University of Victoria

Department of Electrical and Computer Engineering

ECE 355 - Microprocessor-Based-Systems

Laboratory Project

PWM Signal Generation and Monitoring System

Date of Demonstration: November 22, 2024

Report Submitted on: November 29, 2024

To: Dr. D. Rakhmatov

Liam Tanner V00982393
Erich Rueben V00963036

Table of Contents

Problem Description/Specifications

The objective of this laboratory project is to create a PWM signal generation and monitoring system using the STM32F0 Microcontroller, 4N35 Optocoupler, NE555 Timer, a Keysight 33250A Function Generator and the SSD1306 Display. An integrated circuit capable of generating digital signals will be built by our group. The Function Generator will act as a second input source; generating square waves. The input signal will then be measured using the internal timer TIM2, and Resistance and Frequency will be calculated.

The integrated circuit will include a potentiometer that sends analog signals to the microcontroller's Analog to Digital Conversion register. Once converted, the digital signal will be used to calculate the resistance of the circuit then sent to both the SPI to be shown on the SSD1306 Display and to the Digital to Analog Converter within the microcontroller in order to convert the signal back to analog. This analog signal is then sent out of the microcontroller, through the 4N35 Optocoupler and NE555 Timer to create a square wave. Finally, the square wave is sent as input to the microcontroller.

In order to switch between the monitored input sources, the user button, connected to PA0 of the microcontroller, will be pressed (see table 1 for all port specifications). This will cause an EXTI0 interrupt, disabling the measurement of one input signal, and enabling the other. Once the frequency is calculated from the current input source, it will be sent along with the previously calculated resistance value to the microcontroller's SPI which outputs the information to the SSD1306 Display.

Figure 1 displays the different components of the system and their interactions as a whole. This is done at a very high level, each component of the system will be expanded upon in the Design/Specification section of our report.

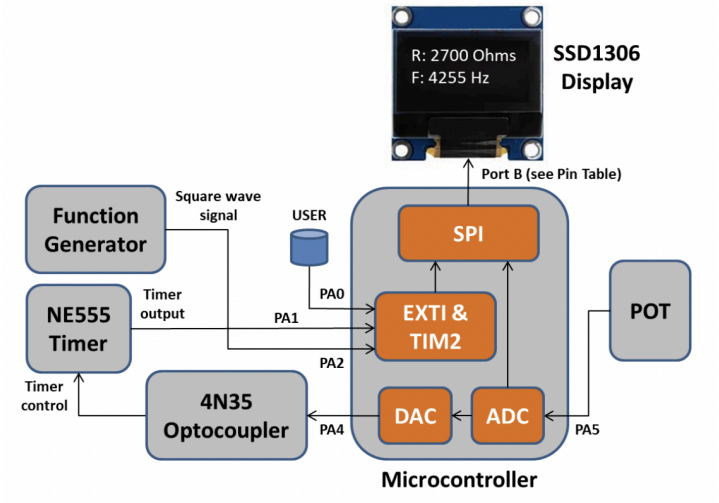


Figure 1. Diagram of the System [1]

Design/Solution

The following design will adhere to the specifications outlined in the previous section, in order to solve the PWM signal generation and monitoring system problem. The bulk of our solution consists of embedded C code written to the STM32F0 Microcontroller. This code works in tandem with an integrated circuit. We describe the components of both the code and circuit in this section so that our solution can be easily reproduced by future engineers. This section will describe the code and circuit but does not include them explicitly. Our source code and images of our circuit wiring can be found in Appendix A.

Global Variables

In order for all the functions of our program to run as a whole, we keep track of several global variables. The resistance and frequency values are used by the refresh_OLED function; however, they are set outside this scope. Therefore, we keep track of them using the global variables “Res” and “Freq”. We must also keep track of the current input source being monitored. In order to accomplish this, we use “inSig” which takes on a value of 1 when the input signal originates from the NE555 Timer and 0 when it comes from the function generator. Finally, the global variable “EdgeState” is used by both the EXTI0_1_IRQHandler() and the EXTI2_3_IRQHandler() in order to monitor whether the current rising edge is the first or second. “EdgeState” takes on the value of 0 for the first edge and 1 for the second.

TIM2

TIM2 is one of the internal timers available in the STM32F0 microcontroller. It is a 32-bit general-purpose timer that can be used for various timing and counting applications. For our purposes, we will be using TIM2 for overflow handling and to measure the period between two rising edges of a square wave input signal which we can then use to calculate frequency.

TIM2 Initialization

The myTIM2_Init() function initializes TIM2 to measure the period of an input signal. It begins by enabling the TIM2 peripheral clock through the RCC register. The timer is then configured to count up, buffer the auto-reload value, and generate update events only on overflow, which ensures precise timing measurements. Now, the prescaler (PSC) and auto-reload (ARR) registers are set to define the timer’s resolution and maximum counting period. In order to apply these settings, the function triggers an update event using the event generation register (EGR). Finally, the interrupt for TIM2 is configured with the highest priority in the NVIC, enabling it to respond to overflow events and ensuring the system can handle signals effectively.

TIM2 Overflow Handler

Upon overflow, a TIM2 interrupt is thrown. Within the handler for this interrupt, we first check if the interrupt flag has indeed been set. If it has, We print an overflow message to the user, clear the flag and restart the stopped timer. The cause of overflow will be explained in the test/results section under “Function Generator”.

ADC

ADC Initialization

The myADC_Init() function initializes the ADC peripheral to convert analog signals from the potentiometer into digital values. It first enables the ADC clock through the RCC register and configures the input pin (PA5) as analog mode. The ADC is set to operate in continuous conversion mode, allowing it to perform successive conversions without re-triggering. The desired ADC channel is selected in the channel selection register (CHSELR), and the sampling time is configured for accurate signal acquisition. The ADC is then enabled using the control register (CR), and the function waits until the ADC is ready for operation, ensuring it is fully initialized before use.

ADC Conversion

After initializing the ADC, we can start converting the potentiometer value from analog to digital. Within the ADC_Converter() function, we begin by starting the ADC process by manipulating the ADC1 Control Register. We then wait for the end of conversion flag by polling the ADC1 Interrupt and Status register. Finally, we save the converted value held in the ADC1 Data Register to our “Res” global variable.

DAC

DAC Initialization

The myDAC_Init() function initializes the Digital-to-Analog Converter (DAC) to convert digital resistance values into an analog signal. It begins by enabling the DAC peripheral clock via the RCC register to ensure the DAC module is powered. The function then configures pin PA4 as an analog output to route the converted analog signal out of the microcontroller. This setup prepares the DAC for use in generating the output signal.

DAC Conversion

Once the microcontroller’s DAC has been initialized, we start the digital to analog conversion by writing the digital resistance value stored in the “Res” global variable to the DAC’s data holding register. The DAC is then enabled through the control register (CR) to begin the conversion process. The resulting analog signal is output through pin PA4, which is subsequently processed by external components, such as the 4N35 optocoupler and NE555 timer, to create a square wave for signal monitoring.

Wiring

The wiring for our system can be described in two parts; the first being a circuit designed for square wave generation and the second being connections from the microcontroller’s SPI to the SSD1306 Display. The circuit includes an external potentiometer, the internal ADC and DAC, a 4N35 Optocoupler, an NE555 Timer, the user button and a function generator.

The circuit makes use of Port A while the Display is connected via Port B. The exact port specifications can be seen in Table 1 below. Additionally, Figure 2 shows all the interactions of the circuit as a whole. Furthermore, images of our exact wiring can be found in Appendix A under “4N35 Optocoupler and NE555 Timer Wiring” and “PBMCUSLK Board Wiring to J5 Connector for OLED Display”.

Port	**Connected I/O Device**	**J5 Connector Pin**	**Direction**
PA0	User Button	NA	Input
PA1	NE555 Timer	NA	Input
PA2	Keysight 33250A Function Generator	NA	Input
PA4	4N35 Optocoupler	NA	Output
PA5	Potentiometer	POT (J10)	Input
PB3	SCLK	21	Output
PB4	RES#	19	Output
PB5	SDIN	17	Output
PB6	CS#	25	Output
PB7	D/C#	23	Output

Table 1. Port Specifications [4]

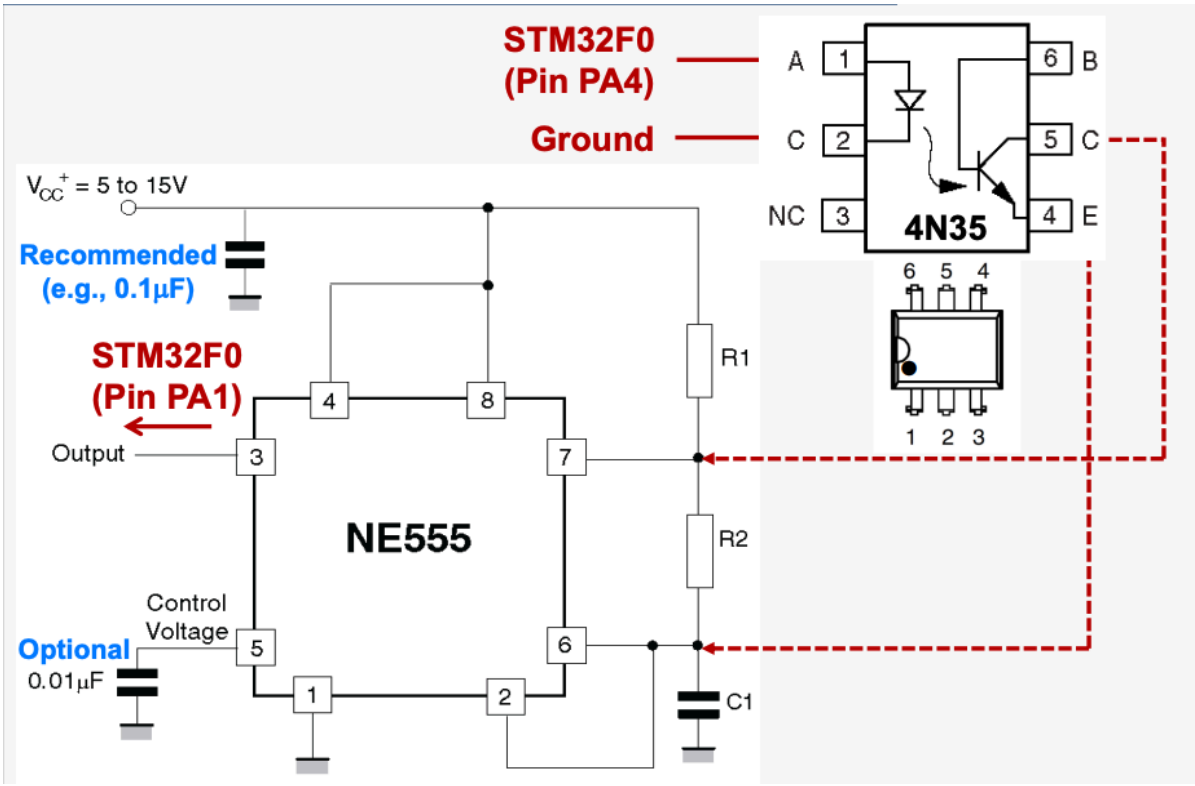


Figure 2. Circuit Schematic for Optocoupler and Timer [4]

4N35 Optocoupler

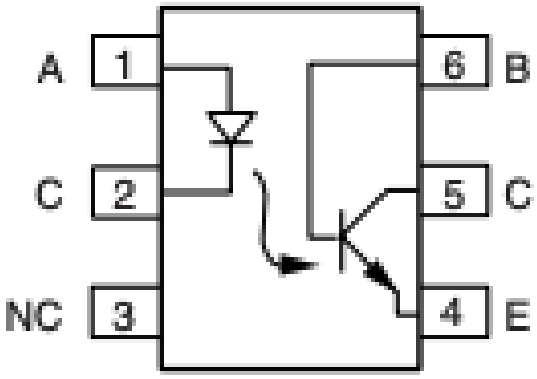


Figure 3. 4N35 Optocoupler [2]

Pin number	**Function**
1	Anode
2	Cathode

3	No Connection
4	Emitter
5	Collector
6	Base

Table 2. Description of the Optocoupler Pins [2]

In our system, the 4N35 Optocoupler is used to control the frequency of the signal emitted by the DAC. With a silicon NPN phototransistor and infrared LED, the optocoupler transfers signals to the NE555 Timer. This isolates the DAC from the rest of the circuit. In order to do this, we had to perform several connections. The anode pin of the Optocoupler was connected to PA4 of the microcontroller while the cathode was connected to ground. The emitter and collector pins were then connected in series with a resistor and then the NE555 Timer. This can be seen in Figure 2 above.

NE555 Timer

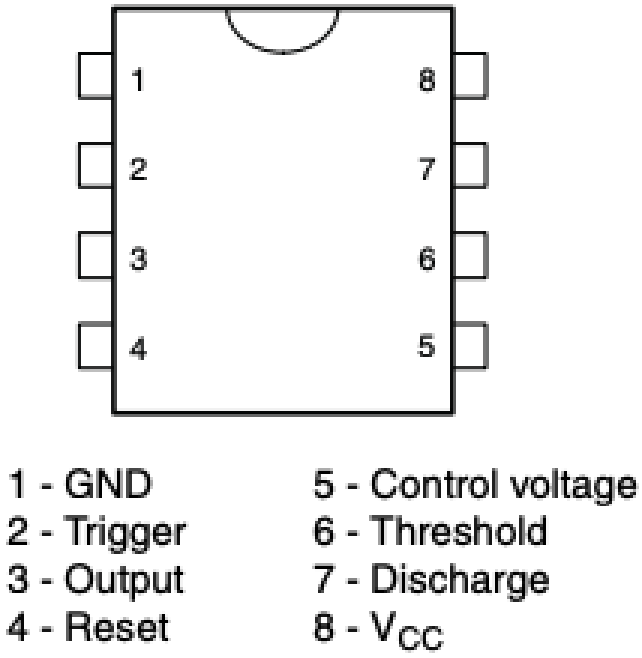


Figure 4. NE555 Timer [3]

We use the NE555 timer to generate square waves of varying frequency based on the output coming from the emitter and collector pins of the 4N35 Optocoupler. The NE555 Timer is then responsible for sending these square waves as input to PA1 of the microcontroller. The exact connections of each of the Timer's pins can be seen in Figure 2. In order to initialize the NE555 Timer we wrote code to GPIOA and EXTI1 which will be expanded upon in the "GPIOA Initialization" and "EXTI Initialization" sections.

Furthermore, the handling of external interrupts caused by incoming square waves from the NE555 Timer will be described in the “EXTI0_1 Handler” section.

PBMCUSLK Board Pins

The PBMCUSLK Board provides the physical interface and connections necessary for the SPI of the microcontroller to connect with external devices. The PBMCUSLK Board is directly connected to Port B of the microcontroller. In order to display our previously calculated frequency and resistance values to the SSD1306 OLED Display, we made connections from Port B of the microcontroller to the J5 Connector. The wiring between the J5 Connector and Display was not implemented by our group and therefore, we cannot describe it.

The relevant PBMCUSLK pins used for this are the SPI1 output MOSI, SPI1 Serial Clock (SCK), LED Display Chip Select (CS#), LED Display Data Command (D/C#) and the LED Display Reset (RES#). The connections between these pins and the J5 Connector pins are described in Table 1.

GPIO Initialization

Initialization for GPIOA and GPIOB are two very similar processes for our system. Our myGPIOA_Init() is a relatively simple function, responsible for port configuration and ensuring no pull-up or pull-down for these ports. Before this, we first enable the clock for the GPIOA peripheral. PA0 is connected to the user button, PA1 is connected to the NE555 Timer and PA2 is connected to the Keysight 33250A Function Generator. Because of this, all three of those ports must be configured as input. We then ensure no pull-up or pull-down for those same ports by manipulating the GPIOA Pull Up Pull Down register. Similarly, myGPIOB_Init() starts by enabling the clock for the GPIOB peripheral. It then sets PB4, PB6 and PB7 as output and ensures no pull-up or pull-down for PB3, PB4, PB5, PB6 and PB7.

EXTI

Our system uses external interrupts to monitor when the user button is pressed and to handle incoming square wave signals from both the function generator and the NE555 Timer. EXTI0 monitors the user button, EXTI1 is responsible for the NE555 input signals and EXTI2 handles the function generator input.

EXTI Initialization

In order to initialize the EXTIs we use the myEXTI_Init() function. This function starts by mapping the EXTI0 line to PA0, the EXTI1 line to PA1 and the EXTI2 line to PA2 using the EXTI Control Register. Once this mapping has been completed, we use the Rising Trigger Selection Register to set the rising edge trigger for the three previously mentioned EXTIs. Then we unmask the interrupts from the EXTI lines using the Interrupt Mask Register. Finally we use The Nested Vectored Interrupt Controller to set the priority of EXTI0_1_IRQn and EXTI2_3_IRQn to 0 and enable EXTI interrupts. This code can be seen in Appendix A under “EXTI Initialization Code”.

EXTI0_1 Handler

The EXTI0_1_IRQHandler() function handles both the user button press and the NE555 Timer input signal. Because of this, we first check to see if the EXTI Pending Register detects an EXTI0 interrupt flag. If it does, then we know the button has been pressed. Otherwise, we confirm that there is indeed an incoming rising edge from the NE555 Timer by checking the Pending Register for an EXTI1 interrupt.

Once we have confirmed that the button was pressed, we check our inSig global variable to determine the current input source. If "inSig" indicates that it is the function generator, we disable EXTI1 interrupts and enable EXTI2 interrupts. If inSig indicates that the current input source is the NE555 Timer, we do the opposite. In both cases we must toggle the "inSig" value to indicate that the input source has changed. We then clear the flag in the EXTI Pending Register and wait for 1 second to ensure that the button press does not cause a double switch.

If instead there is an incoming rising edge from the NE555 Timer, we check "EdgeState". If this indicates that the interrupt was caused by the first rising edge, we start counting with TIM2 and let "EdgeState" take on the value of 1. The next interrupt will be caused by the second rising edge. This means that we stop TIM2 and save the current time to a floating point variable "time". This value is then used to calculate frequency which we save in the global variable Freq. We then let "EdgeState" be 0 again in anticipation for the next rising edge coming from the NE555 Timer. Regardless of whether the rising edge was the first or second, we clear the Pending Register before exiting the function.

EXTI2_3 Handler

For our solution, the EXTI2_3_IRQHandler() function is only concerned with interrupts caused by EXTI2. We start by checking that there is indeed an EXTI2 interrupt flag raised in the Pending Register. If there is, we know that there is an incoming rising edge from the Keysight 33250A Function Generator. At this point, we handle the interrupt in the exact same way as EXTI1 interrupts since the input source has no effect on the process of frequency calculation.

Delay Function

In order to create delays in our program we implemented our own delay function. This allows us to input the number of milliseconds we would like to wait from anywhere in our program. The delay function uses TIM3 to wait for the specified number of milliseconds using a simple while loop. Once the time is up, the program continues to run from where it left off.

OLED Display

OLED Configuration

In order to configure the OLED display we start by enabling the GPIOB and SPI1 clock in RCC. Then we configure PB3 and PB5 as Alternate Function #0. We must then set the SPI control register bits such that the active clock edge is it's first edge, the idle clock polarity is 0, SPI1 is in Master mode, datasize is set to 8 bits, NSS is set to soft, baud rate is pre scaled by $1/2^{257}$, SPI direction is set to bidirectional, and CRC

is set to 7. Then the SPI is initialized and enabled. We then reset the display by setting PB4 to 0 and then 1, performing a 100ms delay after each. Then we write the predefined oled initialization commands to the screen. Lastly, we can clear the screen by writing 0's to each lattice.

OLED Refresh

Compared to the configuration, the OLED refresh is quite simple. The first step is to fill the buffer array with the contents to display, this is done using the snprinf method. Then the initial lattice must be selected by writing commands to the screen. Once the lattice is selected we can send each character of our buffer, converted into its ASCII value, byte by byte.

Main Function

Since we have broken down each component of our program into separate, more complex functions, this allows our main to be very simple and consist of only function calls. Main starts by initializing each of the previously discussed components of the microcontroller and then configures the SSD1306 Display. Main then loops infinitely, refreshing the display, performing ADC conversion and finally, DAC conversion.

Testing/Results

Our testing method consisted primarily of exploratory and sanity testing. We performed the task at hand in the manner we believed was correct, then checked the result with the expected values. If the resulting values varied greatly from the expected values, we knew the implementation was incorrect. If the resulting values were within a realistic range then we knew the implementation was correct.

While every section of our system could be tested or verified, we focused primarily on three major components. These components were the Function Generator, the ADC, and the NE555 Timer. With previous knowledge of the realistic expectations for the values measured by these components, we can confidently say our system implementation is correct.

Component	Testing Method
Function Generator	Measured frequency matched the frequency displayed on the Keysight 33250A Function Generator within a variance of 2%.
ADC	Resistance measured was at 0 Ohms when the potentiometer was turned to the low end (left) and 5000 Ohms when it was turned to the high end (right), consistently changing in between.
NE555 Timer	Measured frequency matched the given range of ~800 Hz to ~1500 Hz changing proportional to the measured resistance.

User Button	The input wave signal switches when the button is pressed

Table 3. Testing Methods

Function Generator

The function generator is capable of sending a square wave to our system at a selected frequency. This frequency was clearly shown in the function generator display. When testing our resulting value, we could look at the set frequency and see if it matched the measured value. If the measured value was within 2% of the set value, the measurement was accepted. To test our system we used frequencies in the range of 10 mHz to 550,000 Hz, due to the erratic behavior presented by any value outside this range. This upper end of this behavior is due to the frequency of interrupts, specifically due to the systems inability to handle an interrupt before the arrival of the next one, leading to incorrect behavior. The lower end is due to the timer interrupt handler, the timer “times out” before the wave frequency can be measured. In other words, the period of the wave is longer than the maximum timer value.

ADC

The Analog-to-digital Converter (ADC) continuously receives an analog value from the potentiometer and converts it to a digital resistance value. The range for the resistance value was discussed in class and determined to be 0 Ohms to 5000 Ohms. When the system output a value that correlated with this range, the implementation was deemed correct.

NE555 Timer

The NE555 Timer is capable of sending a square-wave PWM signal with a frequency determined by the optocoupler. The range for the frequency values was discussed in class, where at a maximum resistance value the frequency should be ~1500 Hz and at a minimum resistance value the frequency should be ~800 Hz. This range was used to verify the measured frequency, where the minimum was found to be ~795 Hz and the maximum was ~1350 Hz.

User Button

The user button used in our design, was configured such that when it is clicked the system switches the source of the input wave (function generator to NE555 timer and vice versa). On the click of the button, the global variable inSig changes value, defining which input signal to listen to. Testing this component was as simple as pressing the user button and watching the behavior of the system. The major issue found during testing was the velocity of the system in comparison to human reaction, leading to the implementation of a delay within the interrupt handler.

Discussion

This project demonstrated the integration of multiple peripherals with the STM32F0 microcontroller, including the Keysight 33250A Function Generator, NE555 timer, optocoupler, OLED display, and ADC/DAC components.

The system successfully measured the frequency from two wave input sources as well as the resistance value of the potentiometer. The first being the Function Generator, which directly sent a square wave input to the system. The second being the NE555 Timer, which received input from the optocoupler controlling the resistance value and thus controlling the square wave sent to the microcontroller. These measured values were then displayed on the SSD1306 Display.

Shortcomings included minor noise in the wave frequency and potentiometer readings. While they did not affect the overall functionality of the system, they distorted the values measured within a factor of ~5%. These issues, while noteworthy, did not take away from the success of the system.

Some challenges we faced during the implementation of our project include issues with trace_printf statements, and the button release delay. The original print statements found in our system created a timing issue. When the frequency of each interrupt was printed to the screen, the system would fail to detect the interrupt caused by the user button. To resolve this issue, we removed the print statements once we were sure the frequencies were measured correctly. Therefore, relying entirely on the OLED display to read the measured values. This issue was likely caused by the delay print statements have during runtime, leading to a longer handling of EXTI1 and EXTI2 interrupts. The second issue with the button release delay was caused by the system's speed at detecting interrupts causing the EXTI0 interrupt to be thrown multiple times during one user button press. This was easily resolved using TIM3 to perform a delay after each EXTI0 interrupt, allowing the handler to release the button before the system continues.

Overall the system worked cohesively, properly detecting interrupts in a seamless manner. Despite minor limitations, the results were consistent with the expectations, and provided insight to the behavior of a STM32F0 microcontroller when coupled with multiple peripheral devices.

Appendices

Appendix A

TIM2 Initialization Code

```
void myTIM2_Init()
{
	trace_printf("In myTIM2_Init\n");
	/* 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 = 0x008C;

	/* Set clock prescaler value */
	// Relevant register: TIM2->PSC
	TIM2->PSC = myTIM2_PRESCALER;

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

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

	/* Assign TIM2 interrupt priority = 0 in NVIC */
	// Relevant register: NVIC->IP[3], or use NVIC_SetPriority
	NVIC_SetPriority(TIM2_IRQn, 0);

	/* 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;
}
```

TIM2 IRQ Handler Code

```
void TIM2_IRQHandler()
{
	trace_printf("Detected TIM2 Interrupt\n");
	/* Check if update interrupt flag is indeed set */
	// Relevant register: TIM2->SR
	if ((TIM2->SR & TIM_SR_UIF) != 0)
	{
		trace_printf("\n*** Overflow! ***\n");
		/* Clear update interrupt flag */
		// Relevant register: TIM2->SR
```

```
                TIM2->SR &= ~(TIM_SR_UIF);

                /* Restart stopped timer */
                // Relevant register: TIM2->CR1
                TIM2->CR1 |= TIM_CR1_CEN;
        }
}
```

ADC Initialization Code

```
void myADC_Init()
{
        trace_printf("In myADC_Init\n");
        /* Enable clock for ADC peripheral */
        // Relevant register: RCC->APB2ENR
        RCC->APB2ENR |= RCC_APB2ENR_ADCEN;

        /* Configure PA5 as analog */
        // Relevant register: GPIOA->MODER
        GPIOA->MODER |= GPIO_MODER_MODER5_Msk;

        /* Configure ADC for continuous conversion */
        // Relevant register: ADC1->CFGR1
        ADC1->CFGR1 |= ADC_CFGR1_CONT;

        /* Select ADC channels */
        // Relevant register: ADC1->CHSELR
        ADC1->CHSELR |= ADC_CHSELR_CHSEL5;

        /* Configure sampling time to 239.5 ADC clock cycles */
        // Relevant register: ADC1->SMPR
        ADC1->SMPR |= 0x7;

        /* Enable ADC process */
        // Relevant register: ADC1->CR
        ADC1->CR = ADC_CR_ADEN;
        while((ADC1->ISR & ADC_ISR_ADRDY_Msk) != 0x01) {}
}
```

ADC Conversion Code

```
void ADC_Converter()
{
```

```
        /* Start ADC process */
        // Relevant register: ADC1->CR
        ADC1->CR |= ADC_CR_ADSTART;

        /* Wait for end of conversion flag */
        // Relevant register: ADC1->ISR
        while((ADC1->ISR & ADC_ISR_EOC_Msk) != ADC_ISR_EOC_Msk) {}

        /* Read value from ADC data register */
        // Relevant register: ADC1->DR
        Res = ( ADC1->DR * 5000 )/ 4095 ;
        digital_res = ADC1->DR;
}
```

DAC Initialization Code

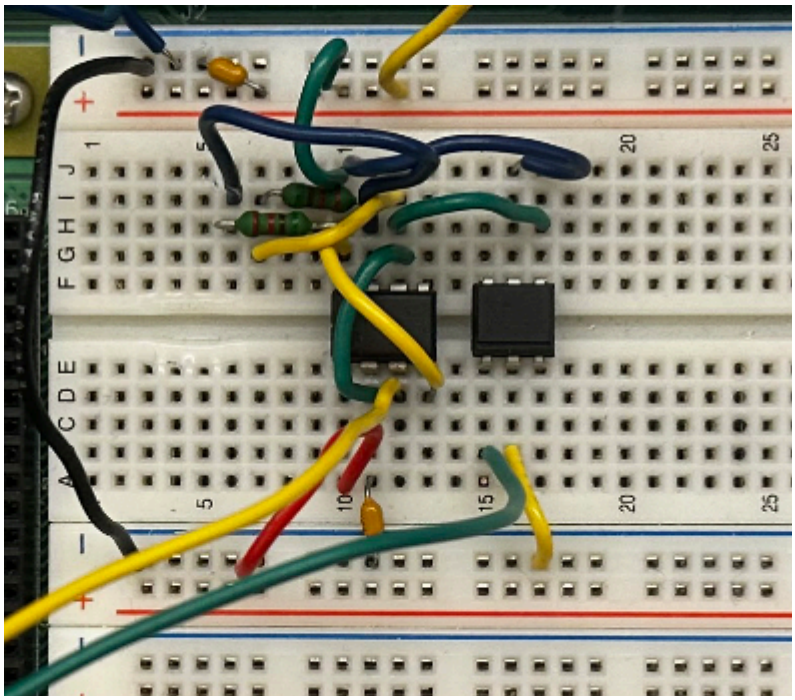
```
void myDAC_Init()
{
        trace_printf("In myDAC_Init\n");
        /* Enable clock for DAC peripheral */
        // Relevant register: RCC->APB2ENR
        RCC->APB1ENR |= RCC_APB1ENR_DACEN;

        /* Configure PA4 as analog */
        // Relevant register: GPIOA->MODER
        GPIOA->MODER |= GPIO_MODER_MODER4_Msk;
}
```

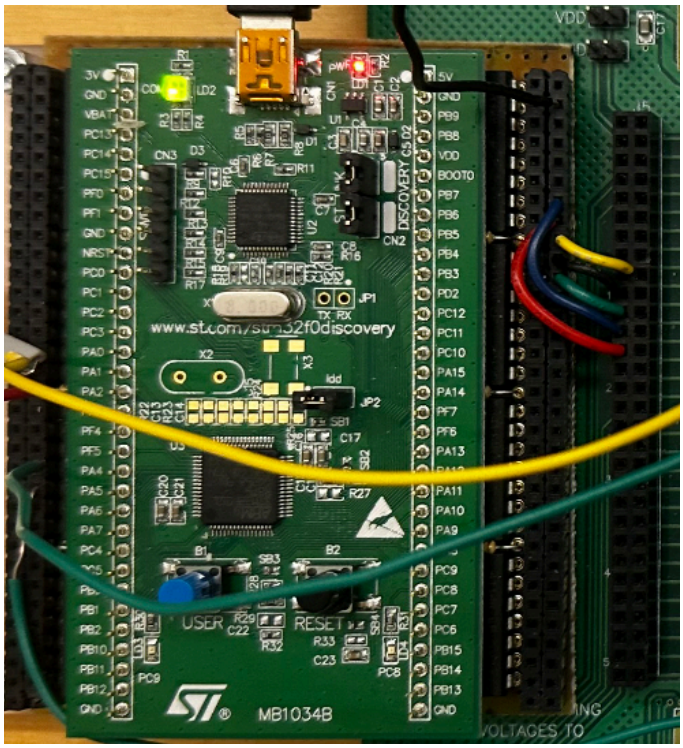
DAC Conversion Code

```
void DAC_Converter()
{
        /* Store Digital value for conversion */
        // Relevant register: DAC->DHR12R1
        DAC->DHR12R1 = digital_res;
        /* Start Conversion */
        // Relevant register: DAC->CR
        DAC->CR |= DAC_CR_EN1;
}
```

4N35 Optocoupler and NE555 Timer Wiring



PBMCUSLK Board Wiring to J5 Connector for OLED Display



GPIOA Initialization Code

```
void myGPIOA_Init()
{
	trace_printf("In myGPIOA_Init\n");

	/* Enable clock for GPIOA peripheral */
	// Relevant register: RCC->AHBENR
	RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

	/* Configure PA0, PA1 and PA2 as input */
	// Relevant register: GPIOA->MODER
	GPIOA->MODER &= ~(GPIO_MODER_MODER0 & GPIO_MODER_MODER1 & GPIO_MODER_MODER2);

	/* Ensure no pull-up/pull-down for PA0, PA1 and PA2 */
	// Relevant register: GPIOA->PUPDR
	GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0 & GPIO_PUPDR_PUPDR1 & GPIO_PUPDR_PUPDR2);
}
```

GPIOB Initialization Code

```
void myGPIOB_Init()
{
	trace_printf("In myGPIOB_Init\n");

	/* Enable clock for GPIOA peripheral */
	// Relevant register: RCC->AHBENR
	RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

	/* Configure PB4, PB6, PB7 as output */
	// Relevant register: GPIOA->MODER
	GPIOB->MODER &= ~(GPIO_MODER_MODER4 | GPIO_MODER_MODER6 | GPIO_MODER_MODER7);
	GPIOB->MODER |= (GPIO_MODER_MODER4_0 | GPIO_MODER_MODER6_0 |
GPIO_MODER_MODER7_0);

	/* Ensure no pull-up/pull-down for PB3, PB4, PB5, PB6, PB7 ports */
	// Relevant register: GPIOA->PUPDR
	GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3 | GPIO_PUPDR_PUPDR4 | GPIO_PUPDR_PUPDR5 |
GPIO_PUPDR_PUPDR6 | GPIO_PUPDR_PUPDR7);
}
```

EXTI Initialization Code

```
void myEXTI_Init()
```

```
{
	trace_printf("In myEXTI_Init\n");
	/* Clear EXTI0 control register */
	// Relevant register: SYSCFG->EXTICR[0]
	SYSCFG->EXTICR[0] &= ~SYSCFG_EXTICR1_EXTI0;
	SYSCFG->EXTICR[0] &= ~SYSCFG_EXTICR1_EXTI1;
	SYSCFG->EXTICR[0] &= ~SYSCFG_EXTICR1_EXTI2;

	/* Map EXTI0 line to PA0, EXTI1 line to PA1, and EXTI2 line to PA2 */
	// Relevant register: SYSCFG->EXTICR[0]
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI0_PA;
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA;
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI2_PA;

	/* EXTI line interrupts: set rising-edge trigger */
	// Relevant register: EXTI->RTSR
	EXTI->RTSR |= EXTI_RTSR_TR0;
	EXTI->RTSR |= EXTI_RTSR_TR1;
	EXTI->RTSR |= EXTI_RTSR_TR2;

	/* Unmask interrupts from EXTI lines */
	// Relevant register: EXTI->IMR
	EXTI->IMR |= EXTI_IMR_MR0;
	EXTI->IMR |= EXTI_IMR_MR1;
	EXTI->IMR &= ~EXTI_IMR_MR2;

	/* Assign EXTI interrupt priority = 0 in NVIC */
	// Relevant register: NVIC->IP[2], or use NVIC_SetPriority
	NVIC_SetPriority(EXTI0_1_IRQn, 0);
	NVIC_SetPriority(EXTI2_3_IRQn, 0);

	/* Enable EXTI interrupts in NVIC */
	// Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
	NVIC_EnableIRQ(EXTI0_1_IRQn);
	NVIC_EnableIRQ(EXTI2_3_IRQn);
}
```

EXTI0 and EXTI1 Handler Code

```
void EXTI0_1_IRQHandler()
{
	/* Check if EXTI0 flag is set, else EXTI1 flag is set */
	// Relevant register: EXTI->PR
```

```
	if((EXTI->PR & EXTI_PR_PR0) != 0){
		trace_printf( "Button Pushed! " );
		if(inSig == 0){

			/* Flip inSig */
			inSig = 1;

			/* Disable EXTI1 interrupts */
			// Relevant register: EXTI->IMR
			EXTI->IMR &= ~EXTI_IMR_MR1;

			/* Enable EXTI2 interrupts */
			// Relevant register: EXTI->IMR
			EXTI->IMR |= EXTI_IMR_MR2;
			trace_printf("Switched to Function Generator input signal.\n");

		} else {
			/* Flip inSig */
			inSig = 0;

			/* Disable EXTI2 interrupts */
			// Relevant register: EXTI->IMR
			EXTI->IMR &= ~EXTI_IMR_MR2;

			/* Enable EXTI1 interrupts */
			// Relevant register: EXTI->IMR
			EXTI->IMR |= EXTI_IMR_MR1;
			trace_printf("Switched to NE555 Timer input signal.\n");
		}

		/* Reset the edge count */
		EdgeState = 0;

		// Give human time to release button
		delay(1000);

		 /* Clear the interrupt flag */
		// Relevant register: EXTI->PR
		EXTI->PR = EXTI_PR_PR0;

	} else if((EXTI->PR & EXTI_PR_PR1) != 0){
```

```
                /* Verify that the system should be reading from NE555 Timer */
                if(inSig == 0){

                        /* Check whether this is the first edge, else it is the second
*/
                        if (EdgeState == 0) {

                                /* Clear the timer */
                                // Relevant register: TIM2->CNT
                                TIM2->CNT = 0x0;

                                /* Start the timer */
                                // Relevant register: TIM2->CR1
                                TIM2->CR1 |= TIM_CR1_CEN;

                                /* Update the edge state */
                                EdgeState = 1;
                        } else {

                                /* Stop the timer */
                                // Relevant register: TIM2->CR1
                                TIM2->CR1 &= ~TIM_CR1_CEN;

                                /* Read the timer value and calculate the frequency */
                                // Relevant register: TIM2->CNT
                                float time = TIM2->CNT;
                                Freq = 48000000/time;

                                /* Reset the edge state */
                                EdgeState = 0;
                        }
                }

                /* Clear the interrupt flag */
                // Relevant register: EXTI->PR
                EXTI->PR = EXTI_PR_PR1;
        }
}
```

EXTI2 and EXTI3 Handler Code

```
void EXTI2_3_IRQHandler()
{
```

```
	/* Check if EXTI2 interrupt flag is set */
	// Relevant register: EXTI->PR
	if ((EXTI->PR & EXTI_PR_PR2) != 0)
	{
		/* Check whether this is the first edge, else it is the second */
		if (EdgeState == 0) {

			/* Clear the timer */
			// Relevant register: TIM2->CNT
			TIM2->CNT = 0;

			/* Start the timer */
			// Relevant register: TIM2->CR1
			TIM2->CR1 |= TIM_CR1_CEN;

			/* Update the edge state */
			EdgeState = 1;
		} else {

			/* Stop the timer */
			// Relevant register: TIM2->CR1
			TIM2->CR1 &= ~TIM_CR1_CEN;

			/* Read the timer value and calculate the frequency */
			// Relevant register: TIM2->CNT
			float time = TIM2->CNT;
			Freq = 48000000/time;

			/* Reset the edge state */
			EdgeState = 0;
		}

		/* Clear the interrupt flag */
		// Relevant register: EXTI->PR
		EXTI->PR |= EXTI_PR_PR2;
	}
}
```

Delay Function Code

```
void delay(unsigned int ms) {
	/* Enable clock for TIM3 */
	// Relevant register: RCC->APB1ENR
```

```
        RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;

        /* Configure TIM3 for a delay */
        // Relevant register: TIM3->PSC and TIM3->ARR
        TIM3->PSC = (48000 - 1);
        TIM3->ARR = ms;

        /* Enable one-pulse mode */
        // Relevant register: TIM3->CR1
        TIM3->CR1 |= TIM_CR1_OPM;

        /* Start the timer */
        // Relevant register: TIM3->CR1
        TIM3->CR1 |= TIM_CR1_CEN;

        /* Wait until the timer finishes */
        // Relevant register: TIM3->SR
        while (!(TIM3->SR & TIM_SR_UIF));

        /* Clear the update interrupt flag */
        // Relevant register: TIM3->SR
        TIM3->SR &= ~TIM_SR_UIF;
}
```

OLED Write Functions Code

```
void oled_Write_Cmd( unsigned char cmd )
{
        /* Make PB6 = CS# = 1 */
        GPIOB->BSRR = GPIO_BSRR_BS_6;

        /* Make PB7 = D/C# = 0 */
        GPIOB->BSRR = GPIO_BSRR_BR_7;

        /* Make PB6 = CS# = 0 */
        GPIOB->BSRR = GPIO_BSRR_BR_6;

        /* Write commands to the display */
        oled_Write( cmd );

        /* Make PB6 = CS# = 1 */
        GPIOB->BSRR = GPIO_BSRR_BS_6;
}
```

```
void oled_Write_Data( unsigned char data )
{
	/* Make PB6 = CS# = 1 */
	GPIOB->BSRR = GPIO_BSRR_BS_6;

	/* Make PB7 = D/C# = 1 */
	GPIOB->BSRR = GPIO_BSRR_BS_7;

	/* Make PB6 = CS# = 0 */
	GPIOB->BSRR = GPIO_BSRR_BR_6;

	/* Write data to the display */
	oled_Write( data );

	/* Make PB6 = CS# = 1 */
	GPIOB->BSRR = GPIO_BSRR_BS_6;
}

void oled_Write( unsigned char Value )
{
	/* Wait until SPI1 is ready for writing (TXE = 1 in SPI1_SR) */
	// Relevant register: SPI1->SR
	while( ( SPI1->SR & SPI_SR_TXE_Msk ) == 0 );

	/* Send one 8-bit character:
	- This function also sets BIDIOE = 1 in SPI1_CR1
	*/
	HAL_SPI_Transmit( &SPI_Handle, &Value, 1, HAL_MAX_DELAY );

	/* Wait until transmission is complete (TXE = 1 in SPI1_SR) */
	// Relevant register: SPI1->SR
	while( ( SPI1->SR & SPI_SR_TXE_Msk ) == 0 );
}
```

OLED Configuration Code

```
void oled_config( void )
{
	/* Enable GPIOB clock in RCC */
	RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

	/* Configure PB3/PB5 as AF0 */
```

```
GPIOB->MODER &= ~(GPIO_MODER_MODER3 | GPIO_MODER_MODER5);
GPIOB->MODER |= (GPIO_MODER_MODER3_1 | GPIO_MODER_MODER5_1);
GPIOB->AFR[0] &= ~((0xF << (3 * 4)) | (0xF << (5 * 4)));
GPIOB->AFR[0] |= (0 << (3 * 4) | (0 << (5 * 4)));

/* Enable clock for SPI1 peripheral */
// Relevant register: RCC->APB1ENR
RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;

/* Configure SPI */
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
*/
// Relevant register: GPIOB->BSRR
GPIOB->BSRR |= GPIO_BSRR_BR_4;
delay(100);
GPIOB->BSRR |= GPIO_BSRR_BS_4;
delay(100);

/* Send initialization commands to LED Display */
for ( unsigned int i = 0; i < 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
        */
        for ( unsigned int i = 0; i < 8; i++ )
        {
                oled_Write_Cmd(0xB0 + i);
                oled_Write_Cmd(0x00); //lower half of segment
                oled_Write_Cmd(0x10); //upper half of segment
                for( unsigned int SEG = 0; SEG < 130; SEG++)
                {
                        oled_Write_Data( 0x00 );
                }
        }
}
```

OLED Refresh Code

```
void refresh_OLED( void )
{

        // DISPLAY NAMES
        /* Initialize Buffer size = at most 16 characters per PAGE + terminating '\0'
*/
        unsigned char Buffer[17];

        /* Store the contents to display in the buffer */
        snprintf( Buffer, sizeof( Buffer ), "Liam and Erich  ");
        /* 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
        */

        /* Select the initial lattice */
        oled_Write_Cmd(0xB1);
        oled_Write_Cmd(0x04); //lower half of segment
        oled_Write_Cmd(0x10); //upper half of segment

        /* Write data to the display */
        for( unsigned int i = 0; i < sizeof(Buffer); i++ )
```

```
{
        for( unsigned int j = 0; j < 7; j++ )
        {
        oled_Write_Data( Characters[Buffer[i]][j] );
        }
}


// DISPLAY INPUT SOURCE

/* If inSig is set to Function Generator, else it is set to NE555 Timer */
if(inSig == 1){
        /* Store the contents to display in the buffer */
        snprintf( Buffer, sizeof( Buffer ), "Function Gen    ");

} else {
        /* Store the contents to display in the buffer */
        snprintf( Buffer, sizeof( Buffer ), "NE555 Timer     ");

}
/* 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
*/

/* Select the initial lattice */
oled_Write_Cmd(0xB3);
oled_Write_Cmd(0x04); //lower half of segment
oled_Write_Cmd(0x10); //upper half of segment

/* Write data to the display */
for( unsigned int i = 0; i < sizeof(Buffer); i++ )
{
        for( unsigned int j = 0; j < 7; j++ )
        {
        oled_Write_Data( Characters[Buffer[i]][j] );
        }
}

// DISPLAY RESISTANCE VALUE
```

```
	/* Store the contents to display in the buffer */
	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
	*/

	/* Select the initial lattice */
	oled_Write_Cmd(0xB4);
	oled_Write_Cmd(0x04); //lower half of segment
	oled_Write_Cmd(0x10); //upper half of segment

	/* Write data to the display */
	for( unsigned int i = 0; i < sizeof(Buffer); i++ )
	{
		for( unsigned int j = 0; j < 7; j++ )
		{
			oled_Write_Data( Characters[Buffer[i]][j] );
		}
	}

	// DISPLAY FREQUENCY VALUE

	/* Store the contents to display in the buffer */
	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
	*/

	/* Select the initial lattice */
	oled_Write_Cmd(0xB5);
	oled_Write_Cmd(0x04); //lower half of segment
	oled_Write_Cmd(0x10); //upper half of segment
	for( unsigned int i = 0; i < sizeof(Buffer); i++ )
	{
		for( unsigned int j = 0; j < 7; j++ )
		{
			oled_Write_Data( Characters[Buffer[i]][j] );
		}
```

```
        }

        /* Wait for ~100 ms (for example) to get ~10 frames/sec refresh rate
        - You should use TIM3 to implement this delay (e.g., via polling)
        */
        delay(100);
}
```

Main Function Code

```
int main(int argc, char* argv[])
{
        SystemClock48MHz();
        trace_printf("This is our project\n");
        trace_printf("System clock: %u Hz\n", SystemCoreClock);
        myGPIOA_Init();                 /* Initialize I/O port A */
        myTIM2_Init();                  /* Initialize timer TIM2 */
        myADC_Init();                   /* Initialize ADC */
        myDAC_Init();                   /* Initialize DAC */
        myEXTI_Init();                  /* Initialize EXTI */
        myGPIOB_Init();                 /* Initialize I/O port B */
        oled_config();                  /* Configure OLED */
        while (1)
        {
                refresh_OLED();
                ADC_Converter();
                DAC_Converter();
        }
        return 0;
}
```

References

[1] B. Sirna and D. Rakhmatov, “ECE 355: Microprocessor-Based Systems Laboratory Manual,” ECE 355 Microprocessor Based Systems Lab Website, https://ece.engr.uvic.ca/~ece355/lab/ECE355-LabManual-2023.pdf (accessed Dec. 4, 2023).

[2] “4N35 Optocoupler Data Sheet,” ECE 355 Microprocessor Based Systems Lab Website, https://ece.engr.uvic.ca/~ece355/lab/supplement/555timer.pdf (accessed Dec. 4, 2023).

[3] “555 Timer Data Sheet,” ECE 355 Microprocessor Based Systems Lab Website, https://ece.engr.uvic.ca/~ece355/lab/supplement/555timer.pdf (accessed Dec. 4, 2023).

[4] D. N. Rakhmatov, “Interface Examples,” ECE 355 Microprocessor Based Systems, https://ece.engr.uvic.ca/~daler/courses/ece355/interfacex.pdf (accessed Dec. 4, 2023).