

Embedded Systems Laboratory Manual

ENCS4101

Shadi Abdelaleem Daana

August 26, 2026

Contents

1	Experiment 1: GPIOs (General Purpose Input/Output), Switch De-bouncing, and Interrupt	1
1.1	Objectives	1
1.2	Pre-Lab	1
1.3	Theory	1
1.3.1	Configuring GPIO Pins	1
1.3.2	Pin Characteristic	2
1.3.2.1	Input mode configuration	2
1.3.2.2	Output mode configuration	2
1.3.3	Switch Bouncing	3
1.3.4	Interrupts	4
1.4	Materials and Equipment	5
1.5	Relevant HAL APIs	5
1.6	Activity	6
1.7	Procedures	6
1.7.1	Hardware Setup:	6
1.7.2	Peripheral Configuration:	6
1.7.3	Software Code	10
1.7.4	After Completing the Lab Activity:	11
1.8	Task	11
2	Experiment 2: LCD, I2C and Keypad	12
2.1	Objectives	12
2.2	Pre-Lab	12
2.3	Theory	12
2.3.1	LCD parallel connection diagram	13
2.3.2	LCD serial interface	14
2.3.3	I2C protocol	15
2.3.4	Interface the 1602 LCD with STM32 microcontroller via I2C	16
2.3.5	Library support	16
2.3.6	Keypad	17
2.3.7	How the Keypad Works:	17
2.4	Materials and Equipment	18
2.5	Relevant External Library Functions	19
2.6	Activity	19
2.7	Procedures	20
2.7.1	Hardware Setup:	20
2.7.2	Peripheral Configuration:	20
2.7.3	Software Code	21
2.8	Tasks	24
2.8.1	Some notes about the firmware code:	25
2.8.2	Post Lab Questions	25

3	Experiment 3: ADC and Analog Sensor Interfacing	26
3.1	Objectives	26
3.2	Pre-Lab	26
3.3	Theory	26
3.3.1	ADC Basics	26
3.3.2	STM32F4 ADC Peripheral	28
3.3.3	ADC Conversion Modes	29
3.3.4	Channel Selection	29
3.3.5	Temperature sensor, V_{REFINT} and V_{BAT} internal channels	30
3.3.6	Light Dependent Resistor (LDR)	30
3.4	Materials and Equipment	31
3.5	Relevant HAL APIs	31
3.6	Activity	31
3.7	Procedure	31
3.7.1	Hardware Setup	31
3.7.2	Peripheral Configuration	32
3.7.3	Software Code	34
3.8	Some notes about the firmware code:	36
3.9	Task	36
4	Experiment 4: Timer Configuration and Output Compare Control	37
4.1	Objectives	37
4.2	Pre-Lab	37
4.3	Theory	37
4.3.1	How does timer work?	38
4.3.2	STM32F4 Timers	38
4.4	Materials and Equipment	39
4.5	Relevant HAL APIs	40
4.6	Activity	41
4.7	Procedure	41
4.7.1	Hardware Setup	41
4.7.2	Peripheral Configuration	41
4.7.3	Software Code	43
4.7.4	After Completing the Lab Activity:	47
4.8	Task	47
5	Experiment 5: DC Motor Control and PWM	48
5.1	Objectives	48
5.2	Pre-Lab	48
5.3	Theory	48
5.3.1	Motor control	49
5.4	Materials and Equipment	51
5.5	Relevant HAL APIs	51
5.6	Activity	51
5.7	Procedure	51
5.7.1	Hardware Setup	51
5.7.2	Peripheral Configuration	52
5.7.3	Software Code	53

5.7.4	After Completing the Lab Activity:	55
5.8	Task	55
6	Experiment 6: Rotary Encoder Interface	56
6.1	Objectives	56
6.2	Pre-Lab	56
6.3	Theory	56
6.3.1	Incremental encoder	57
6.3.2	STM32 Encoder Mode	57
6.3.3	Working Principle	57
6.3.4	Angle Measurement	58
6.3.5	Speed Measurement	58
6.3.6	HW-040 rotary encoder module	58
6.4	Materials and Equipment	59
6.5	Relevant HAL APIs	59
6.6	Activity	59
6.7	Procedure	60
6.7.1	Hardware Setup	60
6.7.2	Peripheral Configuration	60
6.7.3	Software Code	63
6.8	Task	64
6.9	After Completing the Lab Activity:	64
7	Experiment 7: Stepper Motor Control	65
7.1	Objectives	65
7.2	Pre-Lab	65
7.3	Theory	65
7.3.1	Stepper Motor Working Principle	65
7.3.2	Stepper Motor Working Principle	66
7.3.3	Stepper Motor Working Principle	66
7.3.4	28BYJ-48 stepper motor	67
7.3.5	Drive Circuitry	68
7.4	Materials and Equipment	69
7.5	Relevant External Library Functions	69
7.6	Activity	69
7.7	Procedure	69
7.7.1	Hardware Setup	69
7.7.2	Peripheral Configuration	70
7.7.3	Software Code	71
7.8	Task	71
7.9	After Completing the Lab Activity:	71
8	Experiment 8: UART Protocol	72
8.1	Objectives	72
8.2	Pre-Lab	72
8.3	Theory	72
8.4	Materials and Equipment	74
8.5	Relevant External Library Functions	74
8.6	Activity	74

8.7	Procedure	74
8.7.1	Hardware Setup	74
8.7.2	Peripheral Configuration	75
8.7.3	Software Code	76
8.8	After Completing the Lab Activity:	77
9	Experiment 9: Bare Metal Programming	78
9.1	Objectives	78
9.2	Pre-Lab	78
9.3	Theory	78
9.3.1	Bare Metal Programming	78
9.3.2	GPIO Configuration Using Bare Metal Programming	78
9.3.3	Debugging	80
9.3.3.1	Hardware Debugger	80
9.3.3.2	Software Debugger Tools	80
9.3.3.3	Debug View	80
9.3.3.4	Debug Toolbar	83
9.3.4	What is a 7-Segment Display?	84
9.3.5	7-Segment Displays Configurations	84
9.3.6	7-Segment Truth Table	85
9.4	Materials and Equipment	85
9.5	Activity	85
9.5.1	Hardware Setup	85
9.5.2	Software Setup	86
9.6	Required Materials from Reference Manual (STM32F407)	88
9.7	Sample Program	89

1. Experiment 1: GPIOs (General Purpose Input/Output), Switch Debouncing, and Interrupt

1.1 Objectives

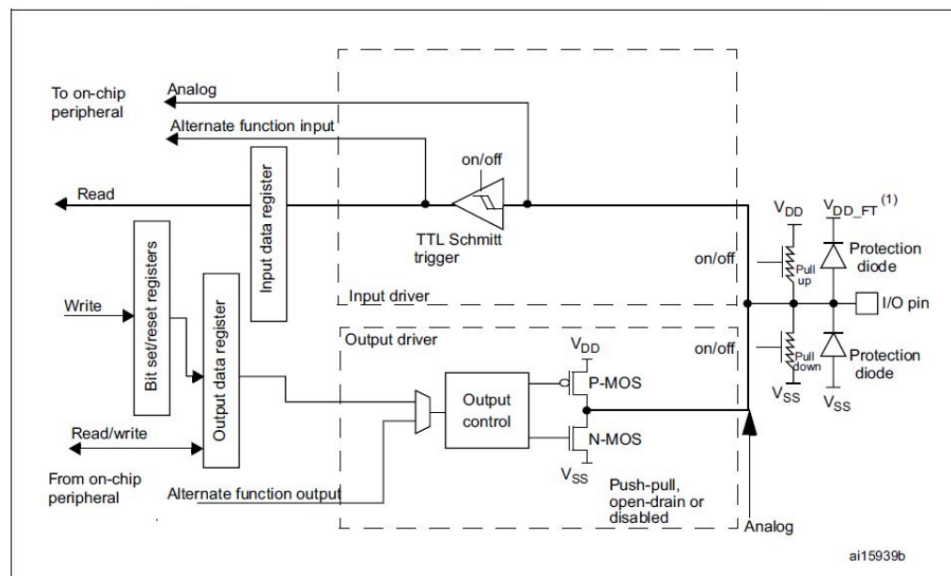
This lab aims to familiarize students with the fundamental concepts and practical skills of embedded systems development. Students will learn how to configure GPIO pins for both input and output operations using the STM32CubeIDE configuration tool. They will create applications using the HAL (Hardware Abstraction Layer) library, understand interrupt handling mechanisms, and implement SW debouncing techniques to ensure reliable button press detection.

1.2 Pre-Lab

- Review GPIO and interrupt concepts.
- How can you solve bouncing in mechanical switches?
- Try to understand the code and the configurations of GPIO pins using STM32CubeIDE.

1.3 Theory

The basic structure of a GPIO pin is illustrated in the following figure: it consists of inner circuitries that can be shaped based on the user configurations written to the GPIO registers.



1. V_{DD_FT} is a potential specific to five-volt tolerant I/Os and different from V_{DD} .

Figure 1.1: Basic structure of a five-volt tolerant I/O port bit

1.3.1 Configuring GPIO Pins

Each pin of the general-purpose I/O (GPIO) port can be individually configured by software in several modes:

- Digital Input
- Digital Output
- Analog
- Alternate function

1.3.2 Pin Characteristic

Each port bit of the general-purpose I/O (GPIO) ports can be individually configured by software in several modes

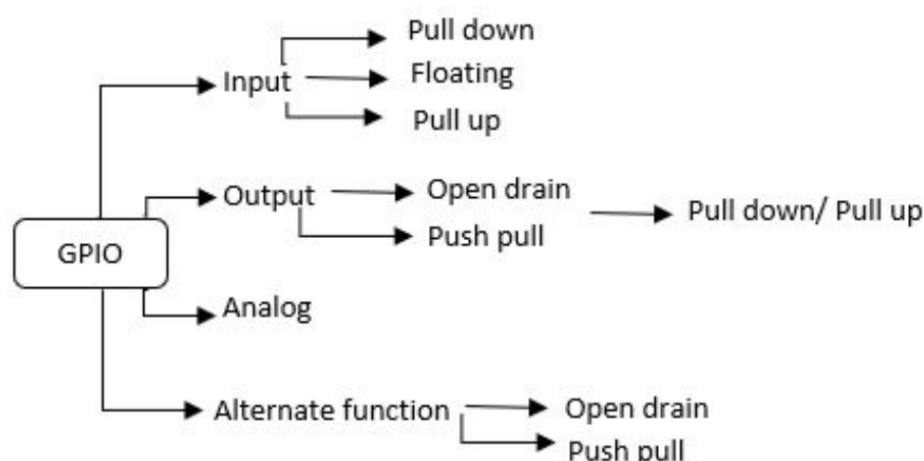


Figure 1.2: GPIO different modes and configurations

1.3.2.1 Input mode configuration

When a STM32 device I/O pin is configured as input, one of three options must be selected:

- Input with internal **pull-up**
- Input with internal **pull-down**
- **Floating** input: signal level follows the external signal. When no external signal is present, the Schmitt trigger randomly toggles between the logical levels induced by the external noise.

1.3.2.2 Output mode configuration

When a STM32 device I/O pin is configured as output, one of two options must be selected:

- **Push-pull** output mode: it uses two transistors, one PMOS and one NMOS. Each transistor is ON to drive the output to the appropriate level.
- **Open-drain** output mode: it does not use the PMOS transistor.
- **Alternate function**: push-pull or open-drain with pull-up or pull-down capability.

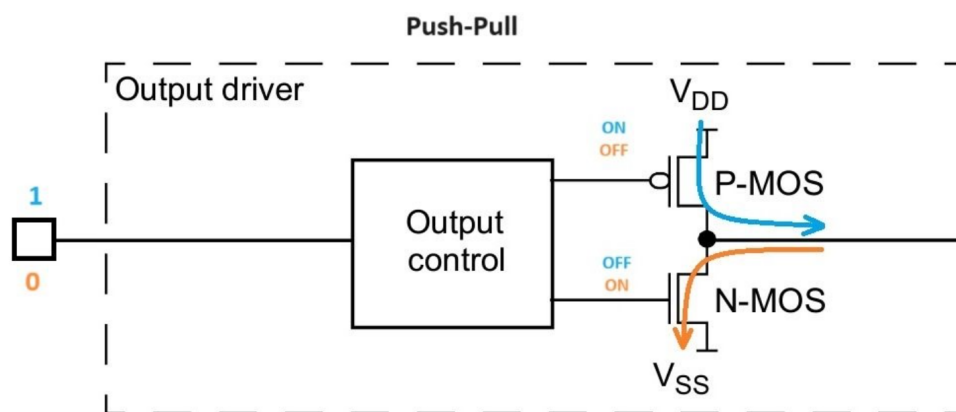


Figure 1.3: Push-Pull Setup

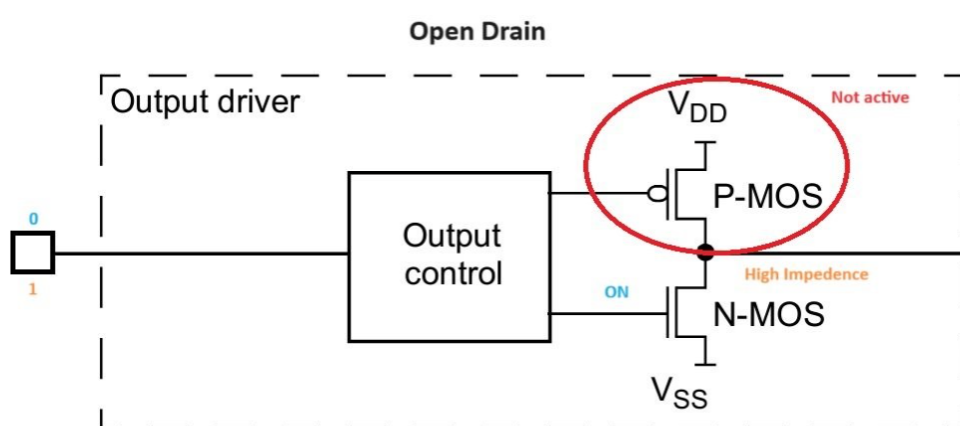


Figure 1.4: Open Drain Setup

1.3.3 Switch Bouncing

When we press a pushbutton or toggle switch or a micro switch, two metal parts come into contact to short the supply. But they don't connect instantly but the metal parts connect and disconnect several times before the actual stable connection is made (as shown in Figure 1.5). The same thing happens while releasing the button. This results the false triggering or multiple triggering like the button is pressed multiple times. It's like falling a bouncing ball from a height and it keeps bouncing on the surface, until it comes at rest.

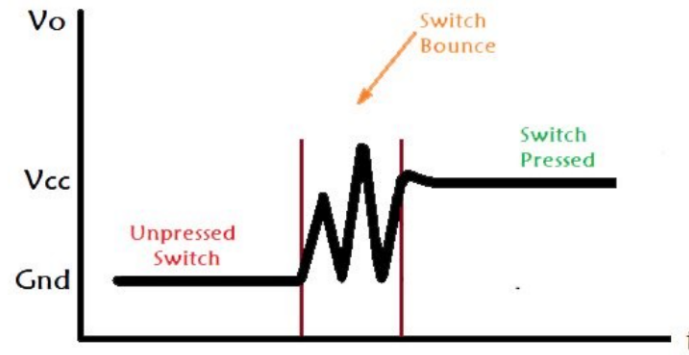


Figure 1.5: Switch bouncing

1.3.4 Interrupts

Interrupt is a mechanism that allows the processor to temporarily suspend its current execution of instructions and switch to a different set of instructions in response to a specific event or condition. Interrupts are usually generated from on-chip peripherals (e.g., a timer) or external inputs (e.g. a tactile switch connected to a GPIO), and in some cases they can be triggered by software

When a peripheral or hardware needs service from the processor, typically the following sequence would occur:

- When an interrupt occurs, the CPU suspends the execution of the current task and saves its context into the stack.
- The processor starts the execution of an Interrupt Service Routine (ISR) to handle the interrupting event.
- The processor resumes the previously suspended task.

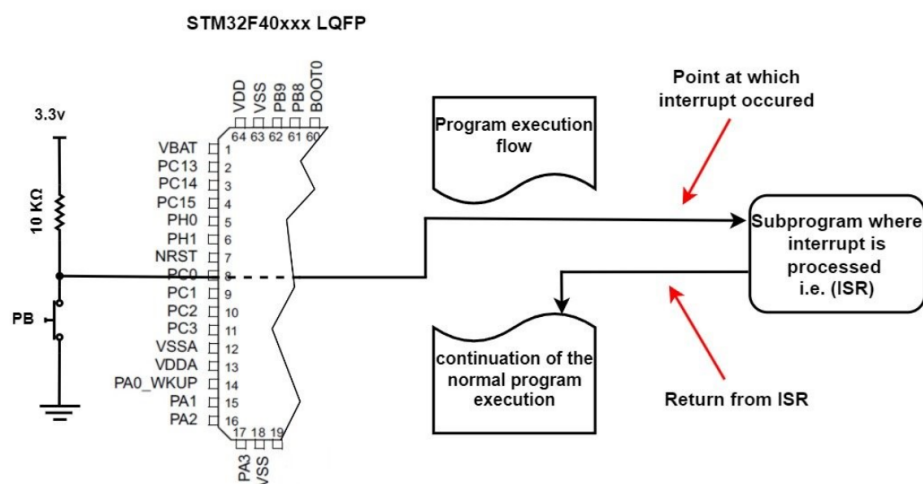


Figure 1.6: Interrupt

1.4 Materials and Equipment

- Breadboard
- STM32F4 evaluation board
- LEDs and push buttons
- Jumper Wires
- Resistors 5x(330 Ω), 1x (10k Ω)

1.5 Relevant HAL APIs

• IO Functions

```
/*Sets or clears the selected data port bit*/
HAL_GPIO_WritePin(GPIOx, GPIO_PIN_x, RESET);
HAL_GPIO_WritePin(GPIOx, GPIO_PIN_x, SET);

/*Toggles the specified GPIO pins*/
HAL_GPIO_TogglePin (GPIOx, GPIO_PIN_x);

/*Reads the specified input port pin*/
HAL_GPIO_ReadPin(GPIOx, GPIO_PIN_x);

/*Input arguments:
GPIOx : x can be (A,B, ..I)
GPIO_PIN_x: x can be (0 ..15)*/
```

- **EXTI line detection callbacks:** EXTI line detection callbacks are functions that get triggered when a specific External Interrupt (EXTI) line detects an event (like a rising or falling edge) on a GPIO pin.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == GPIO_PIN_x) {
        // Your custom code here
    }
}

/*GPIO_PIN_x : x can be (A,B, ..I)*/
```

• Others

```
HAL_GetTick() ;// This function gets current SysTick counter
value
HAL_Delay(100);// This function implements a delay (expressed in
milliseconds)
```

1.6 Activity

Design and implement a four-bit binary counter using LEDs and push buttons with an STM32 microcontroller. The counter starts from 0 (all LEDs off) and is incremented by 1 on every button press. After it reaches the maximum, it overflows and takes the next value 0. This repeats forever. Use interrupt and debouncing techniques to detect the button press.

1.7 Procedures

1.7.1 Hardware Setup:

1. Connect GPIO pins as shown in Figure 1.7 :
 - Connect the STM32 GPIO output pins to LEDs, each in series with a suitable resistor.
 - Connect a push button to a GPIO pin with a pull-down resistor to keep the input at a defined low level when the button is open.
- Note: If you use STM32F411 Nucleo board, avoid using the following pins: (PA2, PA3, PC14, PC15, PA13, and PA14).
- If you use STM32F407 DISC board, avoid using the following pins: (PA13 & PA14).

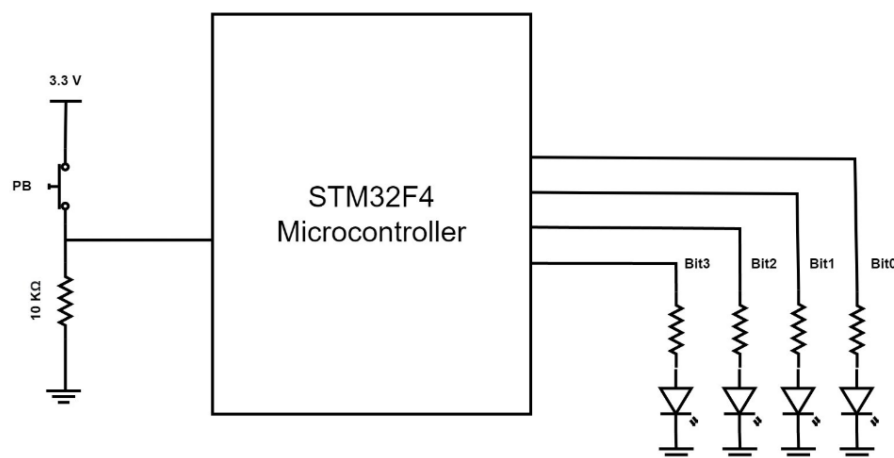


Figure 1.7: Schematic diagram

1.7.2 Peripheral Configuration:

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your microcontroller (STM32F407VGT6 or STM32F411RET6).
- When you create a project, make sure that you select “STM32Cube” in the “Targeted Project type” field.

2. RCC mode and Configuration:

- You can select the clock source from external crystal by enabling it from the HSE drop list as shown in Figure 1.8

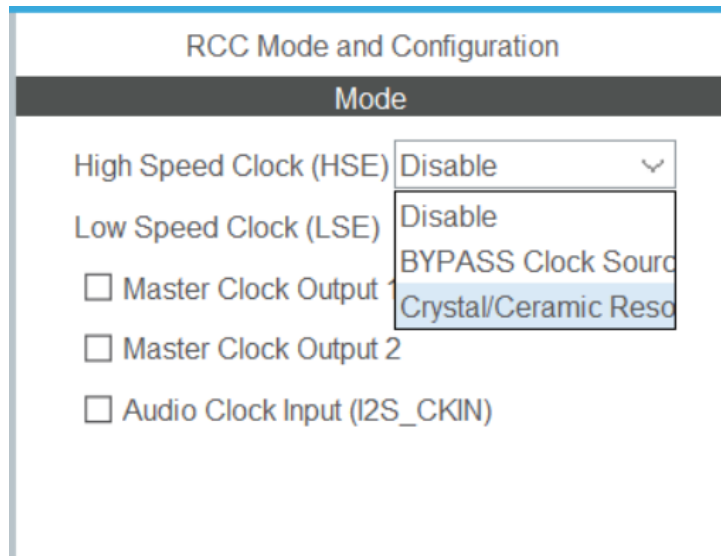


Figure 1.8: RCC mode and Configuration

3. Clock Configuration:

- You need to enter the frequency of the crystal used on the evaluation board in the Input frequency box (i.e. 8Mhz) based on the schematic of the evaluation board
- You can enter the maximum frequency inside the HCLK box
- Clock configuration window is shown in Figure 1.9

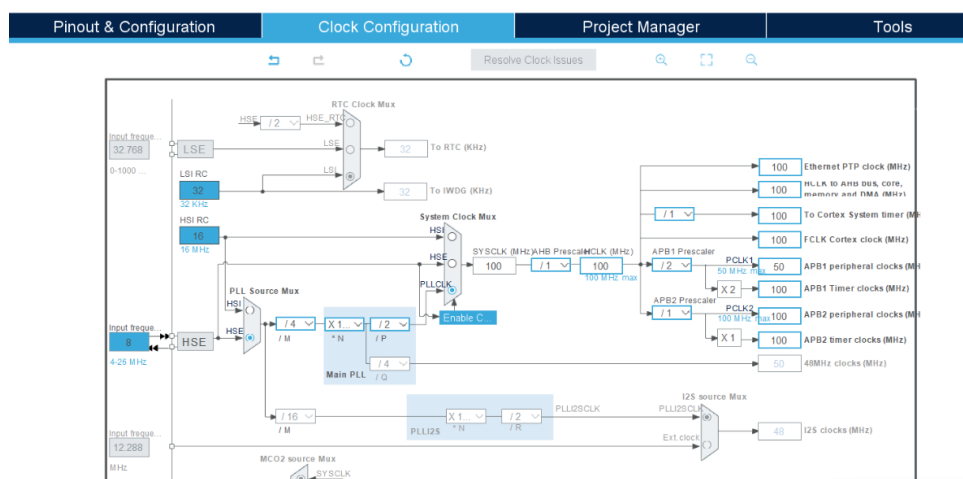


Figure 1.9: Clock Configuration

4. Configure GPIO Pins:

- In the "Pinout & Configuration" tab, configure the GPIO pins connected to buttons/switches as input and those connected to LEDs as output.
- In the pinout view as shown in Figure 1.10, you can assign the function for the required pins, you need to assign four pins as output to drive the 4-LED binary counter LEDs

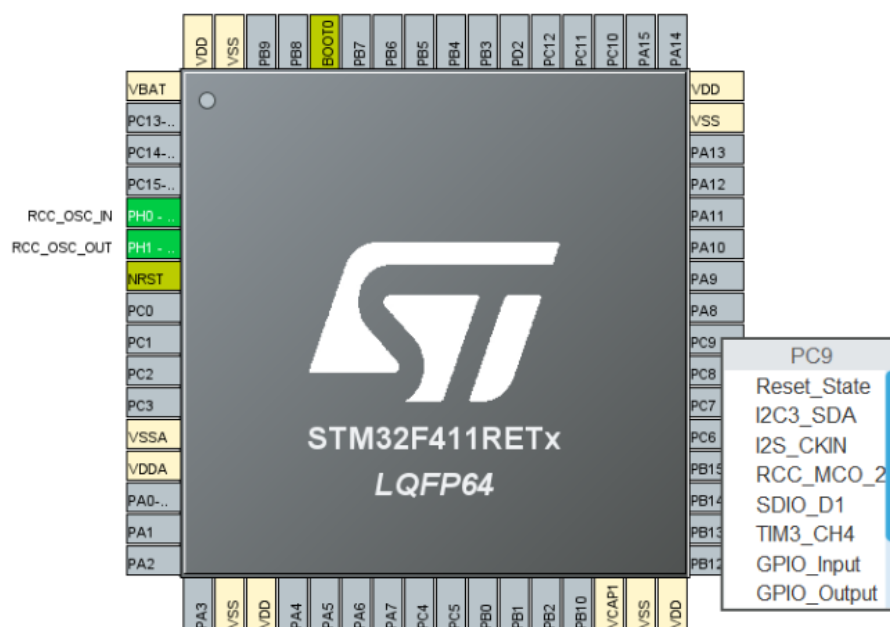


Figure 1.10: Pinout View

- In the configuration tab (Figure 1.11), you can select the GPIO configurations for each pin, you can also assign a **user label** to give custom name to the pins and peripherals in your project. It makes your project more readable and maintainable by assigning meaningful names to the pins, rather than using the default pin names (such as PA10, PB5, etc.)

GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	No pull-up and no pull-down
Maximum output speed	Low
User Label	Bit_0

Figure 1.11: GPIO Configuration

- In the pinout view (Figure 1.12), you can assign a pin as an external interrupt (GPIO_EXTIx) from the drop menu.
- Enable Interrupt in NVIC (Nested Vector Interrupt Controller):
 - On the **Configuration Panel**, click on the "NVIC Settings" (or "Configuration" tab under the "GPIO" section).

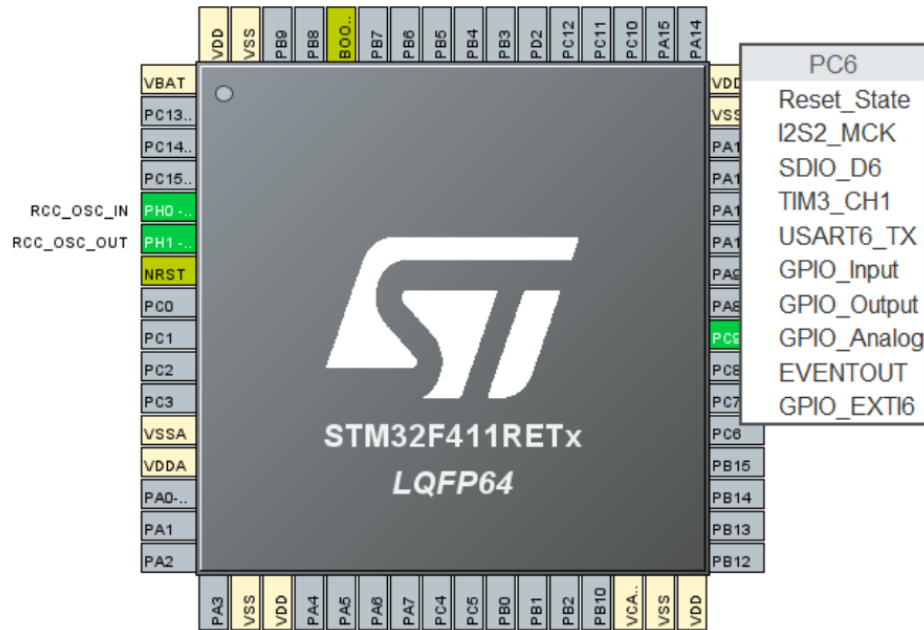


Figure 1.12: Pinout: EXTI

- Enable the interrupt for the corresponding EXTI line (e.g., **EXTI line 0**, **EXTI line 1**, etc.) by checking the box next to it (Figure 1.13). This ensures that the interrupt is active in the NVIC.

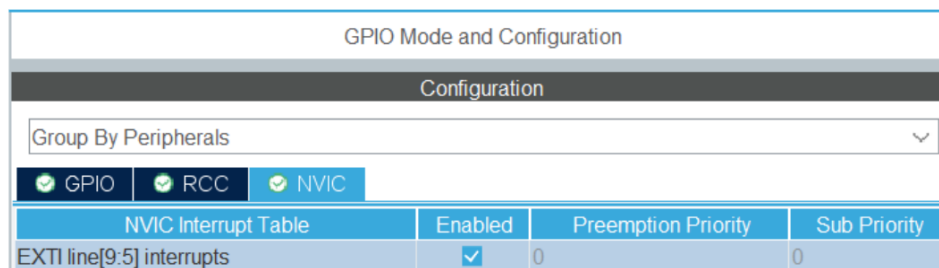


Figure 1.13: Interrupt Enabling

- Configure Edge Detection and enter the user label as shown in Figure 1.14.

PC6 Configuration :

GPIO mode: External Interrupt Mode with Rising edge trigger detection

GPIO Pull-up/Pull-down: No pull-up and no pull-down

User Label: EXTI_PB

Figure 1.14: Pinout: Pin Configuration

5. Generate code:

- click on the "**Project**" menu and select "**Generate Code**" to produce the necessary initialization and configurations.

1.7.3 Software Code

1. Interrupt Handling Code:

- Override the `HAL_GPIO_EXTI_Callback()` function to handle GPIO interrupt events. Debounce the button presses to avoid multiple triggering due to mechanical bouncing.
- You can utilize the `HAL_GetTick()` function provided by the HAL library to handle debouncing in your code. `HAL_GetTick()` returns the current system time in milliseconds since the MCU started running. By comparing timestamps of consecutive button presses or state changes and introducing a suitable delay or timeout, you can effectively debounce the input signals.

```
volatile uint16_t buttonPressCount=0;
uint32_t currentTick=0;
uint32_t previousTick=0;
uint32_t debounceTime =150;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == EXTI_PB_Pin) { //If the interrupt is
        from the specified pin
        currentTick = HAL_GetTick(); // Get current
            system tick
        if ((currentTick - previousTick) > debounceTime)
            { //Debounce check
                buttonPressCount++; // Increment the
                    counter
                if (buttonPressCount > 15)
                    buttonPressCount = 0; //Wrap around
                        count if it exceeds 15
            }
        previousTick = currentTick; //Update the
            previous tick for the next comparison
    }
}
```

- Answer the following questions about the code:
 - (a) Why are the variables defined as global?
 - (b) Explain the reason for using the `volatile` keyword.
 - (c) How can you use `currentTick` and `previousTick` inside the function without affecting the code?

2. Use functions to organize your code

- You can write a function that converts the decimal value of number of counts into their equivalent binary value and activate their corresponding GPIO output (Bit_0 to Bit_3).

```

void WriteBinary4Bit(uint16_t num){
    uint16_t bit0,bit1, bit2,bit3;
    bit0 = (num>>0)&1;
    bit1 = (num>>1)&1;
    bit2 = (num>>2)&1;
    bit3 = (num>>3)&1;
    HAL_GPIO_WritePin(Bit_0_GPIO_Port, Bit_0_Pin, bit0);
    HAL_GPIO_WritePin(Bit_1_GPIO_Port, Bit_1_Pin, bit1);
    HAL_GPIO_WritePin(Bit_2_GPIO_Port, Bit_2_Pin, bit2);
    HAL_GPIO_WritePin(Bit_3_GPIO_Port, Bit_3_Pin, bit3);
}

```

3. Program Code

In the main.c

- Define variables and EXTI callback function used on point 1. section 1.7.3.
- Add WriteBinary4Bit function definition and function prototype. Then use it in your code to drive the LEDs based on the number of button press.
- Build the project and flash it to your STM32 MCU.

1.7.4 After Completing the Lab Activity:

- Press the push-button and observe the corresponding LED behavior.
- Enter into **debug mode**:
 1. Add buttonPressCount to **Live Expression** and observe the changes.
 2. Add a break point inside the EXTI callback function and observe what happens when you press the button.
 3. Check the output data register (**ODR**) and input data register (**IDR**) in **SFRs view**.
- Change the value of debounceTime to (0, 50, 100, 200 &300) and observe the value of buttonPressCount.
- Answer the following questions:
 1. Which function does the ISR call when an external interrupt (EXTI) event occurs?
 2. How is the ISR (Interrupt Service Routine) called in the code?
 3. What happens if you configure the GPIO with pull-up resistor?
 4. What do ODR0 to ODR15 in the output data register (**ODR**) represent?

1.8 Task

Add a second input button that allows to change the counting direction (i.e. increment or decrement) with each press of this button. Use polling method to read the pin status. (i.e. use `HAL_GPIO_ReadPin(GPIOx, GPIO_PIN_x)`)

2. Experiment 2: LCD, I2C and Keypad

2.1 Objectives

The objectives of the LCD and Keypad Lab using STM32F4 microcontrollers are to enable students to interface and control an LCD module and keypad with the microcontroller via I2C, implement firmware logic to display information on the LCD screen based on keypad inputs, and gain practical experience in user interface design and embedded systems development. Through this lab, students will develop skills in GPIO configuration, peripheral interfacing, and firmware programming for interactive applications with STM32F4 microcontrollers.

2.2 Pre-Lab

- What is a SysTick handler?
- Write a program to read a status of an input pin and control another output pin using SysTick handler:
- Override void SysTick_Handler(void) so it can read the status of an input pin every 5ms.
- Use a flag to inform the main function that a pin has been triggered.
- Based on the status of the input pin, control an output pin either ON or OFF from the main function.
- In I2C protocol, each slave has a unique address, in this experiment the slave is an LCD adaptor board. How would you connect more than one LCD to the microcontroller?

2.3 Theory

In embedded systems, LCDs are widely used for displaying information like sensor data, system status, or user interface. They can range from simple character displays to more complex graphical displays. In this lab, we will focus on the first type.

One of the common types used in embedded systems is a character LCD (e.g. 16x2, 20x4) that displays text and numbers for the end user. The prefix in the part number of the LCD indicates the number of rows and columns that can be displayed.

For example, 1602 LCD module consists of 16x2 character cells or (2 rows x 16 columns), each cell of which is 5x8 dots. By controlling all of these individual dots, we can display the character we want. It is not practical to control all of these dots directly by the microcontroller (lots of pins and software overhead). Practically, each LCD module comes with a special function controller (i.e. Hitachi HD44780 controller) that is responsible for

controlling all dots and enabling the user to interface with the display by sending parallel data from the microcontroller.

HD44780U has two 8-bit registers, an **instruction register (IR)** and a **data register (DR)**. The **Instruction Register** receives commands from the microcontroller. These commands control the behavior of the display, such as clearing the display or moving the cursor. You can tell the LCD that you're sending a command to the (**IR**) by driving the **RS (Register Select)** pin to low (**0**).

The **Data Register (DR)** holds the **character data** you want to display on the screen. This is where you send the actual letters, numbers, or symbols that should appear at the current cursor position. Think of the Data Register as a **mailbox** where you place characters to be displayed. The LCD reads from this register and shows the corresponding characters on the screen.

You need to set the **RS pin** to **1** to tell the LCD that you're sending data (characters) to the **Data Register**.

2.3.1 LCD parallel connection diagram

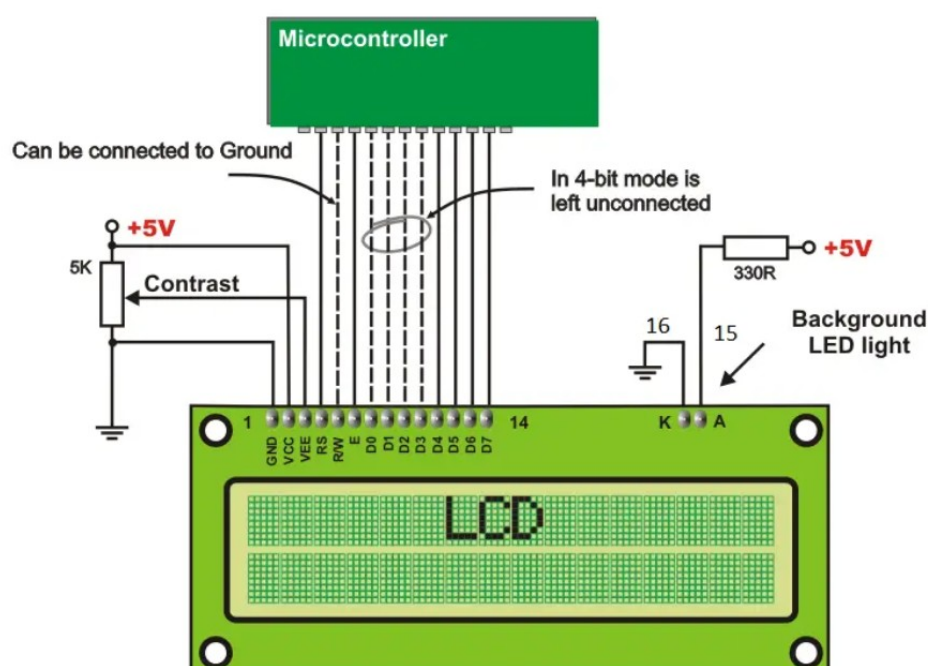


Figure 2.1: LCD parallel data connection with MCU (source: <https://deepbluembedded.com/interfacing-16x2-lcd-with-pic-microcontrollers-mplab-xc8/>)

As illustrated in Figure 2.1, the LCD module has different pins, each pin has its designated function:

- **VCC** is the power supply input pin (+5 V)

- **GND** is the ground pin
- **D0-D7** are the data bus pins (8-bit). The LCD can operate in 4-bit bus mode which means we can send each byte of data in two steps using the 4-bit mode. This will save us four IO pins.
- **RS (Register Select)** pin defines the type of transmitted byte whether it is a **command (0)** or **Data (1)**.
- **R/W** pin allows writing the data to the LCD when it is **Low (0)** and reading the data from the LCD when it is **High (1)**. Usually, we connect it to the ground.
- **E** pin is the **Enable** pin of the LCD. This pin is used to write data to the LCD register by sending a pulse signal with a specific duration after each transmission operation.
- **A** and **K** are the anode and the cathode of the backlight LED of the display.
- **VEE (contrast control)** is used to control the contrast level of the display. We can control the contrast via microcontroller or simply by using a potentiometer.

2.3.2 LCD serial interface

Controlling LCD module directly from the microcontroller requires 6 to 10 pins; 4 or 8 data lines, plus control signals (RS, R/W, and E). This makes the wiring more complex and prone to errors, especially when you deal with breadboards and jumper wires. It also introduces software overhead because the parallel interfaces require careful timing and sequencing of signals.

To address these drawbacks, a very common solution is to use serial-to-parallel ICs (I2C or SPI). These devices simplify the connection by converting a serial bit stream to parallel data, significantly reducing the number of required pins and reducing both wiring and software complexity. Many such devices are available in the market. One of these devices is PCF8574 port extender, which is used to convert two-line (I2C) bus to 8-bit input/output (I/O).

There are many ready-made adapter boards based on the PCF8574 (Figure 2.2). These boards allow easy communication via the SDA and SCL lines. They also include the necessary circuits for controlling the LCD, such as the backlight LED, contrast adjustment, and built-in pull-up resistors.

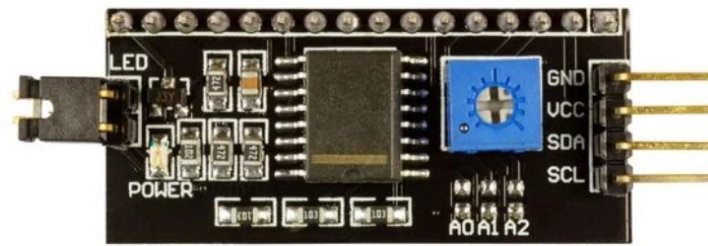


Figure 2.2: PCF8574 adapter board

The adapter board schematic is shown in Figure 2.3. As you can see, PCF8574 has 4 input pins GND, VCC, SDA, SCL for microcontroller connection, and 16 output pins for LCD connection.

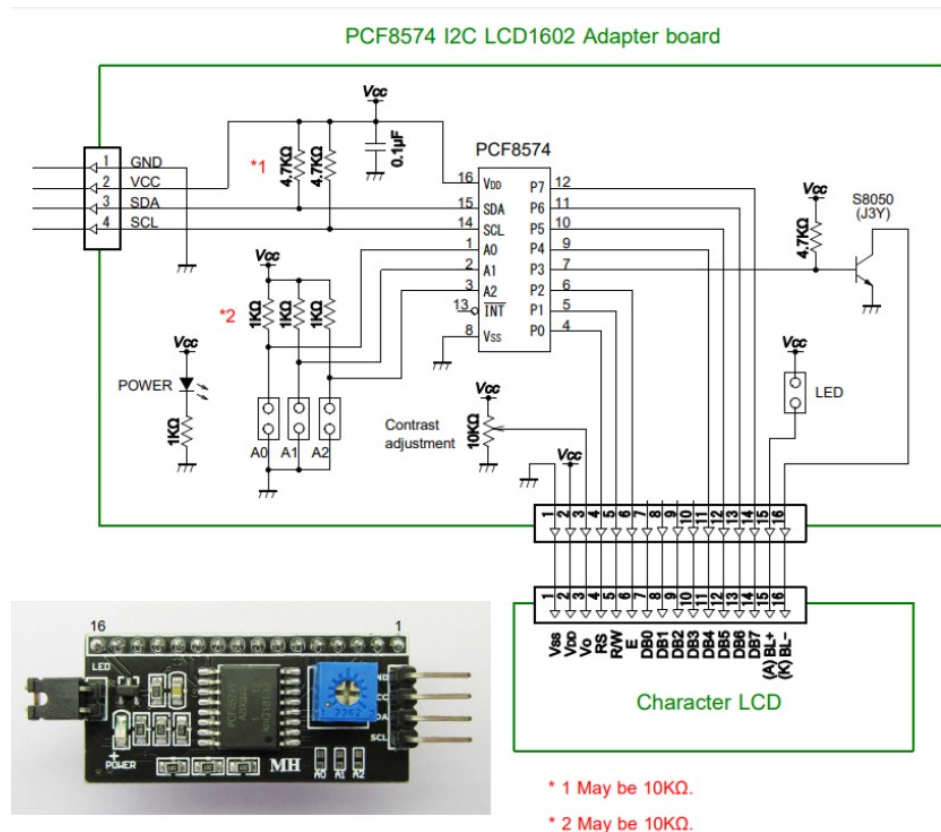


Figure 2.3: PCF8574 adapter board schematic

2.3.3 I2C protocol

I2C (Inter-Integrated Circuit) is a widely used communication protocol that allows multiple devices to communicate with a microcontroller using just two wires: SDA (Serial Data Line) and SCL (Serial Clock Line).

It is a serial communication protocol, meaning data is transferred one bit at a time, but it is highly efficient and flexible because it allows multiple devices (slaves) to share the same two lines, each identified by a unique address. It allows reaching a maximum of 128 devices (with 7-bit address) in standard mode. I2C connection is shown in Figure 2.4

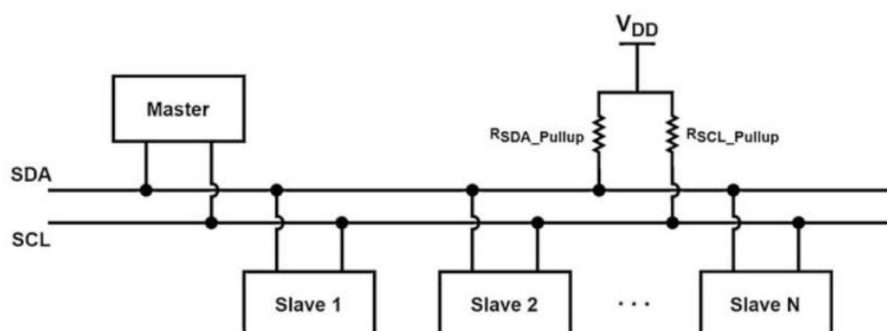


Figure 2.4: I2C master-slaves connection

I2C is especially popular in embedded systems due to its simplicity and minimal pin usage. It is a synchronous master-slave protocol, and it works in half-duplex mode. Both master and slaves can send/ receive data, and the clock is generated by the master device. Moreover, communication is always started by the master device. I2C can transfer data with a speed up to 100 kbits/s in standard mode, and up to 400 kbits/s in the fast data transfer modes

2.3.4 Interface the 1602 LCD with STM32 microcontroller via I2C

To enable communication between the STM32 microcontroller and the LCD via I2C, the PCF8574 adapter board should be connected to the LCD using its 16 pins, while the SDA and SCL pins are connected to the STM32 microcontroller (Figure 3). The next step is to write firmware that sends commands and data to the LCD through the adapter board. In this setup, the STM32 acts as the I2C master, and the PCF8574 board acts as the I2C slave. According to the I2C protocol, each slave device has a unique address.

So, what is the address of the PCF8574? The answer can be found in its datasheet. The upper 4 bits of the address are fixed as 0100, while the lower 3 bits (A2, A1, A0) can be configured by the user based on the state of the address pins. The full address format is: 0 1 0 0 A2 A1 A0 R/W.

By default, the three address pins (A0, A1, A2) are set high in the adapter board, making the address 0100111, which corresponds to 0x4E. The last bit (R/W) is used to specify whether the operation is a read (1) or write (0) operation. In this case, the final address for a write operation would be 0x4E with the R/W bit set to 0.

2.3.5 Library support

There are many libraries available that simplify communication with I2C devices, including LCDs. Sometimes you need to write your own implementation for devices that do

not have available libraries. In the context of this lab, you will not write your own implementation for the device, but you will use a library that enables you to display characters and control the functionality of the LCD via I2C. However, you need to understand how to configure different parameters based on your microcontroller configurations and you also need to understand the function and how you are going to use them.

2.3.6 Keypad

A keypad is a matrix of switches arranged in rows and columns that allows for multiple inputs using fewer microcontroller pins. For example, a 4x4 keypad has 16 keys, but instead of needing 16 GPIO pins, it requires only 8 (4 rows and 4 columns) as shown in Figure 2.5.

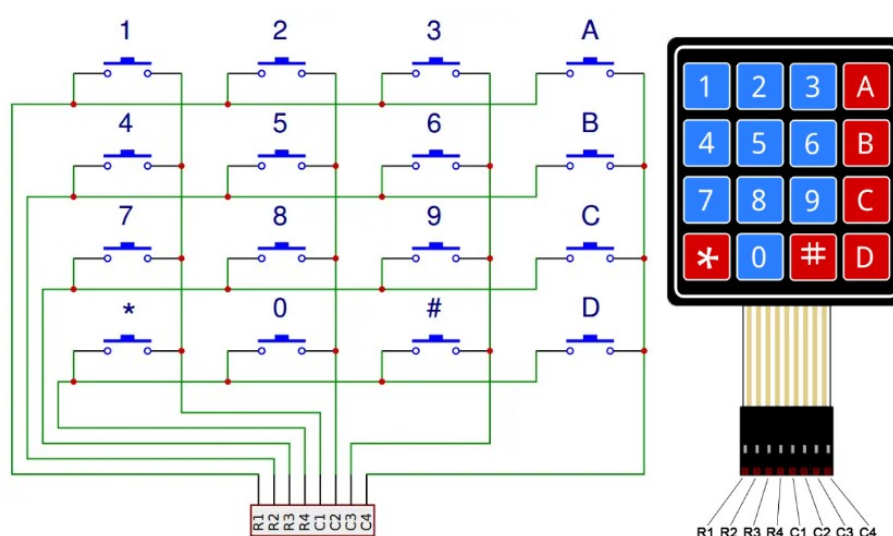


Figure 2.5: 4x4 matrix keypad

2.3.7 How the Keypad Works:

- The rows and columns of the keypad are connected to the microcontroller's GPIO pins.
- When a button is pressed, it connects a row to a column, allowing the microcontroller to detect the specific key that was pressed by checking which row and column are connected (Figure 2.6).

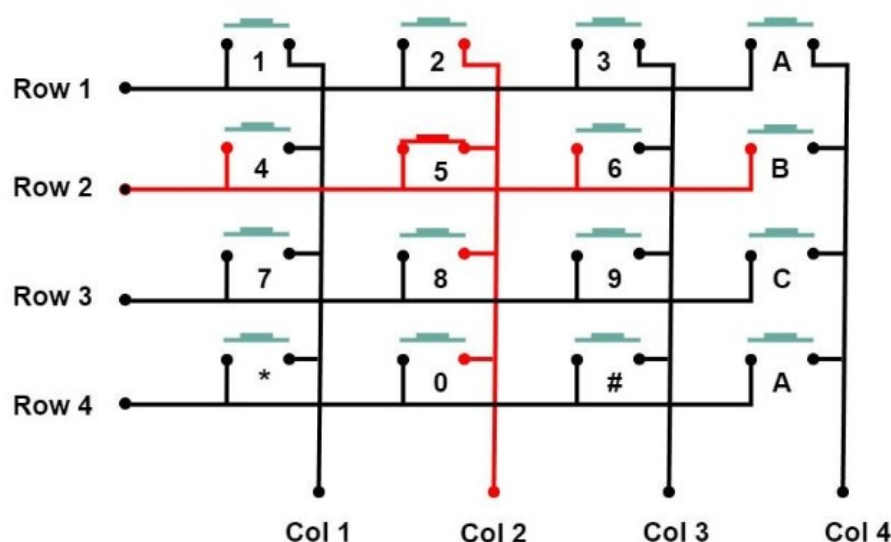


Figure 2.6: 4x4 matrix keypad diagram

- So, how does the microcontroller detect the key? We need to connect the ROWs to output pins and the COLs to the input pins. To identify the key pressed, we will drive one of the ROWs to High (1) and other ROWs to LOW (0) and keep checking the input pins (COLs). Whenever a key is pressed, the connection between ROW and COL gets completed and the COL pin will read 1. Now both the ROW and COL will be 1 and will know which pin has been pressed. The connection between 4x4 keypad and a microcontroller is illustrated in Figure 2.7

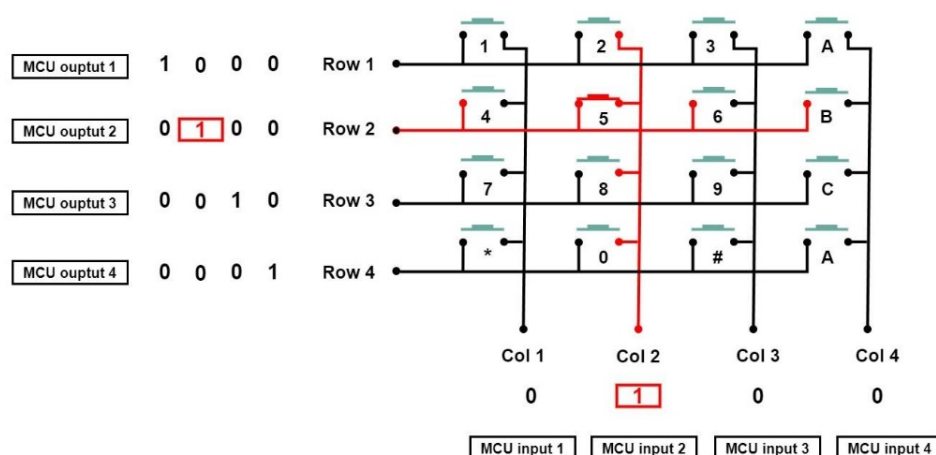


Figure 2.7: 4x4 matrix keypad to microcontroller connection diagram

- In the previous step, we supposed that the input pins connected to COLs are internally pulled down.

2.4 Materials and Equipment

- Breadboard

- STM32F4 evaluation board
- LEDs and push buttons
- Jumper Wires
- 16x2 LCD
- 4x4 Keypad

2.5 Relevant External Library Functions

- **LCD Functions:**

```
void LCD_init(void); // Initializes the LCD module before use.
void LCD_displayText(char *text); //Displays a string of
    characters on the LCD starting at the current cursor position
void LCD_clear (void); // Clears the entire LCD screen and
    resets the cursor to the top-left position (row 0, column 0).
void LCD_setCursor(uint8_t row, uint8_t col); // Moves the
    cursor to a specific position on the LCD.
void LCD_displayInt(int number); // Displays an integer number
    on the LCD as text.
```

- **Keypad Functions:** The following functions are used as part of the keypad scanning process to detect which key is pressed.

```
// Drive one row low at a time
drive_row_high(row);

// Check which column reads low
ColumnInput column = read_columns();
```

2.6 Activity

- Students will first interface the STM32 microcontroller with a 16x2 LCD module using the I2C protocol.
- The program will require users to enter a predefined 4-digit PIN code via the keypad.
- After entering the correct PIN, the LCD will be unlocked, and users will be able to input and display different numbers and characters on the LCD screen.
- If the PIN is incorrect, the system will ask the user to try again

2.7 Procedures

2.7.1 Hardware Setup:

- Before proceeding with hardware connections, you need to assign the I/O pins of the microcontroller to the keypad columns and rows as illustrated in Figure 2.7.
- Notes:
 - If you use STM32F411 Nucleo board, avoid using the following pins: (PA2, PA3, PC14, PC15, PA13, and PA14)
 - If you use STM32F4 Discovery board, avoid using the following pins (PA13 and PA14)
- Connect SCL and SDA pins of the LCD adapter board to PB6 and PB7 respectively.
- Connect +5V and GND to the LCD adapter board

2.7.2 Peripheral Configuration:

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your microcontroller (STM32F407VGT6 or STM32F411RET6).

2. Clock Configuration:

- Enable the external clock from RCC tab, and select the maximum clock frequency from the clock configuration window.

3. Configure GPIO Pins:

- Use the “Device Configuration Tool” to configure GPIO pins for interfacing with the keypad. Select the appropriate GPIO pins and configure them as inputs and outputs.
- Make sure you enter user labels to each input and output pin. Do not forget to pull down the inputs connected to the columns.
- User label should be assigned as follow: COL1, COL2, COL3, COL4, ROW1, ROW2, ROW3, and ROW4

4. Configure I2C :

- Configure the I2C in the standard mode, with the clock speed set to 100KHz as shown in Figure 2.8

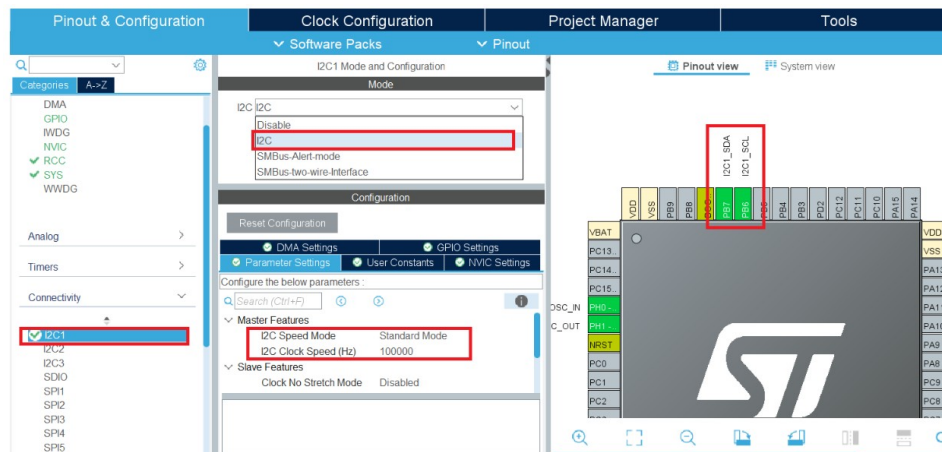


Figure 2.8: I2C Configurations

2.7.3 Software Code

1. Add LCD and keypad libraries to your project

- Copy (or drag and drop) the libraries files found in this [link](#) to the locations shown in Figure 2.9:

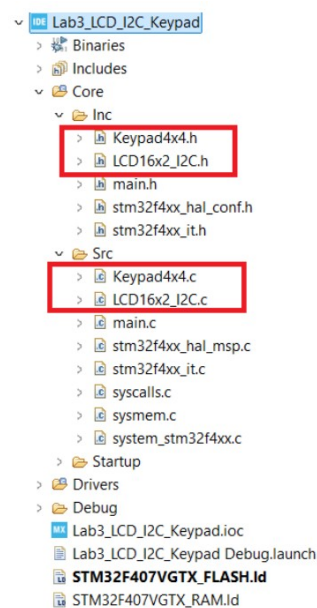


Figure 2.9: LCD and Keypad Libraries

- In order to use libraries, you must include them in main.c file.

```
/* USER CODE BEGIN Includes */
#include <LCD16x2_I2C.h>
#include <Keypad4x4.h>
/* USER CODE END Includes */
```

2. Required global variables and macros

- Add the following definition and global variables

```
/* USER CODE BEGIN PD */
#define PASSWORD "1234" // Predefined password
#define PASSWORD_LENGTH 4
/* USER CODE END PD */

/* USER CODE BEGIN PV */
volatile uint8_t key;
volatile uint8_t pressFlag=0;
volatile uint8_t SysTicks = 0;
volatile uint32_t currentTick=0;
volatile uint32_t previousTick=0;

const uint32_t debounceTime =150;
/* USER CODE END PV */
```

3. Use functions to organize your code

- (a) The following function asks the user to enter a password using the keypad and checks whether it is correct or not.

```
void password_check (void) {
    int pos=-1;
    char input[PASSWORD_LENGTH];

    while(1)
    {
        if (pressFlag==1) {
            pos++ ;
            input[pos] = key ;
            LCD_setCursor(1, pos);
            LCD_displayText("*");
            pressFlag=0;
        }
        if (pos >= PASSWORD_LENGTH) {
            // Compare the entered password with the
            correct one
            if (strncmp(input, PASSWORD,
                PASSWORD_LENGTH) == 0) {
                LCD_clear();
                LCD_setCursor(0, 0);
                LCD_displayText("Correct_PIN");
                break; // Exit the loop and continue
                    the program
            }
            else {
                LCD_clear();
                LCD_setCursor(0, 0);
            }
        }
    }
}
```

```

        LCD_displayText("Try_again:_");

        pos = -1; // Reset the input
                    position
            }
        }
    }
}

```

This function keeps asking the user to enter a 4-digit password. Each time a key is pressed, it shows a star * on the LCD. When four keys are entered, it checks if they match the correct password. If correct, it prints **“Correct PIN”** ; if not, it says **“Try again”** and restarts the input.

Please note that the global variable `pressFlag` becomes 1 when the keypad interrupt detects a key press.

- (b) `handle_keypress` function allows to show a key you pressed on the LCD screen.

```

void handle_keypress(uint8_t key) {
    // Handle the detected keypress
    LCD_clear();
    LCD_setCursor(0, 0);
    char numStr[16];
    sprintf(numStr,"%c", key);
    LCD_displayText(numStr);
}

```

4. `SysTick_Handler` will be used here to periodically scan the keypad and detect which key is pressed.

`SysTick` timer generates `SysTick_Handler` interrupts at regular intervals (every 1ms). In this experiment we write our own routine to detect pressed keys every 5 ms.

```

void SysTick_Handler(void)
{
    static uint8_t row = 0;
    HAL_IncTick();
    SysTicks++;

    if(SysTicks == 5) // Each 5msec
    {
        // Drive one row low at a time
        drive_row_high(row);

        // Check which column reads low
        ColumnInput column = read_columns();

        // If a column is low, register the key press
    }
}

```

```

        if ((column == COL1) || (column == COL2) || (column
            == COL3) || (column == COL4)) {

            currentTick = HAL_GetTick(); // Get current
            system tick
            if ((currentTick - previousTick) >
                debounceTime) { // Debounce check
                key = decode_key(row, column);
                pressFlag=1;
            }
            previousTick = currentTick; // Update the
            previous tick for the next comparison
        }

        // Move to the next row for the next scan
        row = (row + 1) % 4; // Assuming 4 rows
        SysTicks = 0;
    }
}

```

5. Program Code.

In the main.c

- Include all required libraries
- Define variables used on point 2.
- Add password_check and handle_keypress functions with their prototypes
- Override SysTick_Handler using the code on point 4.
- In the main function and before the while loop, initialize the LCD and check for password entry.

```

LCD_init();
LCD_clear();
LCD_setCursor(0, 0);
LCD_displayText("Enter_Password!");
password_check();

```

- In the while loop, check if a button press (indicated by an interrupt flag pressFlag) has occurred; if so, print the character to the LCD and then reset the flag pressFlag=0.
- Build the project and flash it to your STM32 MCU.

2.8 Tasks

1. Add two LEDs to indicate whether the entered password is correct or not.

- Green LED: Lights when the password is correct

- Red LED: Lights when the password is incorrect
2. Modify the system so that after three incorrect password attempts, the system locks for 5 seconds and displays “Access Denied”.
 3. **(Optional):** Change the `password_check()` function to allow users to change the password after entering the correct one.

2.8.1 Some notes about the firmware code:

- In the example code provided for this experiment, the keypad detection is implemented in a non-blocking mode. This means that the main program execution is not blocked during keypad scanning. To achieve this, we used the SysTick timer, which generates interrupts at regular intervals (every 1ms).
- As you know, at every SysTick interrupt, an interrupt service routine (ISR) is called, which is SysTick interrupt handler. In this experiment we write our own routine to detect pressed keys every 5 ms.
- In practice, an Interrupt Service Routine (ISR) should perform only minimal tasks that require very little time. This is because ISRs are designed to respond to events quickly without delaying other processes.
- Typically, instead of executing complex operations within the ISR, we use flags to indicate that an interrupt has occurred. Other functions in the main program can then check these flags and handle the necessary actions outside of the ISR.
- This approach ensures that the system remains responsive and avoids potential delays caused by long ISR routines.
- Keypad buttons may suffer from bouncing, where a single keypress generates multiple signals, therefore, a debouncing logic has been implemented (for more details about software debouncing mechanism, refer to Lab 1)

2.8.2 Post Lab Questions

1. Explain why the SysTick timer is used in this experiment instead of polling inside the main loop.
2. What would happen if the keypad scanning code were placed entirely inside the `while(1)` loop instead of the SysTick handler?
3. Why is it important to minimize the code executed inside an interrupt service routine (ISR)?
4. How does using I2C to control the LCD simplify hardware wiring compared to direct parallel interfacing?
5. Why do we use flags (like `pressFlag`) instead of calling `handle_keypress()` directly inside the `SysTick_Handler`?

3. Experiment 3: ADC and Analog Sensor Interfacing

3.1 Objectives

The lab aims to provide students with practical skills in ADC configuration and sensor interfacing using an STM32F4 microcontroller. Objectives include understanding ADC principles, interfacing with analog sensors (potentiometer, LDR temperature sensor), and integrating an LCD display for real-time data presentation.

3.2 Pre-Lab

- Review ADC basic concepts.
- Study the code and the configurations of ADC using STM32CubeIDE

3.3 Theory

3.3.1 ADC Basics

Real-world signals, such as sound, light, temperature, and motion, are analog or continuous signals, meaning they can vary smoothly over time. In embedded systems, we often need to collect information about these signals to understand the environment. This is achieved using sensors or transducers, which convert analog signals into equivalent voltage values. However, since embedded systems only understand digital data in the form of 0s and 1s, we must convert these analog voltage values into a digital format in order to process them.

Analog-to-Digital converters (ADC) translate analog signals, real world signals like temperature, pressure, voltage, current, distance, or light intensity, into a digital representation of that signal

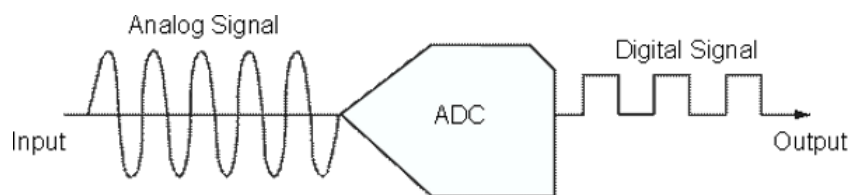


Figure 3.1: Basic ADC Operation

To process an analog signal in a digital system, it needs to be converted into a series of discrete values. Sampling in ADC refers to the process of taking discrete samples of an analog signal at regular intervals or sampling rate. The sampling rate must be at least twice the highest frequency presented in the analog signal to avoid aliasing according to

Nyquist-Shannon sampling theorem. After sampling, the input signal is held between samples using a hold circuit.

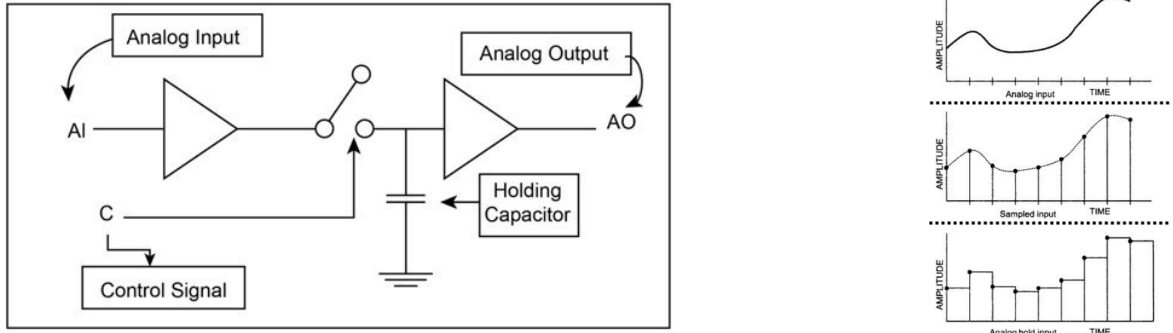


Figure 3.2: Sampling and Hold Circuit in ADC

The digital output of an ADC is represented by a number of binary bits. The number of bits defines the resolution of the ADC.

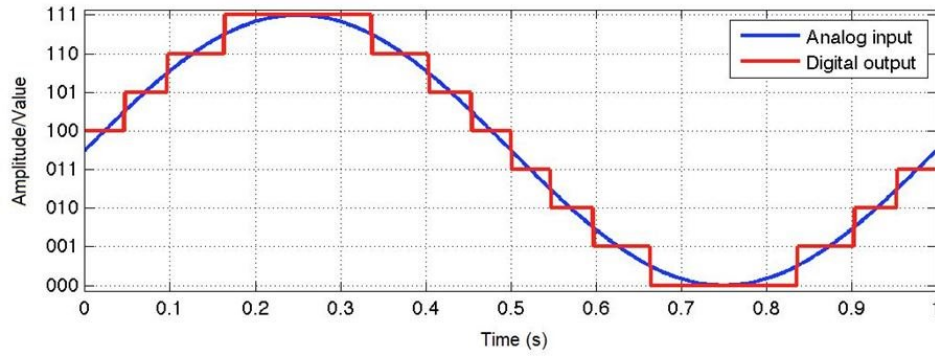


Figure 3.3: ADC Resolution and Quantization Levels

We can calculate the digital output value of the ADC using the following formula:

$$\text{ADC Output} = \text{round} \left(\frac{V_{in}}{V_{ref}} \times (2^n - 1) \right) \quad (1)$$

Where:

- V_{in} is the input voltage to be converted.
- V_{ref} is the reference voltage for the ADC.
- n is the number of bits of the ADC resolution.
- round is the rounding function to the nearest integer.

Example: For a 12-bit ADC with a reference voltage of 3.3V, if the input voltage is 1V, the ADC output can be calculated as follows:

$$\text{ADC Output} = \text{round} \left(\frac{1V}{3.3V} \times (2^{12} - 1) \right) = \text{round}(1240.9) = 1241 \quad (2)$$

3.3.2 STM32F4 ADC Peripheral

The ADC in the STM32F4 microcontroller is a peripheral that converts analog signals into digital values, allowing the microcontroller to process real-world sensor data. It utilizes successive approximation approach to perform analog to digital conversions. It has up to 19 multiplexed channels allowing it to measure signals from 16 external sources, two internal sources, and the VBAT channel. The A/D conversion of the channels can be performed in single, continuous, scan or discontinuous mode. The result of the ADC is stored into a left or right-aligned 16-bit data register.

The resolution of the ADC in the STM32F4 microcontroller is configurable, and it can be 12-bit, 10-bit, 8-bit, or 6-bit.

The sample time in STM32 microcontroller is programmable where the ADC samples the input voltage for a number of ADCCLK cycles that can be modified by using ADC sample registers. Using STM32CubeIDE, you can select the sample cycles from the configuration tool for each channel.

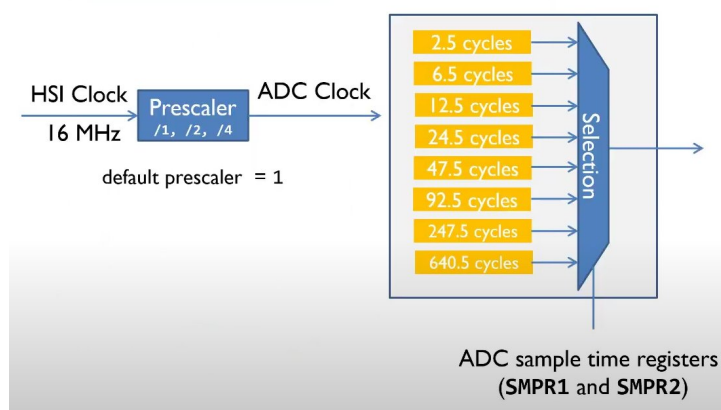


Figure 3.4: STM32F4 ADC Sample Time Selection

The STM32F4 microcontroller has up to three ADC modules (ADC1, ADC2, and ADC3) connected to the APB2 bus. This means that the ADCs Receive their clock source through this bus. (check Functional overview section in the datasheet)

The clock from the APB2 bus might be too fast for the ADCs to operate correctly. To adjust the clock speed to a suitable value for the ADC, there is an internal prescaler dedicated to the ADCs. This prescaler is used to divide the APB2 clock down to a frequency that is compatible with the ADC that does not exceed its maximum operating frequency (typically up to 36 MHz for the STM32F40xx and STM32F41xx ADCs). (check ADC characteristics table in the datasheet)

The time the ADC takes from the moment it starts sampling an analog signal until it produces a corresponding digital result is called ADC conversion time. It consists of sampling time and the conversion time (or data conversion). In our case the data conversion is the time that successive approximation circuit takes to convert the sampled data into digital equivalent.

The data conversion time in STM32 ADC is equal to the resolution (e.g., in 12-bit resolution, the conversion typically takes 12 cycles of the ADC clock). Therefore, the total

conversion time of the ADC is:

$$\text{Total Conversion Time} = \text{Sampling Time} + \text{Data Conversion Time} \quad (3)$$

Assume the ADC clock is 16 MHz and the sampling time is 4 ADC clock cycles for a 12-bit conversion. The total conversion time can be calculated as follows:

$$\text{Total Conversion Time} = \frac{4 \text{ cycles}}{16 \text{ MHz}} + \frac{12 \text{ cycles}}{16 \text{ MHz}} = 0.25\mu s + 0.75\mu s = 1\mu s \quad (4)$$

3.3.3 ADC Conversion Modes

The STM32 microcontroller supports various conversion modes as shown in the following figure:

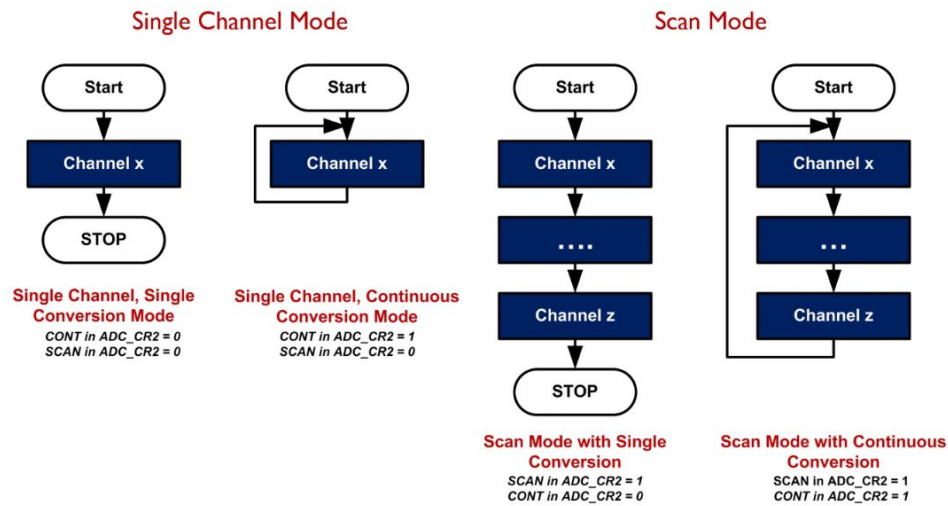


Figure 3.5: STM32F4 ADC Conversion Modes

Note: In this experiment we will only consider Single Channel, Single Conversion Mode.

3.3.4 Channel Selection

There are 16 multiplexed channels. It is possible to organize the conversions in two groups: regular and injected. This lab covers only the regular channels with a polling method.

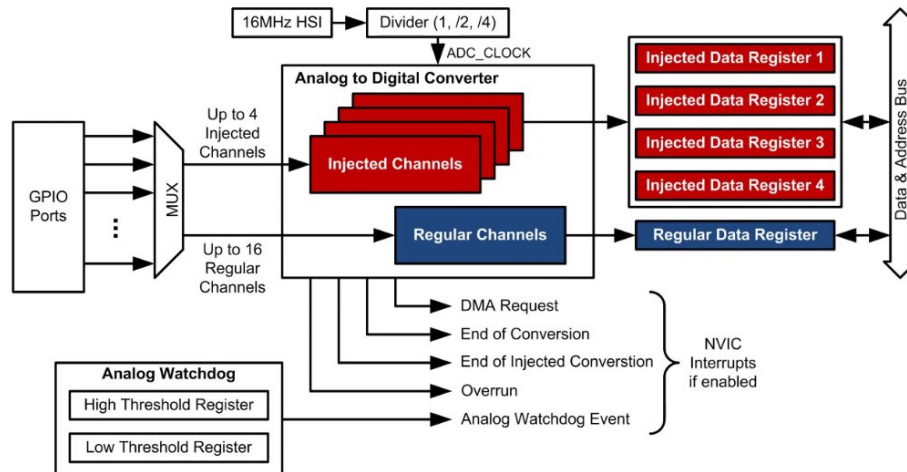


Figure 3.6: STM32F4 ADC Channels

3.3.5 Temperature sensor, V_{REFINT} and V_{BAT} internal channels

STM32 microcontrollers provide several internal channels useful for monitoring and calibration: the temperature sensor, the internal reference voltage (V_{REFINT}), and the V_{BAT} channel. These channels can be read by the ADC for diagnostics or calibration purposes.

- **Temperature sensor:** For STM32F40x/STM32F41x devices the internal temperature sensor is connected to ADC1_IN16.
- **V_{REFINT} (internal reference voltage):** Connected to ADC1_IN17.
- **V_{BAT} (battery voltage):** Connected to ADC1_IN18; it can be converted as either an injected or a regular channel.

Note: The temperature sensor, V_{REFINT} and V_{BAT} channels are available only on the master ADC (ADC1) peripheral.

3.3.6 Light Dependent Resistor (LDR)

An LDR (Light Dependent Resistor), also known as a photoresistor, is a type of resistor whose resistance changes according to the intensity of light. Its resistance decreases when lighting increases. So, in the darkness, an LDR can have very high resistance, while in bright light, the resistance drops to a few hundred ohms or lower.

The LDR can be used in many applications, including automatic lighting systems, measuring light intensity for cameras, and detecting object passes between a light source and an LDR. LDRs are often used in a voltage divider configuration with a fixed resistor to convert the change in resistance into a change in voltage, which can be read by a microcontroller.

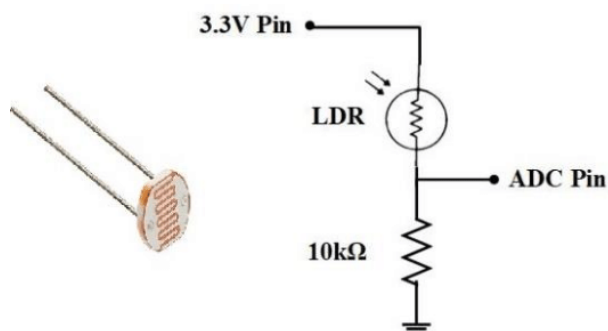


Figure 3.7: LDR Voltage Divider Circuit

3.4 Materials and Equipment

- STM32CubeIDE (software)
- STM32F4 evaluation board
- Breadboard
- Jumper wires
- 16x2 character LCD
- LDR (photoresistor)
- Potentiometer (e.g., 10 kΩ)
- Assorted resistors (for voltage divider, pull-ups, etc.)
- USB cable / power supply for the evaluation board
- Relevant datasheets and the STM32F4 reference manual

3.5 Relevant HAL APIs

- **ADC Single Channel, Single Conversion Mode Relevant APIs**

```
HAL_ADC_Start(); //Starts the ADC conversion process.
HAL_ADC_PollForConversion(); //Waits (polls) until the ADC
    finishes conversion.
HAL_ADC_GetValue(); //Reads the digital value converted by the
    ADC.
HAL_ADC_Stop(); //Stops the ADC conversion process.
```

3.6 Activity

Develop firmware to read sensor data from a potentiometer and a LDR sensor interfaced with an STM32F4 microcontroller and display the readings on an LCD display in real-time.

3.7 Procedure

3.7.1 Hardware Setup

- Connect the LCD as demonstrated in the previous labs

- Connect the output of the voltage divider circuit for the LDR (Figure 3.7) to PA6.
- Connect the signal pin of the potentiometer to PA4.
- Ensure proper wiring and provide necessary power supplies for the sensors.

3.7.2 Peripheral Configuration

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your microcontroller (STM32F407VGT6 or STMF411RET6).

2. Clock Configuration:

- Enable the external clock from RCC tab and select the maximum clock frequency from the clock configuration window.

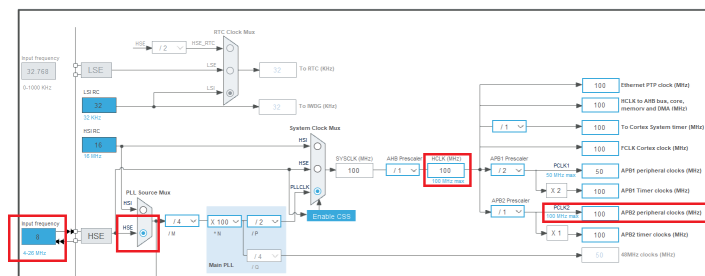


Figure 3.8: STM32F4 Clock Configuration

3. Configure GPIO Pins

- Check the previous lab to configure the I2C interface for the LCD.
- Enable the ADC channels as shown in the figure below.

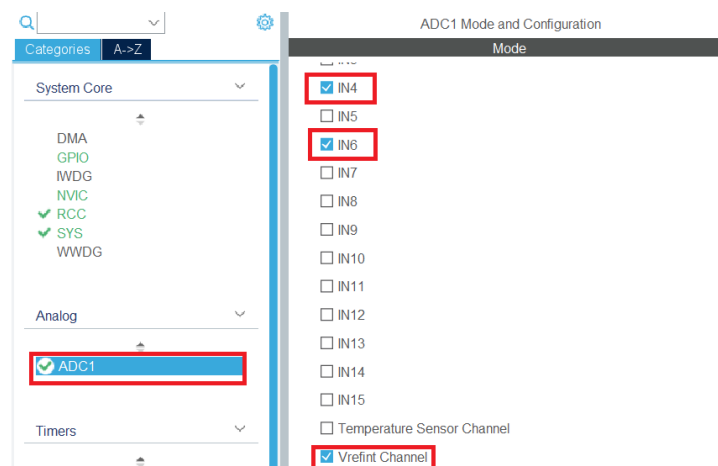


Figure 3.9: STM32F4 ADC Channel Configuration

- Select ADC parameters from the Parameter Setting tab. For this experiment choose a sampling time of 144 cycles. (Can you calculate the total conversion time?)

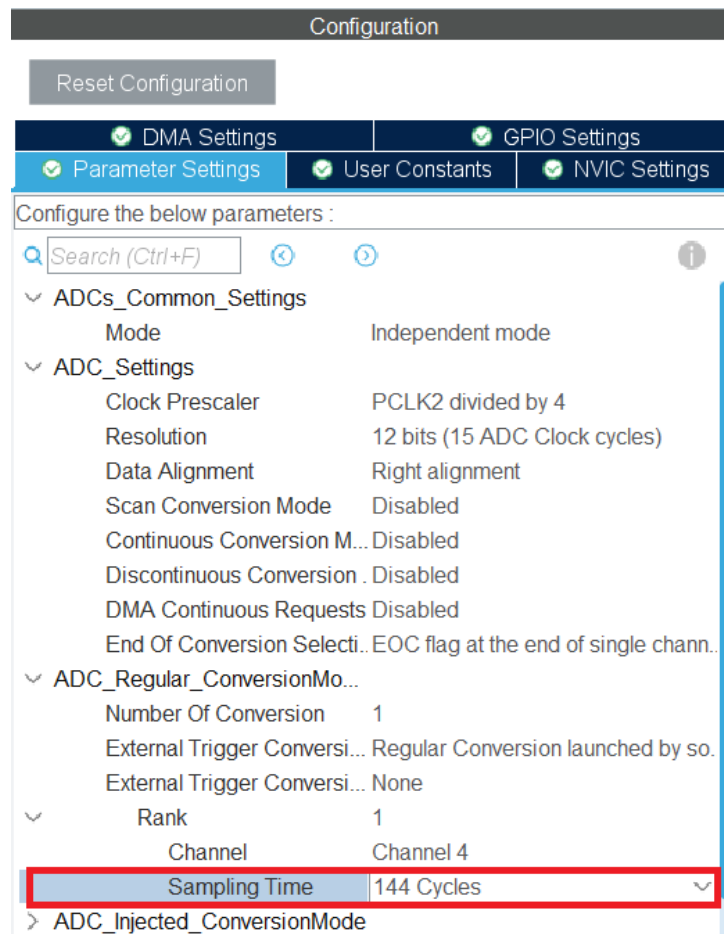


Figure 3.10: STM32F4 ADC Parameter Configuration

- Generate code to produce the necessary initialization files.

4. Enable the float formatting support

- Check your MCU Settings from "Project Properties > C/C++ Build > Settings > Tool Settings.

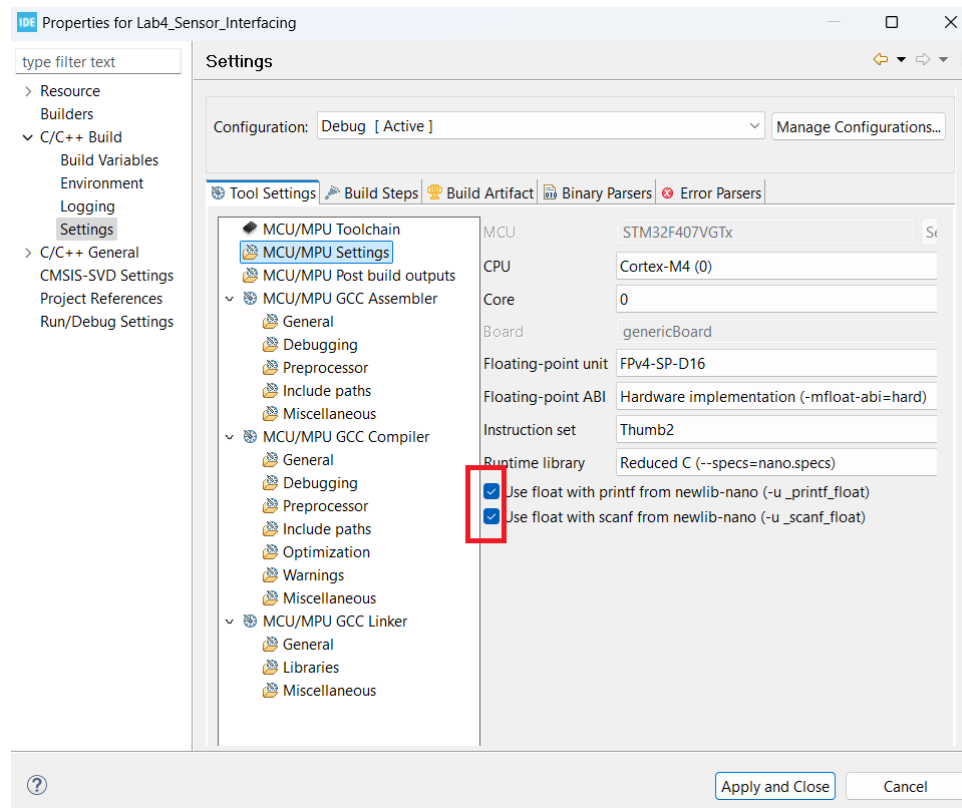


Figure 3.11: Enable Float Support in STM32CubeIDE

5. LCD Library

- Check Lab #2 for the LCD library and initialization code.

3.7.3 Software Code

1. Include the required header files

- In your main.c file, include the necessary header files for the LCD and ADC functionalities.

```
/* USER CODE BEGIN Includes */
#include <LCD16x2_I2C.h>
#include <stdio.h>
#include <string.h>
/* USER CODE END Includes */
```

2. Define global variables

```
/* USER CODE BEGIN 0 */
uint16_t adcValue;
float adcValueflt;
float adcVrefVoltage;
uint32_t vdda_voltage ;
/* USER CODE END 0 */
```

3. Initialize LCD and display an initial message

```
/* USER CODE BEGIN 2 */
LCD_init();
LCD_clear();
LCD_setCursor(0, 0);
LCD_displayText("LAB3_LCD!");
HAL_Delay(1000);
LCD_clear();
/* USER CODE END 2 */
```

4. Add user functions and their prototypes

```
/* USER CODE BEGIN 4 */
void LCD_displayFloat (float flnum){
    char fltChar[7];
    sprintf(fltChar, "%.3f", flnum);
    LCD_displayText(fltChar);
}

uint32_t ADC_Get_Value(ADC_HandleTypeDef *hadc, uint32_t
    adc_channel) {
    ADC_Select_Channel(hadc, adc_channel);
    uint32_t adc_value;
    HAL_ADC_Start(hadc);
    if (HAL_ADC_PollForConversion(hadc, 100) == HAL_OK)
    {
        adc_value = HAL_ADC_GetValue(hadc);
    }
    HAL_ADC_Stop(hadc);
    return adc_value;
}

void ADC_Select_Channel(ADC_HandleTypeDef *hadc, uint32_t
    adc_channel) {
    ADC_ChannelConfTypeDef sConfig = {0};
    sConfig.Channel = adc_channel;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_144CYCLES;
    if (HAL_ADC_ConfigChannel(hadc, &sConfig) != HAL_OK) {
        Error_Handler();
    }
}

float ADC_Get_Vrefint(void) {
    uint16_t vrefint= ADC_Get_Value(&hadc1,ADC_CHANNEL_VREFINT);
    // Read the VREFINT ADC value
    uint16_t vrefint_cal = *VREFINT_CAL_ADDR; // Read VREFINT
        calibrated value
    return 3.3 * ((float)vrefint_cal / vrefint);
}
/* USER CODE END 4 */
```

5. Write the program that runs forever

```

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    adcVrefVoltage= ADC_Get_Vrefint();
    LCD_setCursor(0, 0);
    LCD_displayText("ADC6_value=_");
    LCD_setCursor(0, 11);
    adcValue= ADC_Get_Value(&hadc1,ADC_CHANNEL_6);
    adcValueflt = ((float)adcValue/4095) *adcVrefVoltage;
    LCD_displayFloat(adcValueflt);
    LCD_setCursor(1, 0);
    LCD_displayText("ADC4_value=_");
    LCD_setCursor(1, 11);
    adcValue= ADC_Get_Value(&hadc1,ADC_CHANNEL_4);
    adcValueflt = ((float)adcValue/4095) * adcVrefVoltage;
    LCD_displayFloat(adcValueflt);
    HAL_Delay(200);
}
/* USER CODE END 3 */

```

3.8 Some notes about the firmware code:

- The function `ADC_Get_Vrefint` is used to measure the actual V_{DDA} value. (The calculation inside the function obtained from the reference manual)
- Why would we use that? In some applications, specifically battery-powered ones, it is common for the supply voltage to drop as the battery discharges. Sometimes, both V_{DDA} and V_{REF+} is powered directly by the battery. The problem with this is if the V_{DDA} is based off of this battery voltage, then you may have difficulties to get precise voltage conversions from the ADC. This is due to the math that converts the raw ADC value to a voltage relies on knowing V_{DDA}/V_{REF+} .
- One solution to this is to add some extra hardware
- Another solution is to calculate the voltage of V_{DDA} in software based on V_{REF}

3.9 Task

Write a logic that monitors the reading from the LDR. If the reading indicates low light intensity, turn on an LED. Otherwise, the LED should remain off. Keep in mind that LED should not fluctuate due to minor variations in the LDR reading.

4. Experiment 4: Timer Configuration and Output Compare Control

4.1 Objectives

The lab aims to provide students with practical skills in configuring timers, using their ISR (Interrupt Service Routine), and working with the output compare mode. It involves reading values from an external source at fixed intervals, adjusting the frequency of an output signal, and displaying the frequency in real-time on an LCD.

4.2 Pre-Lab

- What is the function of the prescaler (PSC) in a timer, and how does it affect the output frequency?
- Explain the difference between **ARR (Auto-Reload Register)** and **CCR (Capture/Compare Register)** in timer operation.
- Why do we use output compare mode (**toggle on match**) to generate a square wave, instead of toggling a GPIO pin in software?
- What will happen if you increase the **ARR** value while keeping the prescaler constant?
- If the timer clock is 50 MHz and the prescaler is set to 5000 - 1, calculate the **ARR** value needed to produce:
 - 1 Hz signal
 - 10 Hz signal
 - 100 Hz signal
- What is the purpose of the `HAL_TIM_PeriodElapsedCallback()` function in this experiment?
- The ADC reads a potentiometer voltage between 0 and 3.3 V. Explain how this reading is mapped to change the LED blinking frequency.
- What is the advantage of using hardware timers over software delays (`HAL_Delay`) when generating precise periodic signals?

4.3 Theory

A timer, sometimes referred to as a counter, is a special piece of hardware inside many microcontrollers. You can imagine it as a digital stopwatch that keeps track of time or counts events. For example, an 8-bit timer counts from 0 to 255 and then rolls over back to 0 once it reaches its maximum value, though this maximum value is programmable.

A timer can be used to generate precise timing events, control the execution of specific tasks, and to measure time intervals. Timers are crucial in various applications, including controlling external devices, generating PWM signals, and managing real-time tasks.

4.3.1 How does timer work?

- A timer will tick (increment or decrement by one) each time it receives a clock pulse
- The counting speed depends on the input clock frequency
- Most microcontroller timers have a prescaler (PSC), which divides the input clock frequency to the timer. This allows you to control the timer's counting speed.
- The maximum count of a counter is called the Auto-reload, which is the value that the timer can reach before it resets (rolls over) and starts counting again.
- Auto-reload value is programmable through auto-reload register (ARR)
- A timer can be a count-up, a count down, or a count up/down timer
 - Count-up timer counts from 0 to the auto-reload value then restarts from 0
 - Count-down timer counts from the auto-reload value down to 0 then restarts from the auto-reload value
 - Count up/down timer counts from 0 to the auto-reload value then counts from the auto-reload value down to 0. Then it restarts counting from 0
- Timer resolution is defined by the bit width, which is specified in bits such as 8-bit, 16-bit, or 32-bit.
 - A timer with an N-bit counter can count from 0 to $2^N - 1$.
- To calculate the timer period, you can use the formula

$$Timer\ Period = \frac{(PSC + 1)(ARR + 1)}{Counter\ Clock\ (Hz)}$$

$$Counter\ clock = CK_{CNT}$$

4.3.2 STM32F4 Timers

If you look at the datasheet for your microcontroller, you will often find different sections talking about the various timers available. STM32F4 has different timers (sometimes up to 17 timers). However, we will focus in this lab on the general-purpose timers.

Figure 4.2 illustrates the block diagram of general-purpose timer; it can be divided into 4 main blocks:

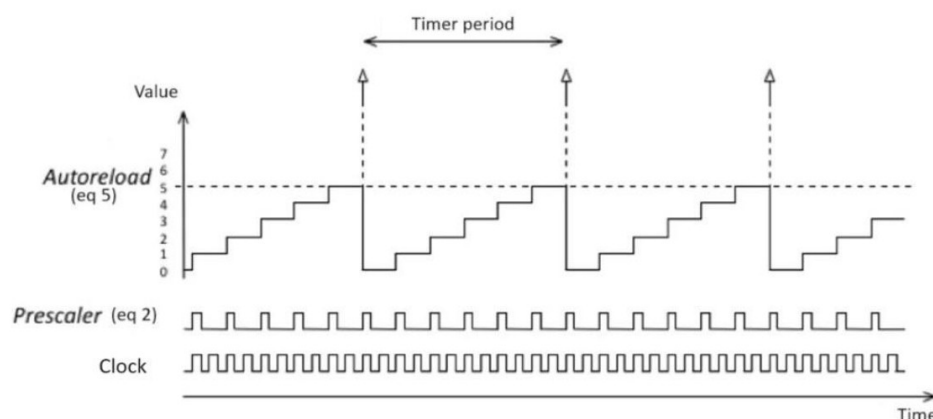


Figure 4.1: Timer working principle

1. The **trigger and clock control block** is responsible for selecting the **clock source** of the timer. It is also used to start/stop or reset the counter.

Note: In this lab, we will only use the internal clock as the trigger source of the counter

2. The **time-base unit** contains:

- (a) the **counting engine**, which manages the incrementing or decrementing of the counter.
- (b) the **auto-reload register (ARR)**; which defines the maximum count value.
- (c) The **prescaler**, which divides the input clock frequency to adjust the speed of the counter.

3. The **output compare** block, the counter is responsible for managing the pins without direct involvement of the CPU. But, how does output compare mode work?
4. The timer has output channels (pins) associated with it and each channel has its dedicated **output compare register (OCR)**. When a **match** detected between the value inside the **OCR** and the **counter**, the output compare function drives the corresponding output pin **high, low, toggle** its state, or stay unchanged based on the configurations.
5. The **input capture** block is used to measure the duration of external events. It is out of the context of the current lab.

4.4 Materials and Equipment

- Breadboard
- STM32F4 evaluation board
- LEDs and push buttons
- Jumper Wires
- 16x2 LCD
- Potentiometer

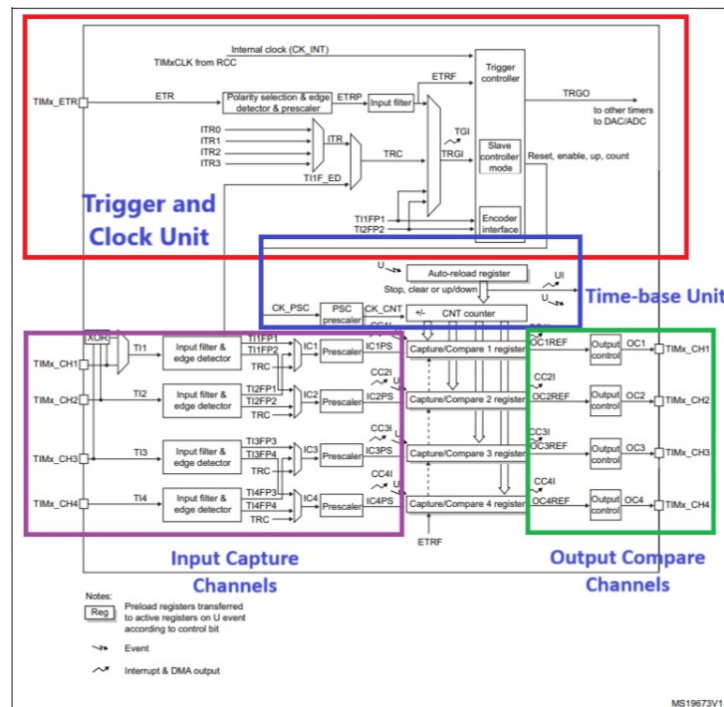


Figure 4.2: General purpose timer block diagram of STM32f4xx

4.5 Relevant HAL APIs

- **Timer HAL APIs:**

```
HAL_TIM_Base_Start_IT(&htimx);
HAL_TIM_Base_Stop_IT(&htimx);
HAL_TIM_OC_Start(&htimx, TIM_CHANNEL_x);
HAL_RCC_GetPCLK1Freq(); // For APB1 timers
HAL_RCC_GetPCLK2Freq(); // For APB2 timers
```

- **interrupt callbacks:**

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

- **Access registers :**

```
// Get the timer prescaler from its register
prescaler= htim->Instance->PSC;

uint32_t prescaler= htim->Instance->PSC; // Get the timer
prescaler from its register (x=1,2,.. etc.)
uint32_t arr= htim->Instance->ARR; // Get the timer ARR value
from its register (x=1,2,.. etc.)
```

4.6 Activity

- Develop firmware to read the potentiometer input every 100ms and use this input to adjust the frequency of a signal (from 1Hz to 100Hz) proportionally. Drive an LED with the generated signal using output compare (OC) mode. Display the frequency value on an LCD display in real-time.

4.7 Procedure

4.7.1 Hardware Setup

- Connect the LCD as demonstrated in the previous labs
- Connect the signal pin of the potentiometer to PA4.
- Connect an LED to an output compare channel 4 of TIM4
 - For Nucleo board: Connect the LED to PB9
 - For Discovery board: Connect the LED to PD15
- Ensure proper wiring and provide necessary power supplies for the potentiometer and LCD.

4.7.2 Peripheral Configuration

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your microcontroller (STM32F407VGT6 or STM32F411RET6).

2. Clock Configuration:

- Enable the external clock source from the **RCC** tab and set the maximum clock frequency in the **Clock Configuration** window (refer to previous labs for guidance). Ensure that the **APB1 Timer Clocks** frequency is set to
 - 100 MHz (for Nucleo board)
 - 50 MHz (for Discovery board)



Figure 4.3: APB1 Timer Clocks frequency

3. Configure GPIO Pins

- Check previous LAB to configure the I2C for the LCD.
- Set up the ADC to read input from channel 4 using the same configuration used in **Lab 4**.
- Enable support for float formatting in the project settings, following the process from **Lab 4**.
- Configure the TIM2 timer to use the internal clock source and set the appropriate **Prescaler** and **ARR** values. **Can you calculate the timer period in μ s?**
- **Note:** The prescaler value should be (50-1) for Discovery board

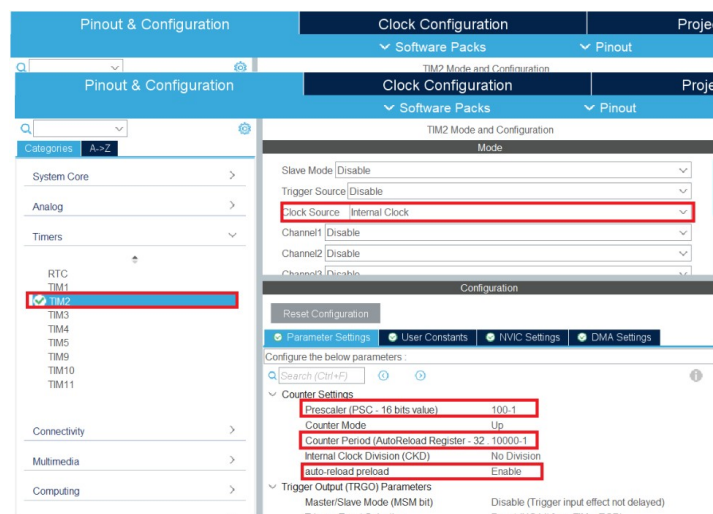


Figure 4.4: TIM2 Configurations

- Enable TIM2 interrupt signal from NVIC setting tab.

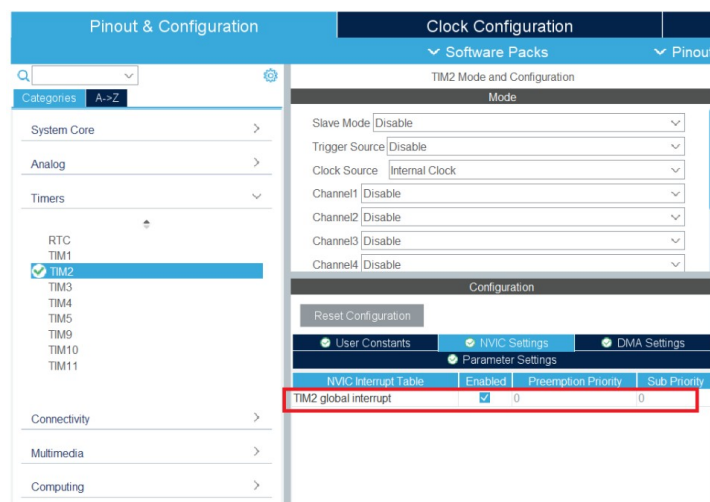


Figure 4.5: TIM2 Interrupt Enable

- Activate the **TIM4** internal clock and set **Channel 4 to Output Compare mode**. Enter the **Prescaler** value as shown in the provided figure, while

leaving the **ARR** value at its default setting, as it will be modified in the program. Enable the **preload** option and select "**Toggle on Match**" from the drop-down menu, to allow the generation of a square wave signal.

- Note: The prescaler value should be (5000-1) for Discovery board

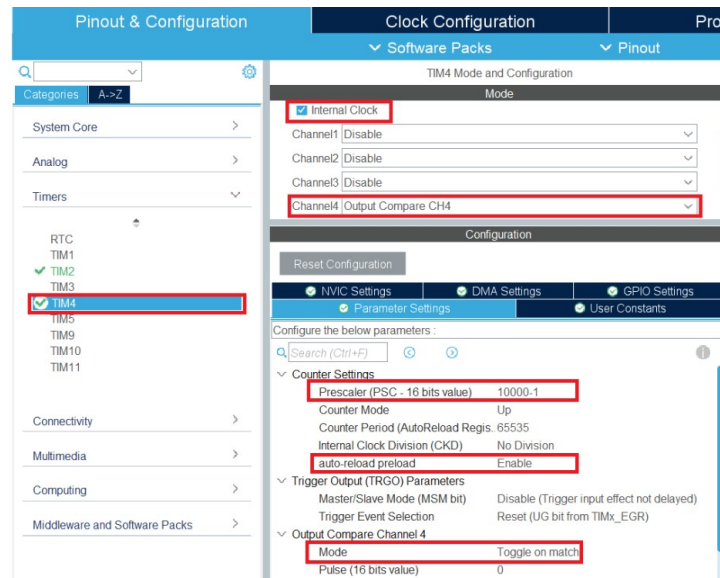


Figure 4.6: TIM4 Configurations

- Generate code to produce the necessary initialization.

4.7.3 Software Code

1. LCD Library

- Check Lab#3

2. Include the required header files and define global variables

```
#include <LCD16x2_I2C.h>
#include <stdio.h>
#include <string.h>

volatile uint16_t adcValue;
volatile float adcValueflt;
volatile float tim4pulsWidth;
float outFreq;
```

3. Add user functions and their prototypes

```
uint32_t ConvertADCToARR(TIM_HandleTypeDef *htim, uint32_t
    adc_value) {
    uint32_t tim_clk ;
```

```

uint32_t prescaler;

// Define the desired frequency range.
const float min_freq = 1.0f / 2; // Minimum frequency (1 Hz
)
const float max_freq = 100.0f * 2; // Maximum frequency
(100 Hz)

// Map the ADC value (0-4095) to the frequency range.
float frequency = min_freq + ((max_freq - min_freq) *
    adc_value / 4095.0f);

// Calculate timer clock frequency
tim_clk= Get_Timer_Clock_Freq(htim);

// Get the timer prescaler from its register
prescaler= htim->Instance->PSC;

// Calculate the corresponding ARR value.
uint32_t arr = (uint32_t)((tim_clk/ (frequency * (
    prescaler + 1))) - 1);
return arr;
}

uint32_t ADC_Get_Value(ADC_HandleTypeDef *hadc, uint32_t
    adc_channel) {
    ADC_Select_Channel(hadc, adc_channel);
    uint32_t adc_value;
    HAL_ADC_Start(hadc);
    if (HAL_ADC_PollForConversion(hadc, 100) == HAL_OK)
    {
        adc_value = HAL_ADC_GetValue(hadc);
    }
    HAL_ADC_Stop(hadc);
    return adc_value;
}

void ADC_Select_Channel(ADC_HandleTypeDef *hadc, uint32_t
    adc_channel) {
    ADC_ChannelConfTypeDef sConfig = {0};
    sConfig.Channel = adc_channel;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_144CYCLES;
    if (HAL_ADC_ConfigChannel(hadc, &sConfig) != HAL_OK) {
        Error_Handler();
    }
}

float TIM_Calculate_Period (TIM_HandleTypeDef *htim){
    uint16_t arr;

```

```

    uint16_t prescaler;
    uint32_t timer_clock;

    prescaler= htim->Instance->PSC;
    arr= htim->Instance->ARR;
    timer_clock= Get_Timer_Clock_Freq(htim);
    float pulsWidth = ((prescaler + 1) * (arr + 1))/(float)
        timer_clock;
    return pulsWidth;
}

uint32_t Get_Timer_Clock_Freq(TIM_HandleTypeDef *htim) {
    uint32_t timer_clock = 0;
    // Check which APB clock the timer is connected to
    if (htim->Instance == TIM1 || htim->Instance == TIM8 ||
        htim->Instance == TIM9 || htim->Instance == TIM10 ||
        htim->Instance == TIM11) {
        // For APB2 timers
        timer_clock = HAL_RCC_GetPCLK2Freq();
        timer_clock = timer_clock;

    } else {
        // For APB1 timers (e.g., TIM2, TIM3)
        timer_clock = HAL_RCC_GetPCLK1Freq();
    }
    return timer_clock*2;
}

void LCD_displayFloat (float flnum){
    char fltChar[7];
    sprintf(fltChar, "%.3f", flnum);
    LCD_displayText(fltChar);
}

```

4. Override Timer Callback Function

```

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    static uint16_t adcValue_k0;
    static uint16_t adcValue_k1;
    static uint16_t adcValue_k2;
    static uint16_t adcValue_k3;
    if (htim->Instance == TIM2) {
        adcValue_k0= ADC_Get_Value(&hadc1,ADC_CHANNEL_4);
        adcValue_k1=adcValue_k0;
        adcValue_k2=adcValue_k1;
        adcValue_k3=adcValue_k2;
        adcValue=(uint16_t)((adcValue_k0+adcValue_k1+
            adcValue_k2+adcValue_k3)/4);
    }
}

```

5. main Function

- (a) **Initialize LCD and display an initial message and start both TIM2 and TIM4:**

```
LCD_init();
LCD_clear();
LCD_setCursor(0, 0);
LCD_displayText("LAB5_LCD!");
HAL_Delay(1000);
LCD_clear();
HAL_TIM_Base_Start_IT(&htim2);
HAL_TIM_OC_Start(&htim4, TIM_CHANNEL_4);
```

- (b) **While loop**

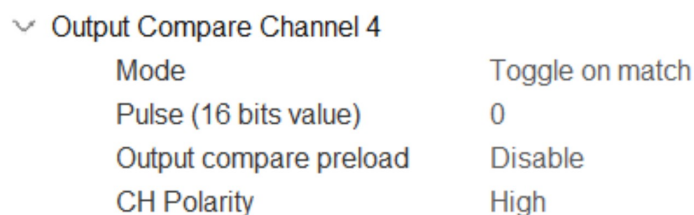
```
while (1) {
    tim4pulsWidth=TIM_Calculate_Period(&htim4);
    outFreq=1/(tim4pulsWidth*2);
    LCD_setCursor(0, 0);
    LCD_displayText("Output=_Hz");
    LCD_setCursor(0, 8);
    LCD_displayFloat(outFreq);
    htim4.Instance->ARR = ConvertADCToARR(&htim4, adcValue);
    HAL_Delay(100);
}
```

6. Some notes about the firmware code:

- The HAL_TIM_PeriodElapsedCallback function is called automatically when a timer interrupt occurs which happens when the timer reaches the Auto-Reload Register (ARR) value and overflows.
 - (a) The user can write his own implementation to define the action when the timer's period elapsed.
 - (b) This function is called whenever a timer update event occurs, which means it is important to identify which timer has triggered the interrupt. In the code, if you need to perform an action when an update event is triggered specifically by TIM2 then you can use the following condition

```
if (htim->Instance == TIM2){}
```

- The function HAL_TIM_OC_Start() is used to start the Output Compare (OC) mode for a specific timer channel. It ensures that the timer starts generating output compare events, which will trigger actions like toggling an output pin when the timer count matches a specified value (the compare value). If you remember, you already specified the mode on the configuration tool in this lab.



Output Compare Channel 4	
Mode	Toggle on match
Pulse (16 bits value)	0
Output compare preload	Disable
CH Polarity	High

Figure 4.7: Ouput Compare Configuration

- The function `TIM_Calculate_Period` used the fomula in Equation 1 to calculate the timer period. The values of PSC and ARR are getting directly from their corresponding registers.
- The function `ConvertADCToARR` convert an ADC value to an Auto-Reload Register (ARR) value for adjusting the frequency of an output compare signal
- The following line of code is used to update the ARR value directly to its corresponding register. As you see from Equation 1, you can change when you change the value of ARR.

```
htim4.Instance->ARR = ConvertADCToARR(&htim4, adcValue);
```

4.7.4 After Completing the Lab Activity:

- Enter into **debug mode**:
 1. Check PSC (Prescaler) and ARR (Auto-Reload Register) values of TIM2 and TIM4:
 - In the Debug view, open: *Window* → *ShowView* → *SFR(SpecialFunctionRegisters)*
 - Navigate to:
 - $TIMx \rightarrow PSC$
 - $TIMx \rightarrow ARR$
 2. Add `adcValueflt` and `outFreq` to **Live Expression** and observe the changes.
 3. Rotate the potentiometer slowly and observe the effect on PSC, and the variables in Live Expression.
- How can we change the timer frequency directly through the code? Give an example from the lab.
- Which register is typically modified to change the timer frequency without stopping the timer? (ARR / PSC / both?)
- Why do we change ARR instead of PSC in this lab?

4.8 Task

Write a logic to blink another LED with a frequency of 20 Hz, which means the LED should turn on and off 20 times per second

5. Experiment 5: DC Motor Control and PWM

5.1 Objectives

The objective of the DC Motor Control and PWM Lab using an STM32F4 microcontroller and L298 motor driver is to provide students with hands-on experience in developing firmware to control the speed and direction of a DC motor. Through this lab, students will learn to utilize PWM signals generated by the microcontroller and L298 motor driver for motor speed control.

5.2 Pre-Lab

1. In the STM32F4 timer configuration, which registers determine the PWM frequency and duty cycle?
2. Why can't the STM32 PWM pin directly drive a DC motor?
3. How is the potentiometer used to control the motor speed?
4. Define the duty cycle and explain its relationship to the average output voltage.
5. If the PWM frequency is 20 kHz and the duty cycle is 40%, sketch one period of the PWM signal. Label the ON and OFF durations clearly.
6. Answer the following questions about the firmware:
 - (a) What is the `MotorDirection` type? Explain why we use it to define `rotatedDir`.
 - (b) Why is float used in the following calculation? Explain what would happen if we did not use it.
`dutyCycle= (float) (TIM4->CCR1+1) / (TIM4->ARR+1) ;`
 - (c) What does the following line of code do?
`TIM4->CCR1 = ConvertADCToCCR(&htim4, adcValue) ;`

5.3 Theory

Pulse Width Modulation (**PWM**) is a method used to create a continuous digital signal that alternates between **high** and **low** states, where the signal's pulse width is modulated while maintaining a **constant frequency**.

$$\text{Frequency [PWM]} = \frac{1}{\text{period [T]}}$$

In PWM, the term "**pulse width**" refers to the duration of time during which the signal is in a high (ON) state within each period of the signal. The ratio of the pulse width to the total period is known as the duty cycle. By adjusting the **duty cycle**, PWM can

effectively control the **average power** delivered to a load.

The duty cycle is usually expressed as a percentage and represents the ratio of the time the signal is high to the total time of one cycle:

$$\text{DutyCycle [PWM]} = \frac{T_{\text{ON}}}{\text{period } [T]} \times 100 [\%]$$

The average output voltage is the duty cycle times the supply voltage:

$$V_{\text{avg}} = \text{DutyCycle} \times V_{\text{Supply}}$$

In STM32F4 microcontrollers, there is a dedicated mode for Pulse width modulation that allows for generating a signal with a **frequency** determined by the value of the ARR register and a **duty cycle** determined by the value of the CCR register.

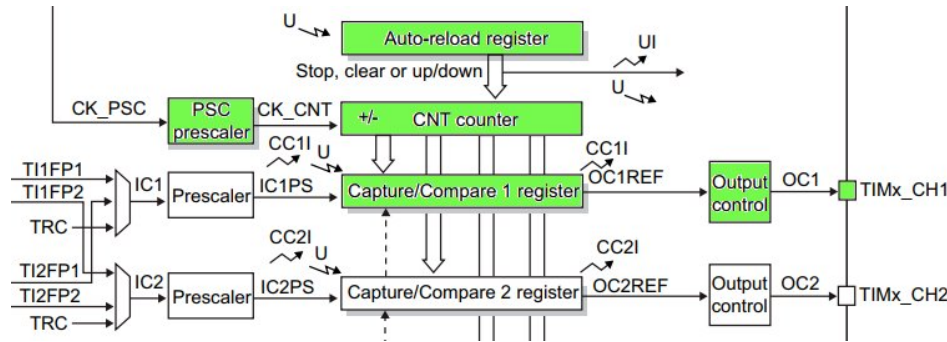


Figure 5.1: PWM registers

The frequency of PWM can be calculated by the following formula:

$$\text{Frequency [PWM]} = \frac{F_{\text{CKPSC}}}{(ARR + 1)(PSC + 1)}$$

The duty cycle can be calculated as follow:

$$\text{DutyCycle [PWM]} = \frac{CCR}{ARR} \times 100 [\%]$$

Embedded systems commonly employ this technique for tasks such as motor speed control, LED dimming, and other applications. In this experiment we will use PWM signal from the microcontroller to control the speed of a DC motor.

5.3.1 Motor control

The PWM from the microcontroller does not have enough power to drive the motor, therefore it needs an interfacing circuit to drive the motor from an external power source.

The circuit that is often used to drive the motor and control its direction is H-bridge and it consists of four bipolar transistors (switches) that deliver power to the motor Figure 5.2.

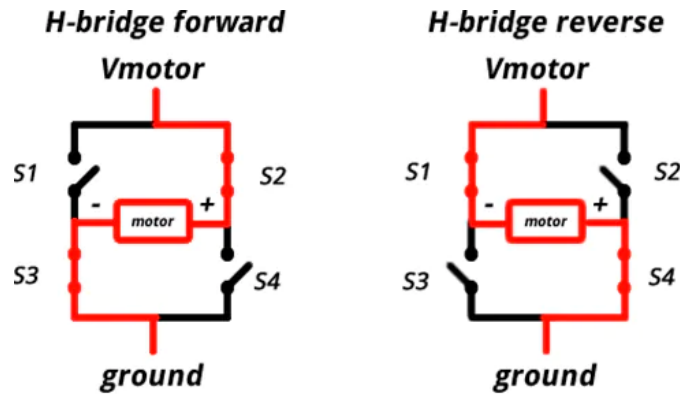


Figure 5.2: H-bridge

The basic operating mode of an H-bridge is fairly simple: if S1 and S4 are open, the right lead of the motor will be connected to the power supply, while the left lead is connected to ground. Current starts flowing through the motor which energizes the motor in (let's say) the forward direction and the motor shaft starts spinning.

In this experiment, we will use L298 bridge to drive the motor and control its direction. This driver has two H-bridges and can drive two motors. The circuit diagram of this circuit is shown in Figure 5.3:

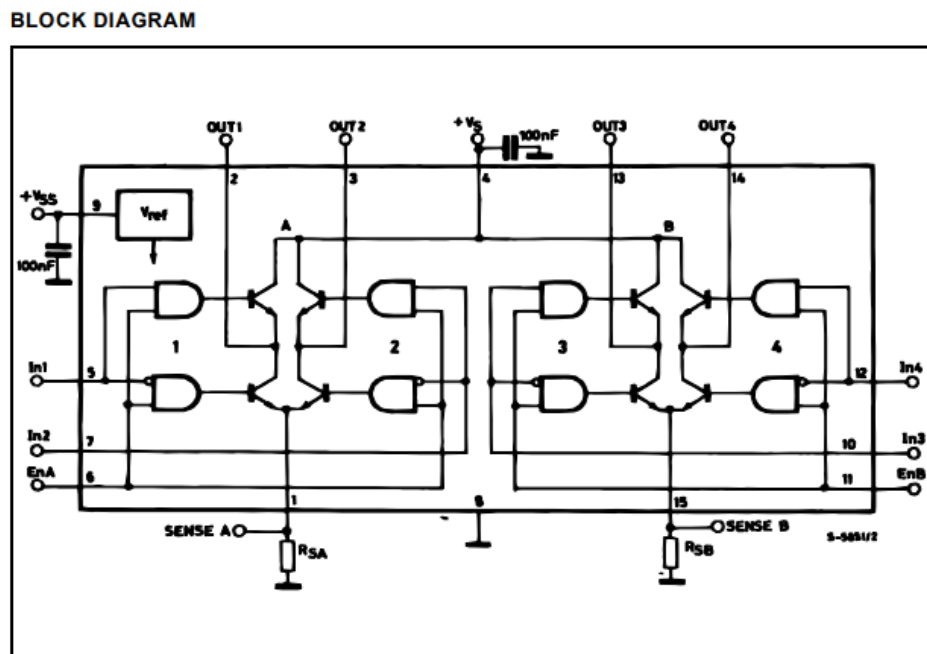


Figure 5.3: L298 Block diagram

5.4 Materials and Equipment

- Breadboard
- STM32F4 evaluation board
- Jumper Wires
- LEDs and push buttons
- Potentiometer
- L298 interfacing board
- DC motor
- External power supply
- 16x2 character LCD

5.5 Relevant HAL APIs

- **Timer HAL APIs:**

```
HAL_TIM_PWM_Start(&htimx, TIM_CHANNEL_x);
```

5.6 Activity

- Develop firmware to control the speed and direction of a DC motor using a potentiometer and push button.

5.7 Procedure

5.7.1 Hardware Setup

1. Connect the potentiometer to an analog input pin (PA5) of the STM32F4 microcontroller
2. Connect L298 Motor Driver
 - (a) Connect the L298 dual H-bridge motor driver to the STM32F4 microcontroller and the motor according to the following schematic.
 - (b) Make sure that H-bridge and STM3F4 have a common ground

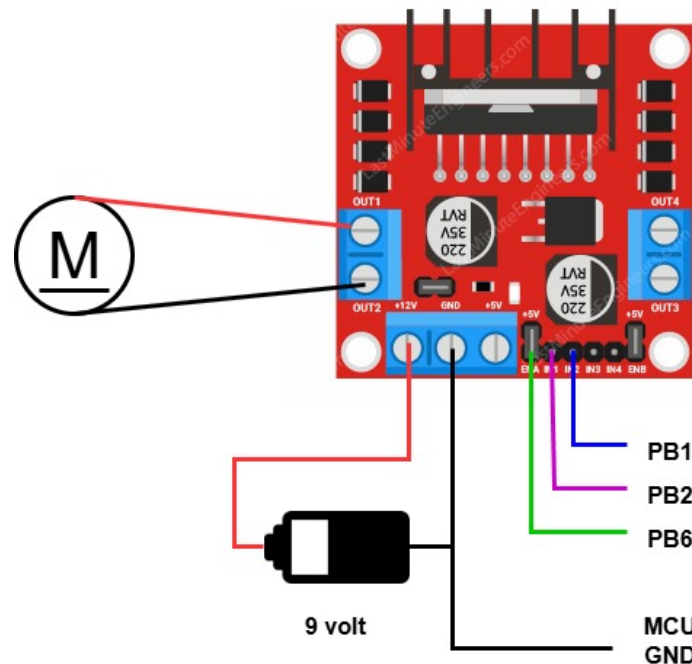


Figure 5.4: L298 Driver Connection Diagram

5.7.2 Peripheral Configuration

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your STM32F4 microcontroller.

2. Configure the Clock:

- Enable the external clock source from the **RCC tab** and set the maximum clock frequency in the **Clock Configuration** window (refer to previous labs for guidance). Ensure that the **APB1 Timer Clocks** frequency is set to
 - 100 MHz (for Nucleo board)
 - 50 MHz (for Discovery board)

3. Configure GPIO Pins:

- Set up the ADC to read input from channel 5 using the same configuration used in **Lab 4**.
- Configure **PB1** and **PB2** as outputs with the following user labels “IN1” and “IN2”.
- Configure **PC13** as an input and add the following user label “BluePushButton”.
- Configure the TIM4 timer to use the internal clock source and set the appropriate **Prescaler** and **ARR** values to get a frequency of 20 KHz.

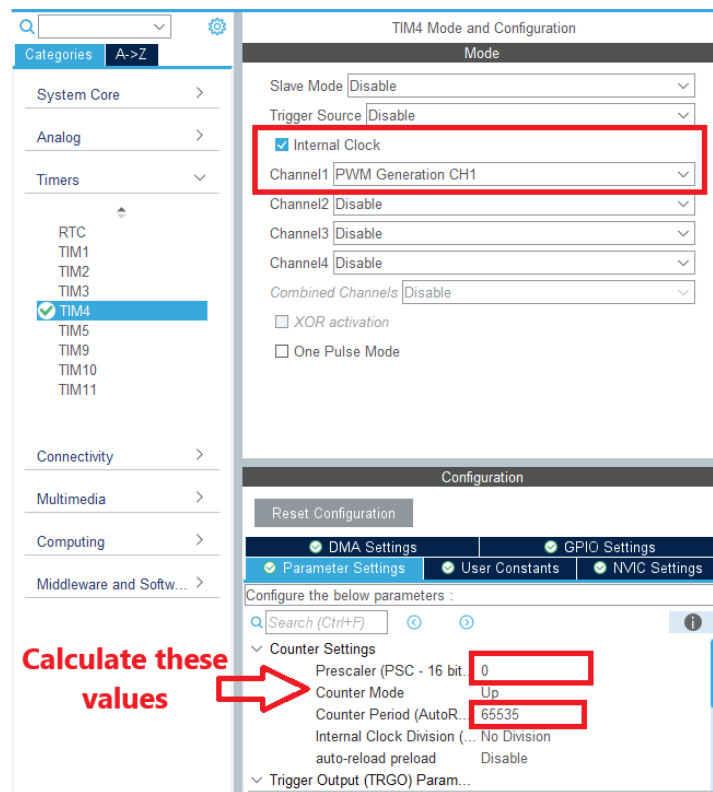


Figure 5.5: PWM Mode

- Generate code to produce the necessary initialization

5.7.3 Software Code

1. Include math.h library
2. Reuse ADC_Get_Value() and ADC_Select_Channel() functions implemented in Lab 3
3. Add the following typedef (outside the main function)

```
/* USER CODE BEGIN PTD */
typedef enum {
    CCW,
    CW
}MotorDirection;
/* USER CODE END PTD */
```

4. Define local variables

```
uint16_t adcValue;
float dutyCycle;
MotorDirection rotatedDir=CCW;
uint32_t adcValue_K0=0;
uint8_t BlueButtonState;
uint32_t currentTick=0;
```

```
uint32_t previousTick=0;
```

5. Start the timer in PWM mode using

```
HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
```

6. Add the following function with its prototype

```
uint32_t ConvertADCToCCR (TIM_HandleTypeDef *htim, uint32_t
    adc_value){

    uint32_t timPeriod= htim->Init.Period;
    uint32_t CCR_value;
    float DutyCycle = (float)(adc_value / 4095.0f);
    CCR_value= (DutyCycle*(float)(timPeriod+1));
    return CCR_value;
}
```

7. While loop

```
BlueButtonState=HAL_GPIO_ReadPin(BluePushButton_GPIO_Port,
    BluePushButton_Pin);
currentTick =HAL_GetTick();
if (BlueButtonState==RESET && (currentTick-previousTick)>150){
    rotateDir=!rotateDir;
    previousTick=currentTick;
}

if (rotateDir==CCW){
    HAL_GPIO_WritePin(IN1_GPIO_Port, IN1_Pin, SET);
    HAL_GPIO_WritePin(IN2_GPIO_Port, IN2_Pin, RESET);
}
else {
    HAL_GPIO_WritePin(IN1_GPIO_Port, IN1_Pin, RESET);
    HAL_GPIO_WritePin(IN2_GPIO_Port, IN2_Pin, SET);
}
adcValue= ADC_Get_Value(&hadc1,ADC_CHANNEL_5);

if(abs(adcValue-adcValue_K0)>30 && adcValue >900){
    TIM4->CCR1=ConvertADCToCCR(&htim4,adcValue);
}
else if (adcValue<800){
    TIM4->CCR1=0;
}
dutyCycle= (float)(TIM4->CCR1+1)/(TIM4->ARR+1);
adcValue_K0 = adcValue;
}
```

5.7.4 After Completing the Lab Activity:

1. Enter into **debug mode**:
 - (a) Check PSC (Prescaler), ARR (Auto-Reload Register), and CCR (Capture-Compare Register) values of TIM4:
 - In the Debug view, open: *Window* $\rightarrow$ *ShowView* $\rightarrow$ *SFR(SpecialFunctionRegisters)*
 - Navigate to:
 - $TIMx \rightarrow PSC$
 - $TIMx \rightarrow ARR$
 - $TIMx \rightarrow CRR$
 - (b) Can we add `rotatedDir` variable to **Live Expression** in debug mode? If not, how can we add it?
2. What is the equation between ADC value and CCR ? Can you show it using debug view?
3. How does the CCR value affect the speed of the motor?
4. Did we used interrupt or polling to detect the button input?

5.8 Task

Write a logic to print the real-time value of duty cycle on an LCD

6. Experiment 6: Rotary Encoder Interface

6.1 Objectives

The goal of the Encoder Interface Lab using an STM32F4 microcontroller is to provide the students with practical experience in writing firmware for interfacing with a rotary incremental encoder. Through this lab, students will learn how to implement the timer encoder mode and calculate both position and speed from the encoder's input signals.

6.2 Pre-Lab

1. Explain the key difference between an **absolute encoder** and an **incremental (quadrature) encoder**. What is the main disadvantage of an incremental encoder?
2. Observing Figure 6.1 (Quadrature encoder working principle), if Channel A leads Channel B (i.e., Channel A's rising edge occurs before Channel B's), is the encoder rotating clockwise or counter-clockwise? Justify your answer
3. How does the STM32's timer in Encoder Mode determine the direction of rotation (counting up or down) from the two input signals (TI1 and TI2)?
4. The lab uses an encoder with 30 PPR
 - (a) How many counts (pulses) will the timer register for one full revolution of the encoder shaft when configured to count on both TI1 and TI2 edges?
 - (b) Using your answer from (a), calculate the angular resolution (degrees per count) of the system.
5. The code uses a timer (TIM4) interrupt every 100 μ s to calculate speed (RPM). Explain the formula used for RPM calculation in your own words. Why is the change in angle (`rotationAngle - prevRotationAngle`) used instead of the absolute angle?

6.3 Theory

Encoders are electronic devices used in many different applications. They can act as sensors to measure motor speed, position and direction, as well as distances and lengths. Unlike potentiometers, encoders can be turned endlessly, making them great input devices for users. Programmers can also detect how fast a user turns the knob, allowing the system to respond accordingly.

You can find rotary encoders in many types of equipment and everyday electronics. Rotary encoders are used in devices such as volume control knobs in audio equipment or in navigation controls on car dashboards.

6.3.1 Incremental encoder

The rotary encoder which we'll be using in this tutorial is an **Incremental Encoder** and it's also called a **Quadrature Encoder**. Unlike an absolute encoder, an incremental encoder does not indicate absolute position. It only tracks and reports changes in position and the direction of movement for each change.

The rotary encoder has two output signals, A and B, which issue pulses when the device is moved. Each channel provides a specific number of equally spaced **pulses per revolution (PPR)** and the direction of motion is detected by the phase relationship of one channel leading or trailing the other channel as showing in Figure 6.1.

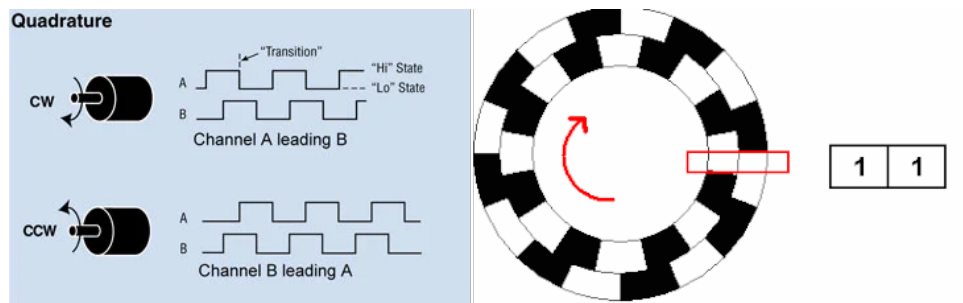


Figure 6.1: Quadrature encoder working principle

6.3.2 STM32 Encoder Mode

As we learnt from previous labs, the timer modules in STM32 can operate in many modes such as regular counter, output compare mode, and PWW mode. In today's experiment we will work with a new mode called **Encoder Mode**. The timer will count based on the input signals from the encoder. You need to select the appropriate configurations, and this mode will detect the signals and decide the direction whether it is counting up or down based on the leading and trailing phase.

6.3.3 Working Principle

As shown in Figure 6.2, the two inputs TI1 and TI2 are used to interface to an incremental encoder. The counter is clocked by each valid transition on TI1FP1 or TI2FP2 (TI1 and TI2 after input filter and polarity selection). The sequence of transitions of the two inputs is evaluated and generates count pulses as well as the direction signal.

Depending on the sequence the counter counts up or down, the DIR bit in the TIMx_CR1 register is modified by hardware accordingly. The DIR bit is calculated at each transition on any input (TI1 or TI2), whatever the counter is counting on TI1 only, TI2 only, or both TI1 and TI2.

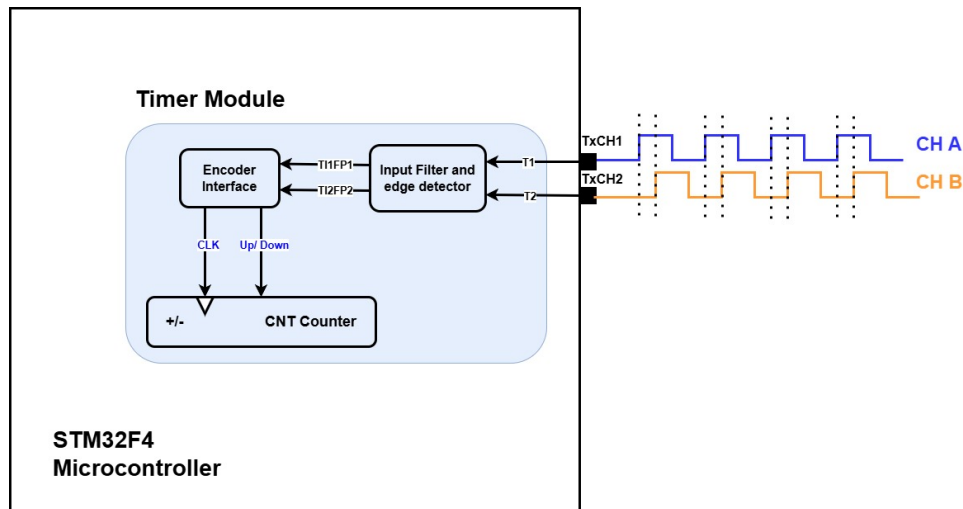


Figure 6.2: Counter operation in encoder interface mode

6.3.4 Angle Measurement

The angle can be calculated based on the encoder resolution. For example, if we have an encoder with a resolution of 30 PPR (pulses per revolution), as in the encoder used in this experiment, one complete revolution corresponds to 30 counts. By dividing 360° by the number of pulses per revolution (30), we can determine the angle represented by each count (pulse).

6.3.5 Speed Measurement

As you know, speed can be calculated by dividing the change in angle by the change in time. The change in angle is determined based on the encoder's input signals. To measure the change in time, we need to use another timer that is configured to generate interrupts at a fixed interval (i.e. $100\ \mu\text{s}$). By dividing the change in angle by this interval, we can calculate the speed.

6.3.6 HW-040 rotary encoder module

We will be using HW-040 rotary encoder module (as shown in Figure 6.3) in this experiment to interface to STM32F4 microcontroller. It is a type of rotary encoder, commonly used as a rotating input knob. It features 30 pulses per revolution (PPR) and includes a built-in push-button switch, activated when the knob is pressed.

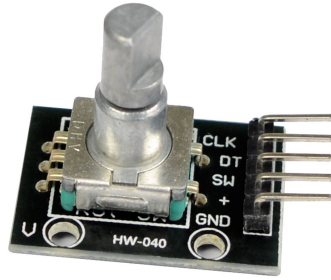


Figure 6.3: Encoder Module

The module has the following pins:

CLK	Encoder Pin A
DT	Encoder Pin B
SW	No push-button switch
VCC (+)	Voltage input (+5V)
GND	Ground (Encoder Pin C)

6.4 Materials and Equipment

- Breadboard
- STM32F4 evaluation board
- Jumper Wires
- LEDs and push buttons
- 16x2 character LCD
- HW-040

6.5 Relevant HAL APIs

- **Timer HAL APIs:**

```
HAL_TIM_Encoder_Start(&htimx, TIM_CHANNEL_ALL);
HAL_TIM_Base_Start_IT(&htimx);
```

6.6 Activity

Develop firmware to read the angle and the speed of the encoder input signal and display it to an LCD in real-time.

6.7 Procedure

6.7.1 Hardware Setup

1. Connect LCD via I2C

- (a) Connect SCL and SDA pins of the LCD adapter board to PB6 and PB7 respectively.
- (b) Connect +5V and GND to the LCD adapter board.

2. Connect HW-040 to STM32F4 pins

- (a) Connect the HW-040 module to the STM32F4 microcontroller and the motor according to the following schematic (Figure 6.3).

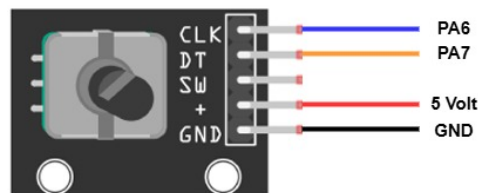


Figure 6.4: HW-040 Connection to STM32F4

6.7.2 Peripheral Configuration

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your STM32F4 microcontroller.

2. Configure the Clock:

- Enable the external clock source from the **RCC tab** and set the maximum clock frequency in the **Clock Configuration** window (refer to previous labs for guidance). Ensure that the **APB1 Timer Clocks** frequency is set to
 - (a) 100 MHz (for Nucleo board)
 - (b) 50 MHz (for Discovery board)

3. Configure I2C

- Set up the I2C (PB6 and PB7) to communicate with LCD using the configurations used in previous Labs.

4. Configure TIM3 (Encoder Mode)

- (a) Configure the TIM3 timer to work on encoder channel mode

TIM3 Mode and Configuration

Mode	
Slave Mode	Disable
Trigger Source	Disable
Clock Source	Disable
Channel1	Disable
Channel2	Disable
Channel3	Disable
Channel4	Disable
Combined Channels	Encoder Mode
☐ Use ETR as Clearing Source	
☐ XOR activation	
☐ One Pulse Mode	

Figure 6.5: TIM3 Mode and Configuration

- (b) Configure the encoder counter to count on both TI1 and TI2 rising edge and enter a value of 10 for the Input filter coefficient.

Configuration

Reset Configuration

☒ DMA Settings ☒ GPIO Settings
☒ Parameter Settings ☒ User Constants ☒ NVIC Settings

Configure the below parameters :

Search (Ctrl+F)

Counter Settings
 Prescaler (PSC - 16 bit... 0
 Counter Mode Up
 Counter Period (AutoRel... 65535
 Internal Clock Division (... No Division
 auto-reload preload Disable

Trigger Output (TRGO) Parame...
 Master/Slave Mode (MS... Disable (Trigger input effect not del...
 Trigger Event Selection Reset (UG bit from TIMx_EGR)

Encoder
 Encoder Mode Encoder Mode TI1 and TI2
 Parameters...
 Polarity Rising Edge
 IC Selection Direct
 Prescaler Division Ratio No division
 Input Filter 10
 Parameters...
 Polarity Rising Edge
 IC Selection Direct
 Prescaler Division Ratio No division
 Input Filter 10

Figure 6.6: Encoder Configuration

5. Configure TIM4 (Encoder Mode)

- (a) Configure the TIM4 timer to use the internal clock source and set the appropriate Prescaler and ARR values for 100 μ s period.

TIM4 Mode and Configuration

Mode

Slave Mode: Disable

Trigger Source: Disable

Clock Source: Internal Clock

Channel1: Disable

Channel2: Disable

Channel3: Disable

Channel4: Disable

Combined Channels: Disable

☐ Use ETR as Clearing Source

☐ XOR activation

Configuration

Reset Configuration

✓ NVIC Settings ✓ DMA Settings

✓ Parameter Settings ✓ User Constants

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

- Prescaler (PSC - 16 bit...): 100-1
- Counter Mode: Up
- Counter Period (AutoRel...): 100
- Internal Clock Division (...): No Division
- auto-reload preload: Disable

Trigger Output (TRGO) Param...

- Master/Slave Mode (MS...): Disable (Trigger input effect not del...)
- Trigger Event Selection: Reset (UG bit from TIMx_EGR)

Figure 6.7: TIM3 Mode and Configuration

- (b) Enable TIM4 interrupt signal from NVIC setting tab.

Configuration

Reset Configuration

✓ NVIC Settings ✓ DMA Settings

✓ Parameter Settings ✓ User Constants

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM4 global interrupt	✓	0	0

Figure 6.8: TIM4 NVIC Settings

- (c) Generate code to produce the necessary initialization

6.7.3 Software Code

1. Include the required header files and define global variables

```
uint32_t counterValue;
uint32_t prevCounterValue=0;
volatile long long int encoderCount=0;
volatile long long int encoderPosition=0;
volatile double rotationAngle;
volatile double prevRotationAngle=0.0;
volatile double RPMvalue;
volatile uint32_t delatT;
volatile uint8_t encoderDir;
```

2. Add the required code for the LCD to display the speed and the angle of the encoder signal.

3. Start TIM3 and TIM4

```
HAL_TIM_Encoder_Start(&htim3, TIM_CHANNEL_ALL);
HAL_TIM_Base_Start_IT(&htim4);
```

4. Write your implementation of the timer callback function

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if (htim->Instance == TIM4) {
        delatT++; // each time interrupt occurs, T2 increases
                  by 100 us
        counterValue = __HAL_TIM_GET_COUNTER(&htim3);
        encoderDir = __HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3);
        if ( prevCounterValue-counterValue !=0 ){
            if (encoderDir == 1 ){
                encoderCount--;
            }
            else {
                encoderCount++;
            }
        }
        encoderPosition=encoderCount/2; //at the rising
                                         edges of TI1 and TI2
        rotationAngle=((double)encoderPosition/30.0)*360.0;
        if (rotationAngle!=prevRotationAngle){
            RPMvalue=(rotationAngle-prevRotationAngle);
            RPMvalue=RPMvalue *60*1000000 /((float)delatT
            *100*360);
            delatT=0;
        }
        else if(delatT>10000){
            RPMvalue=0;
        }
        prevRotationAngle=rotationAngle;
        prevCounterValue=counterValue;
```

```

    }
}

```

6.8 Task

Write logic to control a PWM signal connected to an LED, where the duty cycle is proportional to the angle reading from the encoder (0—360).

For example, if the angle reading is 480°, the duty cycle should be calculated as:

$$\left(\frac{480\%360}{360}\right) \times 100 = \left(\frac{120}{360}\right) \times 100 = 33.33\%$$

Hint: To implement this logic, you will need to define the following variables:

```
float angle2DC;
float dutyCycle2CCR;
```

Use the following formula to calculate the duty cycle from the angle:

```
angle2DC = (float)((int)rotationAngle % 360) / 360;
```

```
dutyCycle2CCR= (TIMx->ARR + 1) * angle2DC - 1;
```

Assign the calculated duty cycle to the timer's CCR register using the formula:

```
TIMx->CCRx = abs((int) dutyCycle2CCR);
```

Replace x and y with the appropriate timer and channel numbers used in your configuration.

6.9 After Completing the Lab Activity:

1. The code uses `encoderPosition = encoderCount / 2;` before calculating the angle. Explain why this division by 2 is necessary.
2. In the provided callback function, the variable `delatT` is reset to 0 after 1 second (if(`delatT>10000`)). What is the purpose of this logic? What problem does it solve in the RPM calculation?
3. With a 30 PPR encoder and the timer counting on both edges, what is the smallest rotational speed (RPM) your system can measure? (Hint: Consider the smallest detectable angle change and the 100 μ s measurement interval). How could you modify the setup to measure slower speeds more accurately?

7. Experiment 7: Stepper Motor Control

7.1 Objectives

The lab aims to demonstrate stepper motor control using STM32F4 microcontroller and ULN2003 Darlington driver. This will allow students to develop firmware to control the position, speed and direction of a stepper motor using full-step and half-step modes.

7.2 Pre-Lab

1. Why do we use a timer in this experiment?
2. Navigate to `stepper.c` and explain the role of each function in the file.
3. Write the mathematical relationship between the motor step speed (steps/second) and the timer parameters.
4. Describe, in simple words, how to write a program to control a stepper motor without using any external library.

7.3 Theory

7.3.1 Stepper Motor Working Principle

A stepper motor is a type of electric motor designed to move in discrete steps. Unlike conventional DC motors, which rotate continuously, stepper motors rotate in precise increments.

The movement is achieved through the use of magnetic field created by coils and interacting with magnetics. When one of the coils is energized, it generates a magnetic field. By energizing the coils in cyclical sequence (by means of input pulses), then the magnetic field will vary. When a magnetic is placed in this fluctuating field, it will align itself in the lowest energy state (equilibrium), producing movement.

This concept is the foundation of the stepper motor, which consists of a stationary part (the stator) made of cyclically energized coils and a rotating part (the rotor) made of ferromagnetic material or magnetic. Figure 7.1 shows Permanent magnet stepper motor

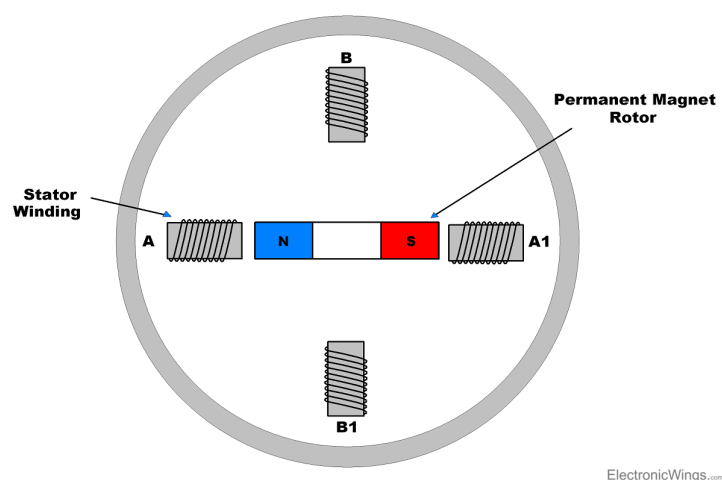


Figure 7.1: Permanent magnet stepper motor

7.3.2 Stepper Motor Working Principle

Stepper motors typically have 2 phases, though some designs may use 3 or 5 phases. Usually in **bipolar stepper motors** (Figure 7.2-a), each phase has a single winding, and the current in the winding can flow in both directions. **Unipolar stepper motors** (Figure 7.2-b) also have one winding per phase, but each winding has a center tap; hence, half of the winding is used for positive current flow and half for negative current flow.

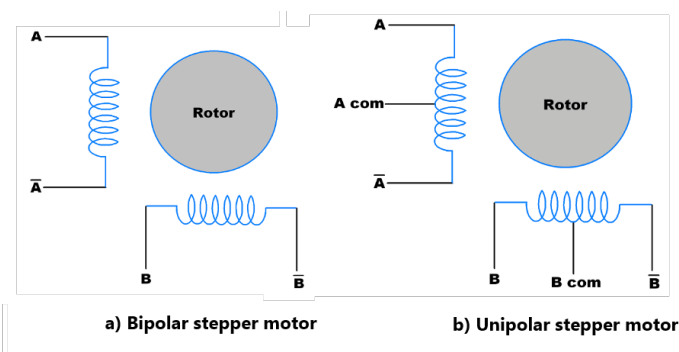


Figure 7.2: Bipolar and unipolar stepper motors

7.3.3 Stepper Motor Working Principle

A stepper motor moves in discrete steps, and to achieve continuous or limited-angle rotation, sequential steps must be provided. Typically, two main step sequences are used to control the rotation of a stepper motor: **full step** and **half step** as shown in Figure 7.3, Table 1 and Table 2.

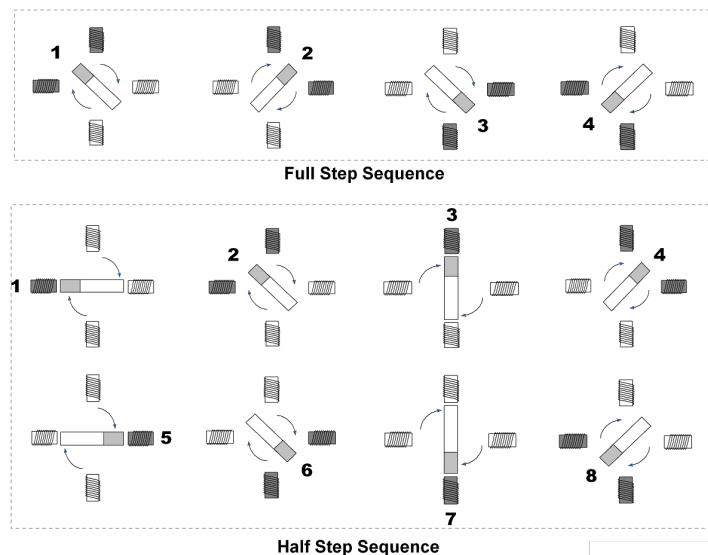


Figure 7.3: Stepper motor control (source: <https://www.electronicwings.com/sensors-modules/stepper-motor>)

Table 1: Full step drive sequence

Step	IN1	IN2	IN3	IN4
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

Table 2: Half step drive sequence

Step	IN1	IN2	IN3	IN4
1	1	0	0	0
2	1	1	0	0
3	0	1	0	1
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

7.3.4 28BYJ-48 stepper motor

In this experiment, we will be using 28BYJ-48 unipolar stepper motor with 32 steps per revolution, that means it has a step angle of 11.25° . However, this motor is connected to an internal small gearbox with a gear ratio of $1/63.68$. Therefore, to achieve a full revolution we need $63.68 \times 32 = 2038$ steps.

The 28BYJ-48 (Shown in Figure 7.4) has two coils, each of which has a center tap. These two center taps are connected internally and brought out as the 5th wire (red wire).

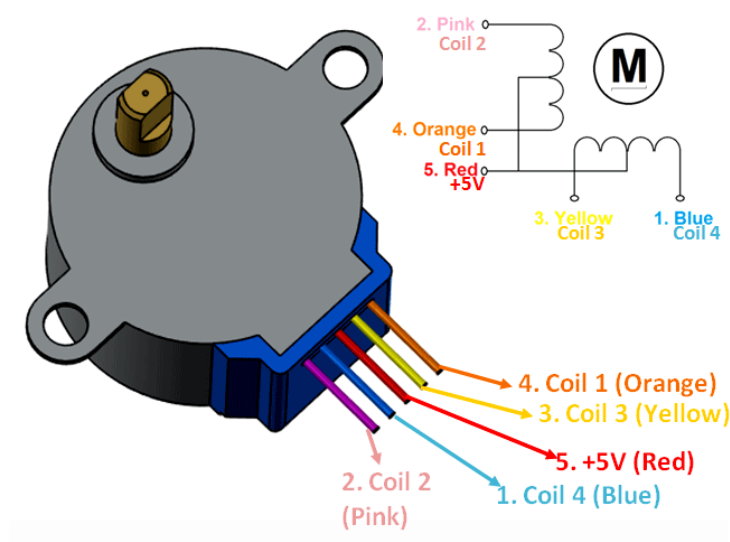
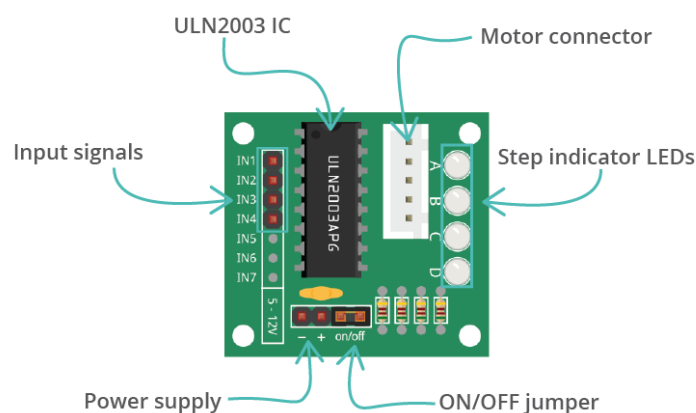


Figure 7.4: 28BYJ-48 stepper motor

The rated operating voltage for this motor is 5-volt and it should not be powered from the microcontroller board, so make sure you use an external power source to drive the motor.

7.3.5 Drive Circuitry

As you know, microcontroller pins are not designed to deliver high-power outputs and therefore cannot drive the stepper motor directly. Hence, to control the motor, a driver IC is required to drive the motor from the control signals generated by the microcontroller. In this experiment, we will use ULN2003 IC (Figure 7.5 shows ULN2003-based driver board). It consists of an array of Darlington transistors that provide high current gain. This allows the low voltage signal of the microcontroller to drive a high current stepper motor.

Figure 7.5: ULN2003-based driver board (source: <https://randomnerdtutorials.com/>)

7.4 Materials and Equipment

- Breadboard
- STM32F4 evaluation board
- Jumper Wires
- LEDs and push buttons
- 16x2 character LCD
- 28BYJ-48 stepper motor
- ULN2003-based driver board

7.5 Relevant External Library Functions

- **Stepper Motor:** The following APIs are used to initialize and move the stepper motor, note that you need to enable a timer and its interrupt to

```
Stepper_Init(); //Initializes all stepper motor variables and
                stores the timer used for step timing.
Stepper_Update(); //Executes one step of the motor and must be
                 called by timer
Stepper_MoveByDegree(); //Converts a desired rotation angle into
                       the corresponding number of steps
```

7.6 Activity

Implement a firmware-based system to drive a stepper motor, and control its position, direction, and speed using an STM32F4 microcontroller.

7.7 Procedure

7.7.1 Hardware Setup

1. Connect Stepper motor

- (a) Connect ULN2003 board to the stepper motor and to GPIO pins of STM32F4 microcontroller according to the following schematic (Figure 7.6):

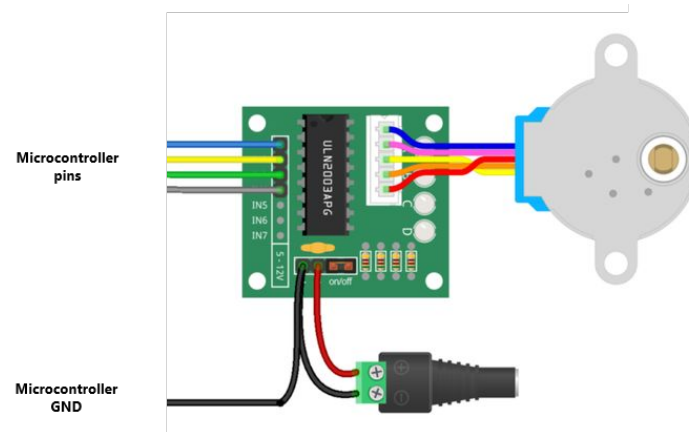


Figure 7.6: Stepper Motor and STM32F4 connection

- (b) Make sure the power supply and the microcontroller have a common ground

7.7.2 Peripheral Configuration

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your STM32F4 microcontroller.

2. Configure the Clock:

- (a) Enable the external clock source from the **RCC tab** and set the maximum clock frequency in the **Clock Configuration** window (refer to previous labs for guidance). Ensure that the **APB1 Timer Clocks** frequency is set to
- i. **100 MHz** (for Nucleo board)
 - ii. **84 MHz** (for Discovery board)

3. Configure GPIO Pins:

- Configure 4 GPIO pins as outputs with the following user labels “IN1”, “IN2”, “IN3”, and “IN4”
 - If you use STM32F411 Nucleo board, avoid using the following pins: (PA2, PA3, PC14, PC15, PA13, and PA14)
 - If you use STM32F4 Discovery board, avoid using the following pins (PA13 and PA14)

4. Configure Timer 4:

- (a) Configure the TIM4 timer to use the internal clock source and set the Prescaler to:
- (100-1) for Nucleo
 - (84-1) for Discovery
- (b) Enable TIM4 interrupt signal from NVIC setting tab.

7.7.3 Software Code

1. Add the stepper library files found in this [link](#) to your project.
2. Use `Stepper_Init()` function to initialize stepper variables and assign a timer for step timing.
3. Use `Stepper_Update()` function must be called by the timer ISR, it executes one step at a time until all steps are done
4. The following is an example of sending a command to the stepper motor using `Stepper_MoveByDegree()` function.

```
/* USER CODE BEGIN 2 */  
Stepper_MoveByDegree(64*360, 0, 500); // maximum speed is 500  
    steps/sec  
while (stepsRemaining!=0);  
/* USER CODE END 2 */
```

7.8 Task

Write a program to control

1. The direction via a push button
2. The speed through potentiometer

7.9 After Completing the Lab Activity:

1. Change between half step and full step modes
2. Explain how changing the timer's ARR value affects the motor speed.

8. Experiment 8: UART Protocol

8.1 Objectives

The goal of the UART lab is to enable students to develop firmware code to initialize and communicate between systems via UART interface using STM32F4 microcontroller.

8.2 Pre-Lab

- Why is UART called an asynchronous communication protocol?
- What is the function of the start bit in UART communication?
- Explain the difference between: Full-duplex, Half-duplex, and Simplex? Which mode is used in this experiment?
- Why must both UART devices have the same configuration (baud rate, parity, stop bits, data bits)?
- Why is interrupt-based UART communication preferred over polling in embedded systems?
- Why is it necessary to use a common ground between the two microcontrollers?

8.3 Theory

UART stands for Universal Asynchronous Receiver /Transmitter and defines a protocol, or set of rules, for exchanging serial data between two devices. In this experiment, we will use UART to communicate with two microcontrollers. UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions as shown in the following figure:

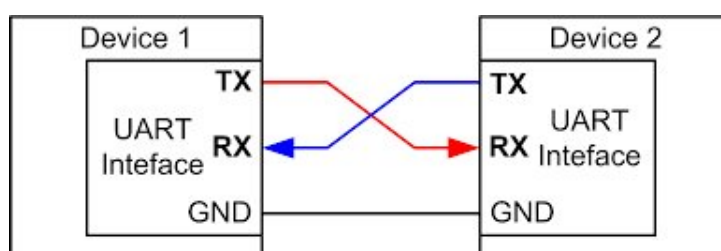


Figure 8.1: UART signals connection

Data transfer in UART can be done in frames. Each frame is composed of the **start bit** and **stop bit**, **data bits**, and an *optional parity bit*.

There can be 5 to 9 user data bits, although 7 or 8 bits is most common. These data bits are usually transmitted with the least significant bit first.

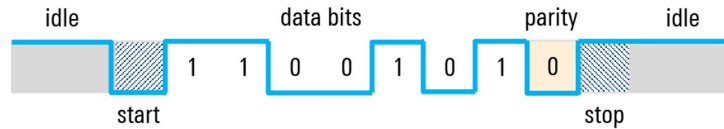


Figure 8.2: UART frame

A UART frame can also contain an optional parity bit that can be used for error detection. This bit is inserted between the end of the data bits and the stop bit

The value of the parity bit depends on the type of parity being used (even or odd):

- In even parity, the bits whose value is 1 are counted. If that count is odd, the parity bit value is set to 1
- In odd parity, the bits whose value is 1 are counted. If that count is even, the parity bit value is set to 1

Table 3: Example of even and odd parity

Data(7-bit)	Count of 1-bits	Even parity	Odd parity
0000000	0	0	1
1010001	3	1	0
1101001	4	0	1
1111111	7	1	0

UART can work in either full duplex, half duplex, or simplex modes.

- In the **full duplex mode**, there are two pins called Rx (receive) and Tx (transmit).
- In the **half duplex mode**, the Tx pin is used for both Tx and Rx.
- In the **simplex mode**, the Tx pin of one device is connected to the Rx pin of the other device.

The master and slave definitions do not apply to UART and are generally preferred when fast data transfer speed is not needed. The most common UART baud rates in use today are 4800, 9600, 19.2K, 57.6K, and 115.2K.

For two devices to communicate successfully through UART, they must have the same configuration settings. Both devices must use the same baud rate, data bits, parity bit, and stop bits.

The STM32F4 microcontroller has four UART modules called UART4, UART5, UART7, and UART8. It also has four USART modules USART1, USART2, USART3, and USART6. These modules can be used in UART mode.

8.4 Materials and Equipment

- Breadboard
- 2x STM32F4 evaluation board
- Jumper Wires
- 16x2 character LCD

8.5 Relevant External Library Functions

- **UART HAL APIs**

```
HAL_UART_Transmit();  
HAL_UART_Receive_IT();  
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){}\\\n    HAL_UART_RxCpltCallback automatically called when a UART\n    receive operation finishes
```

8.6 Activity

Develop firmware to count push-button connected to microcontroller A using interrupt and transmit the button press count via UART to microcontroller B and display the received counts on an LCD.

8.7 Procedure

8.7.1 Hardware Setup

1. Connect Microcontroller A and B together

- (a) Connect Tx pin (PC6) of Microcontroller A to Rx pin (PC7) of Microcontroller B.
- (b) Connect Rx pin (PC7) of Microcontroller A to Tx pin (PC6) of Microcontroller B.
- (c) Connect +5V of the two microcontrollers together.
- (d) Ensure that both devices have a common ground.

2. Connect LCD via I2C to Microcontroller B

- (a) Connect SCL and SDA pins of the LCD adapter board to PB6 and PB7 respectively.
- (b) Connect +5V and GND to the LCD adapter board

8.7.2 Peripheral Configuration

• Microcontroller A

1. Create a New STM32 Project, enable the external clock source from the RCC tab and set the maximum clock frequency.
2. **Configure the UART Peripheral:**
 - (a) In the "Pinout & Configuration" tab, configure the UART6 pins (PC6 and PC7)
 - (b) Go to the "Configuration" tab and enable the USART/UART peripheral. Set the baud rate, data bits, stop bits, and parity as follows.

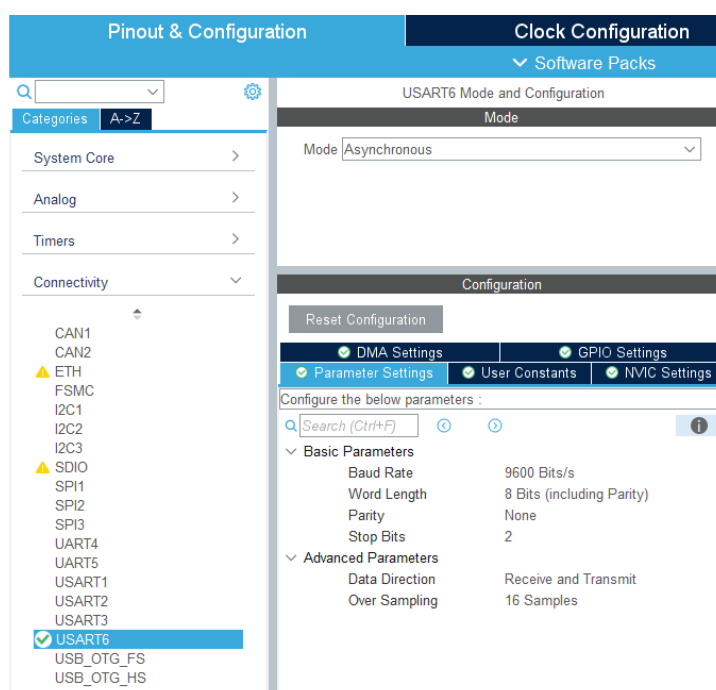


Figure 8.3: UART Configuration

- (c) Enable UART interrupt from NVIC setting tab.
3. **Configure GPIO Pins:**
 - If you use Nucleo board,
 - * Assign pin PC13 as an external interrupt (GPIO_EXTIx) and select falling edge trigger selection
 - If you use Discovery board,
 - * Assign pin PA0 as an external interrupt (GPIO_EXTIx) and select rising edge trigger selection
 - Add the following user label “BluePushButton” to EXTI pin.
 - Enable the interrupt for the corresponding EXTI line in NVIC settings.
4. **Set interrupt priority:**
 - Select a unique priority number for each interrupt.
 - Generate code to produce the necessary initialization

- **Microcontroller B**

1. Repeat points 1-4 from the Peripheral Configuration of Microcontroller A.
2. **Configure GPIO Pins:**
 - Set up the I2C (PB6 and PB7) to communicate with LCD using the configurations used in previous Labs.

8.7.3 Software Code

- **Microcontroller A**

1. Write your customized EXTI callback function to count the presses with de-bouncing logic in interrupt mode.
2. Add the following function that sends integer values:

```
void UART_SendUint32(uint32_t value) {
    uint8_t buffer[5];
    buffer[0] = (value >> 24) & 0xFF; // MSB
    buffer[1] = (value >> 16) & 0xFF;
    buffer[2] = (value >> 8) & 0xFF;
    buffer[3] = value & 0xFF; // LSB
    buffer[4] = '\\0'; // LSB
    HAL_UART_Transmit(&huart6, buffer, 5, HAL_MAX_DELAY);
}
```

3. Write the logic to send the press count via UART **when count change is detected**.

- **Microcontroller B**

1. Include the required LCD header files.
2. Define global variables.

```
volatile uint32_t receivedValue = 0;
uint8_t rxBuffer[5];
```

3. Initialize the LCD and start receiving UART messages in interrupt mode.

```
LCD_init();
LCD_clear();
LCD_setCursor(0, 0);
LCD_displayText("PressCount=");
HAL_UART_Receive_IT(&huart6, (uint8_t *)rxBuffer, 5);
```

4. Override the UART callback function:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART6) { // Check if the
        interrupt is for USART1
    }
}
```

```

        rxBuffer[4] = '\0'; // Ensure the buffer is null
                             -terminated (safe)
        // Reconstruct uint32_t value from received
        bytes
        receivedValue = ((uint32_t)rxBuffer[0] << 24) |
                         ((uint32_t)rxBuffer[1] << 16) | ((uint32_t)
                         rxBuffer[2] << 8) | (uint32_t)rxBuffer[3];
        // Start listening for the next uint32_t value
        HAL_UART_Receive_IT(&huart6, (uint8_t *)rxBuffer
                             , 5);
    }
}

```

5. Add the required code for the LCD to display the number of counts received from **microcontroller A** via UART.

8.8 After Completing the Lab Activity:

1. Why was the button press count sent only when the value changed?
2. Try changing the UART baud rate on only one of the two microcontrollers while keeping the other unchanged. What problems occur due to the baud rate mismatch?
3. Run the program in debug mode on **Microcontroller B**. Add the variable that stores the button press count to the **Live Expressions** window, then press the push button and observe how the variable value changes in real time.
4. Set watchpoints to pause execution inside `HAL_UART_RxCpltCallback` when a UART message received
5. In debug mode, place a breakpoint inside `HAL_UART_RxCpltCallback()`. Then observe whether program execution pauses when a new UART message arrives

9. Experiment 9: Bare Metal Programming

9.1 Objectives

The objectives of this lab are to enable students to gain hands-on experience in low-level firmware development by writing code without relying on abstraction layers, to understand the fundamental principles of interacting directly with hardware peripherals and registers, and to implement basic functionalities such as driving LED and button input handling. Through these objectives, students will develop a solid foundation in embedded systems programming and hardware interaction with STM32F4 microcontrollers.

9.2 Pre-Lab

- Write a single line C code to define the base address of GPIOE
- Define the address of GPIOE_MODER and GPIOE_PUPDR using C code
- Write a C code to configure the following pin at register level:
 - PE4 as output
 - PE5 as input
 - PE6 as analog
 - Activate PE5 internal pull-down resistor
- Write a single line C code to read the state of PE5 pin.
- Write a single line C code to drive the PE4 high

9.3 Theory

9.3.1 Bare Metal Programming

Bare-metal programming is writing software directly on the hardware (registers) without the help of any abstraction layer. Unlike HAL programming, you need to manage the low-level details by yourself. This required the understanding of the microcontroller architecture, memory map, clock tree, and how to configure different peripherals at the register level.

9.3.2 GPIO Configuration Using Bare Metal Programming

1. Define the pins and ports you want to use and their functions. (for example, PD15 is an output pin)

2. Check the memory map and define the base address of the required peripherals from the boundary address table in the reference manual. For example, GPIOD base address is 0x40020C00 (the lower number in the boundary address field).

Bus	Boundary Address	Peripheral
AHB1	0x4002 3800 - 0x4002 3BFF	RCC (Clock)
AHB1	0x4002 0C00 - 0x4002 0FFF	GPIOD (I/O)

Figure 9.1: STM32F4xx AHB1 Peripheral Memory Space

3. To define these addresses in C code, you would write:

```
#define RCC_BASE 0x40023800
#define GPIOD_BASE 0x40020C00
```

4. Peripherals are controlled via registers, which allow you to configure settings and send/receive data to/from the external world. To do this, you must identify the appropriate registers responsible for configuration and data handling.
5. To modify the bits inside any register, you need to know its absolute address, which is calculated as:

$$\text{Register Address} = \text{Base Address of the Peripheral} + \text{Register Offset} \quad (5)$$

The base address can be obtained from the memory map (point 2).

The register offset for each register can be found in the reference manual.

6. **Note:** GPIO clocks are disabled by default. You need to set the corresponding GPIOxEN bit in the RCC_AHB1ENR register to enable the clock for the desired GPIO port. For example, to enable clock for port D, GPIODEN bit which is bit3 should be 1. (check RCC_AHB1ENR in the reference manual)
7. Once the GPIO clock is enabled, you can configure the GPIO pin as input, output, analog, or alternate function using the MODER register. For example, to configure PD15 as output, bit31 should be 0 and bit30 should be 1. (Check MODER register in the reference manual)
8. To define the register addresses using C (Point 4-7):

```
#define RCC_AHB1ENR (*(volatile unsigned int *) (0x40023800 + 0x30))
#define GPIOD_MODER (*(volatile unsigned int *) (GPIOD_BASE + 0x00))
```

9.3.3 Debugging

Debugging is the process of finding and fixing errors or bugs in the source code of any software. When software does not work as expected, computer programmers study the code to determine why any errors occurred.

The debugging tools are used to run the software in a controlled environment, check the code step by step, and analyze and fix the issue. Debugging an STM32 microcontroller requires both hardware and software components working together.

9.3.3.1 Hardware Debugger

The hardware debugger is ST-Link Debugger, and it is built into many STM32 Discovery/Nucleo boards as shown in the following picture:

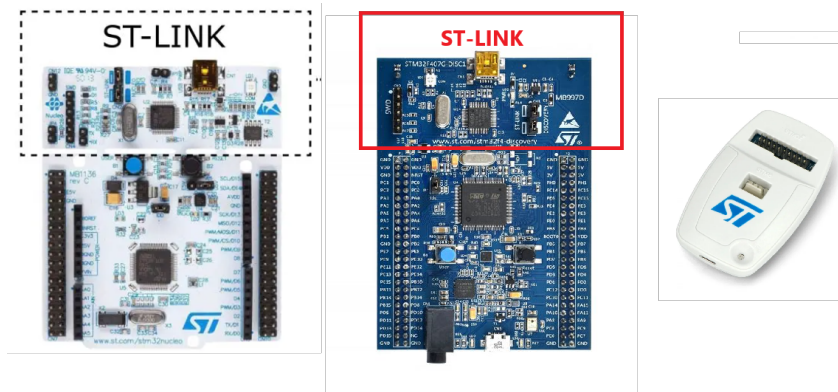


Figure 9.2: ST-Link Debugger on STM32 Development Board

9.3.3.2 Software Debugger Tools

The software debugger tools are integrated with the IDE (i.e STM32CubeIDE) and it is used to:

- Write and compile code.
- Set breakpoints.
- Inspect variables and memory.
- Step through code line-by-line.

9.3.3.3 Debug View

You can access the debug view from the IDE using the debug icon.

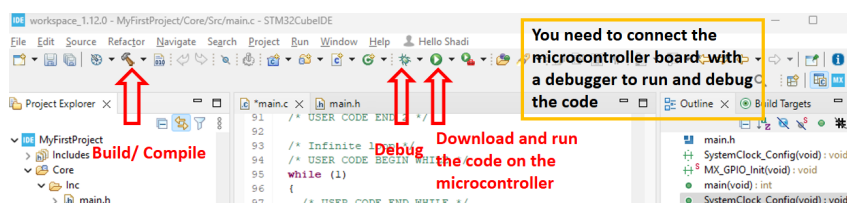


Figure 9.3: Debug View in STM32CubeIDE

The Editor view displays the program file, and it is possible to set breakpoints and follow program execution in the file. A breakpoint is a marker set in the code during debugging, and it helps identify issues by letting you examine how the code runs step-by-step. You can add or remove breakpoints in the editor by double-clicking on the line number.

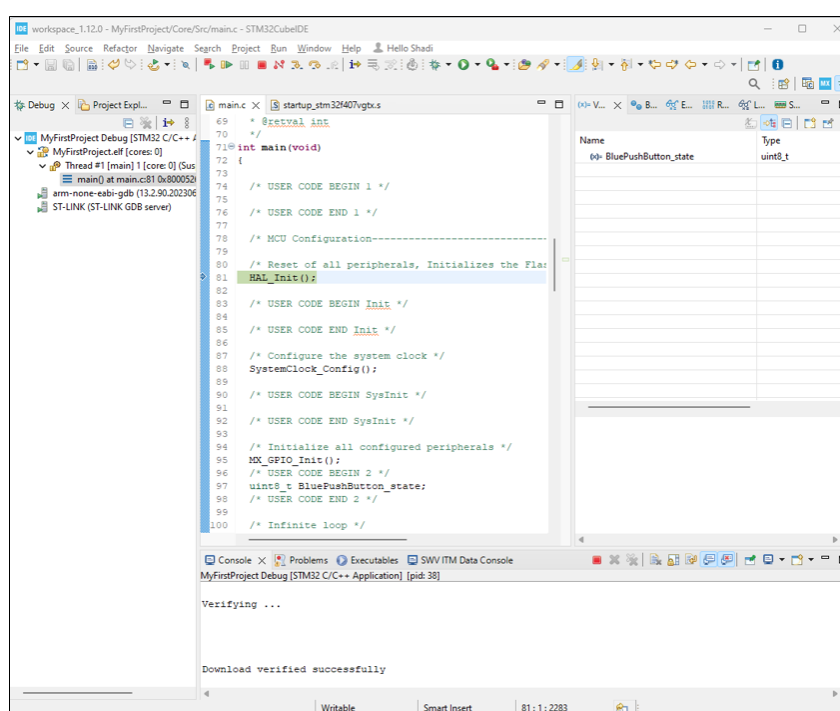


Figure 9.4: Setting Breakpoints in Editor View

The Variables view displays local variables automatically with their current value when the program is not running. We can set a break point in order to check a local variable value.

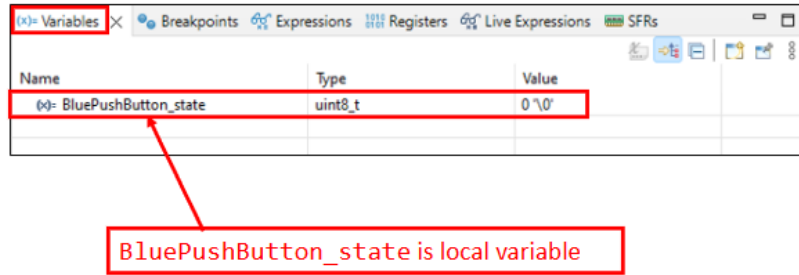


Figure 9.5: Variables View in Debug Mode

The Live Expressions view displays expression values sampled and updated regularly during program execution.

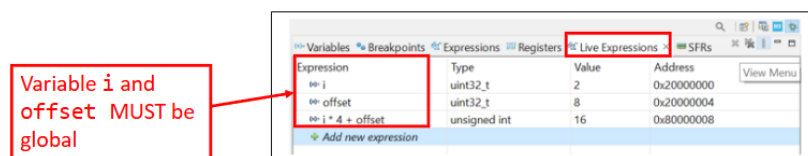


Figure 9.6: Live Expressions View

The SFRs view displays the Special Function Registers in the debugged device, and it is used to observe and manipulate the internal hardware registers of the microcontroller when the program is paused. You can set a break point in order to observe registers values.

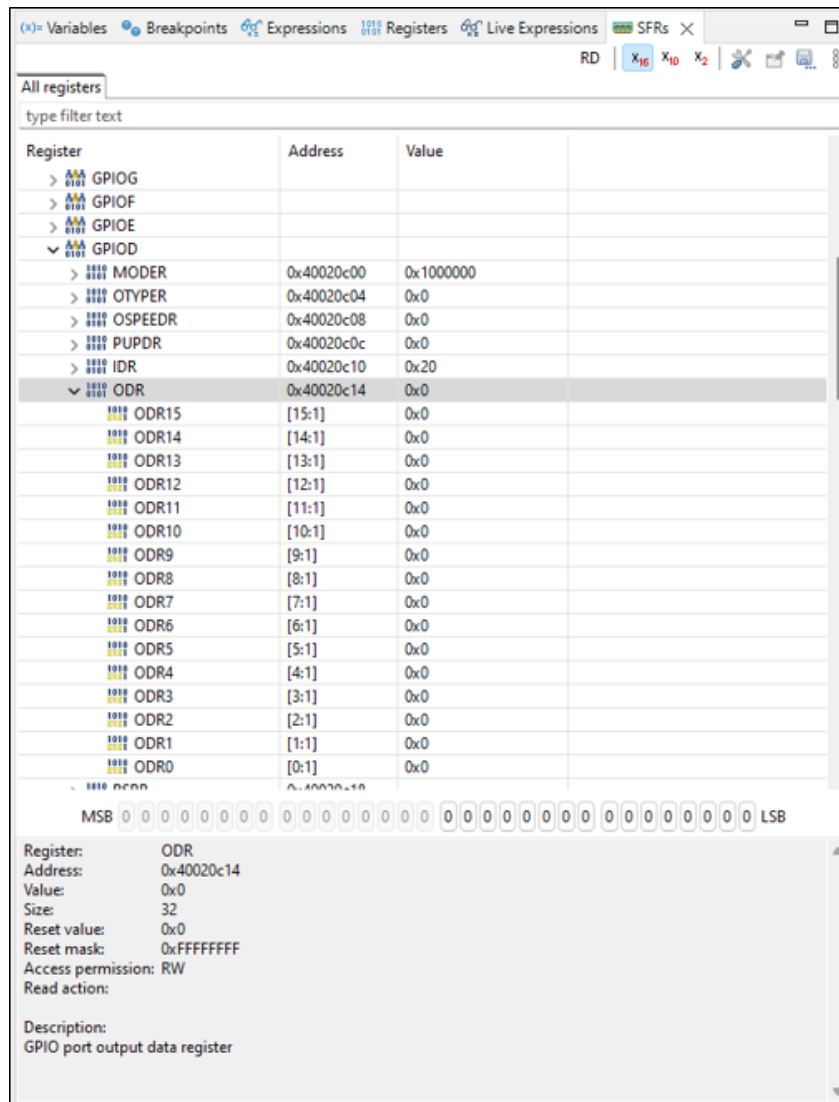


Figure 9.7: Special Function Registers (SFR) View

9.3.3.4 Debug Toolbar

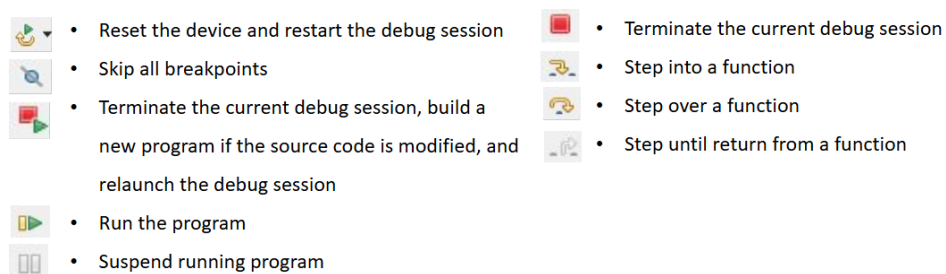


Figure 9.8: Debug Toolbar Controls

9.3.4 What is a 7-Segment Display?

A 7-segment display is an electronic display used to show decimal numbers and some characters. It consists of 7 LEDs (segments) labeled A to G, which can be turned on or off to form numbers or letters. There's usually an additional LED for the decimal point (DP).

Each segment is an individual LED, and by turning specific segments on, you can display numbers from 0 to 9.

9.3.5 7-Segment Displays Configurations

There are two main configurations:

- **Common Cathode:** All the cathodes (negative terminals) of the LEDs are connected together to ground, and you drive the display by applying a positive voltage to the respective segments.
- **Common Anode:** All the anodes (positive terminals) of the LEDs are connected together, and you drive the display by grounding the respective segments.

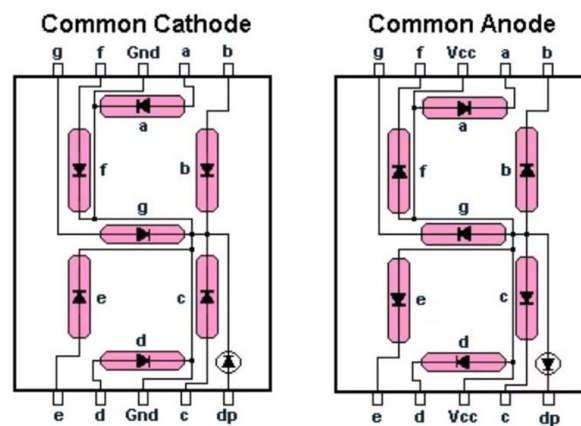


Figure 9.9: 7-Segment Display Configurations

9.3.6 7-Segment Truth Table

Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Figure 9.10: 7-Segment Display Truth Table (Common Cathode)

9.4 Materials and Equipment

- STM32F4 Discovery or Nucleo Board
- 7-Segment Display (Common Cathode or Common Anode)
- Resistors (220Ω or appropriate value)
- Breadboard and Jumper Wires
- USB Cable
- Computer with STM32CubeIDE

9.5 Activity

Your task is to implement a bare metal firmware that counts from 0 to 9 and displays the current count value on a 7-segment display connected to the STM32F4 microcontroller.

9.5.1 Hardware Setup

Draw Schematic with Pin Connections:

Connect the 7-segment display to the STM32F4 microcontroller based on the schematic page. Ensure that the appropriate GPIO pins are connected to the segments of the 7-segment display.

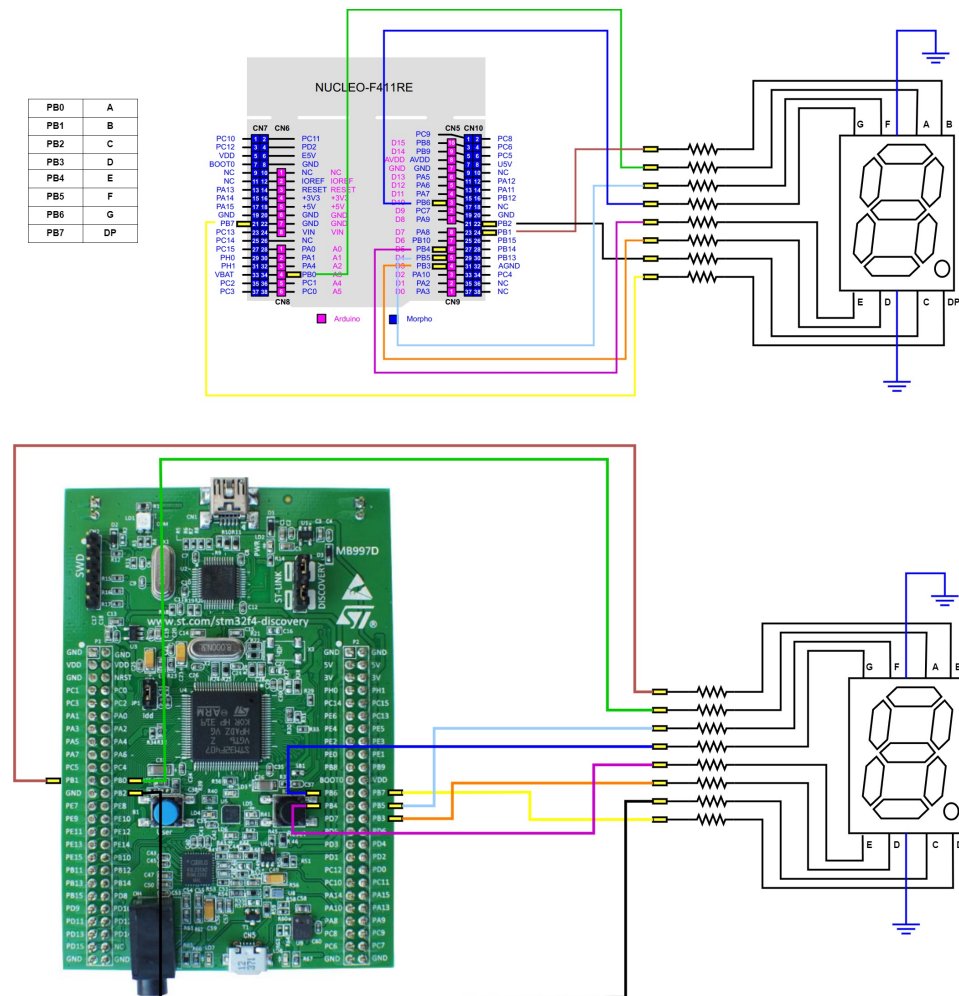


Figure 9.11: Schematic Diagram: 7-Segment Display Connection to STM32F4

Verify and Test Connections:

After making the connections, verify the correctness of the setup by checking continuity, inspecting for short circuits, and ensuring that all components are connected as intended. Test the hardware setup with basic functionality to ensure proper operation before proceeding to firmware development.

9.5.2 Software Setup

1. Create a New STM32 Project:

- Open STM32CubeIDE and create a new STM32 project for your microcontroller (STM32F407VGT6 or STM32F411RET6).

- When you create a project, make sure that you select "Empty" in the "Targeted Project type" field.

2. Configure GPIO Pins:

- Define the base addresses of the required peripherals (Check the memory map in the reference manual)
- Configure the clock and enable the clock for the selected Ports (Check the AHB1ENB register in the reference manual)
- Configure the necessary GPIO pins as output and set initial states for the segments to display digits. (Check the MODER register in the reference manual)

3. Counter Logic:

- Implement a counter variable that increments from 0 to 9.
- Update the 7-segment display based on the current count value. Use a lookup table or direct segment control to display the corresponding digit.
- Ensure that the display is updated at a suitable refresh rate (Delay) to provide smooth and continuous counting.

4. Testing

- Test the firmware on the STM32F4 microcontroller.
- Debug any issues related to GPIO configurations, counter logic, or display refresh to ensure correct functionality.

5. Debugging

- Add a global count variable to The Live Expressions View and observe the changes in real time
- In the SFRs View, check the changes in the register's bits
- Use some breakpoints to follow the flow of the program code

9.6 Required Materials from Reference Manual (STM32F407)

Table 1. STM32F4xx register boundary addresses (continued)

Boundary address	Peripheral	Bus	Register map
0x5006 0800 - 0x5006 0BFF	RNG	AHB2	Section 24.4.4: RNG register map on page 774
0x5006 0400 - 0x5006 07FF	HASH		Section 25.4.9: HASH register map on page 798
0x5006 0000 - 0x5006 03FF	CRYP		Section 23.6.13: CRYP register map on page 766
0x5005 0000 - 0x5005 03FF	DCMI		Section 15.8.12: DCMI register map on page 481
0x5000 0000 - 0x5003 FFFF	USB OTG FS		Section 34.16.6: OTG_FS register map on page 1329
0x4004 0000 - 0x4007 FFFF	USB OTG HS	AHB1	Section 35.12.6: OTG_HS register map on page 1475
0x4002 B000 - 0x4002 BBFF	DMA2D		Section 11.5: DMA2D registers on page 355
0x4002 8000 - 0x4002 93FF	ETHERNET MAC		Section 33.8.5: Ethernet register maps on page 1239
0x4002 6400 - 0x4002 67FF	DMA2		Section 10.5.11: DMA register map on page 338
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 4000 - 0x4002 4FFF	BKPSRAM		-
0x4002 3C00 - 0x4002 3FFF	Flash interface register		Section 3.9: Flash interface registers
0x4002 3800 - 0x4002 3BFF	RCC		Section 7.3.24: RCC register map on page 267
0x4002 3000 - 0x4002 33FF	CRC		Section 4.4.4: CRC register map on page 116
0x4002 2800 - 0x4002 2BFF	GPIOK		Section 8.4.11: GPIO register map on page 290
0x4002 2400 - 0x4002 27FF	GPIOJ		
0x4002 2000 - 0x4002 23FF	GPIOI		
0x4002 1C00 - 0x4002 1FFF	GPIOH		
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		
0x4001 6800 - 0x4001 6BFF	LCD-TFT	APB2	Section 16.7.26: LTDC register map on page 515
0x4001 5800 - 0x4001 5BFF	SAI1		Section 29.17.9: SAI register map on page 966
0x4001 5400 - 0x4001 57FF	SPI6	APB2	Section 28.5.10: SPI register map on page 928
0x4001 5000 - 0x4001 53FF	SPI5		

Figure 9.12: STM32F4xx register boundary address table

6.3.10 RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHSULPIEN	OTGHSSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	DMA2DEN	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	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
Reserved			CRCEEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 9.13: RCC_AHB1ENR Register Description

General-purpose I/Os (GPIO)

RM0090

8.4 GPIO registers

This section gives a detailed description of the GPIO registers.

For a summary of register bits, register address offsets and reset values, refer to [Table 40](#).

The GPIO registers can be accessed by byte (8 bits), half-words (16 bits) or words (32 bits).

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	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
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

Figure 9.14: GPIO_MODER Register Description

9.7 Sample Program

Below is a sample bare metal program structure for the 7-segment display counter:

Listing 1: Bare Metal 7-Segment Counter Program

```
#include <stdint.h>

#if !defined(__SOFT_FP__) && defined(__ARM_FP)
#warning "FPU_is_not_initialized,_but_the_project_is_compiling_for_
an_FPU._Please_initialize_the_FPU_before_use."
#endif

void SevenSeg_writeDigit(uint16_t digit);

/* Register Addresses*/

/* Clock */
#define RCC_BASE 0x40023800UL
#define RCC_AHB1ENR RCC_BASE + 0x30

/* GPIOB Registers*/
#define GPIOB_BASE 0x40020400UL
#define GPIOB_MODER GPIOB_BASE + 0x00
#define GPIOB_BSRR GPIOB_BASE + 0x18

/* GPIOA Registers*/
#define GPIOA_BASE 0x40020000UL
#define GPIOA_MODER GPIOA_BASE + 0x00
#define GPIOA_PUPDR GPIOA_BASE + 0x0C
#define GPIOA_IDR GPIOA_BASE + 0x10

/* delay counter */
#define DELAY_MAX 1000000UL

uint32_t PA0_state =0;
uint16_t count=0;

int main(void)
{
    /* turn on clock on GPIOB */
    *(uint32_t *) (RCC_AHB1ENR) |= (1 << 1);

    /*turn on clock on GPIOA*/
    *(uint32_t *) (RCC_AHB1ENR) |= (1 << 0);

    /* set PB0-PB7 to output mode */
    *(uint32_t *) (GPIOB_MODER) &= ~(1 << 1) & ~(1 << 3) & ~(1 << 5)
        & ~(1 << 7) & ~(1 << 9) & ~(1 << 11) & ~(1 << 13) & ~(1 << 15);
    *(uint32_t *) (GPIOB_MODER) |= (1 << 0) | (1 << 2) | (1 << 4) |
        (1 << 6) | (1 << 8) | (1 << 10) | (1 << 12) | (1 << 14);

    /*Set PA0 to input mode*/
    *(uint32_t *) (GPIOA_MODER) &= ~(1 << 1);
    *(uint32_t *) (GPIOA_MODER) &= ~(1 << 0);
}
```

```

    *(uint32_t *) (GPIOA_PUPDR) &= ~(1 << 1);
    *(uint32_t *) (GPIOA_PUPDR) &= ~(1 << 0);

    /* Loop forever */
    for(;;){
        PA0_state= *(uint32_t *) (GPIOA_IDR) & (0x1);

        SevenSeg_writeDigit(count);

        for(uint32_t i=DELAY_MAX; i--);
        count++;
        if (count==10)count=0;
    }
}

void SevenSeg_writeDigit(uint16_t digit){
    switch (digit) {
        case 0:
            /* set HIGH values on pin PB0->A, PB1->B, PB2->C, PB3->D, PB4
            ->E, PB5->F*/
            *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 1) | (1 << 2) |
            (1 << 3) | (1 << 4) | (1 << 5);
            /* set LOW values on pin PB6->G */
            *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+6));
            break;
        case 1:
            /* set HIGH values on pin PB1->B, PB2->C */
            *(uint32_t *) (GPIOB_BSRR) |= (1 << 1) | (1 << 2);
            /* set LOW values on pin PB0->A, PB3->D, PB4->E, PB5->F, PB6
            ->G*/
            *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+0)) | (1 << (16+3)) |
            (1 << (16+4)) | (1 << (16+5)) | (1 << (16+6));
            break;
        case 2:
            /* set HIGH values on pin PB0->A, PB1->B, PB3->D, PB4->E ,PB6
            ->G */
            *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 1) | (1 << 3) |
            (1 << 4) | (1 << 6);
            /* set LOW values on pin PB2->C, PB5->F */
            *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+2)) | (1 << (16+5));
            break;
        case 3:
            /* set HIGH values on pin PB0->A, PB1->B, PB2->C, PB3->D ,PB6
            ->G */
            *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 1) | (1 << 2) |
            (1 << 3) | (1 << 6);
            /* set LOW values on pin PB4->E, PB5->F */
            *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+4)) | (1 << (16+5));
            break;
        case 4:
            /* set HIGH values on pin PB1->B, PB2->C, PB5->F ,PB6->G */

```

```

*(uint32_t *) (GPIOB_BSRR) |= (1 << 1) | (1 << 2) | (1 << 5) |
    (1 << 6);
/* set LOW values on pin PB0->A, PB3->D, PB4->E */
*(uint32_t *) (GPIOB_BSRR) |= (1 << (16+0)) | (1 << (16+3)) |
    (1 << (16+4));
break;
case 5:
    /* set HIGH values on pin PB0->A, PB2->C, PB3->D, PB5->F , PB6
    ->G */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 2) | (1 << 3) |
        (1 << 5) | (1 << 6);
    /* set LOW values on pin PB1->B, PB4->E */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+1)) | (1 << (16+4));

    break;
case 6:
    /* set HIGH values on pin PB0->A, PB2->C, PB3->D, PB4->E, PB5
    ->F , PB6->G */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 2) | (1 << 3) |
        (1 << 4) | (1 << 5) | (1 << 6);
    /* set LOW values on pin PB1->B */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+1));
    break;
case 7:
    /* set HIGH values on pin PB0->A, PB1->B, PB2->C */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 1) | (1 << 2);
    /* set LOW values on pin PB3->D, PB4->E, PB5->F , PB6->G */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+3)) | (1 << (16+4)) |
        (1 << (16+5)) | (1 << (16+6));

    break;
case 8:
    /* set HIGH values on pin PB0->A, PB1->B, PB2->C, PB3->D, PB4
    ->E, PB5->F , PB6->G */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 1) | (1 << 2) |
        (1 << 3) | (1 << 4) | (1 << 5) | (1 << 6);

    break;
case 9:
    /* set HIGH values on pin PB0->A, PB1->B, PB2->C, PB3->D, PB5
    ->F , PB6->G */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 1) | (1 << 2) |
        (1 << 3) | (1 << 5) | (1 << 6);
    /* set LOW values on pin PB4->E */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+4));
    break;
default:
    /*set HIGH values on pin PB0->A, PB3->D, PB4->E, PB5->F , PB6
    ->G */
    *(uint32_t *) (GPIOB_BSRR) |= (1 << 0) | (1 << 3) | (1 << 4) |
        (1 << 5) | (1 << 6);

```

```
        /*set LOW values on pin PB1->B, PB2->C*/  
        *(uint32_t *) (GPIOB_BSRR) |= (1 << (16+1)) | (1 << (16+2));  
        break;  
    }  
}
```