THE UNIVERSITY OF DANANG
UNIVERSITY OF SCIENCE AND TECHNOLOGY
Faculty of Advanced Science and Technology

LABORATORY REPORT

DEVICE NETWORKS

Instructor : Nguyen Huynh Nhat Thuong

Class : 21ECE

Group members: Luong Nhu Quynh

Vo Van Buu

Nguyen Thi Tam

Tran Hoang Minh

Da Nang, 6th May 2025

Nội dung

1. Project requirements 3
2. Hardware 3
2.1. Block diagram 3
..... 3
2.2. Wiring diagram 4
2.3. Real circuit photo 5
3. Software 5
3.1. Microcontroller configuration 5
3.2. Flowchart 10
3.3. Code 10
4. Results 11
4.1. Testing plan 11
4.2. Picture 12

LAB3: I2C COMMUNICATION

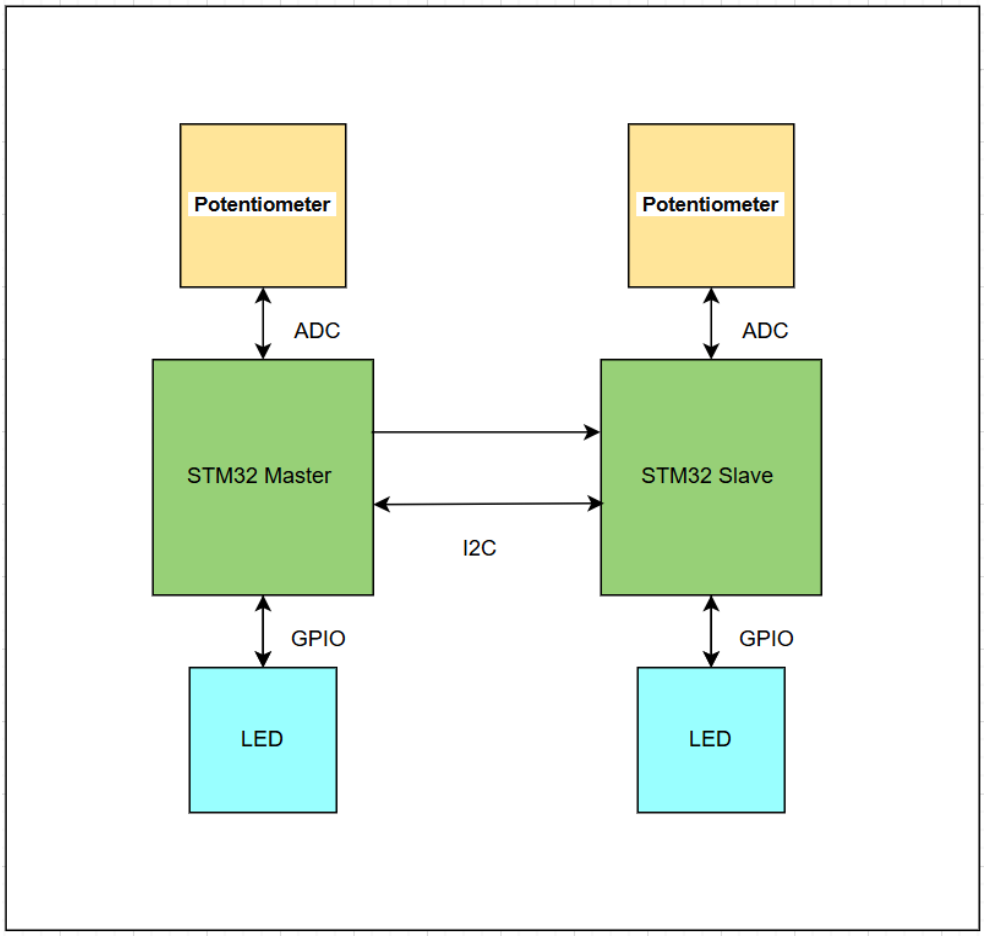
1. Project requirements

This lab requires students to build communication between two STM32 controllers using the I2C protocol.
Lab description:

1. Set up I2C Communication on the STM32
2. Hardware Components: STM32 - 2 pcs, Potentiometer (10K Ohm) - 2 pcs, Jumper wires, Breadboard.
3. Adjust the potentiometer on the master device to control the blink rate of the slave device LED.
4. Adjust the potentiometer on the slave device to control the blink rate of the master device LED

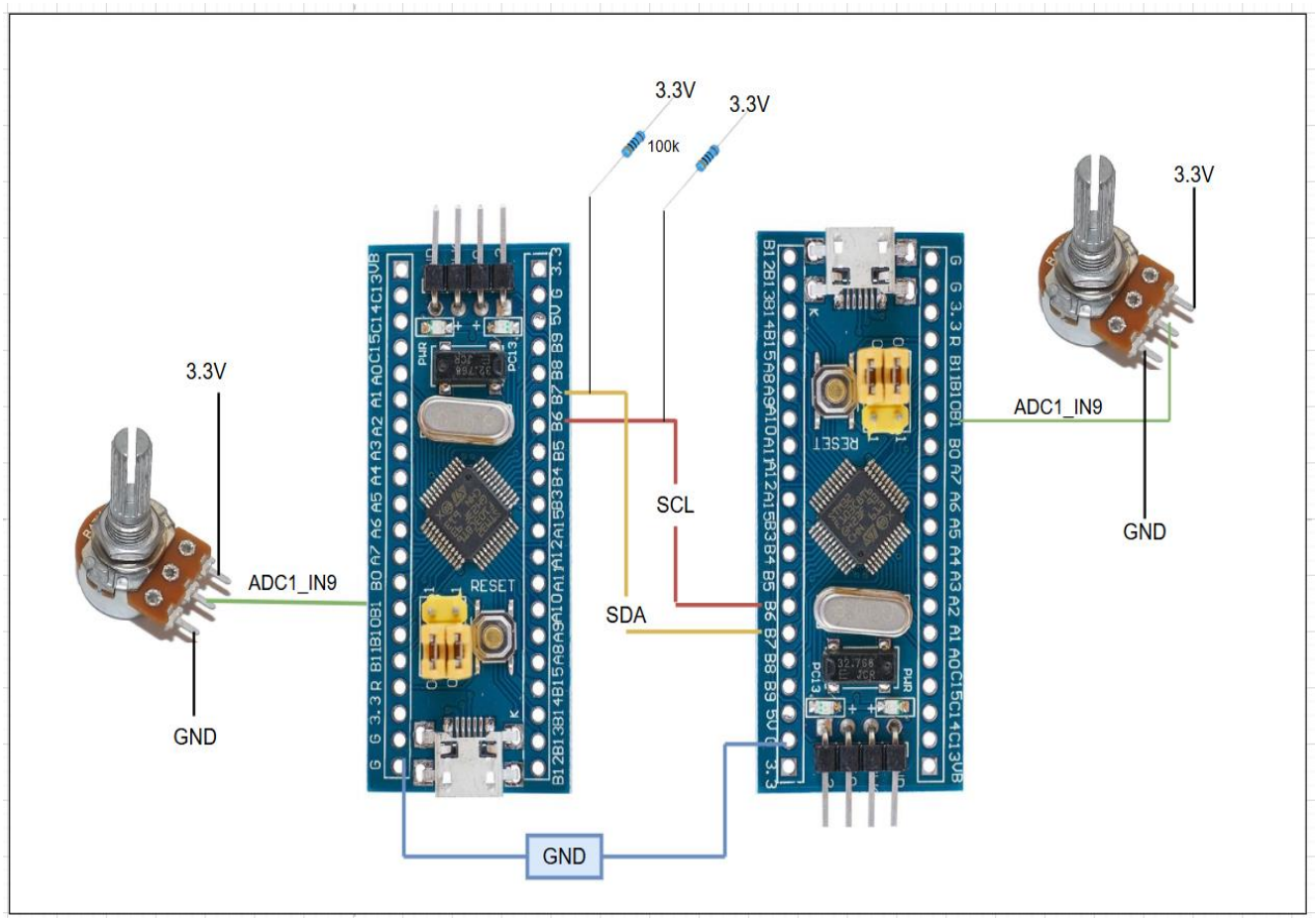
2. Hardware

2.1. Block diagram

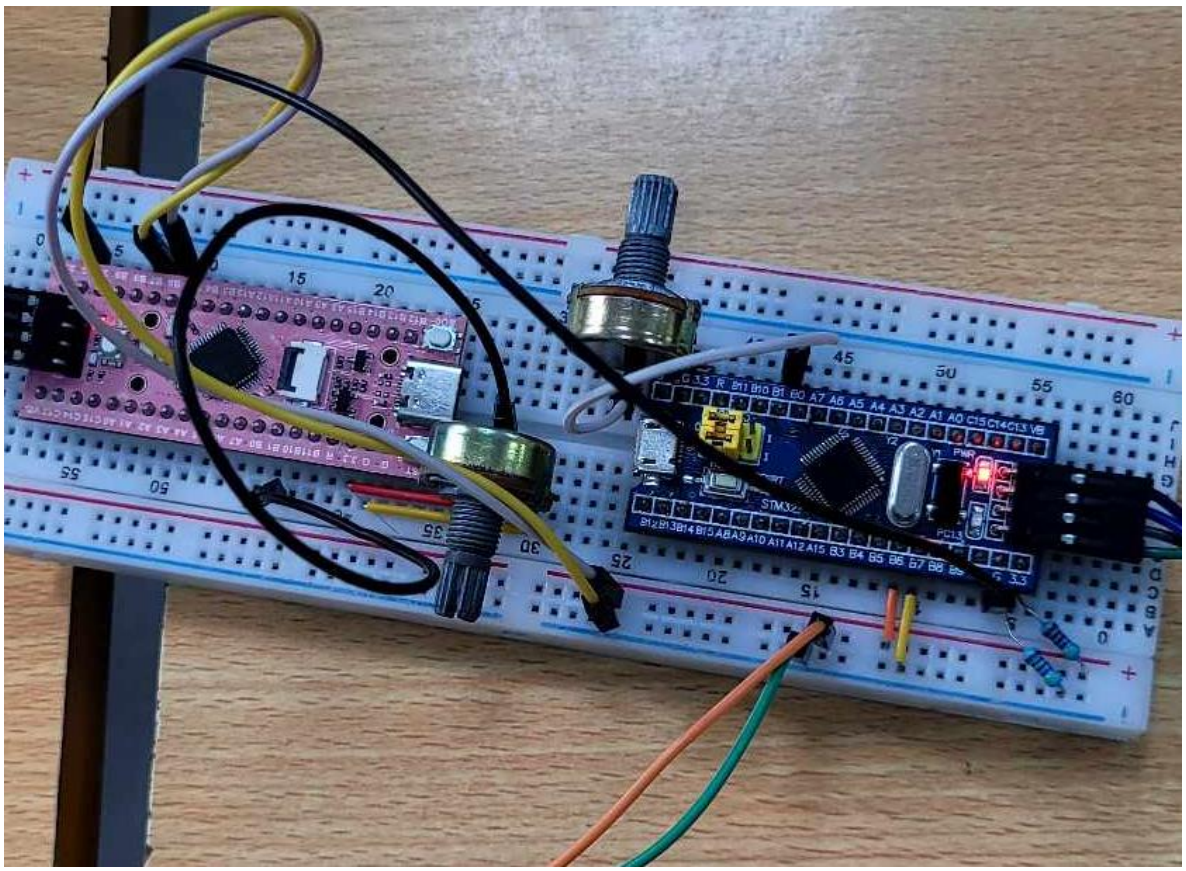


2.2. Wiring diagram

In this project, we use I2C1 in both STM32 devices, so we connect SCLe I2C1 in both STM32 devices, so we connect SCL (serial clock) and SDA (serial data) of them. We use the ADC1 with PIN B1 and connect it to a potentiometer to measure the resistance value. Because I2C lines don't provide a full up resistor function, we add two 100k ohm resistors to create a full up resistor for two I2C lines.



2.3. Real circuit photo

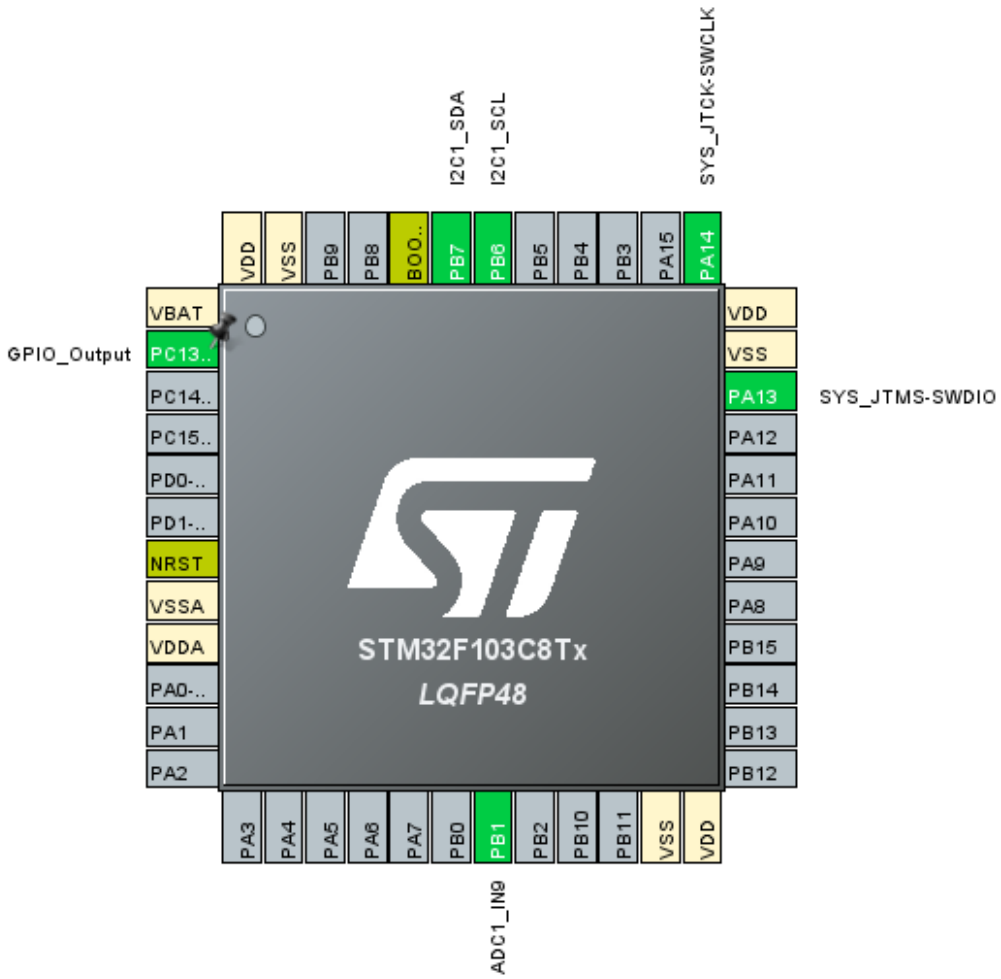


3. Software

3.1. Microcontroller configuration

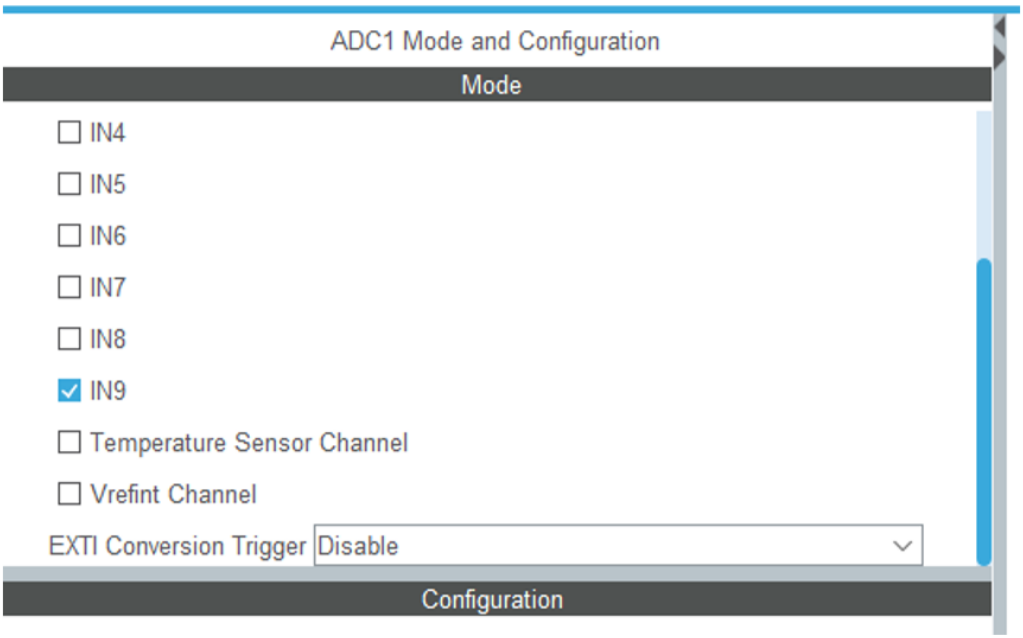
Master

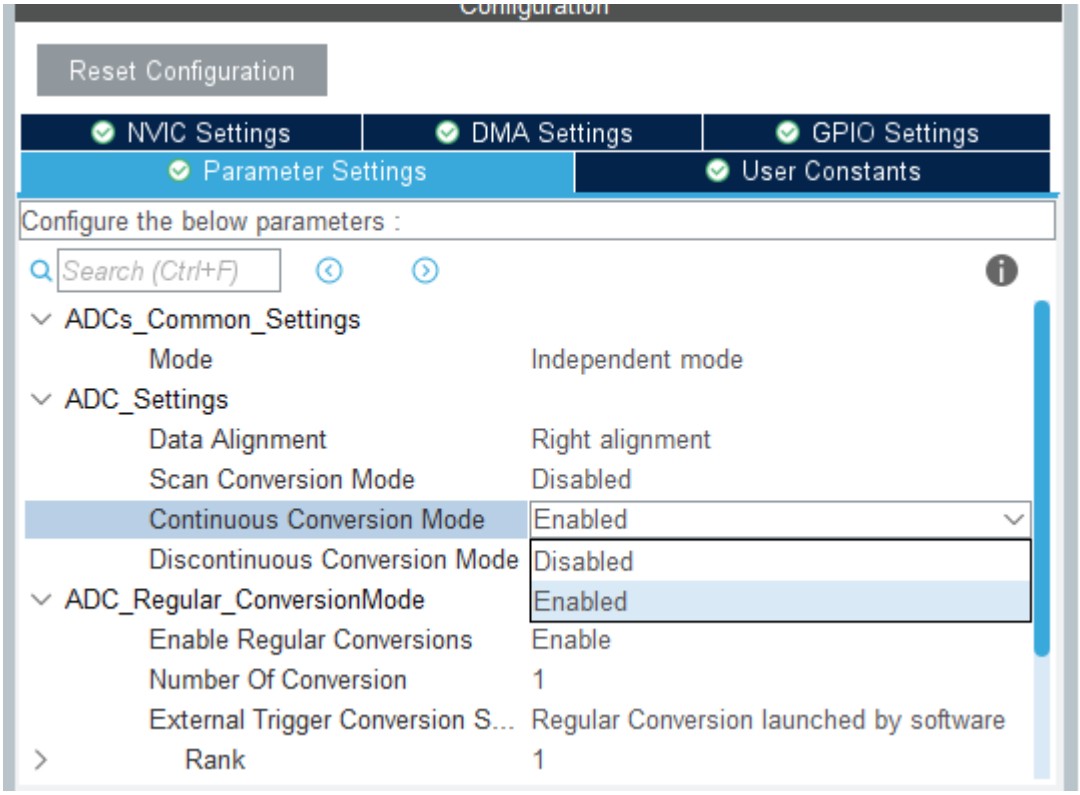
For the master, we use I2C1 with PB7 SDA and PC6 SCL, ADC1 with IN9 with pin PB1, and also use the green LED on the board with pin PC13.



a) ADC configuration

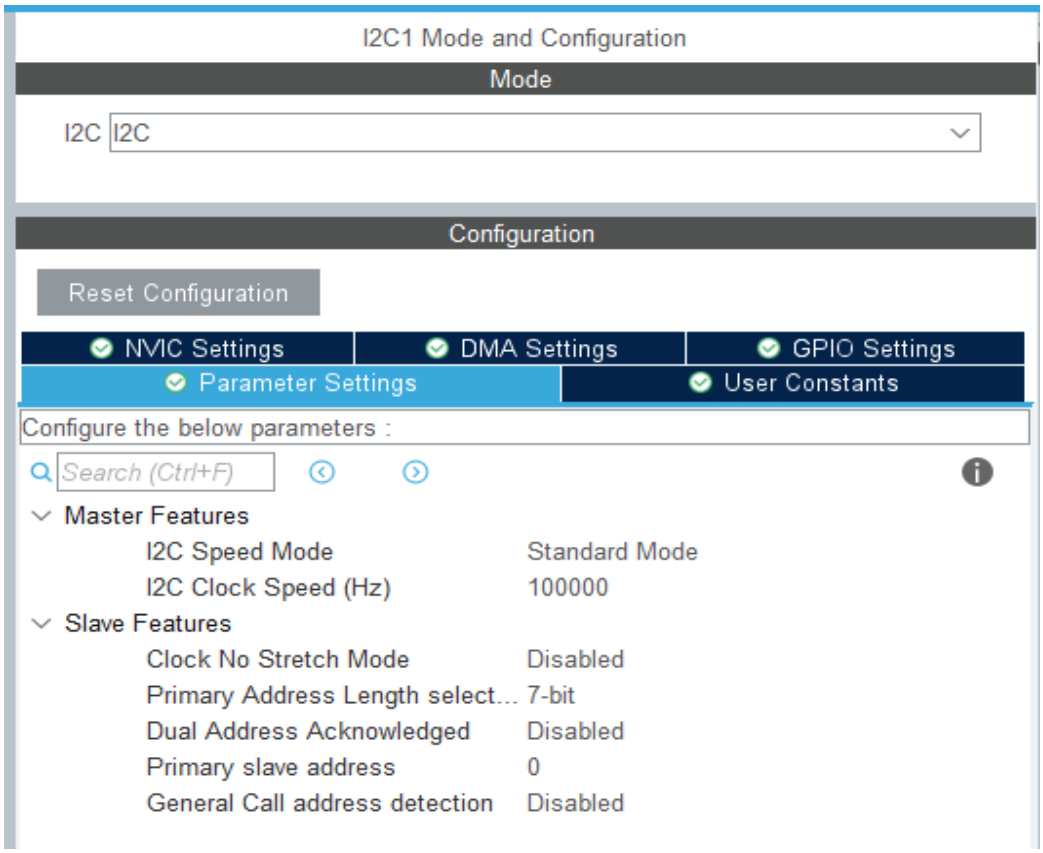
In the Analog setting, we select ADC1 and then choose IN9. And in the parameter setting, we enable Continuous Conversion Mode.





b) I2C configuration

In connectivity, we select I2C1 -> choose I2C. We keep the default settings with standard mode, 100 kHz, and 7-bit slave address length.

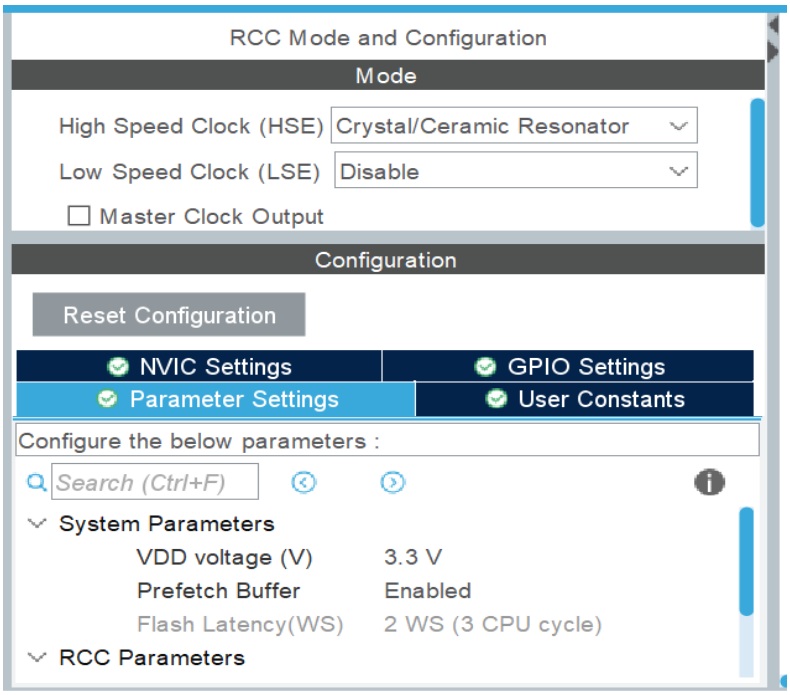


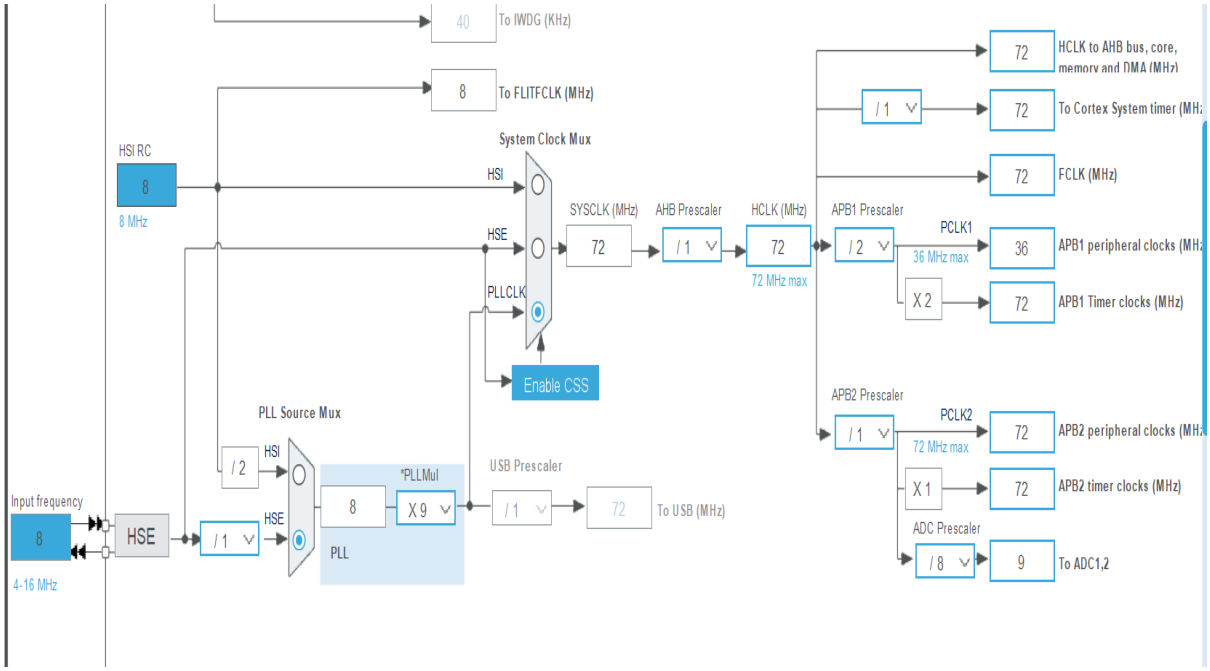
Slave:

Configure the same as Master. But also adding the following settings:

a. Clock configuration.

For the slave, we use the HSE clock source for high-speed communication with the master device.

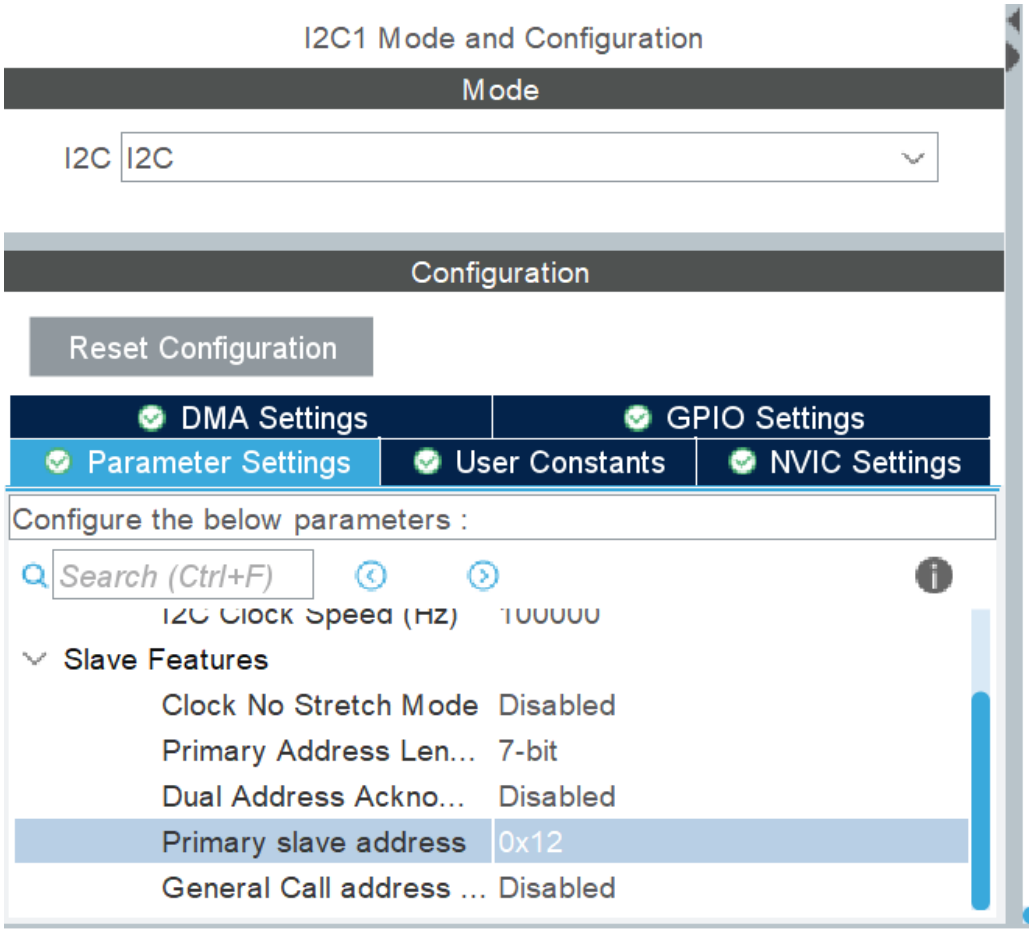


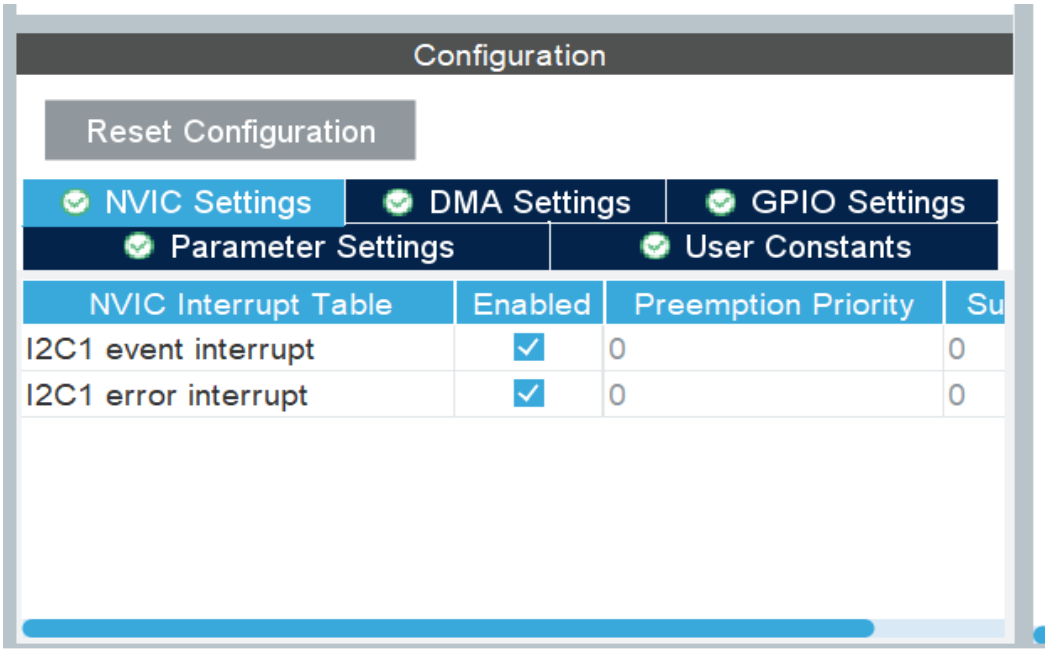


b. I2C configuration

In the different things in Parameter Setting of Slave device, in Primary slave address, we set the address of the slave is 0x12.

And we also enable the interrupts for I2C in the NVIC settings.





3.2. Flowchart

3.3. Code

Master

The code is added in between */* USER CODE BEGIN Includes */* and */* USER CODE END Includes */*

```
uint16_t slaveADDR = 0x12<<1;
uint16_t slaveMemADDR = 0x45<<1;
uint8_t TxData[6] = {0};
uint8_t RxData[6];
uint16_t readValue;
```

The code is added in between */* USER CODE BEGIN BEGIN 2*/* and */* USER CODE END BEGIN 2*/*

```
 HAL_ADC_Start(&hadc1);
```

The code is added in between */* USER CODE BEGIN WHILE*/* and */* USER CODE END WHILE*/*

```
        HAL_ADC_PollForConversion(&hadc1,1000);
        readValue = HAL_ADC_GetValue(&hadc1);
        TxData[5] = readValue>>4;
        HAL_I2C_Master_Transmit(&hi2c1, slaveADDR, TxData, 6, 1000);
        HAL_I2C_Mem_Read(&hi2c1, slaveADDR, slaveMemADDR, 0, RxData, 6,
1000);
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
        HAL_Delay(RxData[3]);
```

Slave:

The code is added in between */* USER CODE BEGIN Include*/* and */* USER CODE END Include*/*

```
#include "i2c_slave.h"
```

The code is added in between */* USER CODE BEGIN 0*/* and */* USER CODE END 0*/*

```
uint8_t i2c_rcv = 255;          // Data received from I2C master
uint32_t time_start = 0;        // Start time for blinking
uint8_t stat_LED = 0;           // LED state: 1 = ON, 0 = OFF
uint8_t value_pot = 0;          // Scaled potentiometer value (0-255)
volatile uint8_t i2c_rx_data = 0; // Buffer for received I2C data
volatile uint8_t i2c_tx_data = 0; // Buffer for transmitted I2C data
uint8_t ScaleADCToByte(uint16_t adc_value) {
    return (uint8_t)((adc_value * 255) / 4095);
}
```

The code is added in between */* USER CODE BEGIN 2*/* and */* USER CODE END 2*/*

```
if (HAL_I2C_EnableListen_IT(&hi2c1) != HAL_OK)
{
        Error_Handler();
}
HAL_ADC_Start(&hadc1);
time_start = HAL_GetTick();
```

The code is added in between */* USER CODE BEGIN WHILE*/* and */* USER CODE END WHILE*/* .

```
        // Read potentiometer via ADC
        if (HAL_ADC_PollForConversion(&hadc1, 10) == HAL_OK) {
            uint16_t adc_value = HAL_ADC_GetValue(&hadc1);
            value_pot = ScaleADCToByte(adc_value);
            i2c_tx_data = value_pot; // Update transmit buffer
            I2C_REGISTERS[3] = i2c_tx_data;
        }
        // Blink LED based on received value
        i2c_rcv = I2C_REGISTERS[4];
        float interval = (i2c_rcv == 0) ? 100 : (1000 * ((float)i2c_rcv / 255.0));
        if ((HAL_GetTick() - time_start) > interval) {
            stat_LED = !stat_LED;
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, stat_LED ? GPIO_PIN_RESET :
GPIO_PIN_SET); // PC13 active-low
            time_start = HAL_GetTick();
        }
```

The code is added in between */* USER CODE BEGIN I2C1_Init 1*/* and */* USER CODE END I2C1_Init 1*/* .

```
hi2c1.Instance = I2C1;
hi2c1.Init.ClockSpeed = 100000;
hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
hi2c1.Init.OwnAddress1 = 36;
hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
hi2c1.Init.OwnAddress2 = 0;
hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
if (HAL_I2C_Init(&hi2c1) != HAL_OK)
{
  Error_Handler();
}
```

4. Results

4.1. Testing plan

Adjust the potentiometer on the master device to control the blink rate of the slave device LED. Use Saleae analyzer to view the bit waveform and Debugger to view the value.

Adjust the potentiometer on the slave device to control the blink rate of the master device LED. Use Saleae analyzer to view the bit waveform and Debugger to view the value.

4.2. Picture

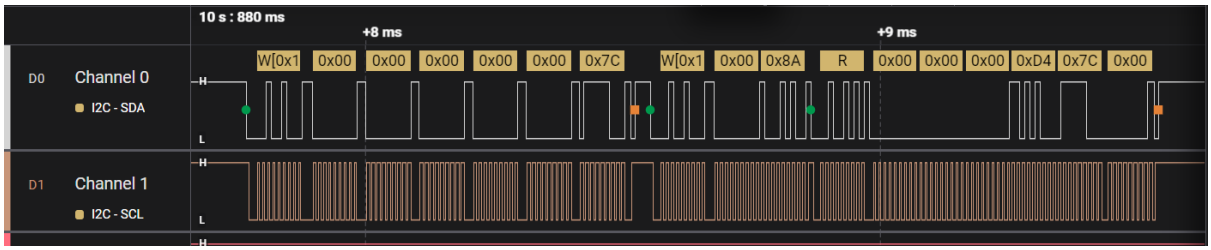
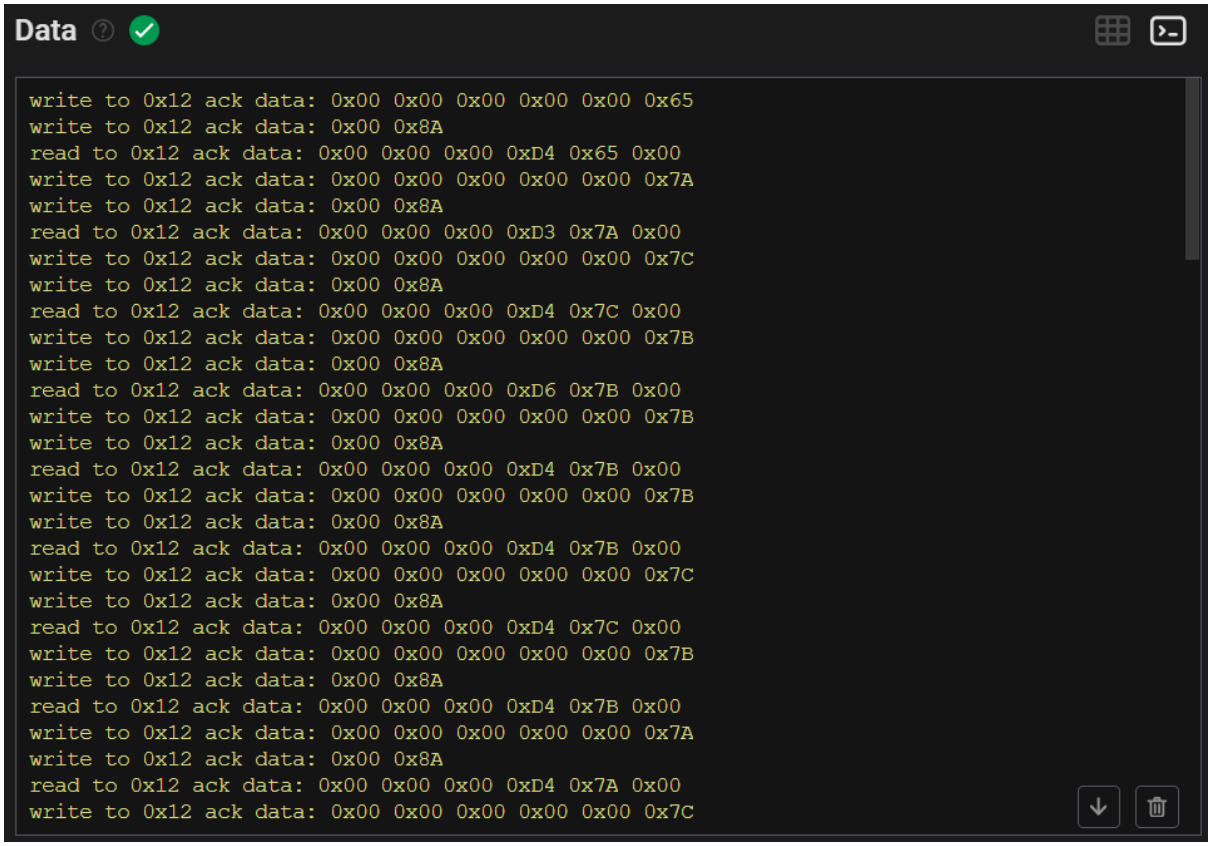


Figure:



Master Debugger:

Expression	Type	Value
TxData	uint8_t [6]	[6]
TxData[0]	uint8_t	0 '\000'
TxData[1]	uint8_t	0 '\000'
TxData[2]	uint8_t	0 '\000'
TxData[3]	uint8_t	0 '\000'
TxData[4]	uint8_t	0 '\000'
TxData[5]	uint8_t	123 '{'
RxData	uint8_t [6]	[6]
RxData[0]	uint8_t	0 '\000'
RxData[1]	uint8_t	0 '\000'
RxData[2]	uint8_t	0 '\000'
RxData[3]	uint8_t	212 'Ô'
RxData[4]	uint8_t	123 '{'
RxData[5]	uint8_t	0 '\000'
readValue	uint16_t	1974
Add new expression		

Slave Debugger:

Expression	Type	Value
RxData[3]	uint8_t	0 '\000'
RxData[4]	uint8_t	0 '\000'
RxData[5]	uint8_t	123 '{'
RxData[6]	uint8_t	0 '\000'
RxData[7]	uint8_t	0 '\000'
RxData[8]	uint8_t	0 '\000'
RxData[9]	uint8_t	0 '\000'
RxData[10]	uint8_t	0 '\000'
I2C_REGISTERS	uint8_t [10]	[10]
I2C_REGISTERS[0]	uint8_t	0 '\000'
I2C_REGISTERS[1]	uint8_t	0 '\000'
I2C_REGISTERS[2]	uint8_t	0 '\000'
I2C_REGISTERS[3]	uint8_t	212 'Ô'
I2C_REGISTERS[4]	uint8_t	124 '|'
I2C_REGISTERS[5]	uint8_t	0 '\000'
I2C_REGISTERS[6]	uint8_t	0 '\000'
I2C_REGISTERS[7]	uint8_t	0 '\000'
I2C_REGISTERS[8]	uint8_t	0 '\000'
I2C_REGISTERS[9]	uint8_t	0 '\000'
value_pot	uint8_t	211 'Ó'

Appendix

i2c_slave.h

```
#ifndef INC_I2C_SLAVE_H_
#define INC_I2C_SLAVE_H_
extern uint8_t I2C_REGISTERS[10];
extern I2C_HandleTypeDef hi2c1;
#endif /* INC_I2C_SLAVE_H_ */
```

i2c_slave.c

```
#include "main.h"
#include "i2c_slave.h"
#include "string.h"
uint8_t I2C_REGISTERS[10] = {0,0,0,0,0,0,0,0,0,0};
extern I2C_HandleTypeDef hi2c1;
#define RxSIZE  11
uint8_t RxData[RxSIZE];
uint8_t rxcount=0;
uint8_t txcount=0;
uint8_t startPosition = 0;
uint8_t bytesRrecvd = 0;
uint8_t bytesTransd = 0;
int process_data (void)
{
	int startREG = RxData[0];  // get the register address
	int numREG = bytesRrecvd;  // Get the number of registers
	int endREG = startREG + numREG -1;  // calculate the end register
	if (endREG>9)  // There are a total of 10 registers (0-9)
	{
		// clear everything and return
		memset(RxData,'\0',RxSIZE);
		rxcount =0;
		return 0;
	}
	int indx = 1;  // set the indx to 1 in order to start reading from RxData[1]
	for (int i=0; i<numREG; i++)
	{
		I2C_REGISTERS[startREG++] = RxData[indx++];  // Read the data from RxData and
save it in the I2C_REGISTERS
	}
	return 1;  // success
}
void HAL_I2C_ListenCpltCallback(I2C_HandleTypeDef *hi2c)
{
	HAL_I2C_EnableListen_IT(hi2c);
```

```
}
void HAL_I2C_AddrCallback(I2C_HandleTypeDef *hi2c, uint8_t TransferDirection, uint16_t
AddrMatchCode)
{
	if (TransferDirection == I2C_DIRECTION_TRANSMIT)  // if the master wants to transmit the
data
	{
		RxData[0] = 0;  // reset the RxData[0] to clear any residue address from previous call
		rxcount =0;
		HAL_I2C_Slave_Seq_Receive_IT(hi2c, RxData+rxcount, 1, I2C_FIRST_FRAME);
	}
	else
	{
		txcount = 0;
		startPosition = RxData[0];
		RxData[0] = 0;  // Reset the start register as we have already copied it
		HAL_I2C_Slave_Seq_Transmit_IT(hi2c, I2C_REGISTERS+startPosition+txcount, 1,
I2C_FIRST_FRAME);
	}
}
void HAL_I2C_SlaveTxCpltCallback(I2C_HandleTypeDef *hi2c)
{
	txcount++;
	HAL_I2C_Slave_Seq_Transmit_IT(hi2c, I2C_REGISTERS+startPosition+txcount, 1,
I2C_NEXT_FRAME);
}
void HAL_I2C_SlaveRxCpltCallback(I2C_HandleTypeDef *hi2c)
{
	rxcount++;
	if (rxcount < RxSIZE)
	{
		if (rxcount == RxSIZE-1)
		{
			HAL_I2C_Slave_Seq_Receive_IT(hi2c, RxData+rxcount, 1,
I2C_LAST_FRAME);
		}
		else
		{
			HAL_I2C_Slave_Seq_Receive_IT(hi2c, RxData+rxcount, 1,
I2C_NEXT_FRAME);
		}
	}
	if (rxcount == RxSIZE)
	{
		process_data();
	}
}
void HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c)
{
	uint32_t errorcode = HAL_I2C_GetError(hi2c);
	if (errorcode == 4)  // AF error
	{
```

```
                if (txcount == 0)  // error is while slave is receiving
                {
                        bytesRrecvd = rxcount-1;  // the first byte is the register address
                        rxcount = 0;  // Reset the rxcount for the next operation
                        process_data();
                }
                else // error while slave is transmitting
                {
                        bytesTransd = txcount-1;  // the txcount is 1 higher than the actual data
transmitted
                        txcount = 0;  // Reset the txcount for the next operation
                }
        }
        /* BERR Error commonly occurs during the Direction switch
         * Here we the software reset bit is set by the HAL error handler
         * Before resetting this bit, we make sure the I2C lines are released and the bus is free
         * I am simply reinitializing the I2C to do so
         */
        else if (errorcode == 1)  // BERR Error
        {
                HAL_I2C_DeInit(hi2c);
                HAL_I2C_Init(hi2c);
                memset(RxData,'\0',RxSIZE);  // reset the Rx buffer
                rxcount =0;  // reset the count
        }
        HAL_I2C_EnableListen_IT(hi2c);
}
```