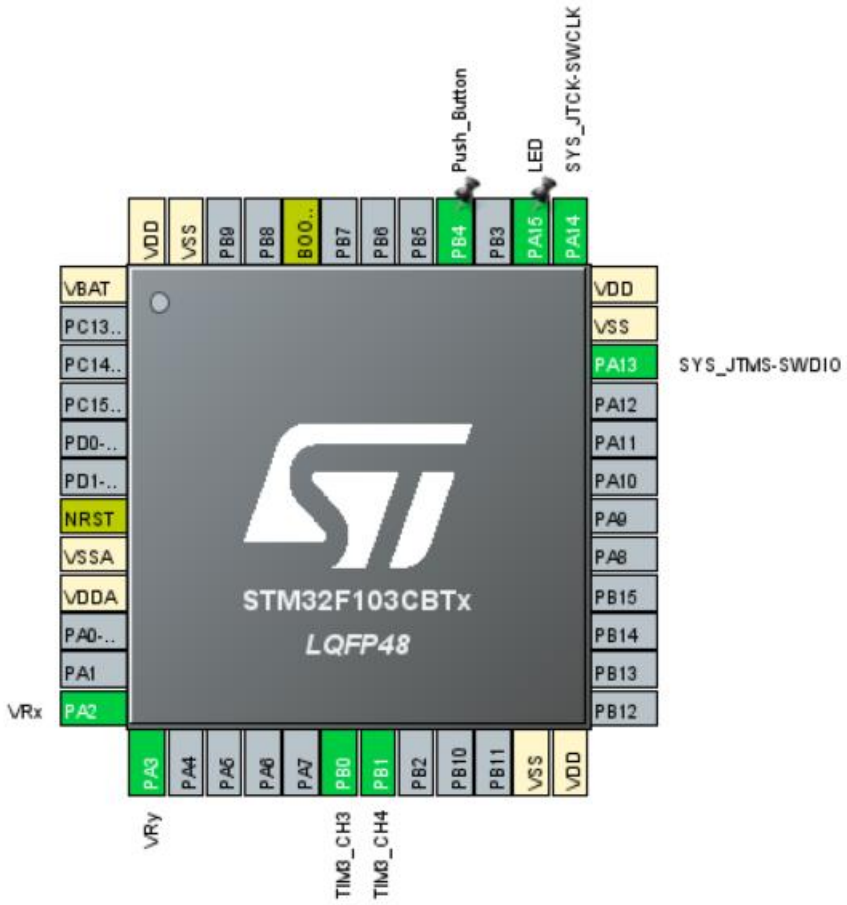
Mobin Khatib – 99106114 mbnkhatib@gmail.com

Professor K. Hajsadeghi

First, we should devise our needs based on the proposal in the CubeMX software, doing so in the CubeMX we will have this shape:



Of course SYS mode, it's debug is always Serial Wire (PA13)

We can see that PA2 which would be Channel2 by ADC (ADC1_IN2) is named VRx here.

We can see that PA3 which would be Channel2 by ADC (ADC2_IN3) is named VRy here.

Pin Na...	Signal on ...	GPIO out...	GPIO mode	GPIO Pull...	Maximum...	User Label	Modified
PA2	ADC1_IN2	n/a	Analog m...	n/a	n/a	VRx	☑
PA3	ADC2_IN3	n/a	Analog m...	n/a	n/a	VRy	☑

Our GPIO PA15 we name it Push_Button and it is Internal Pull up

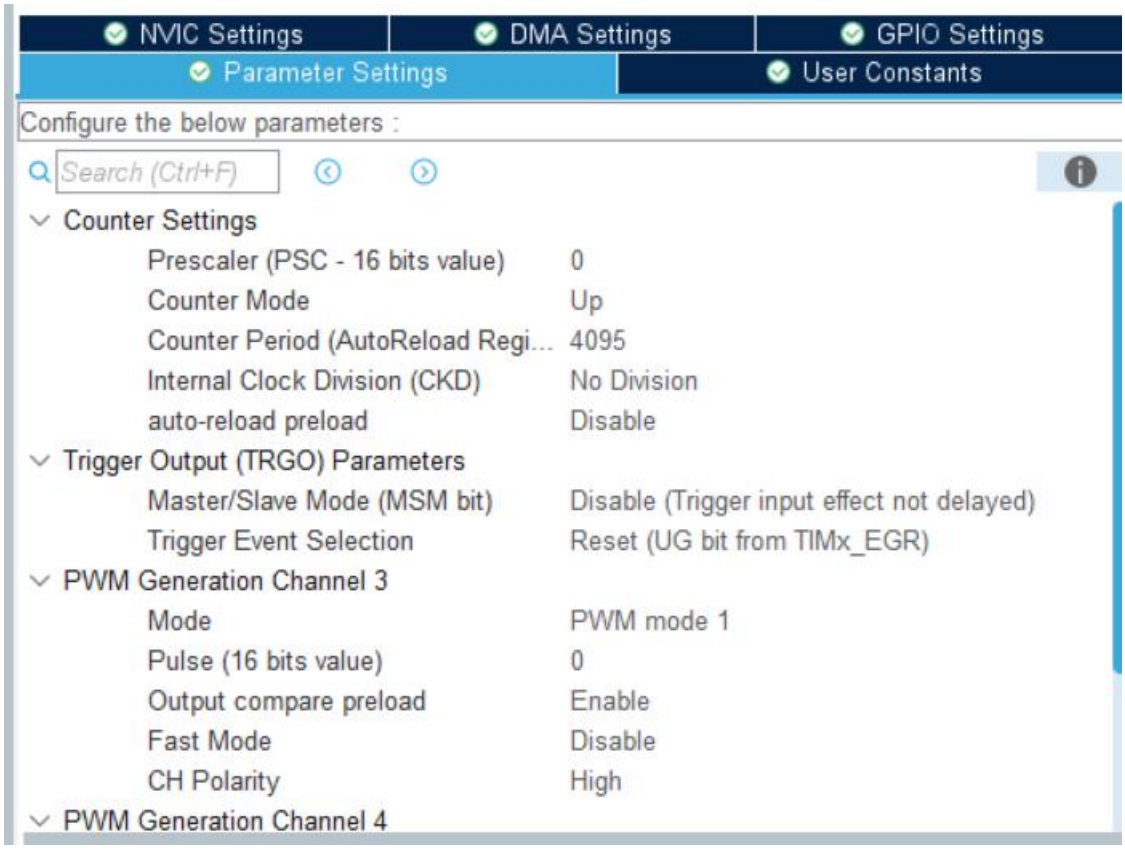
Our GPIO PB4 we name it LED which is Active Low(I'm saying this because it will be important in code)

Pin Na...	Signal on ...	GPIO out...	GPIO mode	GPIO Pull...	Maximum...	User Label	Modified
PA15	n/a	Low	Output P...	No pull-up...	Low	LED	☑
PB4	n/a	n/a	Input mode	Pull-up	n/a	Push_But...	☑

And lastly, our timers will be:

Pin Na...	Signal on ...	GPIO out...	GPIO mode	GPIO Pull...	Maximum...	User Label	Modified
PB0	TIM3_CH3	n/a	Alternate ...	n/a	Low		☐
PB1	TIM3_CH4	n/a	Alternate ...	n/a	Low		☐

Which is set to PWM Generation. Also,



As we know, the frequency of us should not become lower 100Hz, and we know duty cycles are changed with 12-bit resolution, so we can calculate and check the frequency:

f_cpu = 8MHz

prescaler = 0

counter period = $2^{12} - 1 = 4095$

$$f = \frac{\text{f\_cpu}}{(\text{prescaler} + 1)(\text{counter period} + 1)}$$

Then, f = 2kHz which it is ok.

All right, now we are going to complete our code in Keil:

Before our main loop:

Starting PWM:

After initializing the peripherals, start the PWM outputs before entering the main loop. This ensures that the peripherals are active and ready for operation.

```
/* USER CODE BEGIN 2 */


HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3); // Start PWM for RGB LED Blue
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4); // Start PWM for RGB LED Red
/* USER CODE END 2 */
```

In our main loop:

Reading ADC Values and Updating PWM Duty Cycle

Within the main loop (while (1)), continuously read the ADC values and adjust the PWM duty cycles accordingly. This allows real-time control based on the joystick's position.

```
/* USER CODE BEGIN WHILE */
while (1)
{
  HAL_ADC_Start(&hadc1); // Start ADC1 for X-axis
  HAL_ADC_Start(&hadc2); // Start ADC2 for Y-axis
  uint32_t adc_value_x = 0;
  uint32_t adc_value_y = 0;

  // Poll for ADC1 conversion completion and read value
  if (HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY) == HAL_OK)
  {
      adc_value_x = HAL_ADC_GetValue(&hadc1); // X-axis value
  }

  // Poll for ADC2 conversion completion and read value
  if (HAL_ADC_PollForConversion(&hadc2, HAL_MAX_DELAY) == HAL_OK)
  {
      adc_value_y = HAL_ADC_GetValue(&hadc2); // Y-axis value
  }

  // Map ADC values to PWM duty cycle
  //uint32_t pwm_value_blue = (adc_value_x * __HAL_TIM_GET_AUTORELOAD(&htim3)) / 4095;
  //uint32_t pwm_value_red = (adc_value_y * __HAL_TIM_GET_AUTORELOAD(&htim3)) / 4095;

  // Update PWM duty cycles
  __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, adc_value_y); // Set duty cycle for RGB Blue
  __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_4, adc_value_x);  // Set duty cycle for RGB Red
```

The code uses HAL_ADC_PollForConversion to wait until the ADC conversion completes.

After a successful conversion, HAL_ADC_GetValue retrieves the digital value for:

- **adc_value_x**: Represents the X-axis joystick position.
- **adc_value_y**: Represents the Y-axis joystick position.

As we can see in above code, we comment the MAP part because there is no need to use it and we can directly give the values of ADC to PWM duty cycle

Update PWM Duty Cycle:

__HAL_TIM_SET_COMPARE updates the duty cycle for the specific PWM channels controlling the RGB LED:

- **Channel 3**: Adjusts the intensity of the Blue LED based on the Y-axis input.
- **Channel 4**: Adjusts the intensity of the Red LED based on the X-axis input.

Button Press Handling

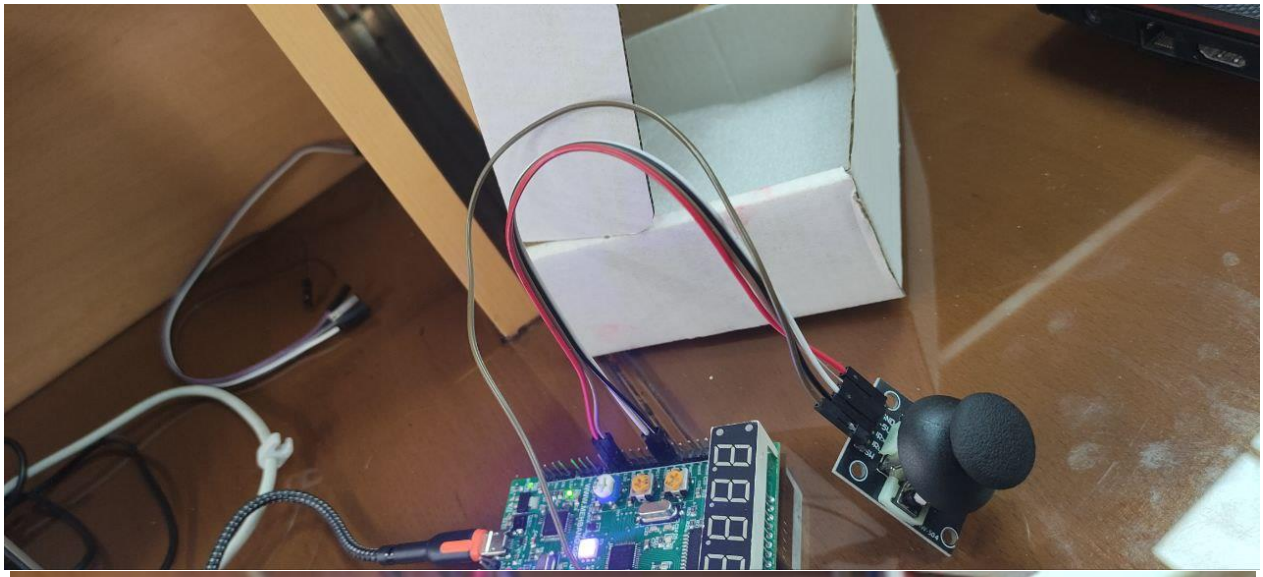
Within the main loop, check the button's state and perform the desired action when it's pressed. This allows our application to respond to user input in real-time.

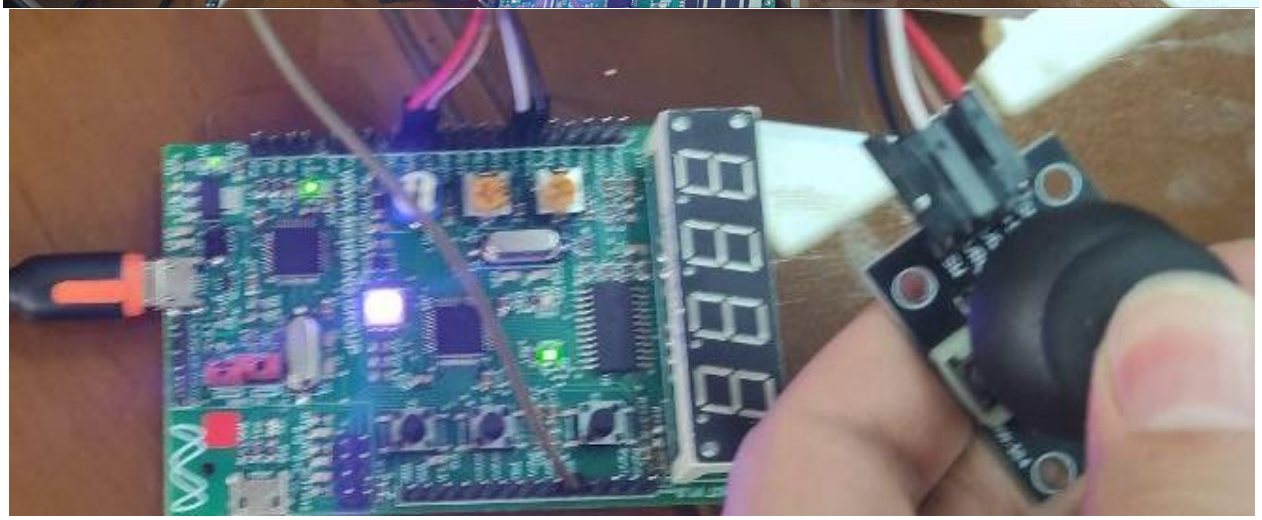
```
// Check if button is pressed
if (HAL_GPIO_ReadPin(Push_Button_GPIO_Port, Push_Button_Pin) == GPIO_PIN_RESET) //Active Low
{
    // Perform action on button press
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET); // Toggle an LED, for example

    // Debounce delay
    HAL_Delay(30);
}
else
{
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET); // Toggle an LED, for example

}
```

The **if** statement is used to handle the state of the joystick's software button. Specifically, it checks whether the button is in the **pressed state** (i.e., **GPIO_PIN_RESET**, indicating an active-low configuration). If the button is pressed, the code executes the block inside the **if**, turning on the LED (or performing a desired action) using **HAL_GPIO_WritePin** and applying a debounce delay. If the button is not pressed, the **else** block ensures the LED is turned off.

Note: Because the configuration of our Joystick is active low, we put GPIO_PIN_RESET not GPIO_PIN_SET in our statement.

The potentiometer in a joystick acts like a changing resistor that adjusts its resistance based on the joystick's position. As you move the joystick, the potentiometer varies its resistance, which can then be used to measure the exact position. This change in resistance is converted into an electrical signal, allowing the joystick to show us the direction and extent of movement in each axis (X and Y). Finally, we connect board to the joystick based on its instruction and then we program the board. We can see some pictures of our results:





Thank you for reading this😊