ECE 355 - Project Report

By: Will Calder V00975012, Manraj Minhas V00998047

Table of Contents

Preliminary Content **4**
Introduction: **5**
1.1 Project Background 5
1.2 Problem Description 6
1.3 Scope of Work 6
System Specifications: **7**
2.1 System Architecture 7
2.2 Hardware Specifications 8
2.2.1 Pin Assignments: 8
2.2.2 External Circuitry 10
2.3 Functional Specifications 11
2.3.1 Frequency Measurement Logic 11
2.3.2 Resistance Measurement and feedback 11
2.3.3 Display Interface 12
Design Approach: **12**
3.1 System Architecture Overview 12
3.2 Hardware Configuration & Initialization 12
3.2.1 System Clock (RCC) 12
3.2.2 General Purpose Timers 13
3.2.3 Analog Peripherals 14
3.3 Signal Measurement Strategy 14
3.3.1 Input Capture Logic 14
3.4 Control System Design 15
3.5 User Interface Design 16
3.5.1 Display Driver (SPI) 16
3.5.2 Practical Application of Displaying Values (OLED) 16
3.5.2 User Input Button 18
3.6 Wiring Design 19
3.6.1 Function Generator Wiring 20
3.6.2 ADC, DAC, and POT Wiring 20
3.6.3 OLED Wiring 21
Experimental Analysis: **22**
4.1 Testing Methodology 22
4.2 Analysis of Frequency Measurement 22
4.2.1 Verification of Input Capture Logic 22
4.2.2 Interrupt Latency Effects 23
4.3 Analysis of Resistance Measurement and Feedback Loop 23
4.3.1 ADC Linearity and Stability 24
4.3.2 Closed-Loop Control Dynamics 24

4.4 System Limitations 24
4.4.1 Input Voltage Range and ADC Resolution 24
4.4.2 Resistance Measurement Range 25
4.4.3 Frequency Measurement Limits 25
4.4.4 Control Loop Non-Linearity 25
Project Discussion **26**
5.1 Results & Challenges 26
5.2 Future Recommendations & Conclusion 26
Appendices **27**
6.1 Code Snippets 27
6.1.1 main.c 27
6.1.2 SetClock.c 35
6.1.3 SetClock.h 36
6.1.4 display.c 36
6.1.5 display.h 49
6.2 Extra Wiring Diagrams 50
References **51**

Preliminary Content

Figures:	
1	High-level system overview illustrating the signal paths between the STM32F0 microcontroller, the NE555 timer circuit, and the user interface peripherals.
2	System Block Diagram illustrating the data flow between the microcontroller, the NE555 timer circuit, and the SSD1306 display [1].
3	Schematic of the external NE555 timer and 4N35 optocoupler interface.
4	Circuit Design top down view.
5	Circuit design side view.
6	Circuit diagram for NE555 and 4N35 Optocoupler Connections [1].
7	NE555 timer pin board diagram [6].
8	4N35 Optocoupler pin board diagram [7].
9	J5 rail located on the Project Base Board User Guide (PBMCUSLK) [5].
10	J10 rail located on the Project Base Board User Guide (PBMCUSLK) [5].
Tables:	
1	Table containing the required pin use for the STM32F0 [1].
2	Pin Description Table.

Introduction:

1.1 Project Background

In the field of embedded systems engineering, the ability to interface digital microcontrollers with analog physical components is a fundamental skill. This project, "PWM Signal Generation and Monitoring System," focuses on the design, implementation, and analysis of a mixed-signal embedded system using the STM32F0 Discovery board.

The core of the system is the STM32F051R8T6 microcontroller, an ARM Cortex-M0 based device running at 48 MHz. The project integrates multiple hardware peripherals, including an NE555 timer, a 4N35 optocoupler, a function generator, and an SSD1306 OLED display, to create a closed-loop frequency control and monitoring device. By leveraging the microcontroller's internal Analog-to-Digital Converter (ADC), Digital-to-Analog Converter (DAC), and General Purpose Timers, the system demonstrates real-time signal processing and hardware actuation.

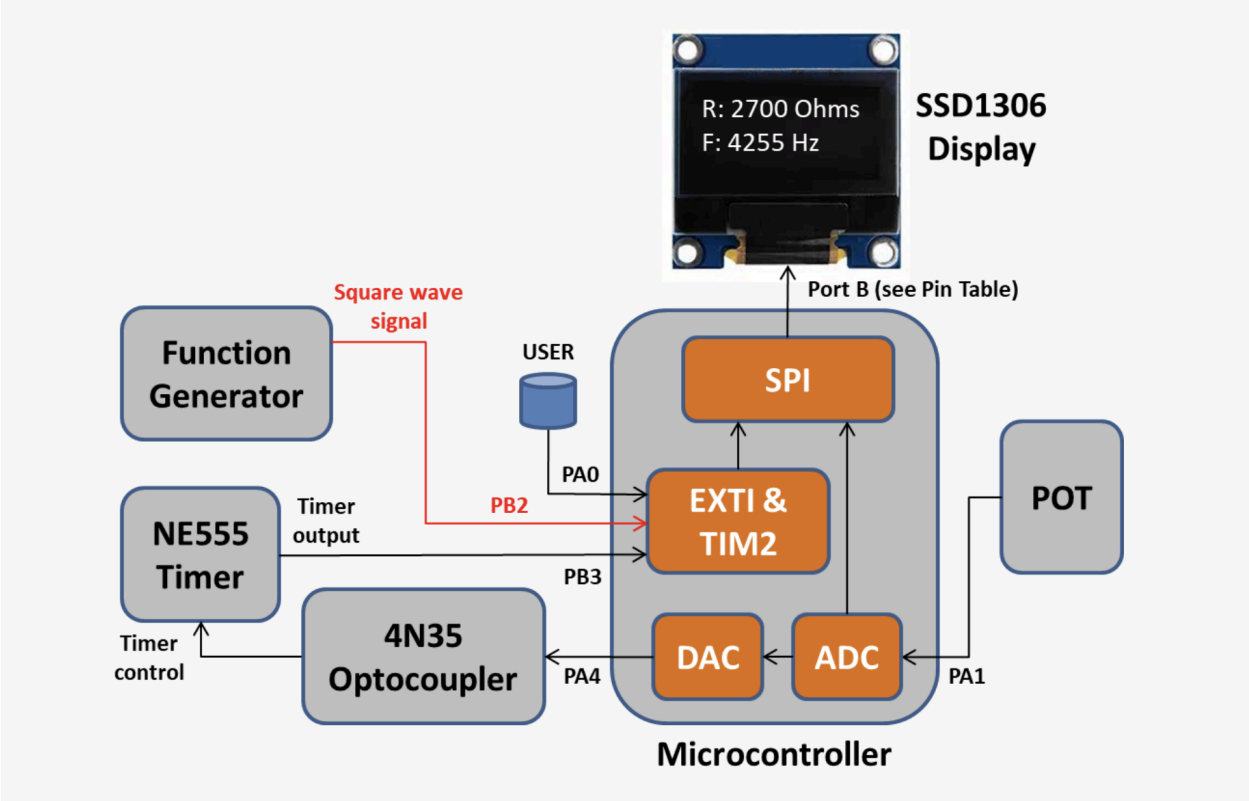

Figure 1: High-level system overview illustrating the signal paths between the STM32F0 microcontroller, the NE555 timer circuit, and the user interface peripherals [1].

1.2 Problem Description

The primary engineering challenge addressed in this project is the precise measurement and manipulation of a Pulse Width Modulated (PWM) signal using a 32-bit microcontroller. Specifically, the system must address two distinct technical problems:

1. Frequency Measurement: The system must accurately capture and calculate the frequency of incoming square-wave signals from two independent sources (a reference Function Generator and a variable NE555 timer) using hardware interrupts and timer input capture logic.

2. Analog Feedback Control: The system requires a mechanism to electronically control the output frequency of the NE555 timer. Since the NE555 is an analog device controlled by voltage, the digital microcontroller must interface with it through a Digital-to-Analog Converter (DAC). A key constraint is the use of a 4N35 optocoupler to isolate and modulate the control voltage pin of the NE555 based on user input from a potentiometer.

1.3 Scope of Work

The scope of this project encompasses both hardware interfacing and embedded firmware development. The specific tasks undertaken include:

- Hardware Interface Design: Establishing physical connections between the STM32F0 GPIO pins and external components, ensuring correct voltage levels and signal integrity for the SPI bus (OLED), analog inputs (ADC), and analog outputs (DAC).

- Firmware Development: Writing C-language firmware using the Eclipse IDE to configure the STM32F0's internal peripherals. This involves direct register manipulation (e.g., RCC, GPIO, TIM, ADC, DAC, SPI) to achieve optimized performance without relying on high-level abstraction libraries.

- Interrupt Management: Implementing a Nested Vectored Interrupt Controller (NVIC) configuration to handle real-time events, such as the user button press (EXTI0) and signal edge detection (EXTI2/EXTI3), ensuring the system remains responsive.

- Data Visualization: Implementing display to visualize real-time data, providing the user with immediate feedback on the measured frequency and the current potentiometer resistance.

System Specifications:

2.1 System Architecture

The system is architected around the STM32F0 Discovery board, utilizing the ARM Cortex-M0 core operating at a system clock frequency of 48 MHz. The architecture relies on a polling-based main loop for analog signal processing and an interrupt-driven mechanism for critical frequency measurement tasks.

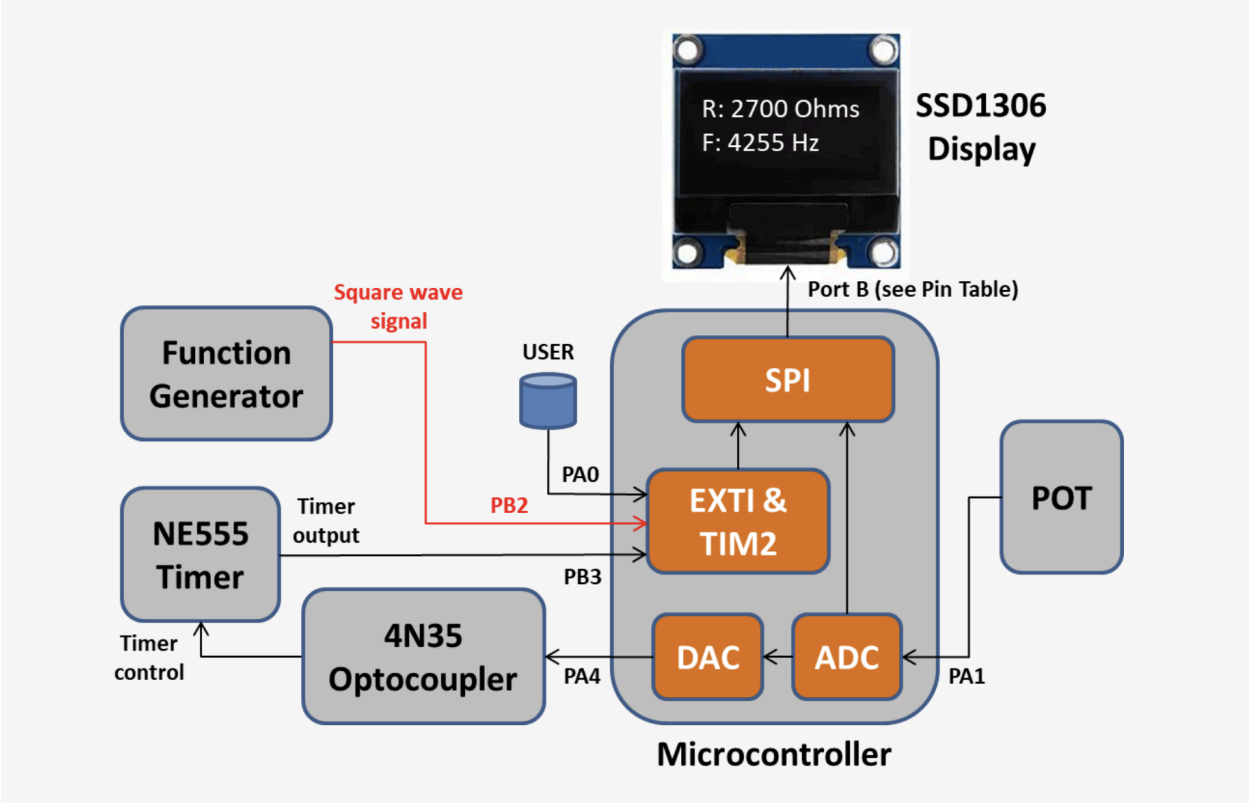

Figure 2: System Block Diagram illustrating the data flow between the microcontroller, the NE555 timer circuit, and the SSD1306 display [1].

2.2 Hardware Specifications

The microcontroller interfaces with the external environment through specific General Purpose Input/Output (GPIO) ports. The configuration is defined in the firmware initialization routines (myGPIOA_Init and myGPIOB_Init).

```
void myGPIOB_Init(){
   /* Enable clock for GPIOB peripheral */
   // Relevant register: RCC->AHBENR
   RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

   /* Configure PB2 as input */
   // Relevant register: GPIOB->MODER
   GPIOB->MODER &= ~(GPIO_MODER_MODER2);
   /* Ensure no pull-up/pull-down for PB2 */
   // Relevant register: GPIOB->PUPDR
   GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR2);

   //555 Timer configuration
   GPIOB->MODER &= ~(GPIO_MODER_MODER3);
   GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3);

}

void myGPIOA_Init(){
   // Enabled clock for GPIOA
   RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

   // Enabling GPIOA0 input to USER PUSH BUTTON.
   GPIOA->MODER &= ~(GPIO_MODER_MODER0);
   GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);

   //Enabling GPIOA1 input to ADC
   GPIOA->MODER |= GPIO_MODER_MODER1;
   GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);

   //Configure output for PA4
   GPIOA->MODER |= GPIO_MODER_MODER4;
   GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
}
```

2.2.1 Pin Assignments:

The following tables detail the specific pin mappings used in the final implementation:

STM32F0	SIGNAL	DIRECTION
PA0	USER PUSH BUTTON	INPUT
PA1	ADC	INPUT (Analog)
PA4	DAC	OUTPUT (Analog)
PB2	FUNCTION GENERATOR	INPUT
PB3	555 TIMER	INPUT
PB8	CS# (Display “Chip Select”)	OUTPUT
PB9	D/C# (Display “Data/Command”)	OUTPUT
PB11	RES# (Display “Reset”)	OUTPUT
PB13	SCLK (Display D0: “Serial Clock” = SPI SCK)	OUTPUT (AF0)
PB15	SDIN (Display D1: “Serial Data” = SPI MOSI)	OUTPUT (AF0)
PC8	BLUE LED	OUTPUT
PC9	GREEN LED	OUTPUT

Table 1: Table containing the required pin use for the STM32F0 [1].

STM32F0 Pin	Signal Name	Peripheral Function	Description
PA0	USER Button	GPIO Input / EXTI0	Triggers EXTI0_1_IRQHandler to toggle measurement source.
PA1	POT Input	ADC1_IN1	Analog input for potentiometer voltage reading.
PA4	DAC Output	DAC_OUT1	Analog output driving the 4N35 Optocoupler LED.
PB2	Func. Gen.	GPIO Input / EXTI2	Input capture for Function Generator signal (rising edge).
PB3	555 Timer	GPIO Input / EXTI3	Input capture for NE555 signal (rising edge).
PB8	CS#	GPIO Output	OLED Chip Select (Active Low).
PB9	D/C#	GPIO Output	OLED Data/Command selection.

PB11	RES#	GPIO Output	OLED Reset signal.
PB13	SCLK	SPI2_SCK (AF0)	SPI Clock for OLED display.
PB15	MOSI	SPI2_MOSI (AF0)	SPI Data for OLED display.

Table 2: Pin Description Table.

2.2.2 External Circuitry

The external control circuit utilizes a 4N35 optocoupler to isolate the digital logic from the analog timing circuit. The DAC output (PA4) drives the anode of the optocoupler's internal LED. The phototransistor side of the optocoupler modulates the voltage at the Control Pin (Pin 5) of the NE555 timer, thereby altering its oscillation frequency.

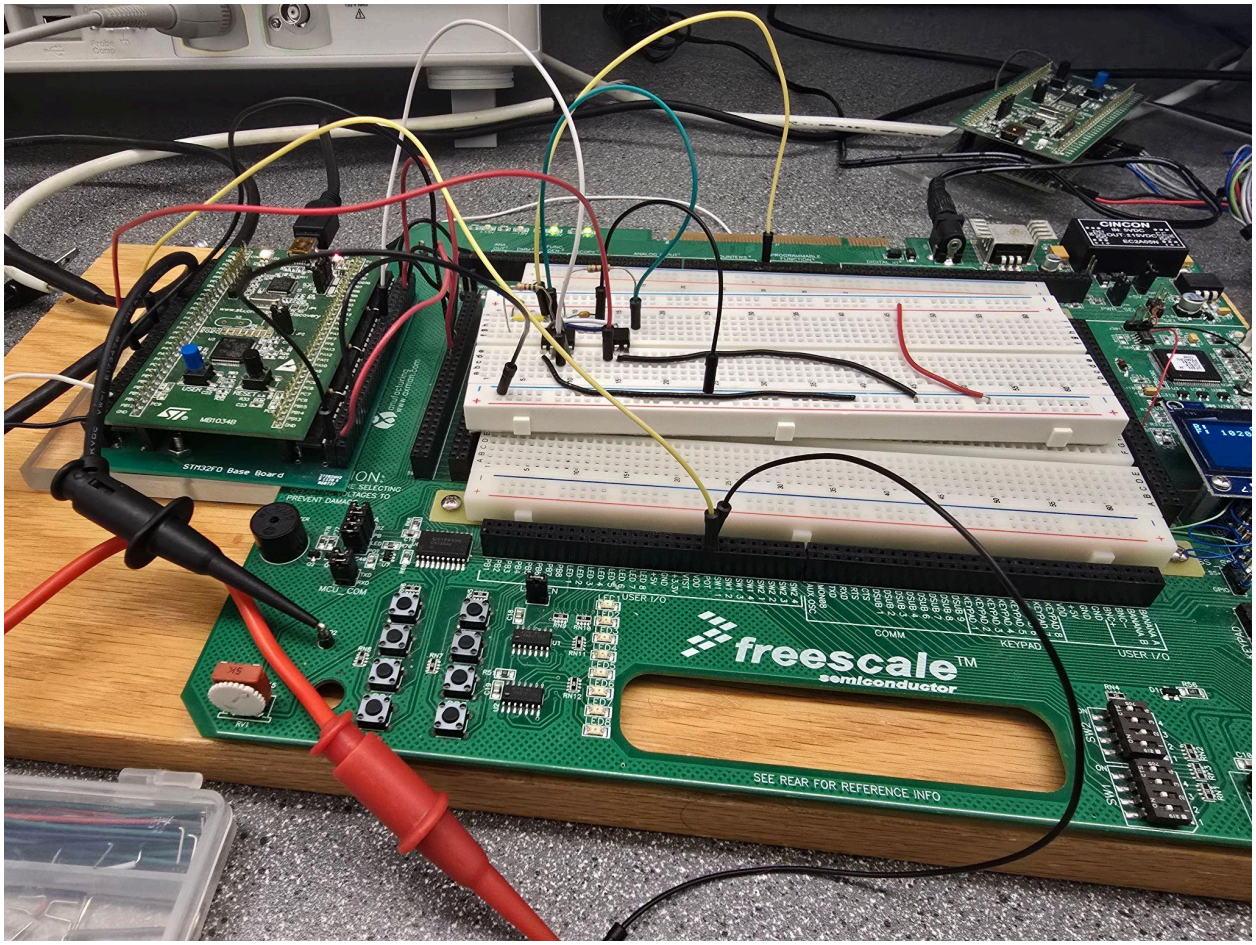

Figure 3: Schematic of the external NE555 timer and 4N35 optocoupler interface.

2.3 Functional Specifications

2.3.1 Frequency Measurement Logic

Frequency calculation is performed inside the EXTI2_3_IRQHandler interrupt service routine. The system uses the 32-bit General Purpose Timer 2 (TIM2) running at the full system core clock speed (f_SYSCLK = 48 MHz) with no prescaler (PSC = 0).

Upon detection of a rising edge on the selected input (PB2 or PB3):

1. The current timer count (N_counts) is captured from TIM2->CNT.
2. The timer is reset to zero.
3. The signal frequency (f_sig) is calculated using the formula:

$$f_{sig} = \frac{f_{sysclk}}{N_{counts}} = \frac{48\,000\,000}{TIM2_{cnt}}$$

```
if(counter > 0){
                    period_us = counter/SystemCoreClock;
                    frequency_hz = SystemCoreClock/counter;
                    Freq = round(frequency_hz);
                    refresh_OLED();
//                     trace_printf("Period: %f Frequency: %f Hz\n", period_us,
frequency_hz);
                    TIM2->CNT = 0;
                }
```

2.3.2 Resistance Measurement and feedback

The system implements a continuous feedback loop in the main() function to control the NE555 frequency.

1. ADC Conversion: The 12-bit ADC continuously samples the voltage at PA1. The sampling time is configured to the maximum duration (239.5 ADC clock cycles) to ensure stability.

2. Resistance Calculation: The firmware maps the 12-bit ADC value (0 to 4095) to a resistance range of 0 to 5000 Omega ohms.

$$R_{pot} = \frac{ADC_{value}(5000)}{4095}$$

3. DAC Output: The raw 12-bit ADC value is written directly to the DAC data register (DAC->DHR12R1), adjusting the optocoupler drive current proportionally to the potentiometer position.

2.3.3 Display Interface

The system utilizes the SSD1306 OLED display communicating via SPI2. The display driver implements a character buffer to render the calculated resistance (R) and frequency (F) in real-time. To prevent display flickering, the refresh rate is throttled using a software delay of approximately 100ms.

Design Approach:

3.1 System Architecture Overview

The system firmware is designed using a continuous cyclic execution architecture. This approach ensures that signal monitoring tasks, such as capturing data for frequency measurement, are integrated directly into the primary workflow, while peripheral operations, such as updating the display and adjusting the DAC output, are managed sequentially within the main application loop.

The software consists of three primary modules:

1. Initialization Layer: Configures the system clock, GPIOs, Timers, ADC, DAC, and SPI peripherals.
2. Interrupt Service Routines (ISRs): Handles real-time inputs from the user button and the external frequency signals.
3. Application Loop: Performs continuous analog data acquisition, control feedback, and user interface updates.

3.2 Hardware Configuration & Initialization

3.2.1 System Clock (RCC)

The system clock is configured to run at 48 MHz using the Phase Locked Loop (PLL) driven by the internal 8 MHz HSI oscillator. The initialization sequence in SetClock.c performs the following critical steps:

1. Disables the PLL (RCC->CR &= ~RCC_CR_PLLON).
2. Configures the PLL multiplication factor to x12 to achieve 48 MHz from the 4 MHz (HSI/2) input source (RCC->CFGR = 0x00280000).
3. Enables the PLL and waits for the RCC_CR_PLLRDY flag.
4. Switches the system clock source to the PLL output.

5. Calls SystemCoreClockUpdate() to update the global clock variable used for timing calculations.

3.2.2 General Purpose Timers

TIM2 (Measurement Timebase): TIM2 is configured as the primary 32-bit up-counter for frequency measurement.

- Prescaler: The timer counts directly at the 48 MHz system clock frequency.
- Auto-Reload (ARR): This maximum value prevents frequent overflows, allowing for the measurement of very low frequencies (long periods).
- Interrupts: Enabled for Update Events (TIM_DIER_UIE) to handle counter overflows if necessary.

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

   /* 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); //Sets up the buffer auto-reload, count up, stop
on overflow. Interrupts on overflow. (TIM_CR1_ARPE | TIM_CR1_URS | TIM_CR1_OPM)

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

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

   /* Update timer registers */
   // Relevant register: TIM2->EGR
   TIM2->EGR = TIM_EGR_UG; //added

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

   /* Enable TIM2 interrupts in NVIC */
   // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
   NVIC_EnableIRQ(TIM2_IRQn); //added
```

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

3.2.3 Analog Peripherals

ADC (Analog-to-Digital Converter): The ADC is initialized in myADC_Init to sample the potentiometer voltage at PA1.

- Mode: Continuous Conversion (ADC_CFGR1_CONT).
- Resolution: 12-bit.
- Sampling Time: Set to the maximum of 239.5 cycles (ADC_SMPR_SMP_0 | ADC_SMPR_SMP_1 | ADC_SMPR_SMP_2) to ensure high impedance signal stability.

DAC (Digital-to-Analog Converter): The DAC is initialized on Channel 1 (PA4) to drive the external optocoupler. The DAC_CR_EN1 bit is set to enable the output buffer, allowing it to drive the current required by the 4N35 LED.

3.3 Signal Measurement Strategy

The core functionality of the system depends on accurately measuring the frequency of incoming square waves. This is achieved using the External Interrupt (EXTI) controller linked to the signal input pins.

3.3.1 Input Capture Logic

The frequency measurement is event-driven, handled by the EXTI2_3_IRQHandler. The logic utilizes a global state variable edge_detected to distinguish between the start and end of a signal period.

1. First Rising Edge:
    - The ISR detects the interrupt flag (EXTI_PR_PR2 or PR3).
    - If edge_detected == 0, the system resets the timer count (TIM2->CNT = 0).
    - The timer is started (TIM2->CR1 |= TIM_CR1_CEN).
    - The flag is set to edge_detected = 1.
2. Second Rising Edge:
    - On the next interrupt, if edge_detected == 1, the ISR captures the current value of TIM2->CNT into a local variable counter.
    - The frequency is immediately calculated:

$$Frenquency = \frac{SystemCoreClock}{Counter}$$

    - The result is stored in the global variable Freq.
    - The edge_detected flag is reset to 0 to prepare for the next measurement.

```
if ((EXTI->PR & EXTI_PR_PR2) != 0){
            if(edge_detected == 0){
                TIM2->CNT = 0x00;
                TIM2->CR1 |= TIM_CR1_CEN;
                edge_detected = 1;
            }else{
                if(counter > 0){
                    period_us = counter/SystemCoreClock;
                    frequency_hz = SystemCoreClock/counter;
                    Freq = round(frequency_hz);
                    refresh_OLED();
//                    trace_printf("Period: %f Frequency: %f Hz\n", period_us,
frequency_hz);
                    TIM2->CNT = 0;
                }
                edge_detected = 0;
            }
            EXTI->PR = EXTI_PR_PR2; // clearing interrupt
        }
```

3.4 Control System Design

The system implements a firmware-based feedback loop in main() to control the NE555 frequency.

1. ADC Sampling: The main loop triggers an ADC conversion (ADC1->CR |= 0x04) and waits for the End of Conversion flag (ADC_ISR_EOC).
2. Resistance Calculation: The raw 12-bit ADC value is converted to a theoretical resistance value (0-5000 Ohms) for display: Res = (ADCValue * 5000) / 0xFFF;
3. Feedback Actuation: The raw ADC value is written directly to the DAC (DAC->DHR12R1 = ADCValue). This changes the voltage at PA4, which drives the 4N35 optocoupler LED. A higher DAC voltage increases the LED brightness, causing the phototransistor to conduct more, which lowers the control voltage on the 555 timer and modifies its frequency.

```
TIM2->CNT = 0x00;
TIM2->CR1 |= TIM_CR1_CEN;
DAC->DHR12R1 = ADCVal; // Write directly to DAC to drive Optocoupler.
ADC1->CR |= ((uint32_t)0x00000004);
while(!(ADC1->ISR & ADC_ISR_EOC)); //((uint32_t)0x00000004))
ADCVal = (uint16_t) ADC1->DR; // Adds the converted ADC value to our variable.
Res = (ADCVal*5000)/0xFFF; // Calculating resistance
Freq = ADCVal; // Setting freq value for ADC
TIM2->CNT = 0; // Prevents overflow from occurring when focused on 555 timer.
```

3.5 User Interface Design

The user interface is managed through the OLED display and the User Input Button.

3.5.1 Display Driver (SPI)

The SSD1306 OLED is driven via SPI2 in Master Mode (display.c).

Configuration: The SPI prescaler is set to 256 (SPI_BAUDRATEPRESCALER_256) to ensure reliable communication.

Initialization: A specific sequence of commands (Charge Pump enable, Addressing Mode, etc.) is sent via oled_init_cmds to configure the panel.

Rendering: The refresh_OLED function formats the global Res and Freq variables into strings (e.g., "R: 2500 Ohm") and writes character bitmaps to the display's GDDRAM.

3.5.2 Practical Application of Displaying Values (OLED)

The following code was used to display the values on the SSD1306 OLED display.

```
void refresh_OLED( void ){
   for (unsigned int page = 0; page < 2; page++){
        // Set cursor to the start of the page (page, column 0)
        oled_SetCursor(page, 3);

        // Write 0x00 to all 128 segments in that page
        // added plus 2 as there were 2 pixels that were writing even with the
128 clear, this fixed that issue
        for (unsigned int seg = 0; seg < 130; seg++) {
            oled_Write_Data(0x00);
        }
    }

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

   snprintf((char *)Buffer, sizeof(Buffer), "R: %5d Ohm", Res);

   oled_SetCursor(0,2); // Initialize location to write the Buffer characters

   for(int i = 0; Buffer[i] != '\0'; i++){
       for(int j = 0; j < 8; j++){
          oled_Write_Data(Characters[Buffer[i]][j]);
       }
   }

   snprintf((char *)Buffer, sizeof(Buffer), "F: %5d Hz", Freq);

   oled_SetCursor(1, 2); // Initialize location to write the Buffer characters

   for(int i = 0; Buffer[i] != '\0'; i++){
```

```
        for(int j = 0; j < 8; j++){
            oled_Write_Data(Characters[Buffer[i]][j]);
        }
    }

    // Delay to prevent flickering (approx 100ms)
    delay_tim3_ms(100);
}
```

The top for loop is added to refresh the top two rows of the OLED display to ensure no overwrite issues occur when new values are written to the display.

```
for (unsigned int page = 0; page < 2; page++){
    // Set cursor to the start of the page (page, column 0)
    oled_SetCursor(page, 3);

    // Write 0x00 to all 128 segments in that page
    // added plus 2 as there were 2 pixels that were writing even with the 128
clear, this fixed that issue
    for (unsigned int seg = 0; seg < 130; seg++) {
        oled_Write_Data(0x00);
    }
}
```

The next image contains the code that writes the values to the oled. First we set the position of the cursor using the function we created called “oled_SetCursor(page, column)”, this function is described next. Once the location has been set we use a for loop to write the characters to the OLED display. This is done twice ensuring both resistance and frequency are displayed correctly on the display.

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

snprintf((char *)Buffer, sizeof(Buffer), "R: %5d Ohm", Res);

oled_SetCursor(0,2); // Initialize location to write the Buffer characters

for(int i = 0; Buffer[i] != '\0'; i++){
   for(int j = 0; j < 8; j++){
      oled_Write_Data(Characters[Buffer[i]][j]);
   }
}

snprintf((char *)Buffer, sizeof(Buffer), "F: %5d Hz", Freq);

oled_SetCursor(1, 2); // Initialize location to write the Buffer characters

for(int i = 0; Buffer[i] != '\0'; i++){
   for(int j = 0; j < 8; j++){
      oled_Write_Data(Characters[Buffer[i]][j]);
```

```
    }
}

// Delay to prevent flickering (approx 100ms)
delay_tim3_ms(100);
```

3.5.2 User Input Button

The User Button (PA0) triggers the EXTI0_1_IRQHandler. This ISR toggles the global user_button_state variable (0 or 1). Based on this state, the EXTI2_3_IRQHandler decides whether to calculate frequency from the Function Generator input (PB2) or the 555 Timer input (PB3), effectively switching the measurement source.

```
void EXTI0_1_IRQHandler(){
   EXTI->IMR &= ~EXTI_IMR_MR0; // Set the interrupt flag.
       if ((EXTI->PR & EXTI_PR_PR0) != 0) { // Check that the interrupt has
been triggered
           if (user_button_state == 0) { // Checks which state the device is
currently in (In this case it is currently record the function generator).
               trace_printf("555 Timer Enabled\n");
               user_button_state = 1; // Ensure that the device is changed so
that it starts recording the 555 timer input.

               TIM2->CR1 |= TIM_CR1_CEN; // Stop timer
               EXTI->IMR |= EXTI_IMR_MR2; // Reseting timer flag.
           } else {
               trace_printf("Function Generator Enabled\n");
               user_button_state = 0;
               TIM2->CR1 |= TIM_CR1_CEN; // Stop timer
               EXTI->IMR |= EXTI_IMR_MR2; // Reseting timer flag.
           }
           EXTI->PR = EXTI_PR_PR0; // Clearing interrupt
       }
   EXTI->IMR |= EXTI_IMR_MR0; // Unmasking interrupt flag.
}
```

3.6 Wiring Design

To ensure the system was correctly wired we followed both the diagram and pin table provided within the lab manual [1]. Below is a photo showing the final wiring configuration of the project.

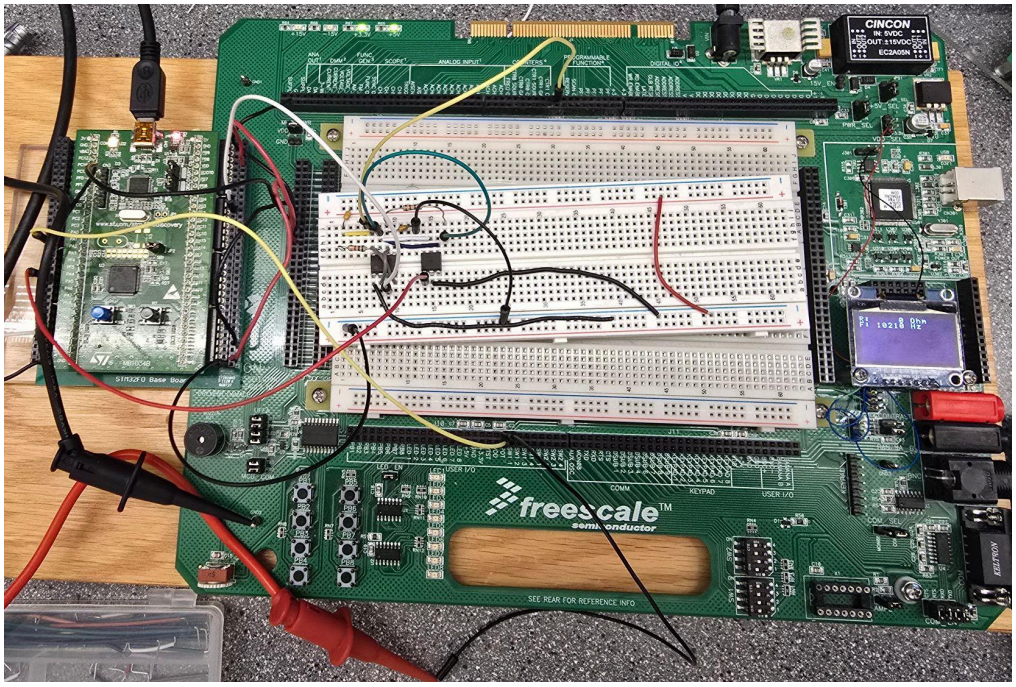

Figure 4: Circuit Design top down view.

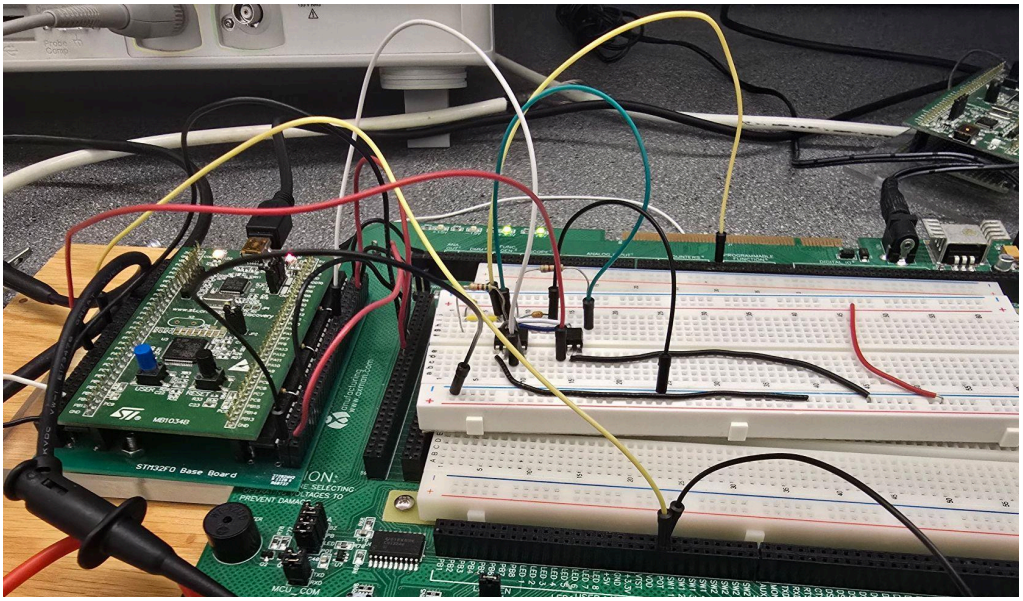

Figure 5: Circuit design side view.

3.6.1 Function Generator Wiring

For creating the alternative input signal to the ADC DAC we used a function generator. This was set up by connecting one cable to any of the ground (GND) pins on the STM32F0 to the ground connector on the function generator's cable. The other connector on the function generator is connected to the STM32F0 pin PB2.

3.6.2 ADC, DAC, and POT Wiring

The first diagram below represents the connections required to accurately wire the NE555 timer to the 4N35 Optocoupler and then into the STM32F0 microcontroller. Below this figure is two diagrams, the first diagram shows the pin layout for the NE555 timer [6]. The second diagram below shows the pin layout for the 4N35 Optocoupler [7].

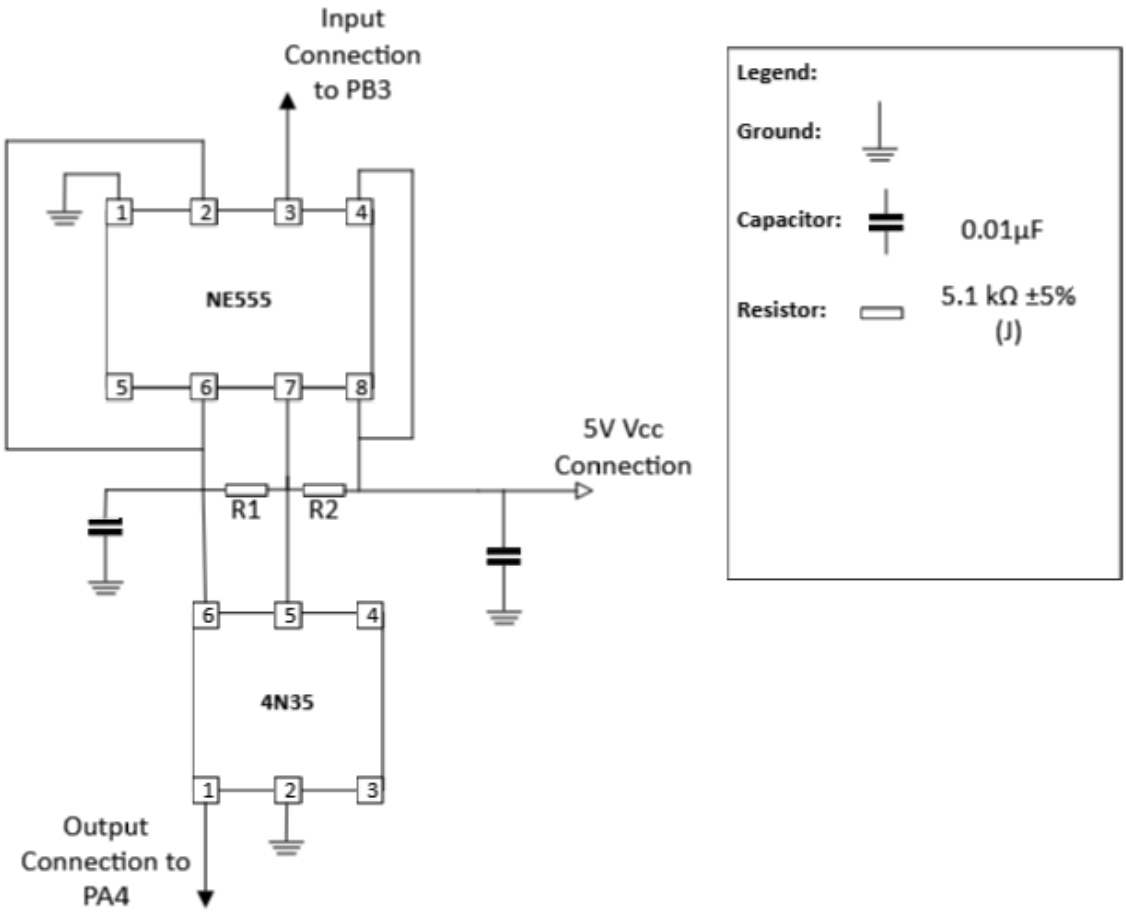

Figure 6: Circuit diagram for NE555 and 4N35 Optocoupler Connections [1].

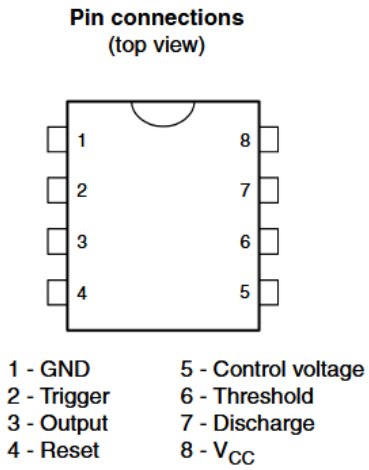

Figure 7: NE555 timer pin board diagram [6].

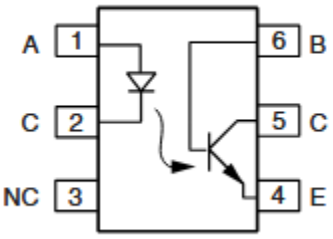

Figure 8: 4N35 Optocoupler pin board diagram [7].

The last portion required for accurately wiring this section of the system is by connecting to the POT pin on the J10 connector rail. More information about where on this rail you can find the pin refer to section 7.2 [1]. The POT pin will be directly connected into the STM32F0 pin PA1 [5].

3.6.3 OLED Wiring

Correctly wiring the OLED display to STM32F0 microcontroller you can follow the following pin guide [8].

- Connection for CS# (Display “Chip Select”) :
  - STM32F0 pin PB8 -> Pin 25 on J5 rail.
- Connection for D/C# (Display “Data/Command”):
  - STM32F0 pin PB9 -> Pin 23 on J5 rail.
- Connection for RES# (Display “Reset”) :
  - STM32F0 pin PB11 -> Pin 19 on J5 rail.
- Connection for SCLK (Display D0: “Serial Clock” = SPI SCK):
  - STM32F0 PB13 -> Pin 21 on J5 rail.
- Connection for SDIN (Display D1: “Serial Data” = SPI MOSI):
  - STM32F0 PB15 -> Pin 17 on J5 rail.

Experimental Analysis:

4.1 Testing Methodology

To validate the system specifications, a series of experiments were conducted using the standard lab equipment provided. The systems signals were verified using the following equipment:

- Digital Multimeter (Volt Meter): Used to visualize the PWM output from the NE555 and the square wave from the function generator, and to measure the actual resistance of the potentiometer for validation against the displayed value.

- Function Generator: Used to provide a calibrated reference square wave (0-3.3V) to verify the input capture accuracy.

4.2 Analysis of Frequency Measurement

The frequency measurement logic, implemented in EXTI2_3_IRQHandler, relies on capturing the TIM2 counter value between two consecutive rising edges. The accuracy of this system is fundamentally determined by the 48 MHz system clock and the interrupt latency.

```
TIM2->CR1 = ((uint16_t) 0x008c); //Sets up the buffer auto-reload, count up, stop on
overflow. Interrupts on overflow. (TIM_CR1_ARPE | TIM_CR1_URS | TIM_CR1_OPM)

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

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

4.2.1 Verification of Input Capture Logic

Testing with the Function Generator confirmed that the EXTI Interrupt correctly triggered on rising edge cases. The state machine implemented in the ISR (toggling the edge_detected flag) successfully isolated single periods of the input signal.

- Low frequency response: At a low Hz input, the timer count TIM2->CNT reached values approximately equal to 48 MHz/ 100Hz). The 32-bit width of TIM2 was critical here; a 16-bit timer would have overflowed at roughly 1.3 ms (732 Hz), making low-frequency measurement impossible without complex overflow counting logic.

- High frequency response: As the input frequency increased, the timer count decreased. While the resolution decreased, it remained sufficient to provide stable readings. The system successfully distinguished between the Function Generator signal (PB2) and the internal 555 Timer signal (PB3) when the User Button was pressed, validating the EXTI0_1_IRQHandler routing logic.

```
if(edge_detected == 0){
                TIM2->CNT = 0x00;
                TIM2->CR1 |= TIM_CR1_CEN;
                edge_detected = 1;
            }else{
                if(counter > 0){
                    period_us = counter/SystemCoreClock;
                    frequency_hz = SystemCoreClock/counter;
                    Freq = round(frequency_hz);
                    refresh_OLED();
//                    trace_printf("Period: %f Frequency: %f Hz\n", period_us,
frequency_hz);
                    TIM2->CNT = 0;
                }
                edge_detected = 0;
            }
```

4.2.2 Interrupt Latency Effects

At higher frequencies , we observed that the displayed frequency began to deviate slightly from the input. This is analyzed as a limitation of the interrupt-driven design. The CPU requires a fixed number of cycles to stack registers and enter the ISR. If the signal period approaches this overhead time, the CPU spends a significant portion of its time entering/exiting interrupts, potentially delaying the execution of the capture code and introducing jitter into the frequency_hz calculation.

4.3 Analysis of Resistance Measurement and Feedback Loop

The analog control subsystem was analyzed by monitoring the linearity of the ADC-to-resistance conversion and the responsiveness of the DAC output.

```
ADCValue = (uint16_t) ADC1->DR;
       Res = (ADCValue*5000)/0xFFF;
```

4.3.1 ADC Linearity and Stability

The firmware calculates resistance using Res = (ADCValue * 5000) / 0xFFF. This linear mapping assumes that the potentiometer acts as a perfect voltage divider. Testing showed a monotonic increase in displayed resistance from 0 ohms to 5000 ohms as the potentiometer was rotated.

- Stability Observations: The decision to use the maximum sampling time (239.5 cycles) in myADC_Init resulted in relatively stable readings. Even with the inherent noise of a breadboard environment, the displayed resistance value remained steady when the potentiometer was static. This indicates that the high input impedance of the ADC was correctly matched to the source impedance of the potentiometer.

```
/* ADC Stability Configuration */
// Set the ADC sample time to the maximum (239.5 cycles) for accuracy
   ADC1->SMPR |= ADC_SMPR_SMP_0 | ADC_SMPR_SMP_1 | ADC_SMPR_SMP_2; // 111
```

4.3.2 Closed-Loop Control Dynamics

The core feature of the project, controlling the NE555 frequency via software, was analyzed by measuring the voltage at the DAC output (PA4) relative to the displayed resistance.

- Control Relationship: As the calculated resistance Res increased, the firmware wrote higher values to DAC->DHR12R1. We verified with the voltmeter that the voltage at PA4 increased linearly from 0V (at 0 ohms) to approximately 3.0V (at 5000 ohms).

- Actuation Response: This rising DAC voltage forward-biased the LED. We observed that the NE555 frequency shifted inversely to the control voltage at pin 5. This confirms the successful operation of the digital-to-analog feedback path. The system effectively acted as a digitally-controlled variable resistor.

```
// analog feedback control loop
      ADCValue = (uint16_t) ADC1->DR;//Read Potentiometer Voltage
      DAC->DHR12R1 = ADCValue;//Write directly to DAC to drive Optocoupler
```

4.4 System Limitations

The system operates within defined electrical and logical boundaries. Exceeding these limits typically results in signal clipping, measurement aliasing, or system instability.

4.4.1 Input Voltage Range and ADC Resolution

The system is constrained by the characteristics of the STM32F0 GPIO pins and the 12-bit ADC architecture.

- Voltage range: The analog input (PA1) and output (PA4) are strictly limited to the 0V to 3.3V rail.

- Resolution range: The 12-bit ADC provides a raw value range of 0 to 4095 (2^12-1). Over the 3.3V range, this yields a voltage resolution of approximately 0.806 mV per bit.

4.4.2 Resistance Measurement Range

The firmware mathematically maps the ADC reading to a resistance value.

- Range: Range: The display is hard-coded to show 0 ohms to 5000 ohms. This assumes the potentiometer is exactly 5k ohms. If a 10k ohms potentiometer were used, the system would still display a maximum of 5000 ohms due to the calculation Res = (ADCValue * 5000) / 0xFFF.

- Precision: Precision: Due to the 12-bit resolution, the smallest detectable change in resistance is 5000 ohms / 4095 ohms which is approximately 2.11 ohms. The system cannot display resistance changes smaller than this step size.

4.4.3 Frequency Measurement Limits

The frequency measurement range is defined by the timer width (low-end limit) and interrupt latency (high-end limit).

- Lower Limit: With a 32-bit timer running at 48 MHz, the counter can run for approximately 89 seconds before overflowing. This allows the system to theoretically measure frequencies as low as 0.011 Hz.

- Upper Limit: The upper limit is determined by the CPU's ability to service the EXTI interrupt. At frequencies approaching 300-400 kHz, the period of the signal approaches the time required to enter and exit the ISR. Beyond this point, the main loop freezes, and the system eventually fails to capture every edge, resulting in erroneous frequency readings.

4.4.4 Control Loop Non-Linearity

While the DAC outputs a linear voltage (0-3.3V), the physical response of the external hardware is non-linear. The 4N35 optocoupler has a non-linear Current Transfer Ratio (CTR), and the NE555's frequency response to control voltage changes is exponential rather than linear.

Project Discussion

5.1 Results & Challenges

This project allowed us to develop a deeper understanding of what is required for developing an embedded system. This included our planning on the original file architecture, circuitry, and how we would split up the work to ensure that it was done in a timely and efficient manner. The first major assumption we made that was incorrect was how to deal with the ADC DAC values.

Specifically, we proceeded on the assumption that we would be using the single interrupt vector that we programmed using the TIM2 timer for both the function generator as well as the tracking of the 555 timer. This led to an issue occurring during demo day where when we demoed our system it was pointed out to us that the system set up as required. However, even with The ADC DAC being incorrectly configured the system still outputs the data as requested and the ADC DAC value can still be modified using the potentiometer. The next issue we found while designing this system was how we could deal with the overwriting of the OLED. Sometimes when we would over write the OLED with the new data it would result in parts of the previous data still being visible. After we spent time discussing the issue we figured out a solution. This solution involved clearing the two lines we added to the OLED each time before writing the new data.

5.2 Future Recommendations & Conclusion

To improve the system for a commercial or industrial application, several enhancements are recommended. Implementing Direct Memory Access (DMA) for the ADC and SPI peripherals would offload data transfer tasks from the CPU, allowing the core to handle higher frequency interrupts without saturation. Additionally, migrating the frequency measurement from a GPIO Interrupt (EXTI) to a dedicated Timer Input Capture Channel (e.g., TIM2_CH2) would allow hardware-based edge detection, significantly reducing CPU overhead.

Overall, going forward we need to ensure we properly understand the requirements asked, as this will help avoid issues like the ADC/DAC feature being improperly implemented. Furthermore, ensuring that each partner has read and fully understands how each separate system works will greatly improve the understanding of how each system can be implemented. This project successfully demonstrated the versatility of the STM32F0 microcontroller in mixed-signal applications, creating a cohesive system that bridges the digital and analog domains.

Appendices

6.1 Code Snippets

Our code was programmed into five separate files: main.c, SetClock.c, SetClock.h, display.c, and display.h. We arranged the file architecture towards readability and good code architecture practices.

6.1.1 main.c

main.c contains the initialization of our interrupts, clocks, and pins for a majority of the project besides the OLED. It also contains the code that drives the main functionality of the system.

```
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//
// ----------------------------------------------------------------------
// School: University of Victoria, Canada.
// Course: ECE 355 "Microprocessor-Based Systems".
// This is template code for Part 2 of Introductory Lab.
//
// See "system/include/cmsis/stm32f051x8.h" for register/bit definitions.
// See "system/src/cmsis/vectors_stm32f051x8.c" for handler declarations.
// ----------------------------------------------------------------------
#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
#include <math.h>
#include "stm32f0-hal/stm32f0xx_hal.h"
#include "stm32f0-hal/stm32f0xx_hal_spi.h"

#include "SetClock.h"
#include "display.h"

// ----------------------------------------------------------------------
//
// STM32F0 empty sample (trace via $(trace)).
//
// 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).
//
// ----- 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
```

```c
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
#pragma GCC diagnostic ignored "-Wreturn-type"

/* Definitions of registers and their bits are
* given in system/include/cmsis/stm32f051x8.h */
/* Clock prescaler for TIM2 timer: no prescaling */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
/* Maximum possible setting for overflow */
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)


void myGPIOB_Init(void);
void myGPIOA_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
void myTIM2_Init(void);
void myEXTI_Init(void);

// Declare/initialize your global variables here...
// NOTE: You'll need at least one global variable
// (say, timerTriggered = 0 or 1) to indicate.
// whether TIM2 has started counting or not.
volatile int edge_detected = 0;
volatile int user_button_state = 0;
volatile double ADCVal = 0.0;
extern int Freq;
extern int Res;
//extern unsigned char CurrentSignal = "FuncGen";


/*****************************************************************/

int main(int argc, char* argv[]){
   // Set timer using timer file function...
   SystemClock48MHz();

   trace_printf("Lab Project Started...\n");
   trace_printf("System clock: %u Hz\n", SystemCoreClock);


   RCC->APB2ENR |= RCC_APB2ENR_SYSCFGCOMPEN;  /* Enable SYSCFG clock */

   myGPIOB_Init();     /* Initialize I/O port PB */
   myGPIOA_Init();     /* Initialize I/O port PA */

   myADC_Init();       /* Initialize ADC */
   myDAC_Init();       /* Initialize DAC */
```

```
    myTIM2_Init();      /* Initialize timer TIM2 */
    myEXTI_Init();      /* Initialize EXTI */

    oled_config(); // Initialize OLED display...

    trace_printf("Initialization Completed!\n");

    while (1) {
        if(user_button_state == 1){ // Set Freq to the ADC value if 555 timer is
enabled.
            Freq = (int)ADCVal;
        }
        refresh_OLED(); // Calls to refresh the OLED with updated data.
    }
    return 0;
}


void myGPIOB_Init(){
    /* Enable clock for GPIOB peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    /* Configure PB2 as input */
    // Relevant register: GPIOB->MODER
    GPIOB->MODER &= ~(GPIO_MODER_MODER2);
    /* Ensure no pull-up/pull-down for PB2 */
    // Relevant register: GPIOB->PUPDR
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR2);

    //555 Timer configuration
    GPIOB->MODER &= ~(GPIO_MODER_MODER3);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3);

}

void myGPIOA_Init(){
    // Enabled clock for GPIOA
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    // Enabling GPIOA0 input to USER PUSH BUTTON.
    GPIOA->MODER &= ~(GPIO_MODER_MODER0);
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);

    //Enabling GPIOA1 input to ADC
    GPIOA->MODER |= GPIO_MODER_MODER1;
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);

    //Configure output for PA4
```

```
    GPIOA->MODER |= GPIO_MODER_MODER4;
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);

}

void myADC_Init(){
// Enable the clock for the ADC peripheral
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;

    RCC->CR2 |= RCC_CR2_HSI14ON;
    while(!(RCC->CR2 & RCC_CR2_HSI14RDY));

    // Configure the ADC for continuous conversion and overrun mode
    ADC1->CFGR1 = ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD;

    // Select Channel 1 (which is connected to PA1)
    ADC1->CHSELR = ADC_CHSELR_CHSEL1;

    // Set the ADC sample time to the maximum (239.5 cycles) for accuracy
    ADC1->SMPR |= ADC_SMPR_SMP_0 | ADC_SMPR_SMP_1 | ADC_SMPR_SMP_2; // 111

    // Enable the ADC
    ADC1->CR |= ADC_CR_ADEN;

    // Wait for the ADC to be ready
    while (!(ADC1->ISR & ADC_ISR_ADRDY))
    {
        // Wait...
    }

    ADC1->IER |= ADC_IER_EOCIE;
    NVIC_SetPriority(ADC1_COMP_IRQn, 0);
    NVIC_EnableIRQ(ADC1_COMP_IRQn);

    ADC1->CFGR1 = ADC_CFGR1_OVRMOD;


    // Start the ADC in continuous mode
    ADC1->CR |= ADC_CR_ADSTART;
}



void myDAC_Init(){
    // Enable the clock for the DAC peripheral
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;

    // Enable DAC Channel 1 (which is connected to PA4)
    DAC->CR |= DAC_CR_EN1;
```

```
}


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

   /* 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); //Sets up the buffer auto-reload, count up,
stop on overflow. Interrupts on overflow. (TIM_CR1_ARPE | TIM_CR1_URS |
TIM_CR1_OPM)

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

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

   /* Update timer registers */
   // Relevant register: TIM2->EGR
   TIM2->EGR = TIM_EGR_UG; //added

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

   /* Enable TIM2 interrupts in NVIC */
   // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
   NVIC_EnableIRQ(TIM2_IRQn); //added

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


void myEXTI_Init(){
   // SYSCFG Configurations:
   /* Mapping EXTI0 to USER BUTTON */
   SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI0_Msk);
   SYSCFG->EXTICR[0] |= (0x00 << 0); //Don't need to shift anywhere so add 0...

   /* Map EXTI2 line to PB2 */
   SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI2_Msk);
   SYSCFG->EXTICR[0] |= (0x01 << 8);
```

```
   /* Map EXTI3 line to PB3 */
   SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI3_Msk);
   SYSCFG->EXTICR[0] |= (0x01 << 12);

   // Configuring EXTI edges for interrupts
   /* Configure EXTI0: falling edge + unmask */
   EXTI->FTSR |= EXTI_FTSR_TR0;    // falling edge
   EXTI->IMR  |= EXTI_IMR_MR0;     // unmask interrupt

   /* EXTI2 line interrupts: set rising-edge trigger + unmask */
   EXTI->RTSR |= EXTI_RTSR_TR2; //added
   EXTI->IMR |= EXTI_IMR_MR2; //added

   /* EXTI3 line interrupts: set rising-edge trigger + unmask */
   EXTI->RTSR |= EXTI_RTSR_TR3;
   EXTI->IMR |= EXTI_IMR_MR2;

   // Clearing pending flags before enabling
   EXTI->PR = EXTI_PR_PR0 | EXTI_PR_PR2 | EXTI_PR_PR3;

  /* Enable EXTI0 interrupt in NVIC, and enabled interrupts for EXTI0 */
   NVIC_SetPriority(EXTI0_1_IRQn, 0);
   NVIC_EnableIRQ(EXTI0_1_IRQn);

   /* Assign EXTI2 and 3 interrupt priority = 0 in NVIC, and enable interrupts
for EXTI2 and 3 */
   NVIC_SetPriority(EXTI2_3_IRQn, 0);
   NVIC_EnableIRQ(EXTI2_3_IRQn);
}


/* This handler is declared in system/src/cmsis/vectors_stm32f051x8.c */
void TIM2_IRQHandler()
{
   /* Check if update interrupt flag is indeed set */
   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); //added

       /* Restart stopped timer */
       // Relevant register: TIM2->CR1
       TIM2->CR1 |= TIM_CR1_CEN; //added
       edge_detected = 1;
   }
}
```

```
// This is the baseline code we attempted to do for the ADC interrupt after
learning our first approach was incorrect,
// unfortunately this code does not actually function and is being left here to
show the attempt and possible building
// blocks for future fixing.
void ADC1_COMP_IRQHandler(void){
   // Check EOC flag
   if ((ADC1->ISR & ADC_ISR_EOC) != 0) {
       uint16_t raw_adc = ADC1->DR; // Read Data
       DAC->DHR12R1 = raw_adc;      // Update DAC
       ADCVal = (double)raw_adc;    // Update Global Variable
       Res = (int)((raw_adc * 5000) / 4095);
   }
}


/* This handler is declared in system/src/cmsis/vectors_stm32f051x8.c */
void EXTI2_3_IRQHandler(){
   volatile uint32_t frequency_hz; // variable for measuring frequency
   double counter = TIM2->CNT; // Counter to track timing.
   EXTI->IMR &= ~EXTI_IMR_MR2; // masking interrupt flag.

   // The if statement checks if 555 timer, or function generator is enabled.
   if(user_button_state == 0){
       /* Check if EXTI2 interrupt pending flag is indeed set */
       if ((EXTI->PR & EXTI_PR_PR2) != 0){
           if(edge_detected == 0){ // Tracks rising edge of square wave for the
function generator
               TIM2->CNT = 0x00; // set timer to 0.
               TIM2->CR1 |= TIM_CR1_CEN; //
               edge_detected = 1; // Set edge detected to one so we can track
for the falling edge of the square wave.
           }else{
               if(counter > 0){ // Ensure that the counter is greater than 0
                   frequency_hz = SystemCoreClock/counter; // Calcualte
frequency.
                   Freq = frequency_hz; // add frequency to Freq so that it
will be displayed on the screen.
                   TIM2->CNT = 0; // set timer count to 0.
               }
               edge_detected = 0; // reset the edge detected to ensure it is
tracking for a rising edge.
           }
       }
   }else{
       TIM2->CNT = 0x00;
       TIM2->CR1 |= TIM_CR1_CEN;
       DAC->DHR12R1 = ADCVal; // Write directly to DAC to drive Optocoupler.
```

```
        ADC1->CR |= ((uint32_t)0x00000004);
        while(!(ADC1->ISR & ADC_ISR_EOC)); //((uint32_t)0x00000004))
        ADCVal = (uint16_t) ADC1->DR; // Adds the converted ADC value to our
variable.
        Res = (ADCVal*5000)/0xFFF; // Calculating resistance
        Freq = ADCVal; // Setting freq value for ADC
        TIM2->CNT = 0; // Prevents overflow from occurring when focused on 555
timer.
    }
    EXTI->PR = EXTI_PR_PR2; // clearing interrupt
    EXTI->IMR |= EXTI_IMR_MR2; // unmasking interrupt flag

}

void EXTI0_1_IRQHandler(){
    EXTI->IMR &= ~EXTI_IMR_MR0; // Set the interrupt flag.
        if ((EXTI->PR & EXTI_PR_PR0) != 0) { // Check that the interrupt has
been triggered
            if (user_button_state == 0) { // Checks which state the device is
currently in (In this case it is currently record the function generator).
                trace_printf("555 Timer Enabled\n");
                user_button_state = 1; // Ensure that the device is changed so
that it starts recording the 555 timer input.

                TIM2->CR1 |= TIM_CR1_CEN; // Stop timer
                EXTI->IMR |= EXTI_IMR_MR2; // Reseting timer flag.
            } else {
                trace_printf("Function Generator Enabled\n");
                user_button_state = 0;
                TIM2->CR1 |= TIM_CR1_CEN; // Stop timer
                EXTI->IMR |= EXTI_IMR_MR2; // Reseting timer flag.
            }
            EXTI->PR = EXTI_PR_PR0; // Clearing interrupt
        }
    EXTI->IMR |= EXTI_IMR_MR0; // Unmasking interrupt flag.
}

#pragma GCC diagnostic pop
// ----------------------------------------------------------------------------
```

6.1.2 SetClock.c

The function SystemClock48MHz() was given in the original main function. Once we started the project we removed it from both the main.c and display.c and moved it into its own file. This allows for a cleaner code base.

```
#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
```

```
#include <math.h>

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 48-MHz 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();
}
```

6.1.3 SetClock.h

SetClock.h is a connection file added to allow both main.c and display.c to access the SystemClock48Mhz() function within SetClock.c

```
#ifndef SETCLOCK_H
#define SETCLOCK_H

void SystemClock48MHz( void );

#endif
```

6.1.4 display.c

display.c holds the initialization and code that allows the upload and use of the OLED display.

```
//
// 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 "stm32f0-hal/stm32f0xx_hal.h"
#include "stm32f0-hal/stm32f0xx_hal_spi.h"

#include "cmsis/cmsis_device.h"
#include "SetClock.h"
#include "display.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 retargetting, 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"


/*** This is partial code for accessing LED Display via SPI interface. ***/

unsigned int Freq = 0;  // Example: measured frequency value (global variable)
unsigned int Res = 0;   // Example: measured resistance value (global variable)
// unsigned CurrentSignal[] = "FuncGen";


// --- Private Function Prototypes ---
static void oled_Write(unsigned char Value);
static void oled_Write_Cmd(unsigned char cmd);
static void oled_Write_Data(unsigned char data);
static void oled_SetCursor(unsigned char page, unsigned char col);
static void delay_tim3_ms(uint32_t ms);

#define HAL_SPI_MODULE_ENABLED

// --- Public Function Prototypes ---
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, //0x14/0x10
    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}   // <-
};

//
// LED Display Functions
//

void refresh_OLED( void ){
   // Clears the first two pages of the OLED (only clears the first two since
those are the only two we are writing to.
```

```
   for (unsigned int page = 0; page < 2; page++){
        // Set cursor to the start of the page (page, column)
        oled_SetCursor(page, 3);

        // Write 0x00 to all 128 segments in that page
        // added plus 2 as there were 2 pixels columns that were writing even
with the 128 clear, this fixed that issue
        for (unsigned int seg = 0; seg < 130; seg++) {
            oled_Write_Data(0x00);
        }
    }

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

   // Adds the "R: <ohms> Ohm" into the buffer array.
   snprintf((char *)Buffer, sizeof(Buffer), "R: %5d Ohm", Res);

   oled_SetCursor(0,2); // Initialize location to write the Buffer characters
(page 0, column 2)

   // Writes each character to the OLED in 8 byte increments.
   for(int i = 0; Buffer[i] != '\0'; i++){
       for(int j = 0; j < 8; j++){
          oled_Write_Data(Characters[Buffer[i]][j]);
       }
   }

   // Adds the "F: <frequency> Hz" into the buffer array.
   snprintf((char *)Buffer, sizeof(Buffer), "F: %5d Hz", Freq);

   oled_SetCursor(1, 2); // Initialize location to write the Buffer characters
(page 1, column 2

   // Writes each character to the OLED in 8 byte increments.
   for(int i = 0; Buffer[i] != '\0'; i++){
       for(int j = 0; j < 8; j++){
          oled_Write_Data(Characters[Buffer[i]][j]);
       }
   }

   // Delay to prevent flickering (approx 100ms)
   delay_tim3_ms(100);
}

void oled_Write_Cmd( unsigned char cmd ){
   // make PB8 = CS# = 1 (Deselect chip to start)
   GPIOB->BSRR = GPIO_BSRR_BS_8;
```

```
    // make PB9 = D/C# = 0 (Command mode)
    GPIOB->BSRR = GPIO_BSRR_BR_9;

    // make PB8 = CS# = 0 (Select chip)
    GPIOB->BSRR = GPIO_BSRR_BR_8;

    oled_Write( cmd ); // Send the command

    // make PB8 = CS# = 1 (Deselect chip when done)
    GPIOB->BSRR = GPIO_BSRR_BS_8;
}

void oled_Write_Data( unsigned char data ){
    // make PB8 = CS# = 1 (Deselect chip to start)
    GPIOB->BSRR = GPIO_BSRR_BS_8;

    // make PB9 = D/C# = 1 (Data mode)
    GPIOB->BSRR = GPIO_BSRR_BS_9;

    // make PB8 = CS# = 0 (Select chip)
    GPIOB->BSRR = GPIO_BSRR_BR_8;

    oled_Write( data ); // Send the data

    // make PB8 = CS# = 1 (Deselect chip when done)
    GPIOB->BSRR = GPIO_BSRR_BS_8;
}


void oled_Write( unsigned char Value ){
    /* Wait until SPI2 is ready for writing (TXE = 1 in SPI2_SR) */
    // NOTE: The blocking HAL_SPI_Transmit function with HAL_MAX_DELAY
    // handles this "wait for TXE" part for you.


    /* Send one 8-bit character:
       - This function also sets BIDIOE = 1 in SPI2_CR1
    */
    HAL_SPI_Transmit( &SPI_Handle, &Value, 1, HAL_MAX_DELAY );


    /* Wait until transmission is complete (BSY = 0 in SPI2_SR)
     * This is critical. We must wait for the BSY flag to clear
     * (meaning the SPI is no longer busy) *before* the calling
     * function raises the CS# pin.
    */
    while ((SPI_Handle.Instance->SR & SPI_SR_BSY) != 0 ){
        // Wait for BSY to clear...
    }
```

```
}


void oled_config( void ){
   // 1. Enable Clocks
   // Enable GPIOB for SPI and control pins
   RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
   // Enable SPI2 peripheral clock
   RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
   // Enable TIM3 peripheral clock (for delays)
   RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;

   // 2. Configure GPIO Pins

   // Configure SPI2 pins: PB13 (SCK) and PB15 (MOSI)
   // Set Mode to 10 (Alternate function)
   GPIOB->MODER &= ~(GPIO_MODER_MODER13 | GPIO_MODER_MODER15);
   GPIOB->MODER |= (GPIO_MODER_MODER13_1 | GPIO_MODER_MODER15_1);

   // Configure Alternate Function AF0 for PB13 (AFRH5) and PB15 (AFRH7)
   // AFR[1] is AFRH (pins 8-15). AF0 is the reset value (0b0000),
   // so we just need to ensure the bits are clear.
   GPIOB->AFR[1] &= ~((0xF << (5*4)) | (0xF << (7*4))); // 5 = 13-8, 7 = 15-8

   // Configure GPIO control pins: PB8(CS#), PB9(D/C#), PB11(RES#)
   // Set Mode to 01 (General purpose output)
   GPIOB->MODER &= ~(GPIO_MODER_MODER8 | GPIO_MODER_MODER9 |
GPIO_MODER_MODER11);
   GPIOB->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0 |
GPIO_MODER_MODER11_0);

   // Set CS# (PB8) high (inactive) by default
   GPIOB->BSRR = GPIO_BSRR_BS_8 | GPIO_BSRR_BS_9;

   // 3. Configure TIM3 for 1ms delays (assuming 48MHz SystemClock)
   // Prescaler = 48,000,000 Hz / 1000 Hz = 48000.
   // Set PSC (Prescaler) register to 47999 (since it's 0-based)
   TIM3->PSC = 47999;
   // Set ARR (Auto-Reload Register) to max value. We'll poll CNT.
   TIM3->ARR = 0xFFFF;
   // Enable the timer
   TIM3->CR1 |= TIM_CR1_CEN;

   // 4. Configure SPI2
   SPI_Handle.Instance = SPI2;

   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# = PB11):
      - make pin PB11 = 0, wait for a few ms
      - make pin PB11 = 1, wait for a few ms
   */
   GPIOB->BSRR = GPIO_BSRR_BR_11; // Pull RES# low
   HAL_Delay(100);              // Wait 100ms
   GPIOB->BSRR = GPIO_BSRR_BS_11; // Pull RES# high
   HAL_Delay(100);              // Wait 100ms


//
// 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 page = 0; page < 8; page++)
   {
       // Set cursor to the start of the page (page, column 0)
       oled_SetCursor(page, 0);

       // Write 0x00 to all 128 segments in that page
       for (unsigned int seg = 0; seg < 130; seg++) // added plus 2 as there
were 2 pixels that were writing even with the 128 clear, this fixed that issue
       {
```

```
            oled_Write_Data(0x00);
        }
    }

}

/*
* Helper function that simplifies the initialization of the location to write
* to on the OLED screen.
*/
static void oled_SetCursor(unsigned char page, unsigned char col){
    // Checks and ensure the function does not go beyond the displays limits
    if (page > 7) page = 7;
    if (col > 127) col = 127;

    oled_Write_Cmd(0xB0 + page);              // Set Page Address (0-7)
    oled_Write_Cmd(0x00 + (col & 0x0F));      // Set Lower Column (0-15)
    oled_Write_Cmd(0x10 + ((col >> 4) & 0x0F)); // Set Upper Column (0-7)
}

/*
* Allows the quick use of TIM3 timer to allow for a small delay to ensure that
* the microprocessor is keeping up with the interrupts writing to the OLED.
*/
static void delay_tim3_ms(uint32_t ms)
{
    TIM3->PSC = 47999;
    TIM3->ARR = 0XFFFF;
    TIM3->CR1 |= TIM_CR1_CEN;

    for(uint32_t i=0; i<ms; i++){
        TIM3->CNT=0;
        while(TIM3->CNT < 1);
    }
}

#pragma GCC diagnostic pop

// ------------------------------------------------------------------------------
```

6.1.5 display.h

display.h is added to allow main.c to access the required functions oled_config() and refresh_OLED(). oled_config() is used in main to initialize the OLED display. refresh_OLED() allows the code in main.c to update the values being displayed on the OLED display.

```
#ifndef DISPLAY_H
#define DISPLAY_H
```

```h
#include "cmsis/cmsis_device.h"

// --- Public Global Variables ---
// These are *defined* in main.c, but the display.c
// file needs to know they exist to read them.

// --- Public Function Prototypes ---

/**
* @brief Initializes all GPIO, SPI2, and Timer peripherals
* required for the OLED display.
*/
void oled_config(void);

/**
* @brief Clears the display and re-prints the Resistance and
* Frequency values. Includes a ~100ms delay.
*/
void refresh_OLED(void);

#endif
```

6.2 Extra Wiring Diagrams

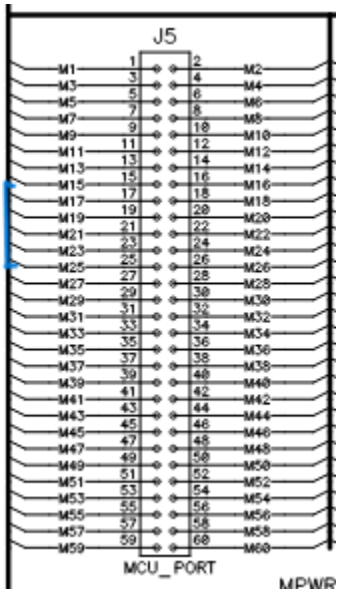

Figure 9: J5 rail located on the Project Base Board User Guide (PBMCUSLK) [5].

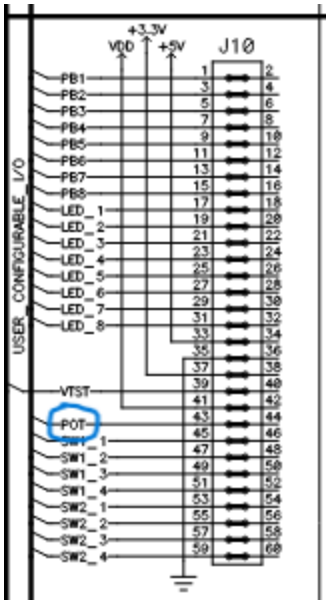

Figure 10: J10 rail located on the Project Base Board User Guide (PBMCUSLK) [5].

References

[1] Brent Sirna and Daler Rakhmatov, University of Victoria Electrical and Computer Engineering ECE 355: Microprocessor-Based Systems Laboratory Manual, Victoria, British Columbia, 2023. https://www.ece.uvic.ca/~ece355/lab/ECE355-LabManual-2025.pdf (Accessed Oct. 9th, 2025).

[2] STMicroelectronics, “STM32F0xxx Cortex-M0 programming manual”, PM0215 Programming manual, ch. 7,9,12,13,14,17 2012.
https://www.ece.uvic.ca/~ece355/lab/supplement/STM32F0xxxProgrammingManual_DM00051352.pdf (Accessed Oct. 9th, 2025).

[3] STMicroelectronics, “STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs”, RM0091
Reference manual, 2014.
https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f0RefManual.pdf (Accessed Nov. 15th, 2025).

[4] Freescale Semiconductor, “MCU PROJECT BOARD STUDENT LEARNING KIT (PBMCUSLK)” pp. 20, Rev. 1, 7/2007.
https://www.ece.uvic.ca/~ece355/lab/supplement/PBMCUSLK_UG.pdf (Accessed Nov. 18th, 2025)

[5] Xiom Manufacturing, https://www.ece.uvic.ca/~ece355/lab/supplement/PBMCUSLK_SCH_D_0-1.pdf. (Accessed Nov. 16th, 2025)

[6] STMicroelectronics, “General-purpose single bipolar timers”, January 2012, Rev 6. https://www.ece.uvic.ca/~ece355/lab/supplement/555timer.pdf (Accessed Nov. 14th, 2025)

[7] Vishay, “Optocoupler, Phototransistor Output, with Base Connection”, Rev: 02-Oct-12. https://www.ece.uvic.ca/~ece355/lab/supplement/4n35.pdf (Accessed Nov. 14th, 2025)

[8] D.N.Rakhmatov, “Interfacing Examples”. https://www.ece.uvic.ca/~ece355/lab/interfacex.pdf (Accessed Nov. 14th, 2025)

[9] D.N.Rakhmatov, “I/O Examples”. https://www.ece.uvic.ca/~ece355/lab/iox.pdf (Accessed Nov. 14th, 2025)

[10] Calculator.net, “Resistor Calculator”. https://www.calculator.net/resistor-calculator.html (Accessed Nov 25th, 2025)