ECE 355
Lab Report

December 9, 2020
ECE 355 B10
Amaan Makhani
V00883520

Marking Rubric

Category	Mark	Section Total
Problem Description/Specifications		5
Design/Solution		15
Testing/Results		10
Discussion		15
Code Design and Documentation		15
Total		60

Table Of Contents

List of Figures

1. Problem Description

The objective of this project was to develop an embedded system for monitoring and controlling a pulse-width modulated (PWM) signal. This PWM signal was to be generated by an external NE555 timer [1].

To control the system, we used an 4N35 optocoupler, which was to be controlled by the microcontroller on the STMF0 Discovery board. The system was to work by using the microcontrollers built-in analog-to-digital converter (ADC) and digital-to-analog converter (DAC). The DAC was to be used to drive the optocoupler to adjust the signal frequency of the 555-timer.The DAC is to convert the digital value to an analog voltage signal to drive the optocoupler. The ADC is used to measure the voltage across the potentiometer. The analog voltage signal coming from the potentiometer was to be measured continuously through polling [1] .

The microcontroller was also used to measure the frequency of the generated signal. The measured frequency and the POT resistance were displayed on the LCD screen. The system diagram is shown below in ***figure 1***.

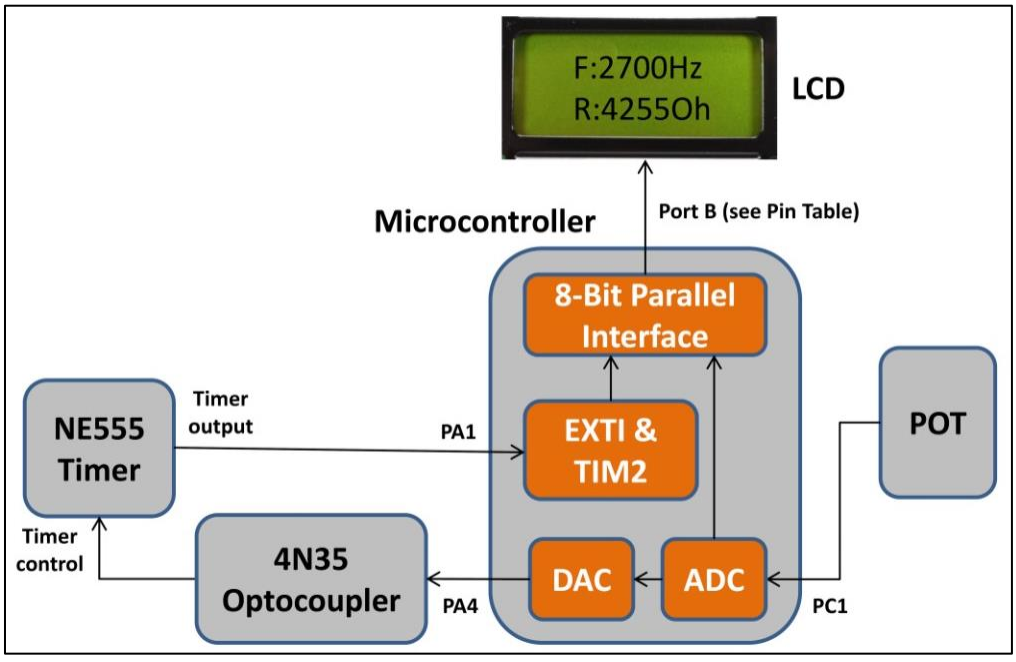

Figure 1: System Design Schematic [1]

To program the microcontroller the Eclipse IDE was used, and the 555-timer program was used to display the LCD and control the resistance across the potentiometer. Due to the nature of our online classes the lab computer was accessed through the UVic Engineering and Computer Science Remote Computing site.

2. Solution Design

This design had five components that needed to work together to complete the specified task.

2.1 ADC

The Analog to Digital Converter built into the microcontroller was used to measure the voltage across the potentiometer. The board used a 5kΩ potentiometer which was connected to an GPIOC through pin 1. To initialize the ADC the program cleared the ADC, enabled GPIOC, set PC1 to analog, enabled the ADC clock, enabled channel 11 and synchronous mode, and then calibrates the ADC. After this initialization I could read the ADC values by turning on the ADC on and polling it to see if it has completed a conversion, then saving the conversion result, and enabling another conversion. These values are read from PC1. The ADC could read values from 0-4095. This is because the ADC register can store 12 bits. Once obtained I used the formula below to convert the digital value into voltage measurement [2].

$$V_{in} = \frac{(\text{ADC Value})(\text{Reference Voltage})}{\text{Resolution}}$$

The board has a reference voltage of 3.3v, as discussed before the resolution value is 4095 due to the 12-bit ADC register. The current was calculated to be V/R_{max} = 0.00066 A. The potentiometer resistance is calculated to be $V_{in}/0.00066$. This leaves us with the formula below and this is what I used to obtain my potentiometer resistance.

$$R_{\text{potentiometer}} = \frac{\left(\frac{(\text{ADC Value})(3.3)}{4095}\right)}{0.00066}$$

2.2 DAC

The Digital to Analog Converter outputs its data using PA4. After measuring the value from the ADC, the measured value is written to the DAC with no conversions occurring. After receiving the 12-bit ADC value the DAC outputs a 0-3.3 V value and

relays it to the optocoupler. To initialize the DAC we clear it, set PA4 as the output pin, then enable the DAC clock, and the DAC. To write to the DAC we set the data holding register to the ADC value. The data holding register then automatically transfer the data to the output register.

2.3 Signal generating circuit

A 4N35 Optocoupler and an NE555 Timer were used to generate the PWM signal. The output from the DAC of the controller was fed into pin 1 of the optocoupler. The signal was generated out of pin 3 of the timer then relayed back to the microcontroller for a frequency measurement. The schematic of the external circuit is shown in ***figure 2***.

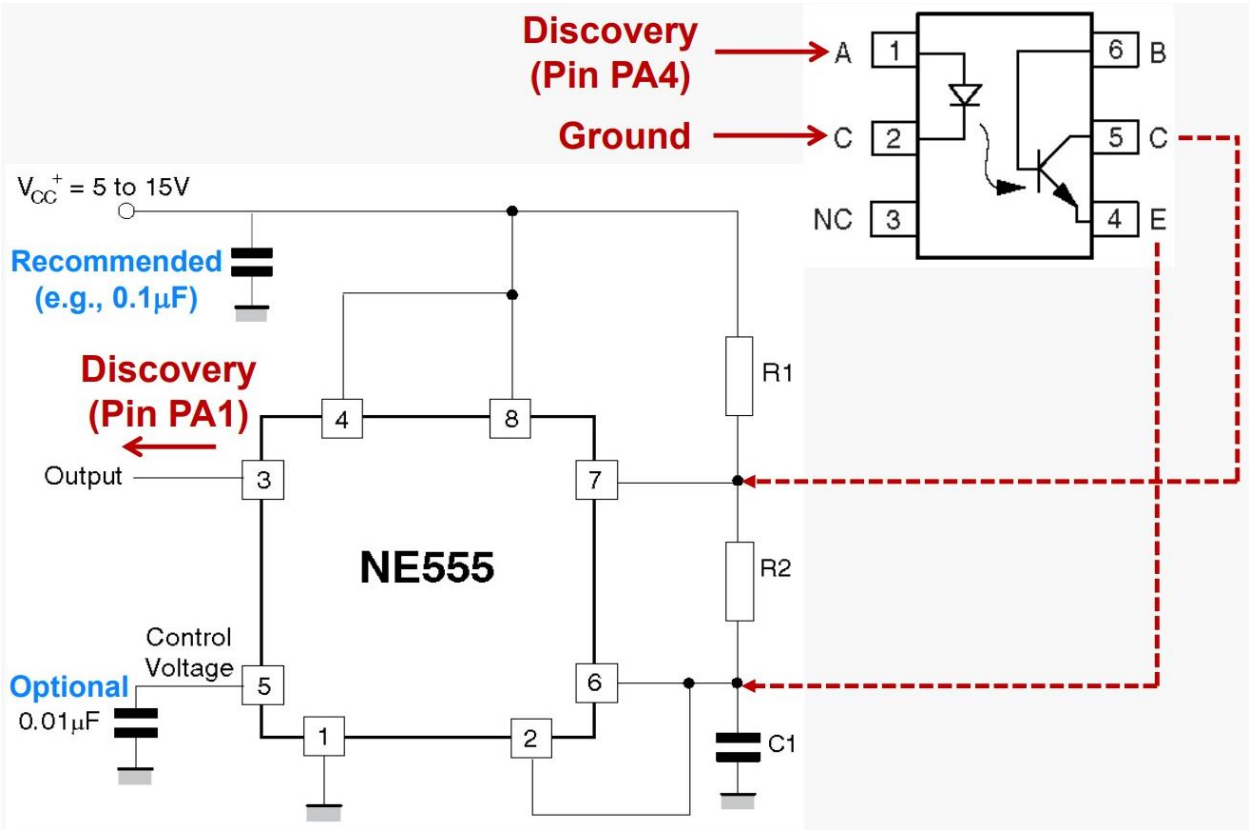

Figure 2: Timer circuit designed to generate PWM signal [3]

Using this circuit, the frequency of the signal out of the NE555 timer was approximately in the range of 800 Hz to 1550 Hz. This was near the expected range

given by the professor. When writing the DAC voltage to the optocoupler we can see the NE555 timer interface shows a voltage change when the resistance is varied. The min and max range of the DAC was shown to be 0.15 V to 2.17 V, the expected range was 0.06 V to 2.2 V. This circuit worked as expected and was well within the operating range values.

2.4 Frequency measurement

The frequency that was measured by the microcontroller were generated by the NE555 timer through PA1. To measure the frequency, an interrupt approach was used. During the EXTI initialization we map EXTI1 to PA1, set the EXTI1 line interrupts to the rising edge as a trigger, then unmask the interrupt line, set the interrupt priority, and enable the interrupt. After this initialization, the EXTI0_1_IRQHandler is responsible for calculating the frequency and handling the interrupts. In the interrupt handler if it is the first edge, we set a flag to indicate we have seen it, clear the current timer, dismiss the interrupt, and start the timer. If the edge is the second edge (we know this is the case given that the flag is set), we retrieve the number of timer pulses elapsed. The timer counts at its clock speed. To calculate the frequency, we use the formula below. The value of the counter is represented as # of clock cycles elapsed and the system clock speed was 48 MHz.

$$Frequency = System\ clock\ speed\ /\#\ of\ Clock\ cycles\ Elapsed$$

2.5 LCD Display

To display the frequency and the potentiometer resistance we outputted the data to an LCD display using an 8-bit parallel interface to communicate with the LCD. The interface uses includes 4 control signals (ENB, RS, R/W, DONE) and it is accompanied 8 data signals (D0-D7). These signals are mapped to specific pins.

To use the LCD display we first initialize GPIOB by enabling it then setting pins 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, and 15 to be output pins. We use PB7 as an input pin to see

if the display has processed the instruction. This is seen by using the pin table in ***figure 3***.

STM32F0	SIGNAL	DIRECTION
PB4	ENB (LCD Handshaking: "Enable")	OUTPUT
PB5	**RS** (0 = COMMAND, 1 = DATA)	OUTPUT
PB6	**R/W** (0 = WRITE, 1 = READ)	OUTPUT
PB7	DONE (LCD Handshaking: "Done")	INPUT
PB8	**D0**	OUTPUT
PB9	**D1**	OUTPUT
PB10	**D2**	OUTPUT
PB11	**D3**	OUTPUT
PB12	**D4**	OUTPUT
PB13	**D5**	OUTPUT
PB14	**D6**	OUTPUT
PB15	**D7**	OUTPUT

Figure 3: LCD Pin Mapping [3]

To initialize the LCD display we first set the function mode, then toggle the display status, set the entry mode, then clear the display. After the initialization we can write data by first clearing the ODR register, then write the data to the ODR register, enable the handshake, wait for the LCD to assert done, desert the handshake, and finally wait for the LCD to be deserted.

That enables us to print data or send instructions to the LCD display. To print our resistance or frequency we isolate each digit through mod operations and division (this process is shown in **Appendix A**). When we have something to write to the first line and first spot for example, we first set the DRAM address using this position mapping shown in ***figure 4***. Once accessed the DRAM will automatically increment a position. To print a digit, we write the digit and or this to the hex

number 0x30 to get the ACII representation of a number. The LCD display is sent 8-bits at a time. We shift the data to PB[8:15], as these are D0-D7 respectively. The way to choose which mode was executed for each instruction is to set the RS bit. RS = 1 and R/W = 0 to write into a DDRAM address. RS = 0 and R/W = 0 is to set a DDRAM address.

Display position	1	2	3	4	5	6	7	8
DDRAM address	00	01	02	03	04	05	06	07
	40	41	42	43	44	45	46	47

Figure 4: DRAM Address Positions [3]

3. Test Procedures and Results

Tests were used to verify individual system components worked as expected prior to their implementation in the system. The components were added and tested as follows:

1. LCD display
2. ADC
3. DAC
4. Signal generating circuit
5. Frequency measurement

The intent of these tests was to ensure the system was functioning correctly and the tests used to verify this are detailed below.

3.1 LCD Display

I chose to begin with the implementation of the LCD display to ensure I could write the data in the correct format when the data was ready. This made it easier for me to debug my measurements in later implementation steps.

Objective

To determine if the LCD display is correctly connected and receiving data from GPIOB. I also tested to see if the data can be written in the format shown in ***figure 5***.

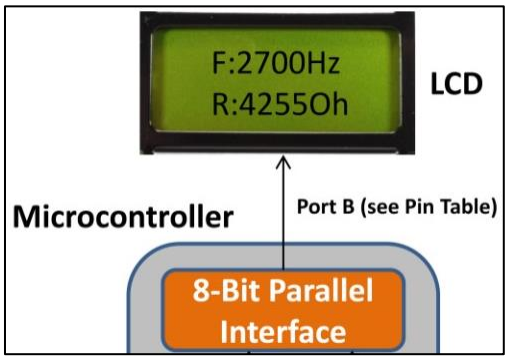

Figure 5: LCD Display Format [1]

Criteria to Meet for Acceptable Performance

The output of the LCD display is in the correct format as mentioned previously, the LCD displays the given test frequency and test voltage, and the LCD display can continually be updated.

Testing Steps

1. Initialize the registers for GPIOB with the correct values. The initialization values include enabling the clock and setting the direction for the output pins.
2. Initialize the LCD to set the function register, set the display state, set the entry mode, and then clear the display.
   NOTE: As a debugging tool the cursor was turned on and set to blink.
3. Run the LCD initialization code after the GPIOB initialization code is run.
4. Ensure that each command is taking place in the LCD initialization code and the effects like clearing the screen is visible.
5. Write the code to write the ASCII representation of the words and numbers to display on the LCD.
6. Run the code to write some test data, i.e. 9000 for both values.
7. Verify that the intended data is correctly displaying on the LCD.
8. Constantly update the display and subtract one each time to verify the display's correct behavior.
9. Verify that data is changing on the LCD and update the update rate as needed.

3.2 ADC

The second completed part of this project was the initialization and usage of the analog to digital converter.

Objective

To determine if the analog voltage input is read correctly.

Criteria to Meet for Acceptable Performance

As the potentiometer resistance is change the ADC should display roughly the correct resistance.

the voltage values read by the

STM are accurate. Calculated resistance and voltage ranges match physical ranges.

Testing Steps

1. Initialize the ADC pins, channel, mode, clear the existing registers, and enable the ADC.
2. Read the ADC value via a polling approach.
3. Display a constantly updated ADC value to ensure data is being transmitted through the ADC.
4. Calculate the resistance value using the formula obtained in the system design phase.
5. Run the code to measure the resistance and output the data to the LCD display.
6. Vary the potentiometer resistance from minimum to maximum values, ensure the values are relatively correct. Relative correctness can be with 10-20% as stated by the professor.

3.3 DAC

Once the ADC had been set up and tested, the DAC can be used to transmit the data to the optocoupler.

Objective

To determine if digital values sent to the DAC are correctly converted into analog voltage values.

Criteria to Meet for Acceptable Performance

The voltage output of the DAC is identical to the voltage measured across the potentiometer.

Testing Steps

Due to the inability to run these tests these tests were different than if we had physical access to the board.

Physical access:

1. Use a Digital Multimeter to measure the voltage between pin PA4 and ground.
2. While turning the potentiometer, ensure that voltage values measured by DMM change from 0.06-2.2V. Again, this is subject to a 10-20% as stated by the professor.

What I tried:

1. Initialize the DAC by clearing it, set PA4 as output, and enabling it.
2. Then send the digital value passed in from the ADC and write it to the data holding register.
3. I tried to verify this value was transferred but was unable to set up the pins and registers correctly in order to verify the transfer, so this verified again after the frequency generation was completed.

3.4 Signal Generating Circuit

Using the output from the DAC, the PWM signal was to be generated and tested.

Objective

To ensure that a correct signal is being generated by the optocoupler and timer circuit.

Criteria to Meet for Acceptable Performance

A PWM wave is being generated out of the output pin of the timer.

Testing Steps

1. Using the NE555 and 4N35, create the circuit shown in discussed in the design section. DAC has already outputted to PA4 and we assume this works until needing to troubleshoot.
2. Initialize EXTI1 using PA1 as the output and create an interrupt handler for EXTI1.

   NOTE: We have tested our interrupt handler frequency measurement in part 2 of Lab 1 so we are assuming this calculation is accurate and reuse majority of this code. However, we change the EXTI we use and the PA we use.
3. Since we cannot use an oscilloscope to measure the signal, we use the timer interface to verify the voltage changes as the resistance changes. This proves our DAC works.
4. This should indicate our circuit works and we will troubleshoot this further if the frequency measurement runs into problem.

3.5 Frequency Measurement

The final part is to measure the frequency generated by the NE555 timer.

Objective

To determine the frequency is generated by the NE555 timer.

Criteria to Meet for Acceptable Performance

The frequency determined by the code varies between the range of 800- 1600 Hz as given by the professor.

Testing Steps

1. We first run our interrupt and ensure the interrupt is triggered.
2. We print the frequency returned from our interrupt handler to the LCD display.
3. Since we were unable to use a function generator, I was unable to truly verify the frequency, but I just ensured it varied with the given range.

4. Now the resistance and frequency will be outputting to the board, once both values and the LCD is operating as expected the system is complete.

4. Discussion

The development of this project was a very educating and difficult experience. Not only were we tasked with a project that could often be hard to debug but we were forced to a remote learning environment due to circumstances beyond our control. We used software like Eclipse and a timer interface to develop this project. These tools were used through the UVIC remote computing facility. In this remote setup I ran into many stations that had Eclipse fail to match the expected clock speed. This forced me to switch stations which was irritating and unproductive. As mentioned before if you fail to set a register you might not be allowing some components to interact which was notoriously hard to debug. I also struggled with learning about some of the STM's functionality without concrete examples. Instead I had to do extensive research through user manuals and reference documents. One such problem like that was you could not just refer to ADC you had to refer to ADC1, this took me a little while to troubleshoot. We also didn't have access to the physical tools for measurement of waves and electrical values. This would have helped to test individual parts prior to adding another component as mentioned in the testing section. I think examples of LCD use would have been helpful as I felt lost and without enough resources to complete this without additional assistance.

After completing the lab demonstration of our final project, the circuit and code executed as expected. My system was well within the ranges and was very accurate which I was proud of.

Having the experience in completing this project and dealing with setbacks, I have a few takeaways for future students. Although seemingly trivial in concept, it is crucial to get a thorough understanding of each section before completing the individual task. Organizing the applicable data sheets and reference notes for easy access when working on each part of this project is extremely important and will save a lot of time. Secondly, this project is not to be underestimated. A substantial amount of time is needed to complete this project, so starting early and asking for help quickly is helpful. Lastly a plan of tasks laid out would also be helpful. This lab was enjoyable overall, and I am proud of my project.

5. References

[1] Rakhmatov, D., 2020. Electrical And Computer Engineering ECE 355: Microprocessor-Based Systems Laboratory Manual (ONLINE). [ebook] Available at: <https://www.ece.uvic.ca/~ece355/lab/ECE355-LabManual-2020.pdf> [Accessed 1 December 2020].

[2] 2014. Stm32f0x1/Stm32f0x2/Stm32f0x8 Advanced ARM-Based 32-Bit Mcus. [ebook] STMicroelectronics. Available at: <https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f0RefManual.pdf> [Accessed 1 December 2020].

[3] Rakhmatov, D., 2020. Interface Examples. [ebook] Available at: <https://www.ece.uvic.ca/~daler/courses/ece355/interfacex.pdf> [Accessed 1 December 2020].

Appendix A: Project Source Code

```
// ----------------------------------------------------------------------------
// School: University of Victoria, Canada.
// Course: ECE 355 "Microprocessor-Based Systems".
// See "system/include/cmsis/stm32f0xx.h" for register/bit definitions.
// See "system/src/cmsis/vectors_stm32f0xx.c" for handler declarations.
// ----------------------------------------------------------------------------
#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
#include "stm32f0xx.h"
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
#pragma GCC diagnostic ignored "-Wreturn-type"

// Clock prescaler for TIM2 timer: no prescaling
#define myTIM2_PRESCALER ((uint16_t)0x0000)
// Maximum possible setting for overflow
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)

void myGPIOA_Init(void);
void myGPIOB_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
void myTIM2_Init(void);
void myTIM3_Init(void);
void wait(int duration);
void myEXTI_Init(void);
int ReadADC(void);
void WriteDAC(uint32_t digitalval);
void myLCD_Init(void);
void write_to_lcd(uint16_t data);
void printValues(int resistance, int frequency);

// Global variables
int isFirstEdge = 0;
float freq = 0;

int main(int argc, char* argv[]) {
    myGPIOA_Init();     /* Initialize I/O port PA   */
    myGPIOB_Init();     /* Initialize GPIO B        */
    myTIM2_Init();      /* Initialize timer TIM2    */
    myTIM3_Init();      /* Initialize timer TIM2    */
    myEXTI_Init();      /* Initialize EXTI          */
    myADC_Init();       /* Initialize ADC           */
```

```
    myDAC_Init();       /* Initialize DAC           */
    myLCD_Init();       /* initialize LCD           */
    while (1)
    {
        int adcval = ReadADC();              /* Save ADC value to relay to DAC                  */
        WriteDAC(adcval);                    /* Relay value from potentiometer to optocoupler   */
        // Calculate potentiometer resistance
        float potResistance = ((3.3 * (DAC->DOR1))/4095)/0.00066;
        printValues(potResistance, freq);    /* Update LCD with frequency and Resistance values */
        wait(100000);                        /* Delay screen update                             */
    }
    return 0;
}

void myADC_Init(){
    ADC1->CR = ((uint16_t)0x0);              /* Clear ADC                   */
    GPIOC->MODER |= ((uint16_t)0x20);        /* PC1 to analog               */
    RCC->APB2ENR |= ((uint16_t)0x0200);      /* Turn on ADC Clock           */
    ADC1->CHSELR |= ((uint16_t)0x0800);      /* Enable Channel 11           */
    ADC1->CFGR2 |= ((uint16_t)0x0000);       /* Enable synchronous mode     */
    ADC1->CR |= ((uint16_t)0x80000000);      /* Calibrate ADC               */
    while((ADC1->CR == 0x80000000));         /* Wait for ADC to calibrate   */
    ADC1->CR |= ((uint16_t)0x1);             /* Enable ADC                  */
}

int ReadADC () {
    ADC1->CR |= ((uint16_t)0x5);             /* Turn on ADC                 */
    while((ADC1->ISR & 0x4) == 1);           /* Wait for end of conversion  */
    int ADCvalue = ADC1->DR;                 /* Save adc value              */
    ADC1->ISR &= ~(0x4);                     /* Clear end of conversion flag */
    return ADCvalue;
}

void myDAC_Init() {
    DAC->CR = ((uint16_t)0x0);               /* Clear DAC                */
    GPIOA->MODER &= ~((uint32_t)0x300);      /* Set PA4 as Analog out    */
    RCC->APB1ENR |= ((uint32_t)0x20000000); /* Enable DAC Clock         */
    DAC->CR = ((uint16_t)0x01);              /* Enable DAC               */
}

void WriteDAC(uint32_t digitalval) {
    // Data holding register automatically transfers to output register
    DAC->DHR12R1 = (unsigned int)digitalval;
}

void myGPIOA_Init(){
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;       /* Enable clock for GPIOA peripheral    */
```

```
    GPIOA->MODER &= ~(GPIO_MODER_MODER1);   /* Configure PA1 as input                */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);   /* Ensure no pull-up/pull-down for PA1  */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);   /* Ensure no pull-up/pull-down for PA4  */
}

void myTIM2_Init(){
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;     /* Enable clock for TIM2 peripheral            */
    TIM2->CR1 |= ((uint16_t)0x008C);        /* Configure TIM2                              */
    TIM2->PSC |= myTIM2_PRESCALER;          /* Set clock prescaler value                   */
    TIM2->ARR |= myTIM2_PERIOD;             /* Set auto-reloaded delay                     */
    TIM2->EGR |= ((uint16_t)0x0001);        /* Update timer registers                      */
    NVIC_SetPriority(TIM2_IRQn, 0);         /* Assign TIM2 interrupt priority = 0 in NVIC  */
    NVIC_EnableIRQ(TIM2_IRQn);              /* Enable TIM2 interrupts in NVIC              */
    TIM2->DIER |= TIM_DIER_UIE;             /* Enable update interrupt generation          */
}

void myTIM3_Init(){
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;     /* Enable Clock for TIM3 Peripheral */
    TIM3->PSC = 480;                        /* Set prescaler                    */
    TIM3->CR1 = ((uint16_t)0x8C);           /* Update & auto reload             */
    TIM3->ARR = 10;                         /* Auto reload default value        */
    TIM3->EGR = ((uint16_t)0x0001);         /* Update timer registers           */
}

void wait(int duration) {
    TIM3->SR = 0x0;                /* Reset status register                          */
    TIM3->CNT = 1;                 /* Clear count                                    */
    TIM3->ARR = duration;          /* Auto reload with duration requested - timeout value */
    TIM3->EGR = 0x01;
    TIM3->CR1 |= 0x01;             /* Enable clock                                   */
    while((TIM3->SR & 0x1) == 0);  /* Wait until count reached                       */
    TIM3->CR1 = ((uint16_t)0x8C);  /* Reset to value in initialization               */
    TIM3->SR = 0x0;                /* Clear status register                          */
}

void myEXTI_Init()
{
    SYSCFG->EXTICR[0] = SYSCFG_EXTICR1_EXTI1_PA;    /* Map EXTI1 line to PA1                    */
    EXTI->RTSR |= EXTI_RTSR_TR1;                /* EXTI1 line interrupts: set rising-edge trigger */
    EXTI->IMR |= EXTI_IMR_MR1;                  /* Unmask interrupts from EXTI1 line            */
    NVIC_SetPriority(EXTI0_1_IRQn, 0);          /* Assign EXTI1 interrupt priority = 0 in NVIC  */
    NVIC_EnableIRQ(EXTI0_1_IRQn);               /* Enable EXTI1 interrupts in NVIC              */
}

void TIM2_IRQHandler(){
    if ((TIM2->SR & TIM_SR_UIF) != 0){      /* Check if update interrupt flag is indeed set */
        TIM2->SR = ((uint16_t)0x00);        /* Clear update interrupt flag                  */
```

```
        TIM2->CR1 = ((uint16_t)0x01);       /* Restart stopped timer                    */
    }
}

void EXTI0_1_IRQHandler()
{
    if ((EXTI->PR & EXTI_PR_PR1) != 0){      /* Check if EXTI2 interrupt pending flag */
        TIM2->CR1 &= ~(TIM_CR1_CEN);
        if(isFirstEdge == 1){
            TIM2->CNT = 0x00000000;          /* Clear count reg                       */
            TIM2->CR1 |= TIM_CR1_CEN;
            isFirstEdge = 0;
        }else{
            EXTI->IMR &= ~(EXTI_IMR_MR1);    /* Mask EXTI1 interrupts                 */
            isFirstEdge = 1;
            unsigned int timerPulse = TIM2->CNT;
            freq = ((double)SystemCoreClock)/((double)timerPulse);
            EXTI->IMR |= EXTI_IMR_MR1;       /* Unmask EXTI1 interrupts               */
        }
        EXTI->PR |= EXTI_PR_PR1;             /* Clear interrupt flag                  */
    }
}

void myGPIOB_Init() {
    // Enable GPIOB clock
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
    // Enable pins for output
    GPIOB->MODER = 0x0;
    GPIOB->MODER |= GPIO_MODER_MODER4_0;
    GPIOB->MODER |= GPIO_MODER_MODER5_0;
    GPIOB->MODER |= GPIO_MODER_MODER6_0;
    GPIOB->MODER |= GPIO_MODER_MODER8_0;
    GPIOB->MODER |= GPIO_MODER_MODER9_0;
    GPIOB->MODER |= GPIO_MODER_MODER10_0;
    GPIOB->MODER |= GPIO_MODER_MODER11_0;
    GPIOB->MODER |= GPIO_MODER_MODER12_0;
    GPIOB->MODER |= GPIO_MODER_MODER13_0;
    GPIOB->MODER |= GPIO_MODER_MODER14_0;
    GPIOB->MODER |= GPIO_MODER_MODER15_0;
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR5);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR6);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR8);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR9);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR10);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR11);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR12);
```

```
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR13);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR14);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR15);
}

void myLCD_Init() {
    write_to_lcd(0x3800);   /* Function set     */
    write_to_lcd(0x0F00);   /* Display on/off   */
    write_to_lcd(0x0600);   /* Entry mode       */
    write_to_lcd(0x0100);   /* Clear display    */
}

void write_to_lcd(uint16_t data) {
    GPIOB-> ODR = 0x0000;                       /* Clear ODR register                */
    GPIOB-> ODR |= data;                        /* Write data                        */
    GPIOB-> ODR |= GPIO_ODR_4;                  /* Make PB[4] = 1 (assert “Enable”)   */
    // Wait for PB[7] to become 1 (“Done” to be asserted)
    while((GPIOB-> IDR & GPIO_ODR_7) == 0);
    GPIOB-> ODR  &= ~(GPIO_ODR_4);              /* Make PB[4] = 0 (dissert “Enable”) */
    // Wait for PB[7] to become 0 (“Done” to be deserted)
    while((GPIOB-> IDR & GPIO_ODR_7) != 0);
}

void printValues(int resistance, int frequency) {
    // Isolate each digit
    int digit4 = (resistance % 10);
    int digit3 = (((resistance - digit4) % 100) / 10);
    int digit2 = (((resistance - digit4 - (10*digit3)) % 1000) / 100);
    int digit1 = (resistance/1000);
    write_to_lcd(0x8000);                           /* Set DDRAM address (first line)   */
    write_to_lcd(0x5220);                           /* R                                */
    write_to_lcd(0x3A20);                           /* :                                */
    write_to_lcd((((0x30 | digit1) << 8) | 0x20));      /* #                                */
    write_to_lcd((((0x30 | digit2) << 8) | 0x20));      /* #                                */
    write_to_lcd((((0x30 | digit3) << 8) | 0x20));      /* #                                */
    write_to_lcd((((0x30 | digit4) << 8) | 0x20));      /* #                                */
    write_to_lcd(0x4F20);                           /* O                                */
    write_to_lcd(0x4820);                           /* H                                */
    // Isolate each digit
    digit4 = (frequency % 10);
    digit3 = (((frequency - digit4) % 100) / 10);
    digit2 = (((frequency - digit4 - (10*digit3)) % 1000) / 100);
    digit1 = (frequency/1000);
    write_to_lcd(0xC000);                           /* Set DDRAM address (second line)  */
    write_to_lcd(0x4620);                           /* F                                */
    write_to_lcd(0x3A20);                           /* :                                */
    write_to_lcd((((0x30 | digit1) << 8) | 0x20));      /* #                                */
```

```
    write_to_lcd((((0x30 | digit2) << 8) | 0x20));      /* #                          */
    write_to_lcd((((0x30 | digit3) << 8) | 0x20));      /* #                          */
    write_to_lcd((((0x30 | digit4) << 8) | 0x20));      /* #                          */
    write_to_lcd(0x4820);                              /* H                          */
    write_to_lcd(0x7A20);                              /* Z                          */
}
#pragma GCC diagnostic pop
// ----------------------------------------------------------------------------------------
```