UNIVERSITY OF VICTORIA

Department of Electrical and Computer Engineering

ECE 355 - Microprocessor-Based Systems

Laboratory Report

Title: PWM Signal Generation and Monitoring System

Date of Experiment: November 7, 2024

Report Submitted on: November 21, 2024

Names: (please print): Anthony Guillen V00954349
Ashley McPherson V00982588

The lab project report marks are distributed as follows:

✓	Problem Description/Specifications:	(5) _____
✓	Design/Solution:	(15) _____
✓	Testing/Results:	(10) _____
✓	Discussion:	(15) _____
✓	Code Design and Documentation:	(15) _____
✓	Total:	(60) _____

Table of Contents

1.0 Problem Description/Specifications

The PWM Signal Generation and Monitoring System lab project aims to develop an embedded system using the STM32F0 Discovery board [1]. Using the microcontroller, the system will measure the frequency of a pulse-width-modulated (PWM) square-wave signal from a 555 timer (NE555 IC) and the frequency of a square-wave signal from a function generator. Additionally, it will calculate the resistance value of a potentiometer (POT). This project integrates various components, including an analog-to-digital converter (ADC), digital-to-analog converter (DAC), 555 timer, function generator, USER button, and LED display, to achieve these objectives.

The potentiometer on the PBMCUSLK board produces an analog voltage signal that is measured constantly by the ADC using a polling approach [1]. The voltage measured is used to calculate the potentiometer resistance value. The ADC's digital value will be converted into an analog signal by the DAC, which will drive an optocoupler (4N35 IC) to adjust the PWM signal's frequency and duty cycle produced by the 555 timer. The 555 timer signal frequency is measured using PA1 and EXTI1.

The function generator uses the SYNC output and a square wave input signal, which is measured by a TIM2 general purpose timer [1]. The clock cycles are tracked in the TIM2_CNT counter register, which must be configured to increment with each cycle of TIM2's clock while TIM2 is active. The microcontroller's PA2 I/O pin generates interrupt requests when a rising (or falling) edge of the input signal is detected. The EXTI2 interrupt handler is utilized here and accesses TIM2.

To alternate between measuring the frequency of the 555 timer and the function generator, the USER button is pressed, triggering an interrupt request [1]. The USER button interrupt is connected to PA0 via EXTI0. Pressing the button switches the system to display the specific frequency and resistance corresponding to either the 555 timer or the function generator on the LED display. The LED display on the PBMCUSLK board communicates with the microcontroller using SPI, which consists of one serial data signal (SDIN = SPI MOSI), one clock signal (SCLK = SPI SCK), and three control signals (RES#, CS#, and D/C#). See Table 1.0 below [1].

TABLE 1.0 Signals and Associated Pin Information [1]

STM32F0	Signal	Direction
PA0	USER push button	Input
PC8	Blue LED	Output
PC9	Green LED	Output
PA1	555 timer	Input
PA2	Function generator	Input
PA4	DAC	Output (Analog)
PA5	ADC	Input (Analog)
PB3	SCLK (Display D0: “Serial Clock” = SPI SCK)	Output (AF0)
PB4	RES# (Display “Reset”)	Output
PB5	SDIN (Display D1: “Serial Data” = SPI MOSI)	Output (AF0)
PB6	CS# (Display “Chip Select”)	Output
PB7	D/C# (Display “Data/Command”)	Output

2.0 Design/Solution

2.1 Design Steps

To accomplish the problem description, we divided the laboratory project into six design steps: implementing the ADC, DAC, USER button, function generator, 555 timer, and LED display. Splitting the project into individual design steps allowed for complete focus on implementing and testing one component at a time. If the previous design steps were functioning successfully, they would not interfere with subsequent steps.

The ADC and DAC were implemented first because they were required for the 555 timer's operation. Then the USER button was implemented to enable switching between the function generator and 555 timer once they were implemented. The function generator was completed before the 555 timer was implemented because the code was already written in the *Part 2: Signal Frequency Measurement* lab. Lastly, the LED display was completed last because it relied on all previous components to be functioning successfully to ensure the displayed values were correct.

2.1.1 Step 1: ADC

Accomplishing task 1 of setting up the analog-to-digital converter consisted of creating an intialize_ADC function that would consist of setting the appropriate bits as seen in **Figure 1**. The summary of this function was configuring the ADC for continuous analog-digital conversions for channel 5.

After initialization, inside main's infinite loop we would manipulate a control register for the ADC1 - specifically bit number 2. Setting this bit would start the ADC conversion process but a precautionary step was used after which used a polling method to check if the end of sequence flag had been set to 1. This is a bug preventing step that ensures the result we are getting from the ADC data register is accurate and complete.

2.1.2 Step 2: DAC

Accomplishing the DAC was a simple task that consisted of setting the corresponding bits that can be seen in **Table 1** which allowed us to initialize the DAC. Afterwards, the ADC data register value is passed to the DAC_DHR12R1 register in the same polling loop discussed, to be used by the optocoupler and then the 555 timer.

2.1.3 Step 3: USER button

Building the USER button for this lab was broken into two parts: first, we focused exclusively on detecting if the user button was being pressed, it wasn't until this step was complete that we worried about transitioning between making measurements for the frequency generator and the 555 timer. In order to accomplish the former, we utilized interrupts and interrupt priorities. To utilize interrupts for the USER button, we had to ensure that PA0 was configured correctly for interrupts by setting the corresponding bits - which can be seen in **Figure 1**. Notability though, this configuration was done in such a way that the external interrupt pending register bit number 0 would be set on an interrupt call so that the status of the flag can be used in the next step. Another notable initialization was setting the priority for this interrupt to be 0 in order to guarantee that the button click was detected. For example - if a timer interrupt had higher priority, due to the frequency it would cause interrupts it would cause conflicts with the user button.

```c
void EXTI0_1_Init()
{
    GPIOA->MODER &= ~(GPIO_MODER_MODER0); //Sets Port A Pin 0 as input which corresponds to USER button
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI0_PA; //Map EXTI0 line to PA0
    EXTI->RTSR |= EXTI_RTSR_TR0; //Sensitive to rising edges
    EXTI->IMR |= EXTI_IMR_MR0; // Unmasks interrupts from EXTI0 line
    NVIC_SetPriority(EXTI0_1_IRQn, 0); //Sets priority to 0 in NVIC
    NVIC_EnableIRQ(EXTI0_1_IRQn); //Enables EXTI0 interrupts in NVIC
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA; //Map EXTI1 line to PA1
    EXTI->RTSR |= EXTI_RTSR_TR1; // Sensitive to rising edges on EXTI1
    EXTI->IMR |= EXTI_IMR_MR1;   // Unmask interrupts from EXTI1 line
    EXTI->IMR &= ~EXTI_IMR_MR1; //Disable interrupt for EXTI1
    EXTI->IMR |= EXTI_IMR_MR2;  //Enable interrupt for EXTI2
}
```

Figure 1: User Button interrupt initializer

The next part of configuring our button was to get it to switch the state of interrupts our program cares about. In our code, we care about the 555 timer or the frequency generator. To accomplish this, the EXTI0_1_IRQHandler is used, which gets called when the user button is pressed as per the initialization seen in **Figure 1** above. Since this handler deals with two interrupts (one of which has not yet been discussed), inside the IRQHandler it was necessary to check which pending flag was set that caused the interrupt. We had to use an if statement checking if bit 0 of the EXTI pending register was set - indicating the interrupt was caused by the user button and can now begin switching states.

In order to switch the states between the 555 timer and the frequency generator, we utilized a global variable that would keep track of which state we were in. With the global variable in place, we used an if/else statement to either enable the interrupts from the frequency generator and disable those from the 555 timer or vice versa. Inside the if statement this was done by setting or clearing the corresponding bits in the EXTI interrupt mask registers as seen in **Figure 2**. At the end of this case, it was necessary to clear the pending flag because hardware itself does not clear the flag.

```c
if(EXTI->PR & EXTI_PR_PR0)
{
    if(globalSignal == 0)
    {
        globalSignal = 1;
        EXTI->IMR &= ~EXTI_IMR_MR1; //Disable interrupt for EXTI1
        EXTI->IMR |= EXTI_IMR_MR2;  //Enable interrupt for EXTI2
        trace_printf("Function generator enabled\n");
    }else if(globalSignal == 1)
    {
        globalSignal = 0;
        EXTI->IMR &= ~EXTI_IMR_MR2; //Disable interrupt for EXTI2
        EXTI->IMR |= EXTI_IMR_MR1;  //Enable interrupt for EXTI1
        trace_printf("555 timer enabled\n");
    }
    EXTI->PR |= EXTI_PR_PR0; //Clear EXTI0 Pending flag
}
```

Figure 2: Switch case inside the EXTI0_1_IRQHandler

2.1.4 Step 4: Function Generator

Constructing the function generator utilized timers, interrupts and knowledge about rising edges. The first step as usual required initializations for the relevant ports and timers which in our case were timer 2 and port A2. The initialization for timer 2 was done in such a way that the timer would count up starting from 0 at a rate of 48MHz - as seen in **Figure 3**. The initialization for PA2 was short and consisted of only 5 lines as seen in **Figure 4**, they key points of this initialization is that PA2 was configured for input to take data in from the frequency generator and produce an interrupt to the EXTI2_3_IRQHandler at every occurring rising edge from the frequency generator. This interrupt was given a priority of 1 to ensure that no delays are caused by other interrupts with a higher priority.

```c
void myTIM2_Init()
{
    /* Enable clock for TIM2 peripheral */
    // Relevant register: RCC->APB1ENR
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;

    /* Configure TIM2: buffer auto-reload, count up, stop on overflow,
     * enable update events, interrupt on overflow only */
    // Relevant register: TIM2->CR1
    TIM2->CR1 = ((uint16_t)0x008C);

    /* Set clock prescaler value */
    //TIM2->PSC = myTIM2_PRESCALER;
    TIM2->PSC = ((uint16_t)0x0000);

    /* Set auto-reloaded delay */
    TIM2->ARR = myTIM2_PERIOD;

    /* Update timer registers */
    // Relevant register: TIM2->EGR
    TIM2->EGR = ((uint16_t)0x0001);

    /* Assign TIM2 interrupt priority = 2 in NVIC */
    // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
    NVIC_SetPriority(TIM2_IRQn, 2);


    /* Enable update interrupt generation */
    // Relevant register: TIM2->DIER
    TIM2->DIER |= TIM_DIER_UIE;

    /* Start Counting Timer Pulses*/
    TIM2-> CR1 |= TIM_CR1_CEN;
}
```

Figure 3: Initialization of TIM2

```c
void myEXTI2_3_Init()
{
    /* Map EXTI2 line to PA2 */
    // Relevant register: SYSCFG->EXTICR[0]
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI2_PA;

    /* EXTI2 line interrupts: set rising-edge trigger */
    // Relevant register: EXTI->RTSR
    EXTI->RTSR |= EXTI_RTSR_TR2;

    /* Unmask interrupts from EXTI2 line */
    // Relevant register: EXTI->IMR
    EXTI->IMR |= EXTI_IMR_MR2;

    /* Assign EXTI2 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
    NVIC_SetPriority(EXTI2_3_IRQn, 1);

    /* Enable EXTI2 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(EXTI2_3_IRQn);
}
```

Figure 4: Initializer for input from the function generator

Inside the EXTI2_3_IRQHandler there was only one interrupt we ended up using, that is, EXTI2 described above. Out of good practice though, we still checked the interrupt pending flag before measuring the frequency of the signal. Inside this handler dealt with 3 global variables, that is, timeTriggered, Freq and Res. timeTriggered was combined with an if else statement to check if the value is either 0 or 1. In the case of the timeTriggered being 0, this indicates a new rising edge and it would set the initial value of TIM2 to 0 and start the timer, additionally, it would set timeTriggered to be 1 so that upon the next rising edge the else statement would be executed. Inside the else statement (when timeTriggered = 1), this is where the calculations for the frequency were done as seen in **Figure 5**. The first necessary step was to stop the TIM2 by using the TIM2_CR1 register and then storing the value the TIM2 counted to which was stored in the TIM2_CNT register. This value needed to be divided by the clock value of 48,000,000 to obtain

the frequency value and was passed to the global variable Freq to be used by the LED display later, code details may be seen in **Figure 5**. At the end of both of these cases the pending flag for EXTI2 was required to be cleared because it does not get cleared by hardware.

```
if(!timerTriggered)
{
    TIM2->CNT = 0;
    TIM2->CR1 |= TIM_CR1_CEN;
    timerTriggered = 1;
}

else
{
    TIM2->CR1 &= ~(TIM_CR1_CEN);
    uint32_t timerValue = TIM2->CNT;
    float_frequency = 48000000 / timerValue;
    float_scaledPeriod = 1/float_frequency;
    frequency = float_frequency;
    Freq = float_frequency;

    Res = 0;
    timerTriggered = 0;
}
```

Figure 5: Method solved to calculate frequency for function generator

2.1.5 Step 5: 555 Timer

The method of solving the 555 timer used the same IRQ handler as the user button - that is EXTI_0_1_IRQHandler but in the case of the 555 timer it utilized line 1 of this routine. The initialization for this can be seen in **Figure 1** where notability EXTI line 1 was connected to PA1 and made sensitive to rising edges.Since the user button and the 555 timer had the same interrupt handler, we had to check if the pending flag for EXTI_PR1 was set, this would indicate that the interrupt was due to the 555 timer. After concluding that the pending flag EXTI_PR_PR1 was set, the calculations for solving the resistance and frequency could be solved. The method for doing this was very similar to that of the frequency generator and utilized the same global variables - that is, timerTriggered, Freq and Res. The only difference between the 555 timer method and the Frequency method was the math to solve for the resistance value which can be seen in **Figure 6**. The derivation from this equation came from knowing the maximum and minimum value, that is, 5000 and 0 respectively, therefore when the ADC data register reached the max value of 4095 there should be a 4095 in the denominator and 5000 in the numerator to result in 5000 as seen in **Figure 6**. As usual, the pending flag corresponding to line 1 must be cleared when finishing this function so that future conflictions don't occur.

```
if(!timerTriggered)
    {
        TIM2->CNT = 0;
        TIM2->CR1 |= TIM_CR1_CEN;
        timerTriggered = 1;
    }

    else
    {
        TIM2->CR1 &= ~(TIM_CR1_CEN);
        uint32_t timerValue = TIM2->CNT;
        float_frequency = 48000000 / timerValue;
        float_scaledPeriod = 1/float_frequency;
        frequency = float_frequency;
        //trace_printf("555 Frequency: %u.%02u Hz\n", frequency, frequency % 100);
        Freq = frequency;
        Res = (ADC1->DR * 5000) / 4095;
        //trace_printf("Resistance: %.2f\n", Res);
        timerTriggered = 0;

    }
EXTI->PR |= EXTI_PR_PR1; //Clear pending flag
```

Figure 6: Method for solving the 555 timer frequency and resistance

2.1.6 Step 6: LED Display

To successfully initialize and operate the LED display for showing the correct frequency and resistance values, five functions are required: oled_config(), oled_Write_Cmd(), oled_Write_Data(), oled_Write(), and refresh_OLED(). The initialization process begins with oled_config(), which prepares the LED display by configuring GPIO pins and SPI connections, sending initialization commands with oled_init_cmds, and clearing the display memory. This ensures that the LED display is powered on and starts with a blank screen. The LED display will continue to show a blank screen until refresh_OLED() is called and executed. The purpose of refresh_OLED() is to continuously update the display to show the frequency and resistance values, whether from the 555 timer or the function generator. In the main function, this function is called in an infinite while loop to constantly update the screen with up-to-date values.

The LED display shows two lines, as shown in **Figure 7**. These lines are updated independently in the refresh_OLED() function, with the resistance displayed on the first line and the frequency on the second. A buffer stores the ASCII codes for each character in a line, with a limit of 16 characters. The function calls oled_Write_Cmd() three times to configure the display settings: selecting the page (line), the lower column start address, and the higher column start address. Next, a for loop iterates through each ASCII character in the buffer to retrieve its corresponding 8-byte character data from the Characters array. 8 bytes equal 1 row within the Characters array, which equals 1 ASCII character. The oled_Write_Data() function is called within this for loop.

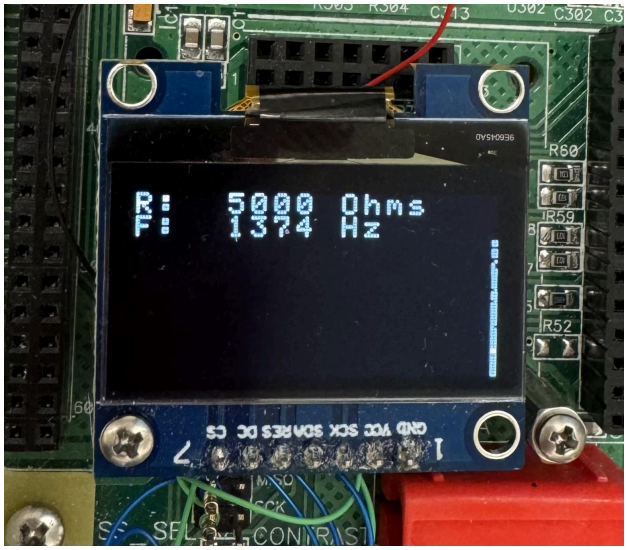

Figure 7: LED display showing maximum resistance and frequency values

```
//Line 1:
snprintf( Buffer, sizeof( Buffer ), "R: %5u Ohms", Res );
/* Buffer now contains your character ASCII codes for LED Display
  - select PAGE (LED Display line) and set starting SEG (column)
  - for each c = ASCII code = Buffer[0], Buffer[1], ...,
      send 8 bytes in Characters[c][0-7] to LED Display
*/
oled_Write_Cmd(0xB0); // Select PAGE
oled_Write_Cmd(0x02); // Lower SEG start address
oled_Write_Cmd(0x10); // Higher SEG start address
for(unsigned int x = 0; Buffer[x] != '\0'; x++)
{
    unsigned char c = Buffer[x];
    for(unsigned int y = 0; y < 8; y++)
    {
        oled_Write_Data(Characters[c][y]); //
    }
}
```

Figure 8: Code to display the first line on the LED display

The frequency and resistance values are calculated within the EXTI0_1_IRQ_Handler() function, as shown in the figures below.

```
float_frequency = 48000000 / timerValue;
float_scaledPeriod = 1/float_frequency;
frequency = float_frequency;
//trace_printf("555 Frequency: %u.%02u Hz\n", frequency, frequency % 100);
Freq = frequency;
```

Figure 9: Frequency calculation

```
Res = (ADC1->DR * 5000) / 4095;
```

Figure 10: Resistance calculation

Both oled_config() and refresh_OLED() functions call oled_Write_Cmd() and oled_Write_Data(). The oled_Write_Data() function sends the 8-byte data to the display as graphical data. The oled_Write_Cmd() function sends a command byte to the LED display to set addresses.

The oled_Write() function is called within the oled_Write_Cmd() and oled_Write_Data() functions. This function uses the Transmit Buffer Empty (TXE) flag to ensure SPI1 is ready for writing (accepting new data), indicated by TXE = 1 in the SPI1_SR register. After sending one 8-bit character, the Busy (BSY) flag checks if the SPI peripheral has completed the transmission process. These flags ensure that the transmission process is not interrupted and writing the data to the LED display is successful.

```
void oled_Write( unsigned char Value )
{
/* Wait until SPI1 is ready for writing (TXE = 1 in SPI1_SR) */
    while(!(SPI1->SR & SPI_SR_TXE))
        {
        trace_printf("Stuck\n");;
        }
    /* Send one 8-bit character:
      - This function also set BIDIOE = 1 in SPI1_CR1
    */
    HAL_SPI_Transmit( &SPI_Handle, &Value, 1, HAL_MAX_DELAY );
    while(SPI1->SR & SPI_SR_BSY); //wait for SPI BSY to not be busy
}
```

Figure 11: oled_Write() function, showing the TXE and BSY flags

2.2 Diagrams

2.2.1 Block Diagrams

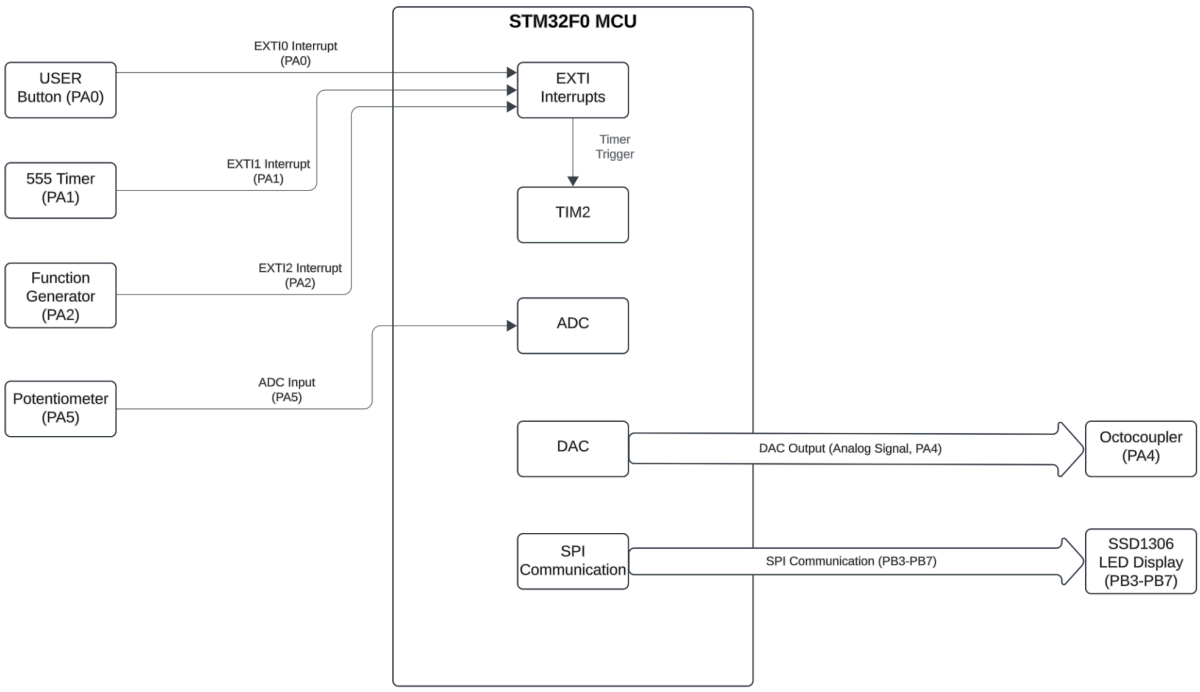

Figure 12: Block diagram to describe the entire PWM Signal Generation and Monitoring System project

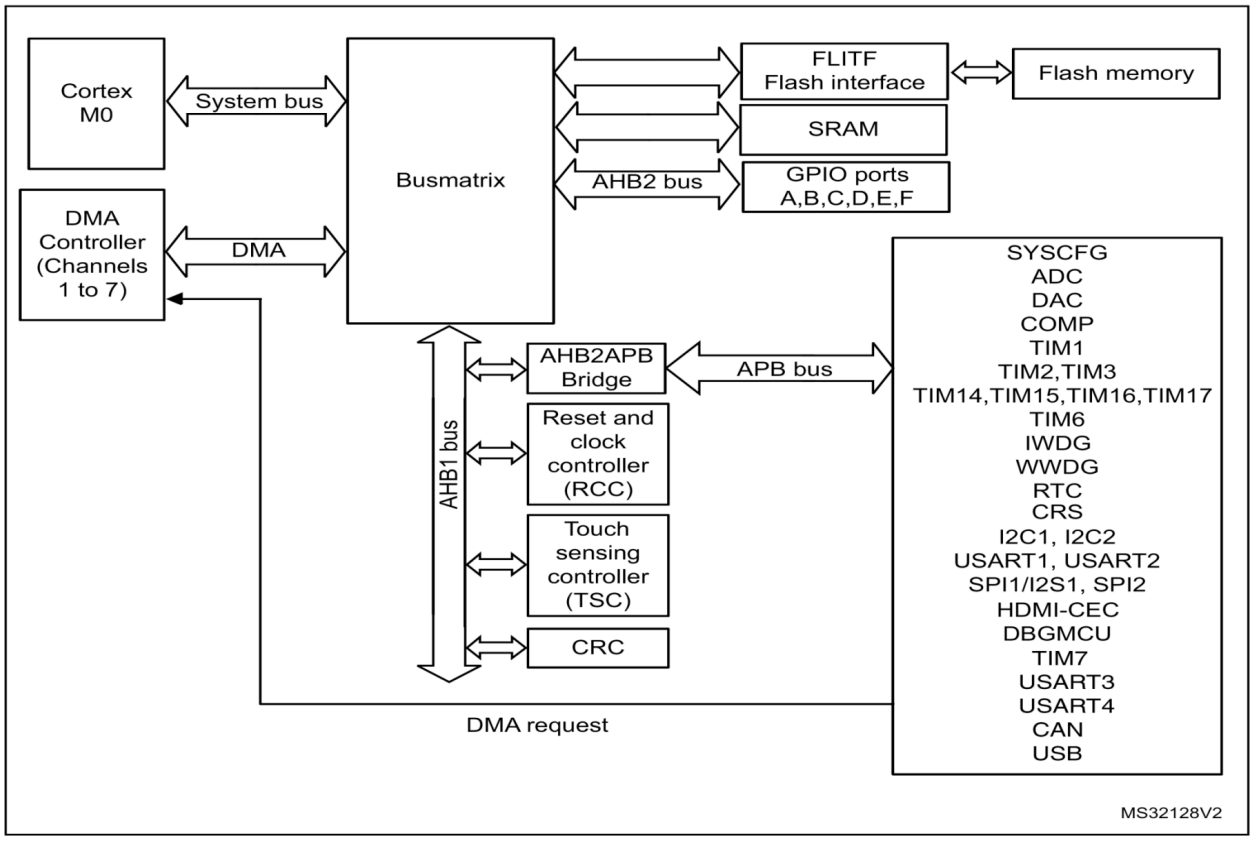

Figure 13: Block diagram describing the STM32F0xx board [2]

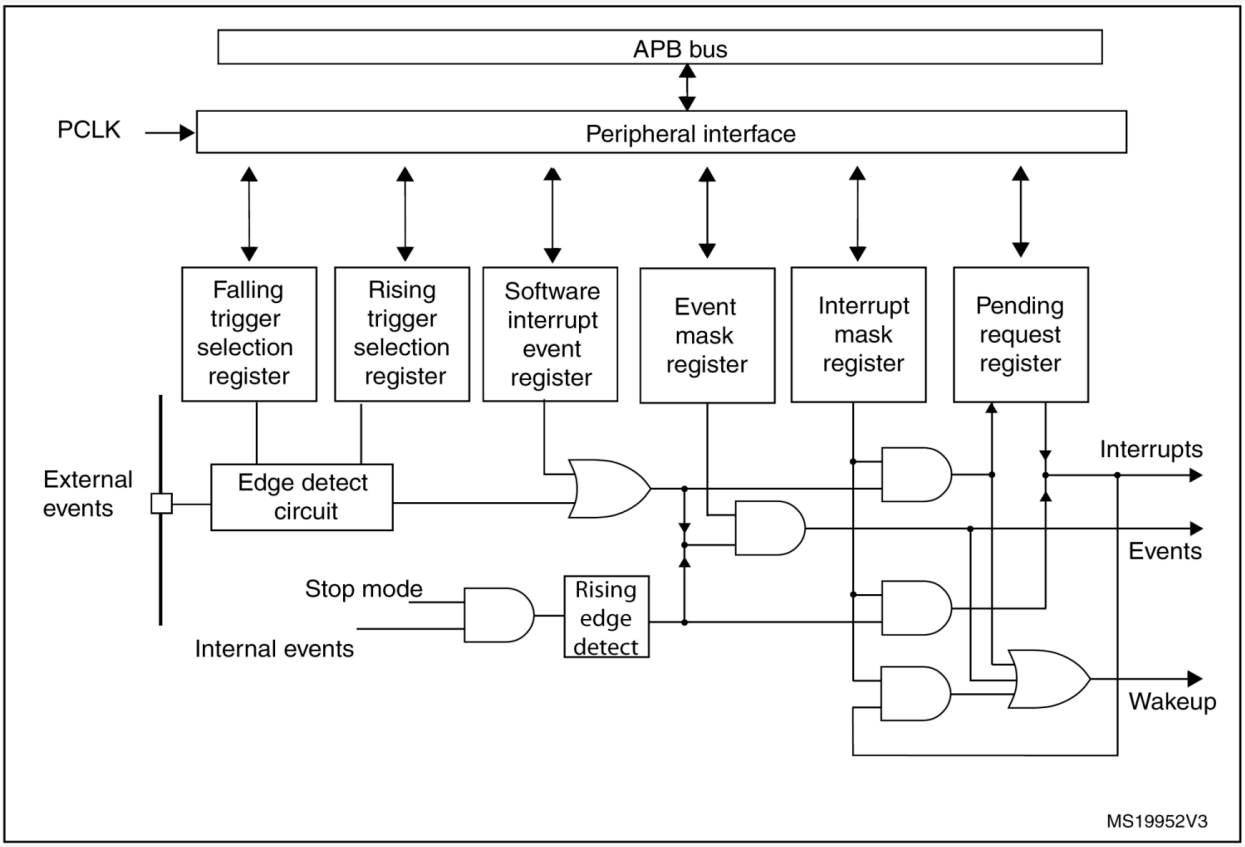

Figure 14: Block diagram describing the EXTI [2]

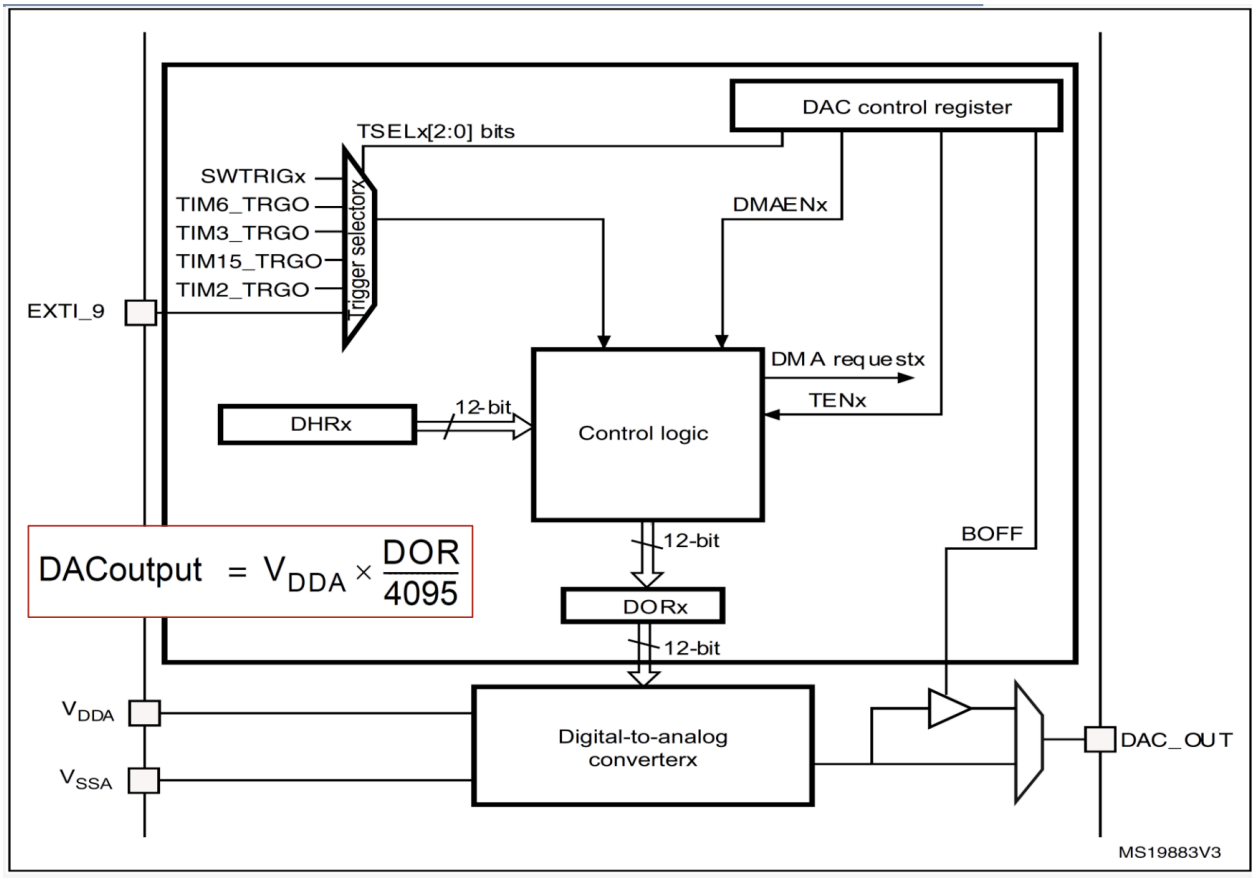

Figure 15: Block diagram describing the DAC [3]

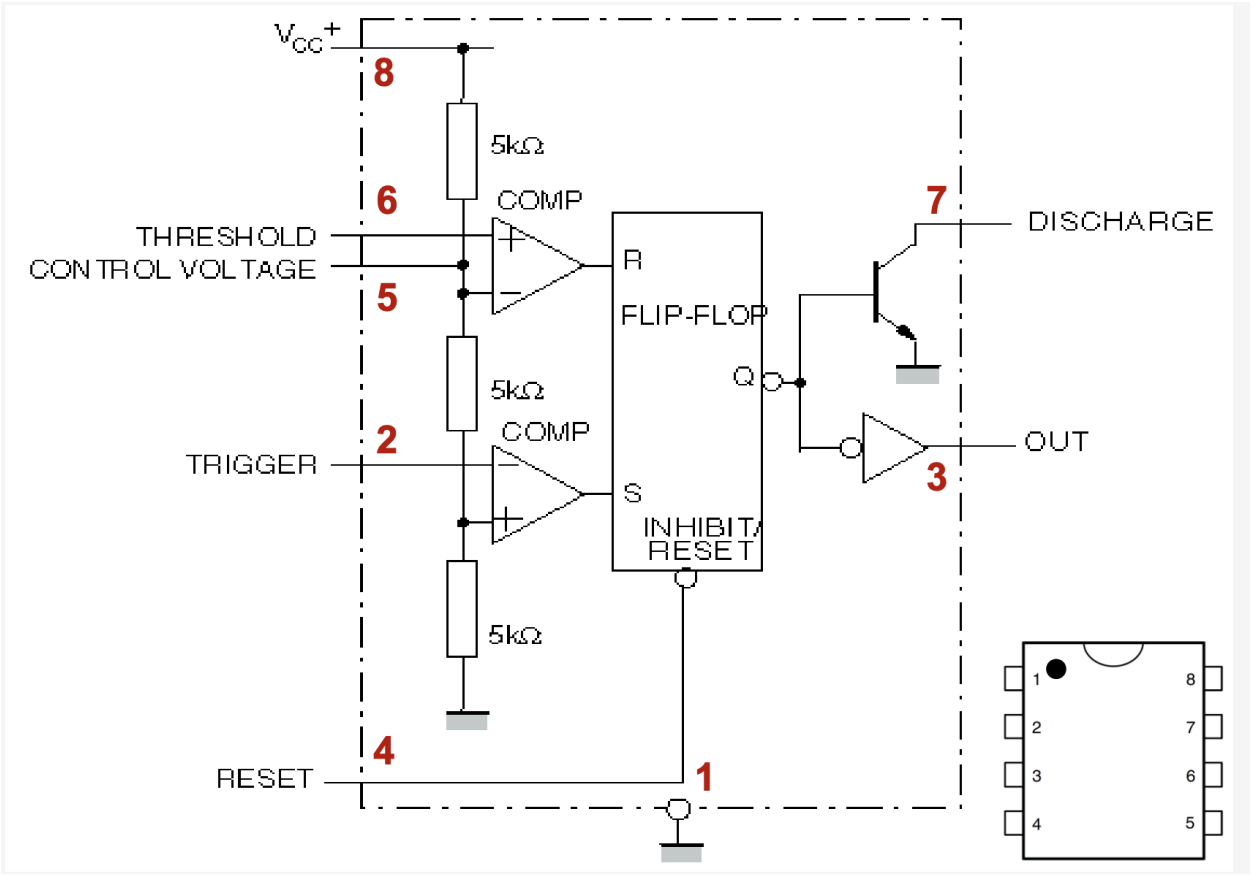

Figure 16: Block diagram describing the 555 timer (NE555) [3]

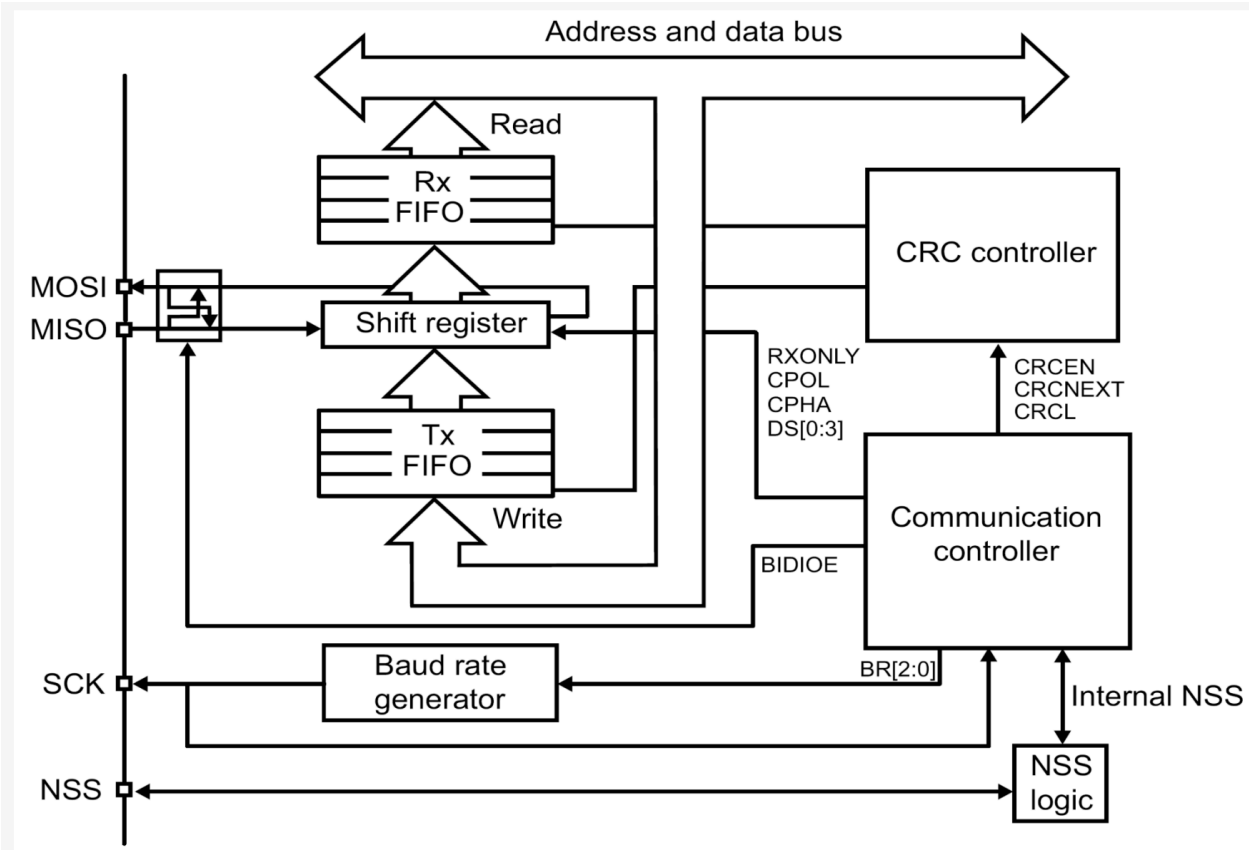

Figure 17: Block diagram describing the SPI1 [3]

2.2.2 Other Diagrams

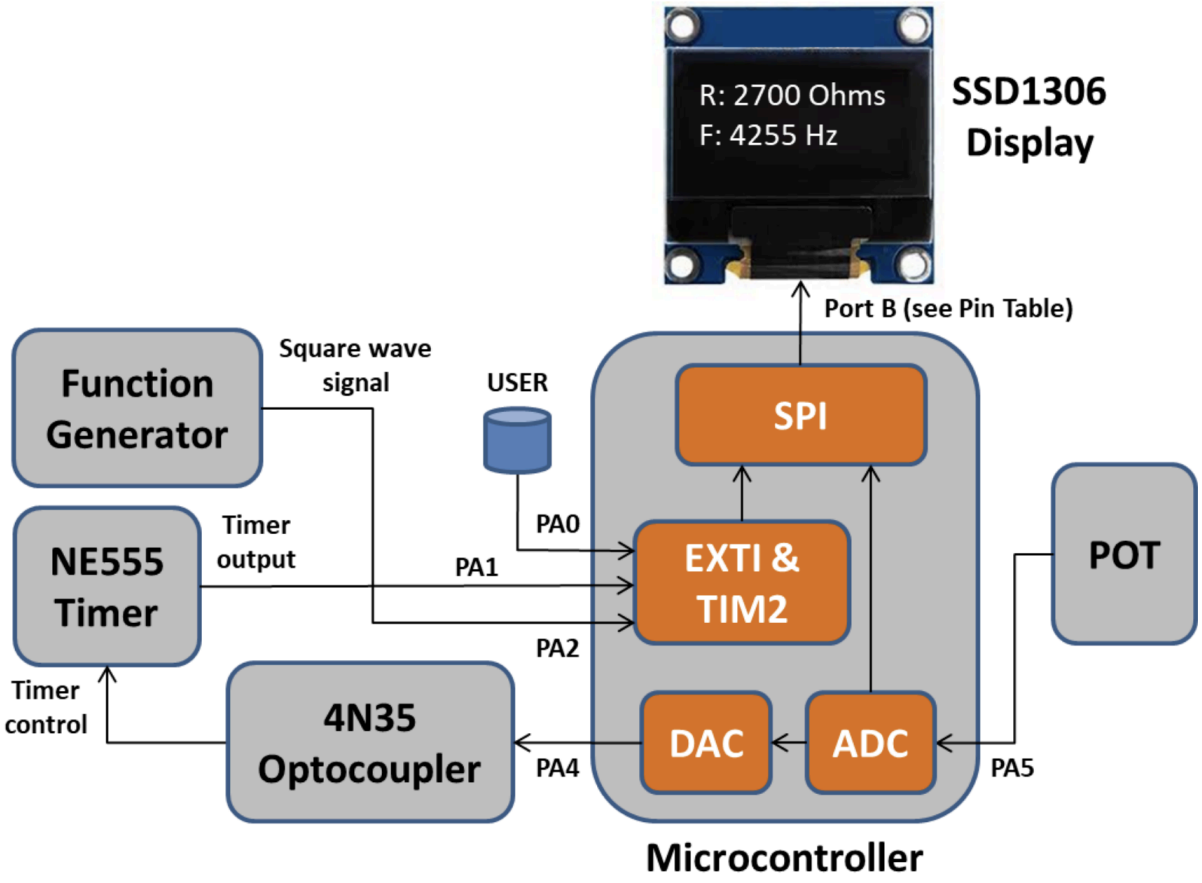

Figure 18: Project diagram [3]

2.3 Lab Work Partitioning

To complete this laboratory report, the majority of the design steps, including the ADC, DAC, USER button, and function generator, were completed together. Closer to the due date, we decided to split up the remaining tasks to complete the project. Anthony completed the code for the 555 timer, while Ashley worked on the code for the LED display. After completing these last two design steps, we tested and troubleshot the code together to ensure all components functioned successfully as a whole.

3.0 Testing/Results

Testing was conducted after implementing the code for each component (design step). Two methods of testing were utilized throughout this project: print statements and breakpoints.

The majority of the testing for this project consisted of using print statements to check if the correct values were being output, if certain parts of the functions were being reached, and to ensure functions were being executed properly. However, excessive use of print statements broke the code. There were a large number of print statements used. After deleting most of the print statements, the code began functioning properly.

After identifying that the print statements were breaking the code, breakpoints were used for debugging. The breakpoints verified that values were correct and functions were running properly. This approach was more efficient and reliable.

3.1 ADC Testing

The ADC was the first component tested in this project. To ensure the ADC was outputting the correct value, a print statement was utilized in the main function to print the integer value. To check the ADC component, the ADC value was verified to output a range between 0 and 4095.

```
506     while (1)
507     {
508         ADC1->CR |= ADC_CR_ADSTART; //Sets bit 2, starts ADC conversion process
509         while(!(ADC1-> ISR & ADC_ISR_EOS_Msk)); // Wait for conversion to finish
510         unsigned int val = ADC1->DR; //Val is between 0 and 4095
511         //trace_printf("ADC Value: %d\n", val);
512         refresh_OLED();
513         DAC->DHR12R1 = val;
514         //trace_printf("DAC Value: %d\n", converted_val);
515     }
516 }
```

Figure 19: Print statement for testing the ADC shown on line 511

3.2 User Button Testing

Testing the functionality of the user button was split into two parts which were 1) detecting if the button was being clicked at all, and then 2) Getting alternating outcomes with each click.
To accomplish the former, we utilized the trace_printf function inside the EXTI0_1_IRQHandler when the PR0 pending flag was set. Next, for the second task of testing we wanted to test if we could alternate the action that happens each click. We also utilized the trace_printf function for this but instead located it inside an if else statement that had a global variable as the condition as seen in **Figure 20**.

```
if(EXTI->PR & EXTI_PR_PR0)
{
    if(globalSignal == 0)
    {
        globalSignal = 1;
        EXTI->IMR &= ~EXTI_IMR_MR1; //Disable interrupt for EXTI1
        EXTI->IMR |= EXTI_IMR_MR2;  //Enable interrupt for EXTI2
        trace_printf("Function generator enabled\n");
    }else if(globalSignal == 1)
    {
        globalSignal = 0;
        EXTI->IMR &= ~EXTI_IMR_MR2; //Disable interrupt for EXTI2
        EXTI->IMR |= EXTI_IMR_MR1;  //Enable interrupt for EXTI1
        trace_printf("555 timer enabled\n");
    }
    EXTI->PR |= EXTI_PR_PR0; //Clear EXTI0 Pending flag
}
```

Figure 20: User Button Testing Method

3.3 Function Generator Testing

Testing the functionality of the function generator was done in Lab #2 and first involved us determining if an interrupt occurred at every rising edge. This was done by using trace_printf inside the EXTI2_3_IRQHandler to give visual feedback that the interrupt was in fact being executed. After we were able to confirm that the interrupts were being executed at rising edges we were able to test the functionality of the TIM2's implementation of measuring the frequency rising edges which was easy to do with a trace_printf statement which printed the frequency because the expected frequency also showed up on the function generator.

3.4 DAC and 555 Timer Testing

Testing the functionality of the 555 timer included the use of a separate external device that was wired to the board called a Digital Storage Oscilloscope which measured the frequency and trace_printf statements. The trace_printf statement here was placed into the EXTI_0_1_IRQHandler when the PR1 flag was set and on the second rising edge occurred. We then were able to look at the Digital Storage Oscilloscope and the value printed to the console to ensure they matched. An accurate reading here indicated to us that the DAC had to be configured correctly as the 555 timer was dependent on the DAC being correct as seen in **Figure 18**.

3.5 LED Display Testing

The LED display was the last component tested in this project. Before implementing and testing the LED display, it was ensured that all other components were functioning properly. This made debugging the LED display easier, as any issues occurring with the display were occurring within the oled_config(), oled_Write_Cmd(), oled_Write_Data(), oled_Write(), and refresh_OLED() functions. Also, this ensured that the values displayed on the LED display were correct.

The testing and debugging of the LED display began once it had powered on. Initially, the screen displayed completely white, with Line 1 “R: [resistance value]” showing for less than a second. After using breakpoints, it was determined that there were issues within oled_Write_Cmd(), oled_Write_Data(), and oled_Write() functions. Line 1 was only showing because the code to display Line 2 had not been written yet. Once these issues were fixed, the LED display functioned successfully.

4.0 Discussion

In this project, we wired and programmed a STMF32F0 board. The objective was to utilize the board's features, such as timers and interrupts, to measure the frequency and resistance of both a frequency generator and the frequency of a 555 timer with the resistance of a potentiometer. With both of these measurements our goal was to display the frequency and resistance on a LED board of one device at a time, being able to switch the input device by pressing the USER button located on the board itself. Careful considerations of interrupt priorities and flow logic were required to ensure precise measurements of a signal's frequency and resistance value.

4.1 Function Generator Discussion

The section of our code that analyzed an input signal from a function generator did give us accurate measurements that were only off by a few Hz to what was actually being outputted. Since it was such a small error it is suspected that it may have come from slight delays within the interrupt being called and TIM2 being stopped that records the measurements as any error here would directly relate to an error in the frequency. Overall though, this error was very minor and we were satisfied with the accuracy of the measurements.

In the *Part 2: Signal Frequency Measurement* lab, the minimum frequency the function generator could measure was 0.25 Hz. Thus, the minimum period was equal to 90 s.

4.1.1 Function Generator Limitations

This section of the project did have some limitations however, for example the measurement between rising edges would be limited by the TIM2 CNT register for example, if the time between rising edges exceeds 32 bits it would cause an overflow and an interrupt to occur. Because our software did not have a handle case for this, it would output the incorrect frequency. This occurred when the frequency was smaller than 0.25 Hz. A patch for this may have been implemented by having a variable that stores how many times an overflow occurs by utilizing the interrupt that would have occurred with an overflow event. This value would then be added to the frequency calculation inside the EXTI_2_3IRQHandler. A similar edge case could occur with a period too small, this would be caused if the period was smaller than the clock speed of

TIM2. A patch for this would be difficult as changing the clock speed for TIM2 would result in needing to change the code for frequency calculations and it would only cause the error to be shifted to smaller periods rather than removing it altogether.

4.2 555 Timer Discussion

Next, when we calculated the values corresponding to the 555 timer we did achieve the values we were expecting - that is, when the potentiometer was turned all the way to the right we got a value of 5000 and when it was turned all the way to the left we got a value of 0. When testing this we did not get any noticeable error with our values. The interrupt that dealt with dealing with the rising edges from the 555 timer shared a interrupt priority of 0 with the USER button. This ensured that there were no delays from other interrupts occurring (possibly from timers). Careful setup of the ADC would also contribute to the success of this section such as ensuring that the ADC had 12 bits of resolution which allowed much more accurate measurements of the incoming voltage readings from the potentiometer.

4.3 USER Button Discussion

The result of our USER button was satisfactory as it would swap between which input device we would receive interrupts from. However, some fine tuning would benefit it.

4.3.1 USER Button Challenges

While the button functioned as expected, it was very touchy and would sometimes double click unless the user pressed it quickly. This is likely because a bouncing effect is causing multiple clicks to be detected by the code despite it seeming to the user it was only pressed once. A proposed solution that is expected would have fixed this is to introduce a short delay after the initial click that masks (disables) interrupts from the EXTI0 line for a few milliseconds before unmasking and re-enabling the use of the user button.

4.4 LED Display Discussion

The LED Display was successfully implemented, efficiently updating the screen each time the frequency and resistance values were refreshed. There was minimal, if any, delay. The two lines displayed were very clear and readable.

4.5 Project Wiring Challenges

In this laboratory project, the biggest difficulty faced was correctly wiring the STM32F0 Discovery Board. As software engineering students with little to no prior experience with hardware wiring or reading specification diagrams, this was a large challenge. This made the beginning stages very difficult, as any correct code would fail. Eventually, with the help of TAs and other students, we learned how to read the diagrams and successfully wire our project.

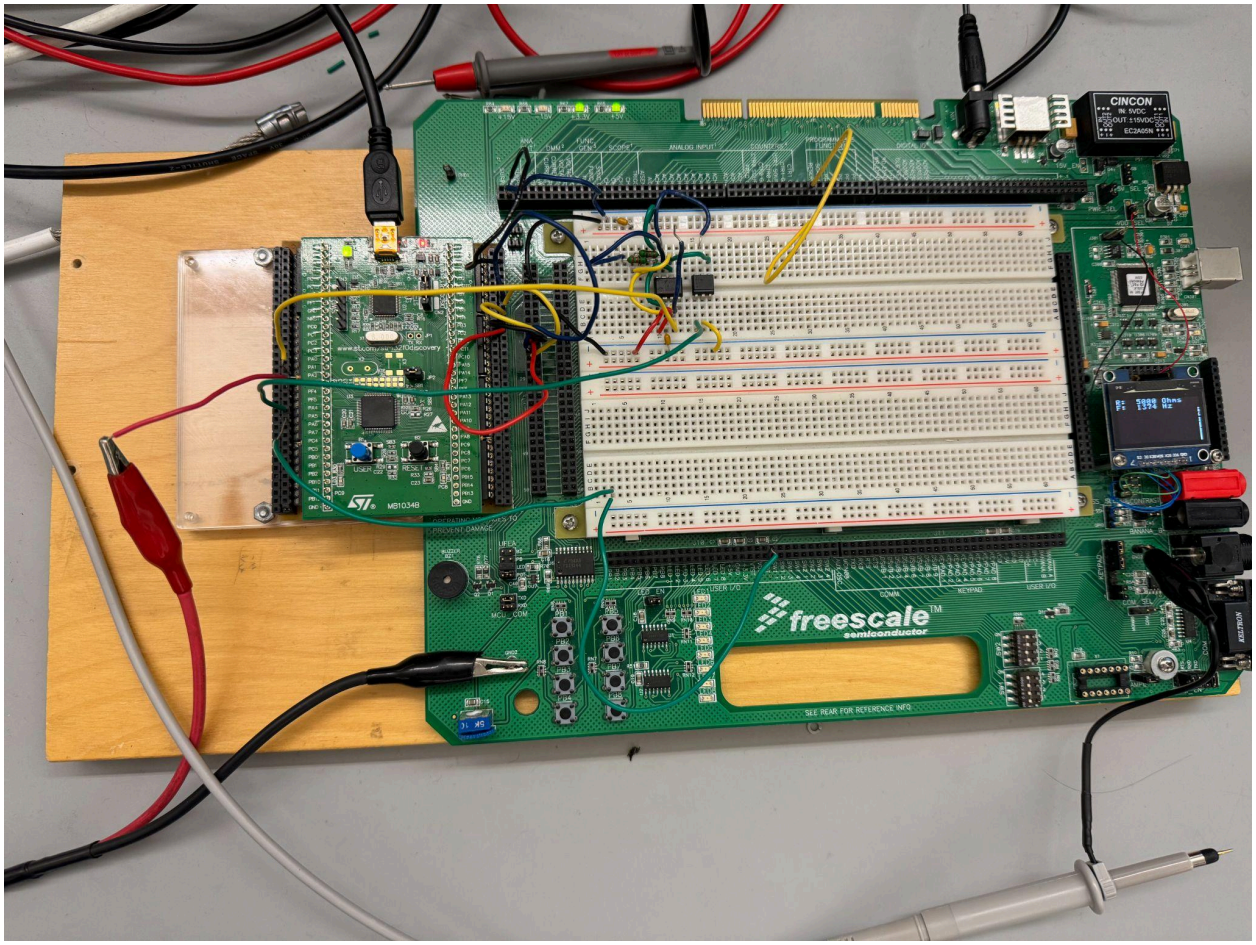

Figure 21: Project wiring

4.6 Advice

In conclusion, all components of the *PWM Signal Generation and Monitoring System* laboratory project were successfully completed. Each component functioned correctly separately and collectively. We would recommend following the six design steps described above if this project were to be repeated because it allowed for focused implementation and testing of one component at a time. This ensured that each design step was successful before moving to the next step. Ultimately, this reduced debugging efforts later.

In addition, getting assistance earlier with the project wiring would have been beneficial, as it would have saved us more time to focus on the code. A good understanding of the wiring diagrams is crucial to successfully complete this laboratory project because it reduces the amount of time spent debugging.

5.0 Appendices

Lab Project (Code) [4]

```
//
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//
// ----------------------------------------------------------------------
#include <stdio.h>
#include "diag/Trace.h"
#include <string.h>
#include "cmsis/cmsis_device.h"
// ----------------------------------------------------------------------
//
// STM32F0 led blink sample (trace via $(trace)).
//
// In debug configurations, demonstrate how to print a greeting message
// on the trace device. In release configurations the message is
// simply discarded.
//
// To demonstrate POSIX retargeting, reroute the STDOUT and STDERR to the
// trace device and display messages on both of them.
//
// Then demonstrates how to blink a led with 1Hz, using a
// continuous loop and SysTick delays.
//
// On DEBUG, the uptime in seconds is also displayed on the trace device.
//
// Trace support is enabled by adding the TRACE macro definition.
// By default the trace messages are forwarded to the $(trace) output,
// but can be rerouted to any device or completely suppressed, by
// changing the definitions required in system/src/diag/trace_impl.c
// (currently OS_USE_TRACE_ITM, OS_USE_TRACE_SEMIHOSTING_DEBUG/_STDOUT).
//
// The external clock frequency is specified as a preprocessor definition
// passed to the compiler via a command line option (see the 'C/C++ General' ->
// 'Paths and Symbols' -> the 'Symbols' tab, if you want to change it).
// The value selected during project creation was HSE_VALUE=48000000.
//
/// Note: The default clock settings take the user defined HSE_VALUE and try
// to reach the maximum possible system clock. For the default 8MHz input
// the result is guaranteed, but for other values it might not be possible,
// so please adjust the PLL settings in system/src/cmsis/system_stm32f0xx.c
//
// ----- main() ---------------------------------------------------------
// Sample pragmas to cope with warnings. Please note the related line at
// the end of this function, used to pop the compiler diagnostics status.
```

```
#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)
/*** This is partial code for accessing LED Display via SPI interface. ***/
//...
volatile unsigned int Freq = 0;  // Example: measured frequency value (global variable)
volatile unsigned int Res = 0;   // Example: measured resistance value (global variable)
volatile int globalSignal = 1; // Toggles the states
volatile uint8_t timerTriggered = 0;
void oled_Write(unsigned char);
void oled_Write_Cmd(unsigned char);
void oled_Write_Data(unsigned char);
void oled_config(void);
void refresh_OLED(void);
SPI_HandleTypeDef SPI_Handle;
//
// LED Display initialization commands
//
unsigned char oled_init_cmds[] =
{
0xAE,
0x20, 0x00,
0x40,
0xA0 | 0x01,
0xA8, 0x40 - 1,
0xC0 | 0x08,
0xD3, 0x00,
0xDA, 0x32,
0xD5, 0x80,
0xD9, 0x22,
0xDB, 0x30,
0x81, 0xFF,
0xA4,
0xA6,
0xAD, 0x30,
0x8D, 0x10,
0xAE | 0x01,
0xC0,
0xA0
};
//
// Character specifications for LED Display (1 row = 8 bytes = 1 ASCII character)
// Example: to display '4', retrieve 8 data bytes stored in Characters[52][X] row
//          (where X = 0, 1, ..., 7) and send them one by one to LED Display.
```

```
// Row number = character ASCII code (e.g., ASCII code of '4' is 0x34 = 52)
//
unsigned char Characters[][8] = {
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  //
SPACE
```

```
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // SPACE
{0b00000000, 0b00000000, 0b01011111, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // !
{0b00000000, 0b00000111, 0b00000000, 0b00000111, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // "
{0b00010100, 0b01111111, 0b00010100, 0b01111111, 0b00010100,0b00000000, 0b00000000, 0b00000000},  // #
{0b00100100, 0b00101010, 0b01111111, 0b00101010, 0b00010010,0b00000000, 0b00000000, 0b00000000},  // $
{0b00100011, 0b00010011, 0b00001000, 0b01100100, 0b01100010,0b00000000, 0b00000000, 0b00000000},  // %
{0b00110110, 0b01001001, 0b01010101, 0b00100010, 0b01010000,0b00000000, 0b00000000, 0b00000000},  // &
{0b00000000, 0b00000101, 0b00000011, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // '
{0b00000000, 0b00011100, 0b00100010, 0b01000001, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // (
{0b00000000, 0b01000001, 0b00100010, 0b00011100, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // )
{0b00010100, 0b00001000, 0b00111110, 0b00001000, 0b00010100,0b00000000, 0b00000000, 0b00000000},  // *
{0b00001000, 0b00001000, 0b00111110, 0b00001000, 0b00001000,0b00000000, 0b00000000, 0b00000000},  // +
{0b00000000, 0b01010000, 0b00110000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // ,
{0b00001000, 0b00001000, 0b00001000, 0b00001000, 0b00001000,0b00000000, 0b00000000, 0b00000000},  // -
{0b00000000, 0b01100000, 0b01100000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // .
{0b00100000, 0b00010000, 0b00001000, 0b00000100, 0b00000010,0b00000000, 0b00000000, 0b00000000},  // /
{0b00111110, 0b01010001, 0b01001001, 0b01000101, 0b00111110,0b00000000, 0b00000000, 0b00000000},  // 0
{0b00000000, 0b01000010, 0b01111111, 0b01000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // 1
{0b01000010, 0b01100001, 0b01010001, 0b01001001, 0b01000110,0b00000000, 0b00000000, 0b00000000},  // 2
{0b00100001, 0b01000001, 0b01000101, 0b01001011, 0b00110001,0b00000000, 0b00000000, 0b00000000},  // 3
{0b00011000, 0b00010100, 0b00010010, 0b01111111, 0b00010000,0b00000000, 0b00000000, 0b00000000},  // 4
{0b00100111, 0b01000101, 0b01000101, 0b01000101, 0b00111001,0b00000000, 0b00000000, 0b00000000},  // 5
{0b00111100, 0b01001010, 0b01001001, 0b01001001, 0b00110000,0b00000000, 0b00000000, 0b00000000},  // 6
{0b00000011, 0b00000001, 0b01110001, 0b00001001, 0b00000111,0b00000000, 0b00000000, 0b00000000},  // 7
{0b00110110, 0b01001001, 0b01001001, 0b01001001, 0b00110110,0b00000000, 0b00000000, 0b00000000},  // 8
{0b00000110, 0b01001001, 0b01001001, 0b00101001, 0b00011110,0b00000000, 0b00000000, 0b00000000},  // 9
{0b00000000, 0b00110110, 0b00110110, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // :
{0b00000000, 0b01010110, 0b00110110, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // ;
{0b00001000, 0b00010100, 0b00100010, 0b01000001, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // <
{0b00010100, 0b00010100, 0b00010100, 0b00010100, 0b00010100,0b00000000, 0b00000000, 0b00000000},  // =
```

```
{0b00000000, 0b01000001, 0b00100010, 0b00010100, 0b00001000,0b00000000, 0b00000000, 0b00000000},  // >
{0b00000010, 0b00000001, 0b01010001, 0b00001001, 0b00000110,0b00000000, 0b00000000, 0b00000000},  // ?
{0b00110010, 0b01001001, 0b01111001, 0b01000001, 0b00111110,0b00000000, 0b00000000, 0b00000000},  // @
{0b01111110, 0b00010001, 0b00010001, 0b00010001, 0b01111110,0b00000000, 0b00000000, 0b00000000},  // A
{0b01111111, 0b01001001, 0b01001001, 0b01001001, 0b00110110,0b00000000, 0b00000000, 0b00000000},  // B
{0b00111110, 0b01000001, 0b01000001, 0b01000001, 0b00100010,0b00000000, 0b00000000, 0b00000000},  // C
{0b01111111, 0b01000001, 0b01000001, 0b00100010, 0b00011100,0b00000000, 0b00000000, 0b00000000},  // D
{0b01111111, 0b01001001, 0b01001001, 0b01001001, 0b01000001,0b00000000, 0b00000000, 0b00000000},  // E
{0b01111111, 0b00001001, 0b00001001, 0b00001001, 0b00000001,0b00000000, 0b00000000, 0b00000000},  // F
{0b00111110, 0b01000001, 0b01001001, 0b01001001, 0b01111010,0b00000000, 0b00000000, 0b00000000},  // G
{0b01111111, 0b00001000, 0b00001000, 0b00001000, 0b01111111,0b00000000, 0b00000000, 0b00000000},  // H
{0b01000000, 0b01000001, 0b01111111, 0b01000001, 0b01000000,0b00000000, 0b00000000, 0b00000000},  // I
{0b00100000, 0b01000000, 0b01000001, 0b00111111, 0b00000001,0b00000000, 0b00000000, 0b00000000},  // J
{0b01111111, 0b00001000, 0b00010100, 0b00100010, 0b01000001,0b00000000, 0b00000000, 0b00000000},  // K
{0b01111111, 0b01000000, 0b01000000, 0b01000000, 0b01000000,0b00000000, 0b00000000, 0b00000000},  // L
{0b01111111, 0b00000010, 0b00001100, 0b00000010, 0b01111111,0b00000000, 0b00000000, 0b00000000},  // M
{0b01111111, 0b00000100, 0b00001000, 0b00010000, 0b01111111,0b00000000, 0b00000000, 0b00000000},  // N
{0b00111110, 0b01000001, 0b01000001, 0b01000001, 0b00111110,0b00000000, 0b00000000, 0b00000000},  // O
{0b01111111, 0b00001001, 0b00001001, 0b00001001, 0b00000110,0b00000000, 0b00000000, 0b00000000},  // P
{0b00111110, 0b01000001, 0b01010001, 0b00100001, 0b01011110,0b00000000, 0b00000000, 0b00000000},  // Q
{0b01111111, 0b00001001, 0b00011001, 0b00101001, 0b01000110,0b00000000, 0b00000000, 0b00000000},  // R
{0b01000110, 0b01001001, 0b01001001, 0b01001001, 0b00110001,0b00000000, 0b00000000, 0b00000000},  // S
{0b00000001, 0b00000001, 0b01111111, 0b00000001, 0b00000001,0b00000000, 0b00000000, 0b00000000},  // T
{0b00111111, 0b01000000, 0b01000000, 0b01000000, 0b00111111,0b00000000, 0b00000000, 0b00000000},  // U
{0b00011111, 0b00100000, 0b01000000, 0b00100000, 0b00011111,0b00000000, 0b00000000, 0b00000000},  // V
{0b00111111, 0b01000000, 0b00111000, 0b01000000, 0b00111111,0b00000000, 0b00000000, 0b00000000},  // W
{0b01100011, 0b00010100, 0b00001000, 0b00010100, 0b01100011,0b00000000, 0b00000000, 0b00000000},  // X
{0b00000111, 0b00001000, 0b01110000, 0b00001000, 0b00000111,0b00000000, 0b00000000, 0b00000000},  // Y
{0b01100001, 0b01010001, 0b01001001, 0b01000101, 0b01000011,0b00000000, 0b00000000, 0b00000000},  // Z
{0b01111111, 0b01000001, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // [
{0b00010101, 0b00010110, 0b01111100, 0b00010110, 0b00010101,0b00000000, 0b00000000, 0b00000000},  //
back slash
{0b00000000, 0b00000000, 0b00000000, 0b01000001, 0b01111111,0b00000000, 0b00000000, 0b00000000},  // ]
{0b00000100, 0b00000010, 0b00000001, 0b00000010, 0b00000100,0b00000000, 0b00000000, 0b00000000},  // ^
{0b01000000, 0b01000000, 0b01000000, 0b01000000, 0b01000000,0b00000000, 0b00000000, 0b00000000},  // _
{0b00000000, 0b00000001, 0b00000010, 0b00000100, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // `
{0b00100000, 0b01010100, 0b01010100, 0b01010100, 0b01111000,0b00000000, 0b00000000, 0b00000000},  // a
{0b01111111, 0b01001000, 0b01000100, 0b01000100, 0b00111000,0b00000000, 0b00000000, 0b00000000},  // b
{0b00111000, 0b01000100, 0b01000100, 0b01000100, 0b00100000,0b00000000, 0b00000000, 0b00000000},  // c
{0b00111000, 0b01000100, 0b01000100, 0b01001000, 0b01111111,0b00000000, 0b00000000, 0b00000000},  // d
{0b00111000, 0b01010100, 0b01010100, 0b01010100, 0b00011000,0b00000000, 0b00000000, 0b00000000},  // e
{0b00001000, 0b01111110, 0b00001001, 0b00000001, 0b00000010,0b00000000, 0b00000000, 0b00000000},  // f
{0b00001100, 0b01010010, 0b01010010, 0b01010010, 0b00111110,0b00000000, 0b00000000, 0b00000000},  // g
{0b01111111, 0b00001000, 0b00000100, 0b00000100, 0b01111000,0b00000000, 0b00000000, 0b00000000},  // h
{0b00000000, 0b01000100, 0b01111101, 0b01000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // i
{0b00100000, 0b01000000, 0b01000100, 0b00111101, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // j
{0b01111111, 0b00010000, 0b00101000, 0b01000100, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // k
{0b00000000, 0b01000001, 0b01111111, 0b01000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // l
{0b01111100, 0b00000100, 0b00011000, 0b00000100, 0b01111000,0b00000000, 0b00000000, 0b00000000},  // m
```

```
{0b01111100, 0b00001000, 0b00000100, 0b00000100, 0b01111000,0b00000000, 0b00000000, 0b00000000},  // n
{0b00111000, 0b01000100, 0b01000100, 0b01000100, 0b00111000,0b00000000, 0b00000000, 0b00000000},  // o
{0b01111100, 0b00010100, 0b00010100, 0b00010100, 0b00001000,0b00000000, 0b00000000, 0b00000000},  // p
{0b00001000, 0b00010100, 0b00010100, 0b00011000, 0b01111100,0b00000000, 0b00000000, 0b00000000},  // q
{0b01111100, 0b00001000, 0b00000100, 0b00000100, 0b00001000,0b00000000, 0b00000000, 0b00000000},  // r
{0b01001000, 0b01010100, 0b01010100, 0b01010100, 0b00100000,0b00000000, 0b00000000, 0b00000000},  // s
{0b00000100, 0b00111111, 0b01000100, 0b01000000, 0b00100000,0b00000000, 0b00000000, 0b00000000},  // t
{0b00111100, 0b01000000, 0b01000000, 0b00100000, 0b01111100,0b00000000, 0b00000000, 0b00000000},  // u
{0b00011100, 0b00100000, 0b01000000, 0b00100000, 0b00011100,0b00000000, 0b00000000, 0b00000000},  // v
{0b00111100, 0b01000000, 0b00111000, 0b01000000, 0b00111100,0b00000000, 0b00000000, 0b00000000},  // w
{0b01000100, 0b00101000, 0b00010000, 0b00101000, 0b01000100,0b00000000, 0b00000000, 0b00000000},  // x
{0b00001100, 0b01010000, 0b01010000, 0b01010000, 0b00111100,0b00000000, 0b00000000, 0b00000000},  // y
{0b01000100, 0b01100100, 0b01010100, 0b01001100, 0b01000100,0b00000000, 0b00000000, 0b00000000},  // z
{0b00000000, 0b00001000, 0b00110110, 0b01000001, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // {
{0b00000000, 0b00000000, 0b01111111, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // |
{0b00000000, 0b01000001, 0b00110110, 0b00001000, 0b00000000,0b00000000, 0b00000000, 0b00000000},  // }
{0b00001000, 0b00001000, 0b00101010, 0b00011100, 0b00001000,0b00000000, 0b00000000, 0b00000000},  // ~
{0b00001000, 0b00011100, 0b00101010, 0b00001000, 0b00001000,0b00000000, 0b00000000, 0b00000000}   // <-
};
void SystemClock48MHz( void )
{
        //
        // Disable the PLL
        //
        RCC->CR &= ~(RCC_CR_PLLON);
        //
        // Wait for the PLL to unlock
        //
        while (( RCC->CR & RCC_CR_PLLRDY ) != 0 );
        //
        // Configure the PLL for a 48MHz system clock
        //
        RCC->CFGR = 0x00280000;
        //
        // Enable the PLL
        //
        RCC->CR |= RCC_CR_PLLON;
        //
        // Wait for the PLL to lock
        //
        while (( RCC->CR & RCC_CR_PLLRDY ) != RCC_CR_PLLRDY );
        //
        // Switch the processor to the PLL clock source
        //
        RCC->CFGR = ( RCC->CFGR & (~RCC_CFGR_SW_Msk)) | RCC_CFGR_SW_PLL;
        //
        // Update the system with the new clock frequency
        //
        SystemCoreClockUpdate();
```

```
}
void ADC_initialize()
{
	//Enable the ADC clock
	RCC->APB2ENR |= RCC_APB2ENR_ADCEN; //Enable clock for ADC with macro
	RCC->APB2ENR |= (1 << 9); //Sets bit 9, same effect as above but without a macro
	//Set Port A pin 5 to analog mode
	GPIOA->MODER |= (3 << (2 * 5)); //SET PA5(ADC_IN5) as an analog mode pin
	//Step 1, ADC->CFGR1 (page 10 of example PDF)
	ADC1->CFGR1 &= ~(0b11000); //Clears Bits[4:3] which creates 12 bit resolution
	ADC1->CFGR1 &= ~(0b100000); //Clears Bit[5] which creates right alignment data
	ADC1->CFGR1 |= 0b1000000000000; //Sets bit 12 which means contents are overwritten when overrun is
detected
	ADC1->CFGR1 |= 0b10000000000000; //Sets bit 13 which sets continuous conversion mode
	//Step 2, Channel select register
	ADC1->CHSELR |= (1 << 5); //Set bit 5 to 1 to indicate ADC is channel 5
	//Step 3, Set the sampling time register
	ADC1->SMPR |= 0b111; //This allows as many clock cycles as needed
	//Step 4, Set control register
	ADC1->CR |= 0b1; //Enable the ADC process
	//THE BASIC INITIALIZATION OF ADC IS COMPLETE
	//Make it wait for the ADC1 -> ISR[0] to become 1
	while(!(ADC1->ISR & 0b1)); //Stuck on this line until ADC1->ISR[0] becomes 1 - This is the ADC ready
flag
}
void DAC_initialize()
{
	//Enable the clock for DAC
	RCC->APB1ENR |= (1 << 29); //Sets bit 29
	//Set Port A pin 4 to analog mode
	GPIOA->MODER |= (3 << (2 * 4)); //SET PA4 to analog mode for the DAC
	DAC -> CR |= 0b1; // Enable DAC channel 1
	DAC -> CR &= ~(0b10); //Enable DAC channel tri-state buffer
	DAC -> CR &= ~(0b100); //Disable channel 1 trigger enable
}
void myTIM2_Init()
{
	/* Enable clock for TIM2 peripheral */
	// Relevant register: RCC->APB1ENR
	RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
	/* Configure TIM2: buffer auto-reload, count up, stop on overflow,
	 * enable update events, interrupt on overflow only */
	// Relevant register: TIM2->CR1
	TIM2->CR1 = ((uint16_t)0x008C);
	/* Set clock prescaler value */
	//TIM2->PSC = myTIM2_PRESCALER;
	TIM2->PSC = ((uint16_t)0x0000);
	/* Set auto-reloaded delay */
	TIM2->ARR = myTIM2_PERIOD;
```

```
	/* Update timer registers */
	// Relevant register: TIM2->EGR
	TIM2->EGR = ((uint16_t)0x0001);
	/* Assign TIM2 interrupt priority = 0 in NVIC */
	// Relevant register: NVIC->IP[3], or use NVIC_SetPriority
	NVIC_SetPriority(TIM2_IRQn, 2);
	/* Enable TIM2 interrupts in NVIC */
	// Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
	//NVIC_EnableIRQ(TIM2_IRQn);
	/* Enable update interrupt generation */
	// Relevant register: TIM2->DIER
	TIM2->DIER |= TIM_DIER_UIE;
	/* Start Counting Timer Pulses*/
	TIM2-> CR1 |= TIM_CR1_CEN;
}
void myTIM3_Init()
{
	/* Enable clock for TIM2 peripheral */
	// Relevant register: RCC->APB1ENR
	RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
	/* Configure TIM2: buffer auto-reload, count up, stop on overflow,
	 * enable update events, interrupt on overflow only */
	// Relevant register: TIM2->CR1
	TIM3->CR1 = ((uint16_t)0x008C);
	/* Set clock prescaler value */
	//TIM2->PSC = myTIM2_PRESCALER;
	TIM3->PSC = ((uint16_t)0x0000);
	/* Set auto-reloaded delay */
	TIM3->ARR = myTIM2_PERIOD;
	//TIM2->ARR = ((uint32_t)12000000);
	/* Update timer registers */
	// Relevant register: TIM2->EGR
	//TIM2->EGR = ((uint16_t)0x001);
	TIM3->EGR = ((uint16_t)0x0001);
	/* Assign TIM2 interrupt priority = 0 in NVIC */
	// Relevant register: NVIC->IP[3], or use NVIC_SetPriority
	NVIC_SetPriority(TIM3_IRQn, 4);
	/* Enable TIM2 interrupts in NVIC */
	// Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
	//NVIC_EnableIRQ(TIM2_IRQn);
	/* Enable update interrupt generation */
	// Relevant register: TIM2->DIER
	TIM3->DIER |= TIM_DIER_UIE;
	/* Start Counting Timer Pulses*/
	TIM3-> CR1 |= TIM_CR1_CEN;
}
void EXTI0_1_Init()
{
```

```
	GPIOA->MODER &= ~(GPIO_MODER_MODER0); //Sets Port A Pin 0 as input which corresponds to
USER button
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI0_PA; //Map EXTI0 line to PA0
	EXTI->RTSR |= EXTI_RTSR_TR0; //Sensitive to rising edges
	EXTI->IMR |= EXTI_IMR_MR0; // Unmasks interrupts from EXTI0 line
	NVIC_SetPriority(EXTI0_1_IRQn, 0); //Sets priority to 0 in NVIC
	NVIC_EnableIRQ(EXTI0_1_IRQn); //Enables EXTI0 interrupts in NVIC
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA; //Map EXTI1 line to PA1
	EXTI->RTSR |= EXTI_RTSR_TR1; // Sensitive to rising edges on EXTI1
	EXTI->IMR |= EXTI_IMR_MR1;   // Unmask interrupts from EXTI1 line
	EXTI->IMR &= ~EXTI_IMR_MR1; //Disable interrupt for EXTI1
	EXTI->IMR |= EXTI_IMR_MR2;  //Enable interrupt for EXTI2
}
void myEXTI2_3_Init()
{
	/* Map EXTI2 line to PA2 */
	// Relevant register: SYSCFG->EXTICR[0]
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI2_PA;
	/* EXTI2 line interrupts: set rising-edge trigger */
	// Relevant register: EXTI->RTSR
	EXTI->RTSR |= EXTI_RTSR_TR2;
	/* Unmask interrupts from EXTI2 line */
	// Relevant register: EXTI->IMR
	EXTI->IMR |= EXTI_IMR_MR2;
	/* Assign EXTI2 interrupt priority = 0 in NVIC */
	// Relevant register: NVIC->IP[2], or use NVIC_SetPriority
	NVIC_SetPriority(EXTI2_3_IRQn, 1);
	/* Enable EXTI2 interrupts in NVIC */
	// Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
	NVIC_EnableIRQ(EXTI2_3_IRQn);
}
void EXTI2_3_IRQHandler()
{
	// Declare/initialize your local variables here...
	double float_frequency;
	double float_scaledPeriod;
	unsigned int frequency;
	unsigned int scaledPeriod;
	/* Check if EXTI2 interrupt pending flag is indeed set */
	if ((EXTI->PR & EXTI_PR_PR2) != 0)
	{
		//
		// 1. If this is the first edge:
		//		- Clear count register (TIM2->CNT).
		//		- Start timer (TIM2->CR1).
  if(!timerTriggered)
  {
    TIM2->CNT = 0;
    TIM2->CR1 |= TIM_CR1_CEN;
```

```
    timerTriggered = 1;
  }
                    //   Else (this is the second edge):
                    //        - Stop timer (TIM2->CR1).
                    //        - Read out count register (TIM2->CNT).
                    //        - Calculate signal period and frequency.
                    //        - Print calculated values to the console.
                    //          NOTE: Function trace_printf does not work
                    //          with floating-point numbers: you must use
                    //          "unsigned int" type to print your signal
                    //          period and frequency.
  else
  {
    TIM2->CR1 &= ~(TIM_CR1_CEN);
    uint32_t timerValue = TIM2->CNT;
    float_frequency = 48000000 / timerValue;
    float_scaledPeriod = 1/float_frequency;
    frequency = float_frequency;
    Freq = float_frequency;
    Res = 0;
    timerTriggered = 0;
  }
                    //
                    // 2. Clear EXTI2 interrupt pending flag (EXTI->PR).
                    // NOTE: A pending register (PR) bit is cleared
                    // by writing 1 to it.
                    //
  EXTI->PR |= EXTI_PR_PR2;
        }
}
void myGPIOA_Init()
{
        /* Enable clock for GPIOA peripheral */
        // Relevant register: RCC->AHBENR
        RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
        /* Configure PA4 as output */
        // Relevant register: GPIOA->MODER
        GPIOA->MODER |= (0b01 << (2*4));
        /* Ensure no pull-up/pull-down for PA4 */
        // Relevant register: GPIOA->PUPDR
        GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
        //Configure PA1 as input
        GPIOA->MODER &= ~(0b11 << 2 *1);
        //No pull-up, no pull-down
        GPIOA->PUPDR &= ~(0b11 << (2 * 1));
}
int main(int argc, char* argv[])
{
        SystemClock48MHz();
```

```
	RCC->AHBENR |= (1 << 0); // Enable clock for GPIOA
	myGPIOA_Init();
	ADC_initialize();
	DAC_initialize();
	myTIM2_Init();
	myTIM3_Init();
	oled_config();
	EXTI0_1_Init();
	myEXTI2_3_Init();
	while (1)
	{
		ADC1->CR |= ADC_CR_ADSTART; //Sets bit 2, starts ADC conversion process
		while(!(ADC1-> ISR & ADC_ISR_EOS_Msk)); // Wait for conversion to finish
		unsigned int val = ADC1->DR; //Val is between 0 and 4095
		//trace_printf("ADC Value: %d\n", val);
		refresh_OLED();
		DAC->DHR12R1 = val;
		//trace_printf("DAC Value: %d\n", converted_val);
	}
}
//
// LED Display Functions
//
void refresh_OLED( void )
{
	// Buffer size = at most 16 characters per PAGE + terminating '\0'
	unsigned char Buffer[17];
	//Line 1:
	snprintf( Buffer, sizeof( Buffer ), "R: %5u Ohms", Res );
	/* Buffer now contains your character ASCII codes for LED Display
	  - select PAGE (LED Display line) and set starting SEG (column)
	  - for each c = ASCII code = Buffer[0], Buffer[1], ...,
		  send 8 bytes in Characters[c][0-7] to LED Display
	*/
	oled_Write_Cmd(0xB0); // Select PAGE
	oled_Write_Cmd(0x02); // Lower SEG start address
	oled_Write_Cmd(0x10); // Higher SEG start address
	for(unsigned int x = 0; Buffer[x] != '\0'; x++)
	{
		unsigned char c = Buffer[x];
		for(unsigned int y = 0; y < 8; y++)
		{
			oled_Write_Data(Characters[c][y]); //
		}
	}
	//Line 2:
	snprintf( Buffer, sizeof( Buffer ), "F: %5u Hz", Freq );
	/* Buffer now contains your character ASCII codes for LED Display
	  - select PAGE (LED Display line) and set starting SEG (column)
```

```
            - for each c = ASCII code = Buffer[0], Buffer[1], ...,
                    send 8 bytes in Characters[c][0-7] to LED Display
          */
 oled_Write_Cmd(0xB1); // Select PAGE
 oled_Write_Cmd(0x02); // Lower SEG start address
 oled_Write_Cmd(0x10); // Higher SEG start address
 for(unsigned int x = 0; Buffer[x] != '\0'; x++)
 {
            unsigned char c = Buffer[x];
            for(unsigned int y = 0; y < 8; y++)
            {
                    oled_Write_Data(Characters[c][y]);
            }
 }
}
void oled_Write_Cmd( unsigned char cmd )
{
          //trace_printf("Made it to oled_write_cmd\n");
          GPIOB->ODR |= (1 << 6);  // make PB6 = CS# = 1
          GPIOB->ODR &= ~(1 << 7); // make PB7 = D/C# = 0
          GPIOB->ODR &= ~(1 << 6); // make PB6 = CS# = 0
          oled_Write( cmd );
          GPIOB->ODR |= (1<<6); // make PB6 = CS# = 1
}
void oled_Write_Data( unsigned char data )
{
          GPIOB->ODR |= (1 << 6); // make PB6 = CS# = 1
          GPIOB->ODR |= (1 << 7); // make PB7 = D/C# = 1
          GPIOB->ODR &= ~(1 << 6); // make PB6 = CS# = 0
          oled_Write( data );
          GPIOB->ODR |= (1 << 6); // make PB6 = CS# = 1
}
void oled_Write( unsigned char Value )
{
/* Wait until SPI1 is ready for writing (TXE = 1 in SPI1_SR) */
          while(!(SPI1->SR & SPI_SR_TXE))
                    {
                    trace_printf("Stuck\n");;
                    }
          /* Send one 8-bit character:
            - This function also sets BIDIOE = 1 in SPI1_CR1
          */
          HAL_SPI_Transmit( &SPI_Handle, &Value, 1, HAL_MAX_DELAY );
          while(SPI1->SR & SPI_SR_BSY); //wait for SPI BSY to not be busy
}
void oled_config( void )
{
          trace_printf("Start\n");
          RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
```

```
// Configure PB3 and PB5 as Alternate Function (AF0) for SPI1
	GPIOB->MODER &= ~(GPIO_MODER_MODER3 | GPIO_MODER_MODER5);
	GPIOB->MODER |= GPIO_MODER_MODER3_1; //alternate mode
	GPIOB->MODER |= GPIO_MODER_MODER5_1; //alternate mode
	GPIOB->AFR[0] |= (0X0 << GPIO_AFRL_AFSEL3_Pos) | (0X0 << GPIO_AFRL_AFSEL5_Pos); //AF0
	//Configure PB4 as output for RES
	GPIOB->MODER |= GPIO_MODER_MODER4_0;
	//Configure PB6 as output for CS
	GPIOB->MODER |= GPIO_MODER_MODER6_0;
	//Configure PB7 as output for D/C
	GPIOB->MODER |= GPIO_MODER_MODER7_0;
	//Enable SPI1 clock
	RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
	trace_printf("Start2\n");
	SPI_Handle.Instance = SPI1;
	SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
	SPI_Handle.Init.Mode = SPI_MODE_MASTER;
	SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
	SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
	SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
	SPI_Handle.Init.NSS = SPI_NSS_SOFT;
	SPI_Handle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
	SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;
	SPI_Handle.Init.CRCPolynomial = 7;
	// Initialize the SPI interface
	HAL_SPI_Init( &SPI_Handle );
	// Enable the SPI
	__HAL_SPI_ENABLE( &SPI_Handle );
	/* Reset LED Display (RES# = PB4):
	  - make pin PB4 = 0, wait for a few ms
	  - make pin PB4 = 1, wait for a few ms
	*/
	GPIOB->ODR &= ~(0b10000);
	for(int x = 0; x < 100000 * 3; x++);
	trace_printf("Start3\n");
	GPIOB->ODR |= 0b10000;
	for(int x = 0; x < 100000 * 3; x++);
	trace_printf("Start4\n");
	// Send initialization commands to LED Display
	for ( unsigned int i = 0; i < sizeof( oled_init_cmds ); i++ )
	{
		//trace_printf("oled_init_cmds size = %d\n", sizeof(oled_init_cmds[i]));
	    oled_Write_Cmd( oled_init_cmds[i] );
	}
	/* Fill LED Display data memory (GDDRAM) with zeros:
	  - for each PAGE = 0, 1, ..., 7
		set starting SEG = 0
		call oled_Write_Data( 0x00 ) 128 times
	*/
```

```
	//Clearing:
	for(int x = 0; x < 8; x++)
	{
		oled_Write_Cmd(0xB0 + x);
		oled_Write_Cmd(0x00);
		oled_Write_Cmd(0x10);
		for (int y = 0; y < 128; y++) {
			oled_Write_Data(0x00);
		}
	}
	trace_printf("End of config\n");
}
void EXTI0_1_IRQHandler(void)
{
	/*
	In EXTI0_1_IRQHandler() do the following:
	Check if the EXTI1 flag is set → 555 timer stuff
	Check if EXTI0 flag is set → button press
	Check if button is pressed
	If the global signal is 0, set it to 1 and disable EXTI1 and enable EXTI2
	Testing: print function generator enable
	Else: reset global signal to 1, disable EXTI2 and enable EXTI1
	Testing: print 555 timer enable
	Clear pending flag EXTI0 pending flag
	Clear pending flag EXTI1 pending flag
	*/
	//trace_printf("Interrupt called\n");
	if(EXTI->PR & EXTI_PR_PR1)
	{
		double float_frequency;
		double float_scaledPeriod;
		unsigned int frequency;
		unsigned int scaledPeriod;
		if(!timerTriggered)
		    {
		      TIM2->CNT = 0;
		      TIM2->CR1 |= TIM_CR1_CEN;
		      timerTriggered = 1;
		    }
				//  Else (this is the second edge):
				//	- Stop timer (TIM2->CR1).
				//	- Read out count register (TIM2->CNT).
				//	- Calculate signal period and frequency.
				//	- Print calculated values to the console.
				//	  NOTE: Function trace_printf does not work
				//	  with floating-point numbers: you must use
				//	  "unsigned int" type to print your signal
				//	  period and frequency.
		    else
```

```
                {
                        TIM2->CR1 &= ~(TIM_CR1_CEN);
                        uint32_t timerValue = TIM2->CNT;
                        float_frequency = 48000000 / timerValue;
                        float_scaledPeriod = 1/float_frequency;
                        frequency = float_frequency;
                        //trace_printf("555 Frequency: %u.%02u Hz\n", frequency, frequency
% 100);
                        Freq = frequency;
                    Res = (ADC1->DR * 5000) / 4095;
                        //trace_printf("Resistance: %.2f\n", Res);
                        timerTriggered = 0;
                }
            EXTI->PR |= EXTI_PR_PR1; //Clear pending flag
        }
        if(EXTI->PR & EXTI_PR_PR0)
        {
            if(globalSignal == 0)
            {
                globalSignal = 1;
                EXTI->IMR &= ~EXTI_IMR_MR1; //Disable interrupt for EXTI1
                EXTI->IMR |= EXTI_IMR_MR2;  //Enable interrupt for EXTI2
                trace_printf("Function generator enabled\n");
            }else if(globalSignal == 1)
            {
                globalSignal = 0;
                EXTI->IMR &= ~EXTI_IMR_MR2; //Disable interrupt for EXTI2
                EXTI->IMR |= EXTI_IMR_MR1;  //Enable interrupt for EXTI1
                trace_printf("555 timer enabled\n");
            }
            EXTI->PR |= EXTI_PR_PR0; //Clear EXTI0 Pending flag
        }
}
#pragma GCC diagnostic pop
// ----------------------------------------------------------------------------
```

References

[1] B. Sirna and D. Rakhmatov, “ECE 355: Microprocessor Based Systems - Laboratory Manual.” University of Victoria, Victoria, 2023

[2] D. Rakhmatov, “I/O Examples.” University of Victoria, Victoria

[3] D. Rakhmatov, “Interfacing Examples.” University of Victoria, Victoria

[4] “STM32F0 Reference Manual.” STMicroelectronics, 2014