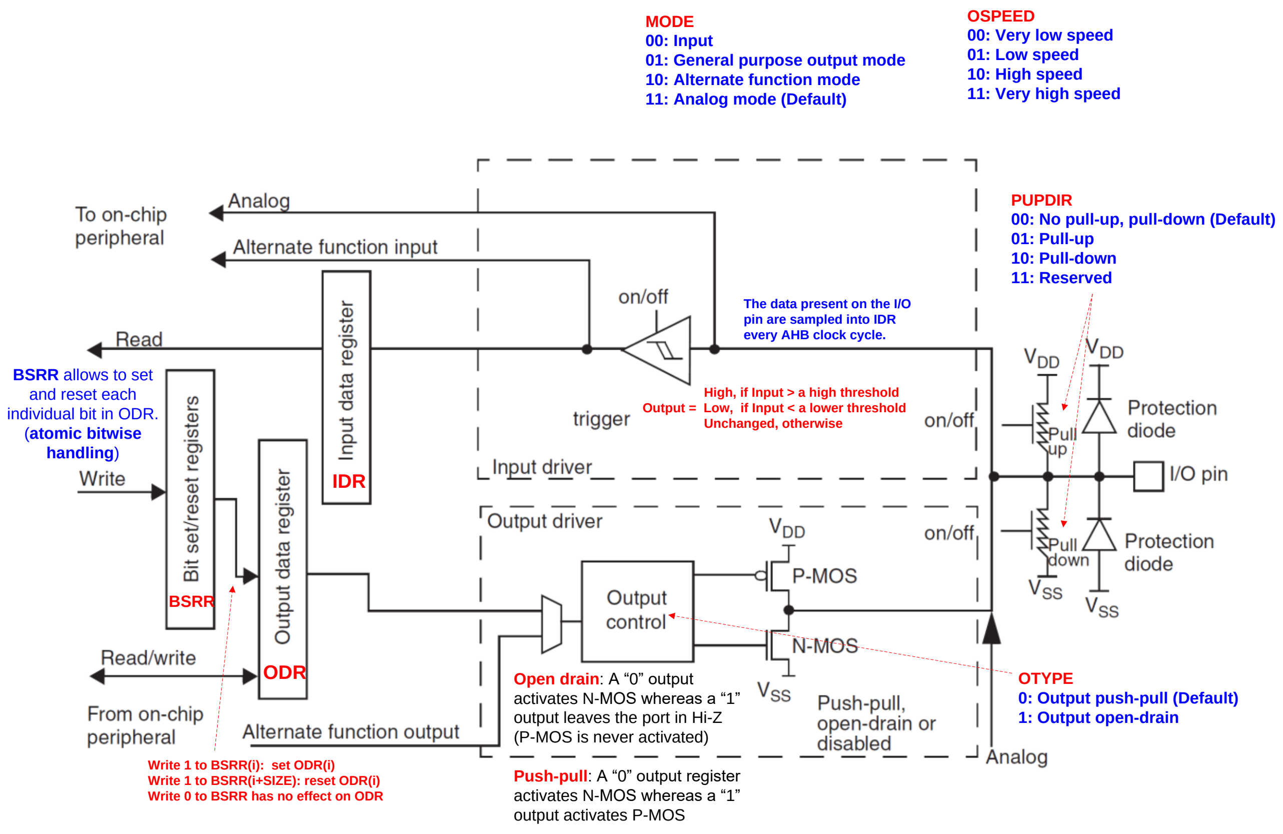
3. Switching to PLL: The last step is to switch system to PLL clock.

The software provided in this lab uses the 16MHz HSI as the input to the PLL clock. Appropriate scaling factors have been selected to achieve the maximum allowed clock speed (80 MHz). See the function void **System_Clock_Init**() for details.

```
static void enable_HSI(){

        /* Enable Power Control clock */
        /* RCC->APB1ENR |= RCC_APB1LPENR_PWRLPEN; */

        // Regulator voltage scaling output selection: Scale 2
        // PWR->CR |= PWR_CR_VOS_1;

        // Enable High Speed Internal Clock (HSI = 16 MHz)
        RCC->CR |= ((uint32_t)RCC_CR_HSION);
        while ((RCC->CR & RCC_CR_HSIRDY) == 0); // Wait until HSI ready

        // Store calibration value
        PWR->CR |= (uint32_t)(16 << 3);

        // Reset CFGR register
        RCC->CFGR = 0x00000000;

        // Reset HSEON, CSSON and PLLON bits
        RCC->CR &= ~(RCC_CR_HSEON | RCC_CR_CSSON | RCC_CR_PLLON);
        while ((RCC->CR & RCC_CR_PLLRDY) != 0); // Wait until PLL disabled

        // Programming PLLCFGR register
        // RCC->PLLCFGR = 0x24003010; // This is the default value

        // Tip:
        // Recommended to set VOC Input f(PLL clock input) / PLLM to 1-2MHz
        // Set VCO output between 192 and 432 MHz,
        // f(VCO clock) = f(PLL clock input) × (PLLN / PLLM)
        // f(PLL general clock output) = f(VCO clock) / PLLP
        // f(USB OTG FS, SDIO, RNG clock output) = f(VCO clock) / PLLQ

        RCC->PLLCFGR = 0;
        RCC->PLLCFGR &= ~(RCC_PLLCFGR_PLLSRC);                  // PLLSRC = 0 (HSI 16 Mhz clock
selected as clock source)
        RCC->PLLCFGR |= 16 << RCC_PLLCFGR_PLLN_Pos;     // PLLM = 16, VCO input clock = 16 MHz /
PLLM = 1 MHz
        RCC->PLLCFGR |= 336 << RCC_PLLCFGR_PLLN_Pos;  // PLLN = 336, VCO output clock = 1 MHz *
336 = 336 MHz
        RCC->PLLCFGR |= 4 << RCC_PLLCFGR_PLLP_Pos;      // PLLP = 4, PLLCLK = 336 Mhz / PLLP = 84
MHz
        RCC->PLLCFGR |= 7 << RCC_PLLCFGR_PLLQ_Pos;      // PLLQ = 7, USB Clock = 336 MHz / PLLQ =
48 MHz

        // Enable Main PLL Clock
        RCC->CR |= RCC_CR_PLLON;
        while ((RCC->CR & RCC_CR_PLLRDY) == 0);  // Wait until PLL ready

        // FLASH configuration block
        // enable instruction cache, enable prefetch, set latency to 2WS (3 CPU cycles)
        FLASH->ACR |= FLASH_ACR_ICEN | FLASH_ACR_PRFTEN | FLASH_ACR_LATENCY_2WS;

        // Configure the HCLK, PCLK1 and PCLK2 clocks dividers
        // AHB clock division factor


        RCC->CFGR &= ~RCC_CFGR_HPRE; // 84 MHz, not divided
        // PPRE1: APB Low speed prescaler (APB1)
        RCC->CFGR &= ~RCC_CFGR_PPRE1;
        RCC->CFGR |= RCC_CFGR_PPRE1_DIV2; // 42 MHz, divided by 2
        // PPRE2: APB high-speed prescaler (APB2)
        RCC->CFGR &= ~RCC_CFGR_PPRE2; // 84 MHz, not divided

        // Select PLL as system clock source
        // 00: HSI oscillator selected as system clock
        // 01: HSE oscillator selected as system clock
        // 10: PLL selected as system clock
        RCC->CFGR &= ~RCC_CFGR_SW;
        RCC->CFGR |= RCC_CFGR_SW_1;
        // while ((RCC->CFGR & RCC_CFGR_SWS_PLL) != RCC_CFGR_SWS_PLL);

        // Configure the Vector Table location add offset address
//      VECT_TAB_OFFSET  = 0x00UL; // Vector Table base offset field.
                // This value must be a multiple of 0x200.
        SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; // Vector Table Relocation in Internal FLASH

}
```

See this tutorial for a comprehensive guide on how to set different clock sources: https://ecoderlenz.com/?p=830

Appendix E: General Purpose I/O


MODE
00: Input
01: General purpose output mode
10: Alternate function mode
11: Analog mode (Default)
OSPEED
00: Very low speed
01: Low speed
10: High speed
11: Very high speed
To on-chip peripheral
Analog
Alternate function input
Read
Input data register
IDR
on/off
trigger
The data present on the I/O pin are sampled into IDR every AHB clock cycle.
Output = High, if Input > a high threshold
Low, if Input < a lower threshold
Unchanged, otherwise
on/off
Input driver
PUPDIR
00: No pull-up, pull-down (Default)
01: Pull-up
10: Pull-down
11: Reserved
V_DD
V_DD
Pull up
Protection diode
I/O pin
Pull down
Protection diode
V_SS
V_SS
BSRR allows to set and reset each individual bit in ODR. (atomic bitwise handling)
Write
Bit set/reset registers
BSRR
Output data register
ODR
Read/write
From on-chip peripheral
Write 1 to BSRR(i): set ODR(i)
Write 1 to BSRR(i+SIZE): reset ODR(i)
Write 0 to BSRR has no effect on ODR
Output driver
V_DD
on/off
Output control
P-MOS
N-MOS
V_SS
Alternate function output
Push-pull, open-drain or disabled
Analog
Open drain: A "0" output activates N-MOS whereas a "1" output leaves the port in Hi-Z (P-MOS is never activated)
Push-pull: A "0" output register activates N-MOS whereas a "1" output activates P-MOS
OTYPE
0: Output push-pull (Default)
1: Output open-drain

Appendix F: Code Comments and Documentation

Program comments are used to improve code readability, and to assist in debugging and maintenance. A general principal is "***Structure and document your program the way you wish other programmers would***" (McCann, 1997).

The book titled "*The Elements of Programming Style*" by Brian Kernighan and P. J. Plauger gives good advices for beginners.

- **Format your code well.** Make sure it's easy to read and understand. Comment where needed but don't comment obvious things it makes the code harder to read. If editing someone else's code, format consistently with the original author.

- Every program you write that you intend to keep around for more than a couple of hours ought to have documentation in it. Don't talk yourself into putting off the documentation. A program that is perfectly clear today is clear only because you just wrote it. Put it away for a few months, and it will most likely take you a while to figure out what it does and how it does it. If it takes you a while to figure it out, how long would it take someone else to figure it out?

- **Write Clearly** - don't be too clever - don't sacrifice clarity for efficiency.

- **Don't over comment.** Use comments only when necessary.

- Format a program to help the reader understand it. **Always Beautify Code.**

- Say what you mean, **simply and directly**.

- **Don't patch bad code** - rewrite it.

- **Make sure comments and code agree.**

- Don't just echo code in comments **- make every comment meaningful.**

- **Don't comment bad code** - rewrite it.

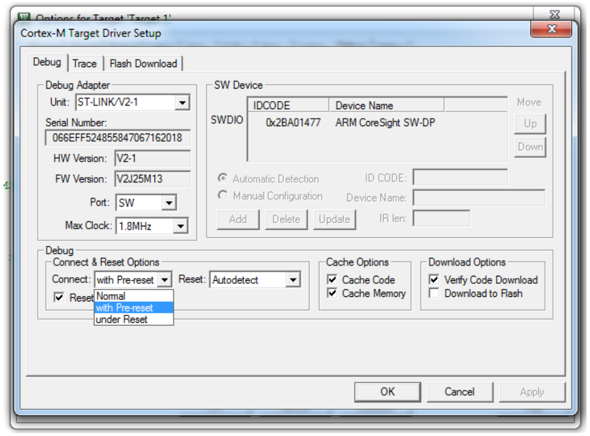
- **The single most important factor in style is consistency.** The eye is drawn to something that "doesn't fit," and these should be reserved for things that are actually different.

Appendix G: “Target not found” error

When you use the STM32F4 discovery board at the very first time, you might not be able to program it in Keil and receive an error of “***Target not found***” when you download the code to the board. This is because the demo program quickly puts the STM32F4 microcontroller into a very low power mode after a reset. There are several ways to solve it. Below are two simplest ones. The error will go away permanently.

Solution 1: In Keil,

1. Click the icon "***Options for Target***"
2. Click "***Debug***" and then "***Settings***"
3. Change the connect from the default value "***normal***" to "***with pre-reset***", as shown below.


Cortex-M Target Driver Setup

Debug Adapter
Unit: ST-LINK/V2-1
Serial Number: 066EFF524855847067162018
HW Version: V2-1
FW Version: V2J25M13
Port: SW
Max Clock: 1.8MHz

SW Device
SWDIO 0x2BA01477 ARM CoreSight SW-DP

Debug — Connect & Reset Options
Connect: with Pre-reset
Reset: Autodetect
☑ Reset
Normal / with Pre-reset / under Reset

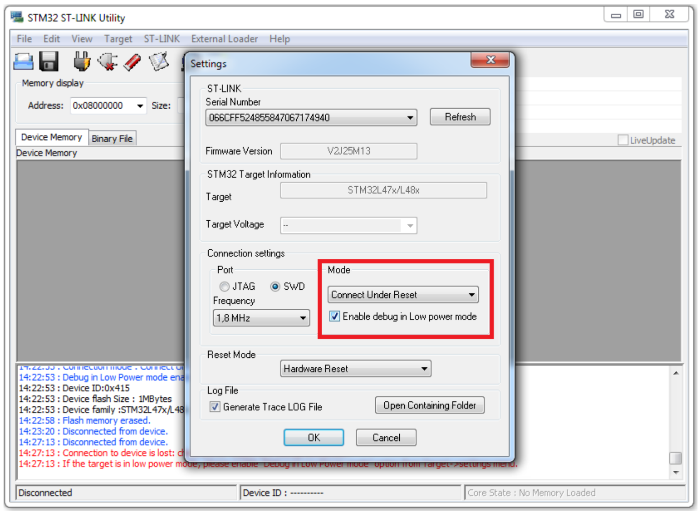
Cache Options
☑ Cache Code
☑ Cache Memory

Download Options
☑ Verify Code Download
☐ Download to Flash

Solution 2: If the previous solution fails, you can download and install ***STM32 ST-Link Utility***
http://www.st.com/en/development-tools/stsw-link004.html

Follow the following steps:

1. Run ST-Link Utility, click menu "***Target***”, click "***Settings***"
2. Select "***Connect Under Set***" as the connection mode
3. Click "***Target***" and "***Connect***", and then click "***Target***" and "***Erase Chip***"!
4. Click "***Target***" and "***Disconnect***"


STM32 ST-LINK Utility

Settings
ST-LINK
Serial Number: 066CFF524855847067174940
Firmware Version: V2J25M13
STM32 Target Information
Target: STM32L47x/L48x
Target Voltage: --
Connection settings
Port: ○ JTAG ● SWD
Frequency: 1,8 MHz
Mode: Connect Under Reset
☑ Enable debug in Low power mode
Reset Mode: Hardware Reset
Log File: ☑ Generate Trace LOG File

Memory display — Address: 0x08000000

```
14:22:53 : Debug in Low Power mode ena
14:22:53 : Device ID:0x415
14:22:53 : Device flash Size : 1MBytes
14:22:53 : Device family :STM32L47x/L48
14:22:58 : Flash memory erased.
14:23:20 : Disconnected from device.
14:27:13 : Disconnected from device.
14:27:13 : Connection to device is lost: c
14:27:13 : If the target is in low power mode, please enable "Debug in Low Power mode" option from Target->settings menu.
```

Disconnected | Device ID : ---------- | Core State : No Memory Loaded

Appendix H: Acknowledgement and References

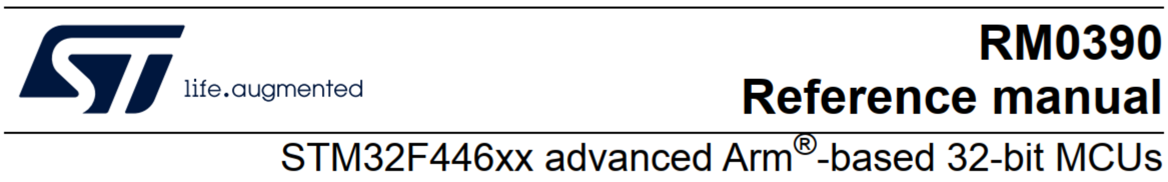
The labsheet is prepared by **Dr. Sajid Muhaimin Choudhury**, Dept of EEE, BUET, on 23/06/2022

The labsheet is based on Lab Materials provided as Instuctor Suppliment of "Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, Third Edition." By Dr. Yifeng Zhu, Arm Documentation. Materials used are copyrighted to Dr. Zhu. The Labsheets are modified from STM32L4 architecture to STM32F4 architecture. Texts has been updated and difference between the architectures are highlighted.

Clock Generation Tutorial is taken from STM32F4xx Clock Sources (HSI, HSE, PLL)
https://ecoderlenz.com/?p=830

STM32F4 Documentations are taken from st.com

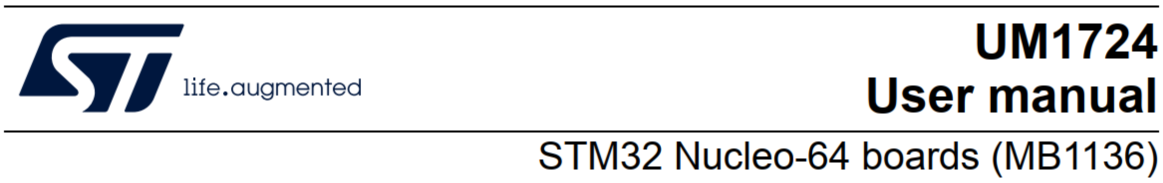
The following documents are strongly recommended as reference:


life.augmented

RM0390
Reference manual

STM32F446xx advanced Arm®-based 32-bit MCUs

https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

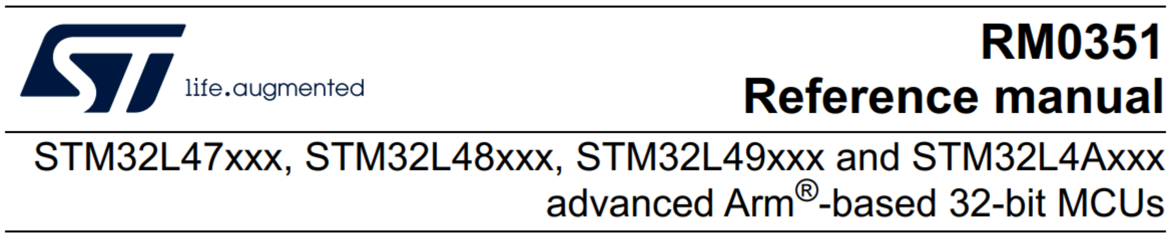

life.augmented

UM1724
User manual

STM32 Nucleo-64 boards (MB1136)

https://www.st.com/resource/en/user_manual/dm00105823-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

The following Document is recommended for Yifeng Zhu's book literature


life.augmented

RM0351
Reference manual

STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs

https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf