University of Victoria

Department of Electrical and Computer Engineering

Lab Project Report

Frequency Measurement and ADC/DAC System Implementation

Submitted By:

Group Number:	01
Section:	B06
Members:	Arfaz Hossain (V00984826)
	Aly Mooltazeem (V00962689)

Faculty:

Lecture Professor:	Daler Rakhmatov
Lab Technologist:	Brent Sirna

Date of Submission: Nov 29, 2024

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

Contents

Abstract

This project focuses on the development and implementation of a system for measuring the frequencies of the periodic signals from various devices input to a microcontroller. The design incorporates ADC-based signal acquisition, DAC-driven signal generation, precise frequency and resistance calculations, and real-time data visualization via an OLED display. The implementation leverages multiple microcontroller peripherals—GPIO, TIM2, ADC, DAC, and EXTI—to facilitate seamless operation between Function Generator and 555 Timer modes, controlled through a user interface button. Experimental validation demonstrated measurement accuracy within 2% deviation under standard operating conditions. Technical challenges encompassing signal interference and interrupt timing were resolved through robust synchronization protocols and signal conditioning techniques. The final system architecture exemplifies efficient peripheral utilization and systematic design methodology, while identifying potential enhancements in sampling efficiency and noise reduction strategies.

1 Problem Description and Technical Specifications

1.1 Objectives

This project aims to develop a sophisticated embedded system that combines signal generation, measurement, and control capabilities. The system is built around the STM32F0 Discovery microcontroller board interfacing with a PBMCUSLK project board, with the following core objectives:

- **System Architecture:** Implementation based on STM32F051R8 microcontroller:
  - Integration of multiple peripherals: GPIO, ADC, DAC, TIM2, EXTI
  - SPI communication protocol for OLED display control
  - Dual-mode operation with button-based switching
- **Dual Signal Generation and Monitoring:** Design and implementation of a system capable of working with two distinct signal sources:
  - A PWM signal generated by an NE555 timer circuit [1]
  - A square wave signal from an external Function Generator
  - 12-bit ADC resolution for high-accuracy signal capture
- **Dynamic Signal Control:** Development of a feedback system where:
  - The ADC measurement from the potentiometer controls the 555 timer's PWM characteristics
  - An optocoupler (4N35) provides electrical isolation and signal control [2]
  - The DAC output modulates the optocoupler's behavior
  - Real-time DAC signal generation for testing and analysis
- **Operational Modes:** Implementation of two distinct operational modes:
  - **Function Generator Mode:** Displays measured frequency from the function generator
  - **NE555 Timer Mode:** Display the 555 Timer Frequency
  - External button-based mode switching capability
- **Real-time Measurement System:** Creation of a measurement system that:
  - Continuously monitors potentiometer voltage through ADC polling
  - Calculates actual resistance values using voltage divider formulas
  - Accurately measures signal frequencies from both sources
- **User Interface:** Implementation of an OLED-based display system that:

- Shows current frequency measurements
- Displays calculated potentiometer resistance
- Provides visual feedback for system operation
- Supports real-time data visualization for both operational modes

1.2 Technical Specifications

The implementation must adhere to specific technical requirements and constraints:

- **Microcontroller Interface Requirements [3]:**
  - USER button configuration on PA0 with EXTI0 interrupt capability
  - 555 timer signal measurement on PA1 utilizing EXTI1
  - Function Generator signal measurement on PA2 using EXTI2
  - DAC output on PA4 for optocoupler control
  - ADC input on PA5 for potentiometer voltage measurement
  - OLED screen on PB4 to PB7 to display measured values
- **Signal Processing Requirements: [3]**
  - Continuous ADC polling for potentiometer voltage measurement
  - Real-time conversion of voltage readings to resistance values
  - Accurate frequency measurement of both signal sources
  - Interrupt-driven source switching capability
- **System Integration Features:**
  - Seamless switching between signal sources via USER button
  - Dynamic update of display information
  - Proper electrical isolation through optocoupler
  - Stable operation across varying input conditions
- **Pedagogical Constraints and Development Guidelines** [3]:
  - **Potentiometer Voltage Polling:** Voltage values from the potentiometer must be obtained using a polling approach.
  - **Specific Pin Assignments:** Fixed pin assignments must be followed (e.g., PA0 for USER button, PA1 for 555 timer signal input, etc.).
  - **USER Button Interrupts:** The USER button (PA0) must trigger an interrupt using EXTI0 to switch frequency measurements.
  - **Frequency Measurement via TIM2:** TIM2 must be used to measure both the Function Generator and 555 timer signal frequencies.

- **DAC-Controlled PWM Frequency:** The DAC (PA4) must drive the optocoupler to adjust the 555 timer's PWM frequency.
- **SPI Communication for LED Display:** SPI pins (PB3, PB4, PB5, PB6, PB7) must be used to drive the SSD1306 LED display.
- **Voltage Measurement Limits:** The lower and upper limits of the potentiometer voltage must be determined to calculate the corresponding resistance.
- **Reserved Pins:** PA13 and PA14 are reserved for ST-LINK communication and must not be used.

2 Design and Solution

2.1 System Overview

The system is designed as an integrated hardware-software solution focusing on real-time signal processing. The core system architecture includes:

2.1.1 Signal Processing Components

- ADC (12-bit resolution) for input signal capture
- DAC for scaled analog signal output
- TIM2 for precise frequency measurements
- GPIO for button input handling
- OLED Display for user interface

2.1.2 Operational Modes

- **Function Generator Mode:** Displays measured frequency from the function generator
- **NE555 Timer Mode:** Display the 555 Timer Frequency

2.1.3 System Integration

- Centralized control via STM32F051R8 microcontroller
- Interrupt-based event handling
- Real-time data processing pipeline
- User interface management

[0pt] The block diagram in Figure 1 The diagram illustrates the interconnections between various hardware modules and highlights the system's input, processing, and output stages, making it a typical system-level design representation for an embedded electronic device.

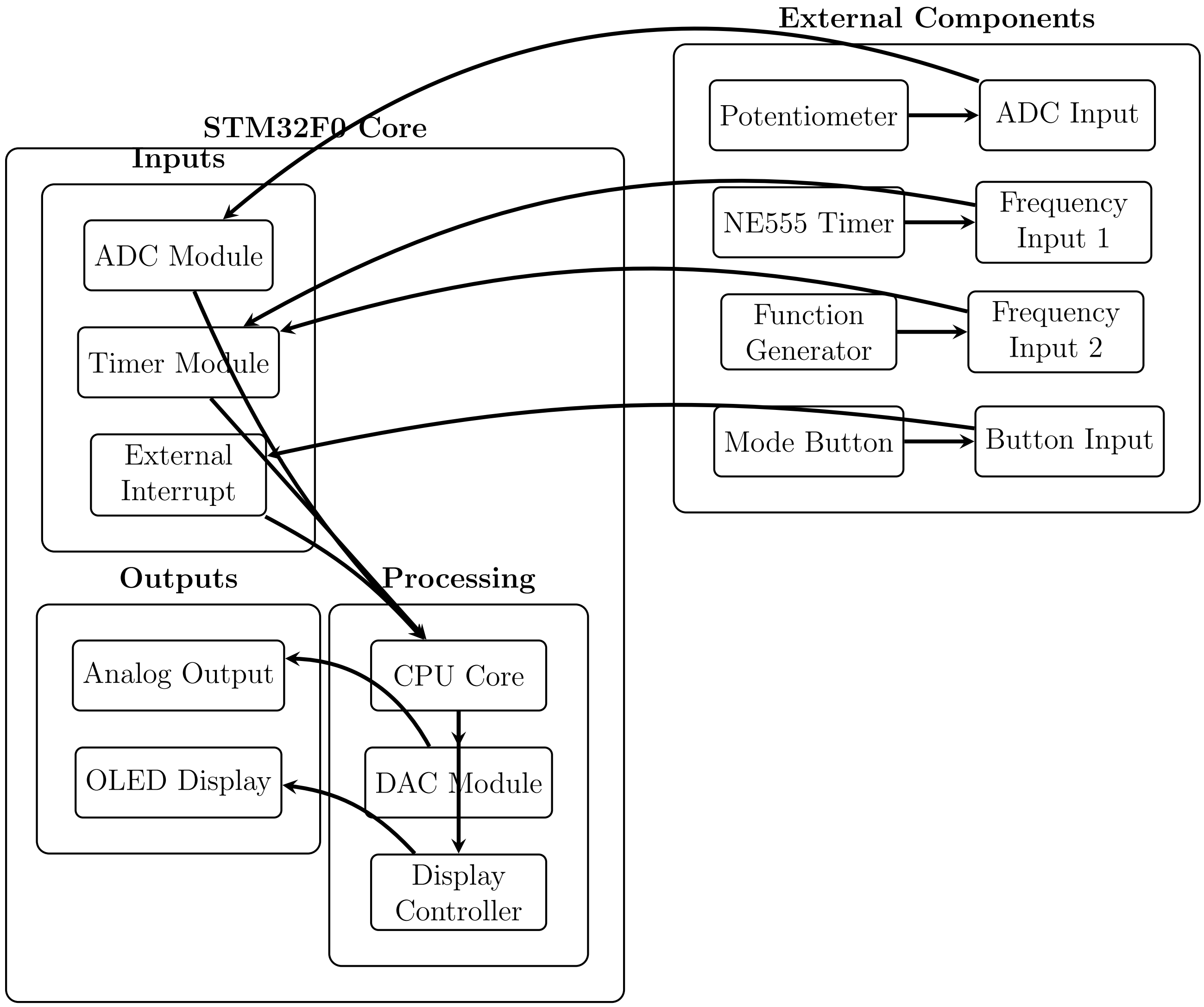

Figure 1: STM32F0 Microcontroller System Block Diagram

2.2 Hardware Design

2.2.1 Block Diagram

The hardware architecture (Figure 2) consists of the following interconnected components:

- STM32F051R8 microcontroller (central processor)
- Input devices (potentiometer, function generator)
- Output devices (DAC, OLED display)
- Control interfaces (mode-switch button)

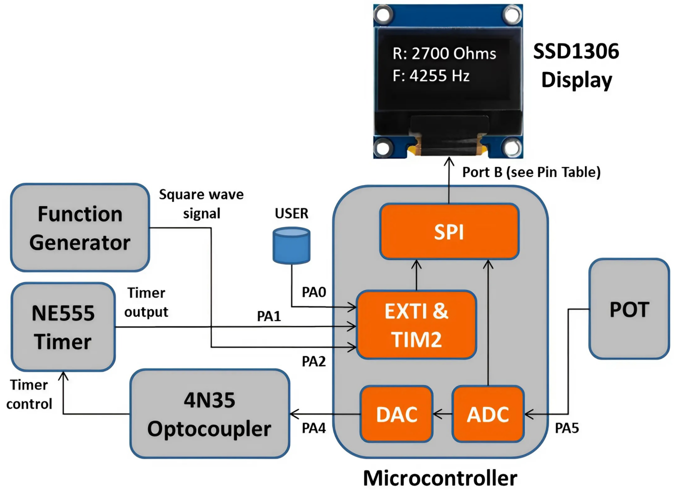

Figure 2: System architecture showing the interaction between major components of the frequency measurement and signal generation system [3]

2.2.2 Hardware Components [3]

- **STM32F051R8 Microcontroller:** Central processing unit
- **Potentiometer:** Analog input source
- **OLED Display (SSD1306):** User interface display
- **Mode-Switch Button:** System control
- **Passive Components:** Signal conditioning

2.2.3 Pin Configuration [3]

- PA0: USER button interrupt handling (EXTI0)
- PA1: NE555 timer signal measurement (EXTI1)
- PA2: Function generator frequency measurement (EXTI2)
- PA4: DAC output for optocoupler control
- PA5: ADC input for potentiometer measurement
- PB3-PB7: SPI and control signals for OLED display

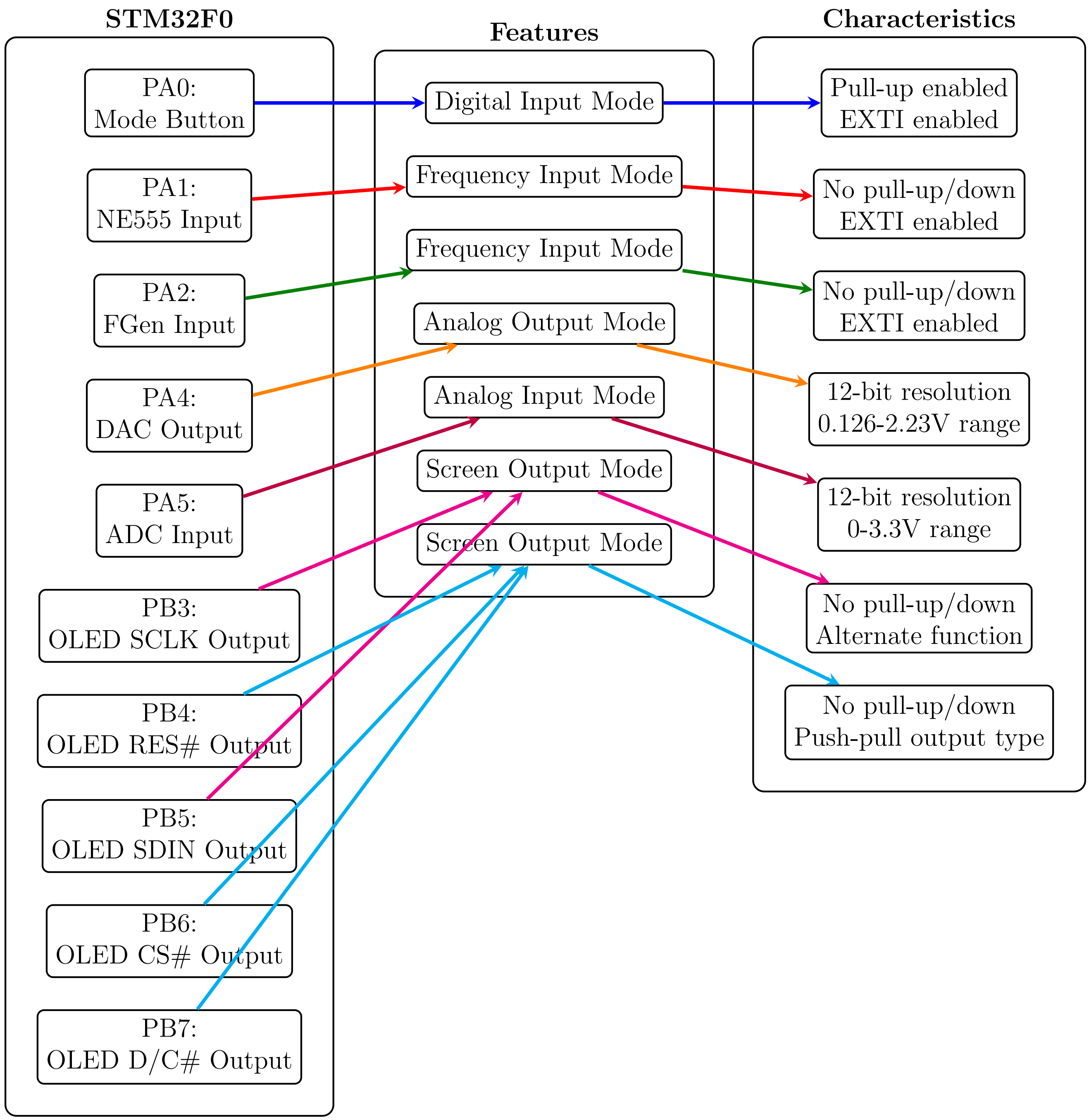

Figure 3: Pin Configurations of the GPIO Pins [3]

2.2.4 Power Distribution

- Main supply: 3.3 V Power Supply (Allowable V_{DD}: 2.0 to 3.6 V [4])
- Separate analog/digital grounds
- Decoupling capacitors for noise reduction

3 Software Design

The software is designed with the intent of maintainability and debugging. However, there is room for improvement in modularization (except for the OLED screen) to better achieve these goals. Key initialization functions and operational logic are outlined below.

3.1 Initialization Functions

The examples of the initialization functions from the project configure the system's peripherals, including the GPIO, ADC, DAC, and timers. (See Appendices 7.1 and 7.3 for the full code.)

3.1.1 System Clock

```c
void SystemClock48MHz(void) {
    // Disable the PLL
    RCC->CR &= ~(RCC_CR_PLLON);
    // Wait for the PLL to unlock
    while ((RCC->CR & RCC_CR_PLLRDY) != 0);
    // Configure the PLL for a 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 to the PLL as the clock source
    RCC->CFGR = (RCC->CFGR & (~RCC_CFGR_SW_Msk)) | RCC_CFGR_SW_PLL;
    // Update the system clock variable
    SystemCoreClockUpdate();
}
```

Listing 1: System Clock Initialization Function [5]

3.1.2 GPIO Initialization

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

    // Configure PA0 (button) as input
    GPIOA->MODER &= ~(GPIO_MODER_MODER0);

    // Configure PA1 (555 timer) as input
    GPIOA->MODER &= ~(GPIO_MODER_MODER1);

    // Configure PA2 (function generator) as input
```

```
    GPIOA->MODER &= ~(GPIO_MODER_MODER2);

    // Configure PA4 and PA5 as analog mode
    GPIOA->MODER |= GPIO_MODER_MODER4;
    GPIOA->MODER |= GPIO_MODER_MODER5;

    // Ensure no pull-up/pull-down for PA1 and PA2
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1 | GPIO_PUPDR_PUPDR2);
}
```

Listing 2: GPIO Port A Initialization Function [5]

3.1.3 ADC and DAC Initialization

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

    // Configure ADC settings
    ADC1->SMPR = 0x7;                  // Maximum sampling time
    ADC1->CHSELR = ADC_CHSELR_CHSEL5;// Select channel 5

    // Calibrate ADC if enabled
    if (ENABLE_CAL) {
        ADC1->CR = ADC_CR_ADCAL;
        while (ADC1->CR == ADC_CR_ADCAL);
    }

    // Enable ADC and wait for ready
    ADC1->CR |= ADC_CR_ADEN;
    while (!(ADC1->ISR & ADC_ISR_ADRDY));

    // Configure continuous conversion mode
    ADC1->CFGR1 |= (ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD);
}

void myDAC_init(void) {
    // Enable DAC Clock
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;

    // Clear and configure DAC control register
    DAC->CR &= ~(0x7);
    DAC->CR |= DAC_CR_EN1;
}
```

Listing 3: ADC and DAC Initialization Function [6]

3.1.4 Timer (TIM2) Initialization

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

    // Configure TIM2
    TIM2->CR1 = ((uint16_t)0x008C);
    TIM2->PSC = myTIM2_PRESCALER;
    TIM2->ARR = myTIM2_PERIOD;

    // Update timer registers
    TIM2->EGR |= ((uint16_t)0x0001);

    // Configure and enable interrupts
    NVIC_SetPriority(TIM2_IRQn, 0);
    NVIC_EnableIRQ(TIM2_IRQn);
    TIM2->DIER |= TIM_DIER_UIE;
}
```

Listing 4: Timer 2 Initialization Function [5]

3.1.5 External Interrupt Initialization

```c
void EXTI_Init(void) {
    // Map EXTI2 and EXTI0 lines to PA2 and PA0 respectively
    SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI0 |
       SYSCFG_EXTICR1_EXTI1 | SYSCFG_EXTICR1_EXTI2);
    SYSCFG->EXTICR[0] |= (SYSCFG_EXTICR1_EXTI0_PA |
       SYSCFG_EXTICR1_EXTI1_PA | SYSCFG_EXTICR1_EXTI2_PA);

    // Set rising-edge trigger for EXTI2 and EXTI0 lines
    EXTI->RTSR |= (EXTI_RTSR_TR0 | EXTI_RTSR_TR1 | EXTI_RTSR_TR2);

    // Unmask interrupts from EXTI2 and EXTI0 lines
    EXTI->IMR |= (EXTI_IMR_IM0 | EXTI_IMR_IM1);

    // Configure interrupt priorities and enable in NVIC
    NVIC_SetPriority(EXTI0_1_IRQn, 0);
    NVIC_EnableIRQ(EXTI0_1_IRQn);

    NVIC_SetPriority(EXTI2_3_IRQn, 1);
    NVIC_EnableIRQ(EXTI2_3_IRQn);
}
```

Listing 5: EXTI Initialization Function [5]

3.2 Core Logic

3.2.1 Signal Measurement and Reconstruction

```c
// ADC Reading (Measurement)
uint32_t readADC(void) {
    // Start ADC conversion
    ADC1->CR |= ADC_CR_ADSTART;

    // Wait for conversion completion
    while (!(ADC1->ISR & ADC_ISR_EOC));

    // Return the ADC result
    return ADC1->DR;
}

// Write to DAC (Reconstruction)
void writeDAC(uint32_t adc_val)
{
    // DHR12R1 is the 12-bit right-aligned data
    DAC->DHR12R1 = adc_val;
}
```

Listing 6: ADC Reading and DAC Writing Function [6]

3.2.2 Frequency and Resistance Computation

```c
void measure_frequency(unsigned int bit_number, unsigned int*
   var_address) {
    // Assign some necessary variables
    unsigned int count = 0;
    float period = 0;
    float frequency = 0;
    // Compute the register mask by bit number
    uint32_t register_mask = EXTI_PR_PR0 << bit_number;

    /* Check if EXTI2 interrupt pending flag is indeed set */
    if ((EXTI->PR & register_mask) != 0) {
        // If this is the first 'rising' edge:
        if((TIM2->CR1 & TIM_CR1_CEN) == 0) {
            TIM2->CNT = 0;    // Clear count register
            TIM2->CR1 |= TIM_CR1_CEN;    // Start the timer
        } else {     // Otherwise, this is the second edge
            TIM2->CR1 &= ~(TIM_CR1_CEN);    // Stop timer
            // Read out count register
            count = TIM2->CNT;
            // Calculate the frequency
```

```c
                period = (float)count / (float)SystemCoreClock;
                frequency = 1 / period;
                // 'var_address' - variable where the frequency is
                   stored
                // Update the 'var_address'
                *var_address = (unsigned int)(frequency);
            }
            EXTI->PR |= register_mask;
        }
}
```

Listing 7: Frequency and Resistance Computation

3.2.3 Mode Switching Logic

The software includes a toggle function to switch between NE555 timer mode and function generator mode.

```c
void toggle_mode(void) {
    // Toggle the mode
    funcGen_mode = !funcGen_mode;

    // Enable or disable interrupts based on the mode
    if (!funcGen_mode) { // NE555 timer mode
        EXTI->IMR &= ~(EXTI_IMR_IM2);
        EXTI->IMR |= EXTI_IMR_IM1;
    } else { // Function generator mode
        EXTI->IMR &= ~(EXTI_IMR_IM1);
        EXTI->IMR |= EXTI_IMR_IM2;
    }

    // Debug output (optional)
    if (TOGGLE_DEBUG) {
        trace_printf(funcGen_mode ? "<<<< FUNCTION GENERATOR >>>>\n"
            : "<<<< NE555 TIMER >>>>\n");
    }
}
```

Listing 8: Toggle Mode Function

3.2.4 Button Press Handling [5]

```c
void button_push(void) {
    // Check for a pending interrupt on PA0
    if ((EXTI->PR & EXTI_PR_PR0) != 0) {
        if ((GPIOA->IDR & GPIO_IDR_0) != 0) {
            // Wait for button release
```

```
            while ((GPIOA->IDR & GPIO_IDR_0) != 0) {}

            // Trigger the mode toggle
            toggle_mode();
        }
        // Clear the pending interrupt flag
        EXTI->PR |= EXTI_PR_PR0;
    }
}
```

Listing 9: Button Push Handler Function

3.3 Utilities

3.3.1 Timer Interrupt Handling [5]

```
void TIM2_IRQHandler(void) {
    if ((TIM2->SR & TIM_SR_UIF) != 0) {
        // Handle timer overflow
        trace_printf("\n*** Overflow in TIM2! ***\n");

        // Clear interrupt flag and restart the timer
        TIM2->SR &= ~TIM_SR_UIF;
        TIM2->CR1 |= TIM_CR1_CEN;
    }
}
```

Listing 10: Timer 2 Interrupt Handler

3.3.2 Value Computation

From the 12-bit ADC Value, the resistance is calculated using the following equation:

$$R_{pot} = \frac{ADC_{value}}{4095} \times 5000\ \Omega$$

```
unsigned int toOhms(uint32_t adc_val) {
    return (unsigned int)(((float)adc_val/4095.0) * 5000.0);
}
```

Listing 11: Resistance Calculation Function

3.3.3 External Interrupt Handlers

```
void EXTI0_1_IRQHandler(void) {
    // Handle button press
    button_push();
```

```c

    // Measure frequency from PA1 if in NE555 timer mode
    if (!funcGen_mode) {
        measure_frequency(1, &ne555_frequency);
    }
}

void EXTI2_3_IRQHandler(void) {
    // Measure frequency from PA2 if in function generator mode
    if (funcGen_mode) {
        measure_frequency(2, &fgen_frequency);
    }
}
```

Listing 12: EXTI0 and EXTI2 Handlers

3.4 Key Features

3.4.1 Signal Measurement and Generation

- `readADC()`: Captures analog signals using the ADC with 12-bit resolution. This function is critical for converting the potentiometer's voltage into digital values for resistance calculation.

- `measure_frequency()`: Accurately measures the frequency of input signals by utilizing hardware timers (TIM2) and external interrupt-driven edge detection. The function processes input from either the NE555 timer or the function generator, depending on the operational mode.

  The period is determined by the number of system clock counts (48 MHz) between the two rising edges. This leads to the following equation:

  $$T = \frac{f_{\text{system clock}}}{N_{count}}$$

  Since $F = \frac{1}{T}$,

  $$F_{measured} = \frac{N_{count}}{f_{\text{system clock}}}$$

3.4.2 Signal Generation

- `writeDAC()`: Outputs analog signals to external devices through the DAC, which is synchronized with ADC inputs to provide scaled responses. This functionality ensures smooth signal generation for external circuit testing.

- Continuous signal monitoring allows seamless integration between input (ADC) and output (DAC) processes.

3.4.3 Mode Switching

- The system supports two operational modes:
  1. **NE555 Timer Mode**: Captures the frequency of a signal from a NE555 timer circuit and displays it alongside resistance values from the potentiometer.
  2. **Function Generator Mode**: Measures the frequency of a signal from an external function generator, providing accurate real-time updates.
- `toggle_mode()`: Enables seamless switching between operational modes via the user button (PA0) interrupt, ensuring intuitive and responsive control.
- Real-time updates to OLED displays allow users to monitor changes instantly.

3.4.4 Synchronization

- **Interrupt-Driven Architecture [5]**:
  - Utilizes external interrupts (EXTI) for precise event handling, enabling accurate frequency and signal measurements.
  - Minimizes CPU overhead by offloading tasks to hardware interrupts, improving efficiency and responsiveness.
- **Hardware-Timer-Based Synchronization [5]**:
  - TIM2 is configured to provide high-resolution timing for frequency measurement, ensuring accuracy even at high signal rates.
  - Overflow detection and interrupt handling [5] prevent timing inaccuracies during long-duration measurements.
- **ADC-DAC Synchronization [4]**: Ensures seamless operation between input signal acquisition and output signal generation, minimizing latency and improving system performance.
- **OLED Display Updates [7]**: The system synchronizes signal measurements with visual output, ensuring that displayed data is both current and accurate.

3.4.5 OLED Screen Operation

- **Initialization:** The OLED screen is initialized through a series of configuration steps. These include setting up GPIO pins, SPI communication, and a timer (TIM3) [6]. Initialization commands are then sent to the OLED controller to configure the display's operating mode. These commands include the following:
  1. Enabling the display
  2. Setting addressing modes

3. Clearing the screen memory (GDDRAM) by writing zeros across all segments on each page

- **Writing to the Screen:** Characters are displayed on the OLED by sending their bitmap representations from a predefined character map to the screen's GDDRAM. Each character is represented by an 8-byte array, with each byte corresponding to a segment of the character by multiples of 8 [6]. This uses the following functions:
  - `oled_print()`: Iterates through the input buffer of characters, selecting the appropriate page and segment, and writing the character data using the `oled_write_data`.
  - `oled_Write_Data()`: Sends a data byte to the OLED controller for display in its memory (GDDRAM).

- **Page Segmenting:** The display memory is divided into pages (rows) and segments (columns) as shown in Figure 4 [7]. This makes use of the following functions:
  - `set_Page`: selects the row where data will be written. This uses `oled_write_cmd` which passes 8 bits of data, where the second nybble should be of a hexadecimal digit `0xB`. (e.g. For page 4, the system should send **0xB4**.)
  - `set_Segment`: Specifies the column where data will be written. This also uses `oled_write_cmd`, where the second nybble should be:
    * **Hexadecimal digit 0x0** for the first four bits of the column (e.g. for column 65 (0x41), 0x1 represents the first four bits, so send **0x01**.)
    * **Hexadecimal digit 0x1** for the last four bits of the column (e.g. for column 65 (0x41), 0x4 represents the last four bits, so send **0x14**.)

  These functions send appropriate commands to the OLED to position the address pointer, and then moves on to the next segment everytime a character is written. For example, to print "HELLO!" starting at page 3, column 1, the program fills out 48 segments from segment 8 to 55 in the screen memory.

	COL0	COL 1	.....	COL 126	COL 127
PAGE0					
PAGE1					
:	:	:	:	:	:
PAGE6					
PAGE7					

Figure 4: Rows and columns of the address memory, with the address pointer movement [7]

- **Memory Location and Character Position:** Each character's position on the screen is determined by its row and column, corresponding to the page and segment in memory. For example, ASCII characters are mapped to the predefined character array, where the ASCII code determines the row in the array, and the bytes represent the visual representation of the character. This mechanism ensures a precise correspondence between memory locations and the screen's display layout.

- **Refreshing the OLED:** The OLED display is periodically refreshed using the TIM3 timer to ensure that updated information is consistently displayed. The `refresh_OLED` function is triggered based on the timer's count reaching a predefined threshold of about 4800 counts. Meaning, that the screen must refresh every $\frac{4800}{48000\text{ Hz}} = 100$ ms [6]. During each refresh cycle, relevant data such as resistance and frequency values are formatted into strings and displayed on specific pages of the OLED screen using the `oled_print` function. The timer ensures accurate timing for updates, and the `TIM3_delay` function is used to introduce small delays during initialization and resetting processes. This mechanism maintains a responsive and stable display update rate.

The designed C module for the OLED screen header and source file is shown in Appendix 7.3.

3.5 Peripheral Justification

- **ADC/DAC** [6]:
  - The ADC (12-bit resolution) is crucial for precise analog-to-digital conversion of input signals from the potentiometer.
  - The DAC enables scalable analog output, supporting external signal generation and circuit testing.
- **TIM2** [5]:
  - Provides precise timing for frequency measurement by counting clock cycles between signal edges.
  - Supports high-speed signal processing with overflow detection to ensure robustness in varying signal conditions.
- **GPIO** [5]:
  - Handles user button inputs and external signal connections.
  - Configures specific pins (PA0, PA1, PA2) for input signals and mode control.
- **EXTI** [5]:
  - Enables hardware-based edge detection for signal frequency measurement.
  - Critical for capturing precise timing events without CPU intervention, ensuring low-latency performance.
- **OLED Display** [6]:
  - The SSD1306-based display provides a user-friendly interface to present real-time measurements and system status.
  - SPI communication ensures efficient data transfer, supporting real-time updates.

4 Testing and Results

4.1 Testing Procedure

To validate the functionality and performance of the system, a structured testing procedure was employed. Each aspect of the system was rigorously examined to ensure reliability and accuracy under real-world conditions.

4.1.1 System Initialization and Peripheral Verification

1. Power on the STM32F0 Discovery board and ensure stable voltage levels across all peripherals.
2. Verify system clock configuration by outputting the clock frequency to the console and ensuring it matches the expected 48 MHz.
3. Test GPIO pin configurations by probing each pin and confirming the expected input/output states using an oscilloscope.
4. Check ADC and DAC initialization by generating a test analog signal using a function generator and confirming correct ADC readings and DAC voltage outputs using a multimeter.
5. Initialize the OLED display and verify proper communication via SPI by displaying test data.

4.1.2 Frequency Measurement Validation

1. Generate a square-wave signal (max. 0–3.3 V amplitude) using a function generator at variable frequencies up to 80 MHz [8].
2. Connect the signal to PA2 and monitor the EXTI2 interrupt behavior using the debugger or console outputs.
3. Measure the time elapsed between two rising edges of the input signal using TIM2 and confirm accuracy by comparing the calculated frequency to the function generator's set frequency.
4. Determine the minimum detectable frequency by lowering the input signal frequency until the system fails to provide consistent measurements.
5. Determine the maximum detectable frequency by increasing the input signal frequency until the timer fails to capture events accurately.

4.1.3 Resistance Calculation Validation

1. Connect a potentiometer to PA5 and adjust its resistance across the full range (0–5 kΩ).
2. Confirm ADC readings at various potentiometer positions by comparing the measured resistance (using the conversion from ADC to resistance) to values obtained using a multimeter.
3. Validate real-time updates on the OLED display for resistance measurements using the console output or the OLED output.

4.1.4 Mode Switching Functionality

1. Test the system's response to pressing the USER button (PA0) by toggling between NE555 Timer Mode and Function Generator Mode.
2. Ensure proper reconfiguration of EXTI interrupts for PA1 and PA2 when switching modes.
3. Validate that frequency measurements from the correct input source are displayed on the OLED after each mode switch.
4. Confirm seamless transitions without data loss or system crashes.

4.1.5 Signal Monitoring and DAC Control

1. Adjust the potentiometer and monitor the real-time ADC readings.
2. Validate DAC outputs by connecting an oscilloscope to PA4 and confirming the output voltage corresponds to the scaled ADC input.
3. Ensure that the DAC signal drives the optocoupler to adjust the PWM frequency and duty cycle of the NE555 timer circuit.
4. Verify continuous synchronization between ADC input, DAC output, and OLED display updates.

4.1.6 Noise and Stability Testing

1. Introduce electrical noise to the system using a signal generator or by varying the power supply voltage.
2. Verify the robustness of the ADC readings and DAC outputs under noisy conditions by observing the stability of displayed data and oscilloscope waveforms.
3. Test the impact of simultaneous mode switching and signal input variations on system performance.
4. Confirm the effectiveness of decoupling capacitors in mitigating noise.

4.2 Results

The system demonstrated robust performance across all testing scenarios, meeting or exceeding the project requirements.

4.2.1 Frequency Measurement Performance

In the STM32 microcontroller, the ARR in the timer is limited 32 bits. Therefore, the maximum count is determined by the following:

$$N_{\text{max}} = 2^{32} - 1 = 4,294,967,295$$

Therefore, the **minimum frequency** would be:

$$f_{\text{min}} = \frac{f_{\text{clock}}}{N_{\text{max}}} = \frac{48,000,000}{4,294,967,295} \approx 0.0112 \text{ Hz}$$

Moreover, the graph in Figure 5 shows the plot of measured frequencies with the frequencies from the function generator. And the second graph (Figure 6) shows the percentage error from the measured frequencies against the expected frequencies.

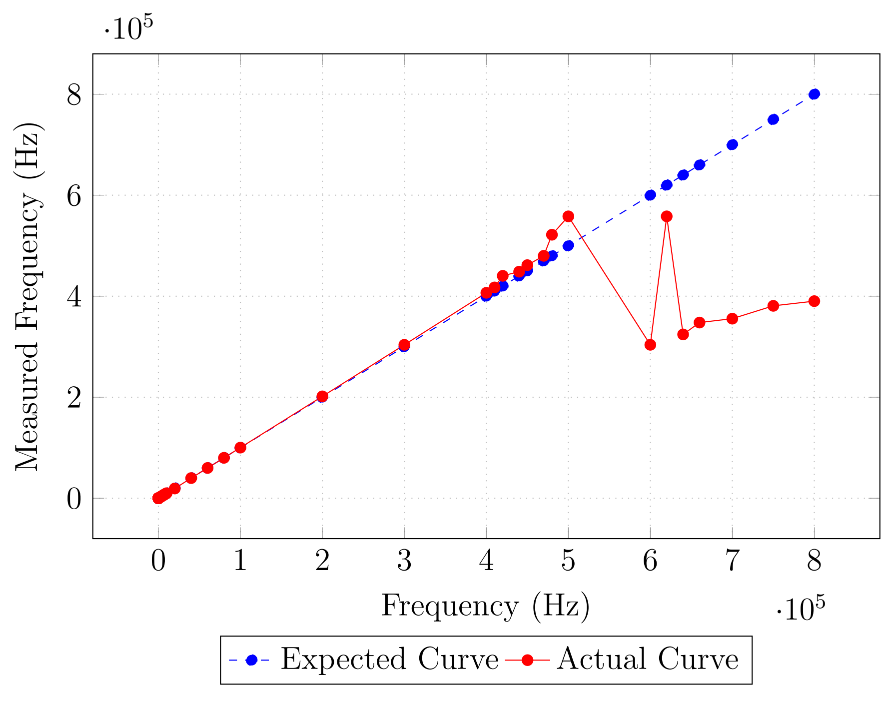

Figure 5: Graph of Function Generatior Frequency vs. Measured Frequency from the MCU

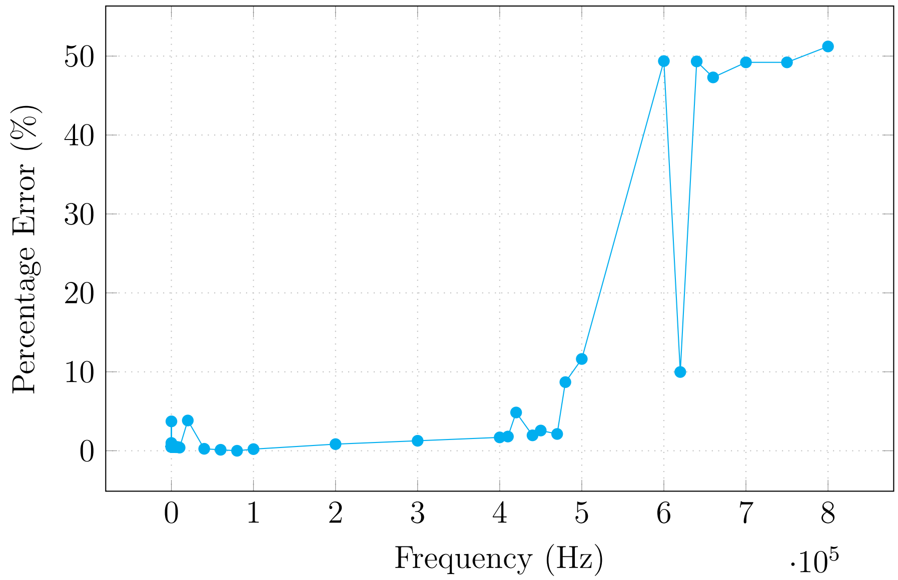

Figure 6: Graph of Percentage Error against Inputted Frequency

Judging by the graphs above, the system can only read up to 480 kHz, with an error of less than 9%. Therefore, the maximum frequency is **480 kHz**.

Higher frequencies have smaller time period, T. The system clock counts the number of counts (N) using the TIM2 counter between two rising edges using the prescaler value set at zero (i.e. at system clock frequency of 48 MHz). However, the system clock lacks adequate sampling frequency to measure very small periods, which makes the frequency measurement inaccurate.

For a 48 MHz clock, each clock cycle is **20.83 ns**. The number of counts, N, recorded by the timer for one signal period (T) is:

$$N = T \times f_{\text{clock}} \tag{1}$$

Where T is the period of the signal (in seconds), and f_{clock} is the system clock frequency (48 MHz).

For example, a high-frequency signal like 600 kHz, the period, T of the signal is:

$$T = \frac{1}{f_{\text{signal}}} = \frac{1}{600,000} = 1.6667\,\mu\text{s}$$

Therefore, the number of counts for this frequency are:

$$N = T \times f_{\text{clock}} = 1.6667 \times 10^{-6} \times 48 \times 10^{6} = 80\,\text{counts}$$

With only 80 counts, only one missed or extra clock cycle introduces significant error. For example, a missing **1 clock cycle** changes N to 79, resulting in:

$$\text{Frequency error} = \frac{f_{\text{clock}}}{N} - \frac{f_{\text{clock}}}{N-1}$$

$$\text{Error} = \left| \frac{48,000,000}{80} - \frac{48,000,000}{79} \right| \approx |600,000 - 607,595| = 7,595\,\text{Hz}$$

This results in a **quantization error** of approximately **1.27%** just from missing 1 clock cycle. However, it missed many clock cycles which gave a relative error of about 49.36%. It is very likely caused by the following criteria:

- **Interrupt Latency:** The microcontroller might not be able to process interrupts fast enough due to delays in handling and returning from interrupts. Therefore, the interrupt handling has caused some synchronization issues.
- **Jitter and Noise:** At such high frequencies, any signal noise or jitter can cause additional rising edges to be detected erroneously or edges to be missed altogether, introducing further inaccuracies.

To summarize, the system suceeded with the following results:

- **Accuracy:** Measured frequencies with a relative error within 9% from the function generator's reference values across a range of **0.0112 Hz to 480 kHz**.
- **Minimum Detectable Frequency: 0.0112 Hz**, limited by the timer's ability to measure long periods without overflow.
- **Maximum Detectable Frequency: 480 kHz**, constrained by EXTI interrupt latency and TIM2 resolution.

4.2.2 Resistance Measurement Performance

- **Accuracy:** Resistance calculations matched multimeter readings within 5% across the full range of the potentiometer (0–5 kΩ).
- **Response Time:** Real-time updates to the OLED display occurred with minimal latency, ensuring user-friendly interaction.

4.2.3 NE555 Timer Frequencies against Potentiometer Readings

The graph in Figure 7 below shows the NE555 timer frequency readings with varying potentiometer resistance. (The blue curve shows the frequencies measured from the MCU, and the red curve shows the actual frequency measured from the oscilloscope.)

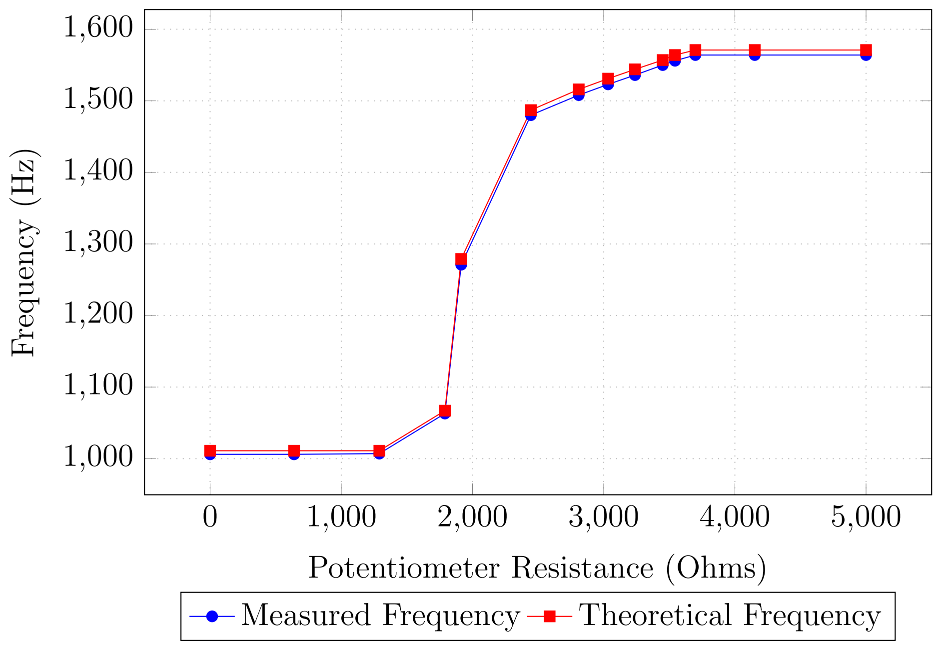

Figure 7: Graph of 555 Timer Frequency against Potentiometer Resistance

It is observed that the frequency remains constant at 1011 Hz until 1290 Ω. After 1290 Ω, the frequency skyrockets until at 3700 Ω, it gradually reaches a constant value at 1571 Hz. Therefore, with the DAC voltages generating from 0.126-2.23 V, the system can generate PWM signals through the NE555 timer at frequencies ranging from **1011 Hz** to **1571 Hz**.

4.2.4 Mode Switching Reliability

- **Seamless Transitions:** Mode switches occurred instantaneously, with accurate updates to frequency and resistance measurements.
- **Interrupt Handling:** No missed events or system crashes were observed during rapid mode toggling.

4.2.5 Noise Mitigation and Stability

- **Noise Rejection:** Effective decoupling capacitors [1] and robust interrupt-driven control minimized noise-induced errors.
- **Stable Operation:** The system maintained consistent performance even under noisy conditions and power supply variations.

4.2.6 System Limitations

- The maximum detectable frequency (480 kHz) is limited by EXTI and TIM2 latency, and further optimization may require higher-performance hardware.
- The ADC sampling rate (239.5 clock cycles) [6] restricts the system's ability to process rapidly changing analog inputs.

5 Discussion and Conclusion

5.1 Challenges

Throughout the development process, several challenges were encountered that required innovative solutions:

- **Signal Noise:** The presence of electrical noise in the ADC and DAC circuits impacted measurement accuracy, necessitating the implementation of decoupling capacitors and basic filtering strategies to stabilize readings.
- **Interrupt Synchronization:** Managing multiple interrupts for mode switching and signal frequency measurement introduced timing conflicts, which were mitigated through careful prioritization and interrupt-driven control logic.
- **Frequency Range Limitations:** It is observed from Section 4.2.1 that the frequency can be measured within a certain range. The system's maximum detectable frequency was constrained by the resolution of interrupt latency, presenting a challenge in achieving higher measurement ranges. Similarly, the system's minimum frequency is only limited by the auto-reload register of the clock timer.

5.2 Future Work

The current system provides a solid baseline, but there are several areas for enhancement to expand its capabilities:

- **Improved Sampling Rates:** Upgrading the ADC sampling rate and optimizing data acquisition would allow more precise measurements, particularly for rapidly varying signals.
- **Advanced Noise Reduction:** Implementing digital signal processing (DSP) techniques, such as low-pass filters or averaging algorithms, could further reduce the impact of noise on system performance.
- **Utilize timer chaining for extended range:** Combine multiple timers (e.g. TIM2, TIM3, TIM16, etc.) to measure very low frequencies without encountering timer overflow, extending the detectable frequency range.
- **Introduce dynamic prescaling:** Implement an adaptive prescaler to automatically adjust timer resolution based on signal frequency, ensuring better accuracy across a wide range of frequencies.
- **Extended Frequency Range:** Hardware upgrades or optimized software configurations could extend the system's maximum detectable frequency range, improving versatility.
- **Enhanced User Interface:** Expanding the OLED display to include graphical visualizations of signals and system states could improve user interaction.

5.3 Conclusion

For this lab project, an efficient and reliable system has been successfully developed for real-time signal measurement and generation. By leveraging the STM32F0 microcontroller and carefully integrating hardware and software components, the design meets the requirements for frequency and resistance measurement while maintaining responsiveness and precision at certain ranges. Despite challenges such as noise, latencies, and synchronization complexities, the system performed well under testing and offers a solid foundations for embedded systems technology. This work demonstrates the potential of embedded systems in real-time signal processing and lays the groundwork for beginner-level implementations.

6 References

[1] STMicroelectronics, *General-purpose single bipolar timers*, 2014. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab/supplement/555timer.pdf.

[2] V. Semiconductors, *Optocoupler, phototransistor output, with base connection*, 2010. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab/supplement/4n35.pdf.

[3] D. o. E. University of Victoria and C. Engineering, *Ece 355 lab manual*, 2024. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab.

[4] STMicroelectronics, *Stm32f051r8t6 datasheet*, 2014. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab/supplement/stm32f051r8t6_datatsheet_DM00039193.pdf.

[5] D. Rakhmatov, *I/o examples*, Adopted from STM32F0xx Advanced ARM-based 32-bit MCUs Reference Manual, © 2014 STMicroelectronics and other datasheets., University of Victoria, 2024. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab/iox.pdf.

[6] D. Rakhmatov, *Interfacing examples*, Adopted from STM32F0xx Advanced ARM-based 32-bit MCUs Reference Manual, © 2014 STMicroelectronics and other datasheets., University of Victoria, 2024. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab/interfacex.pdf.

[7] "Ssd1306 oled display driver datasheet," Solomon Systech, Tech. Rep., 2008. [Online]. Available: https://www.ece.uvic.ca/~ece355/lab/supplement/SSD1306.pdf.

[8] University of Victoria, Department of Electrical and Computer Engineering, *Equipment handbook for undergraduate students*, University of Victoria, 2018. [Online]. Available: https://www.uvic.ca/ecs/ece/assets/docs/current/undergraduate/Equipment-handbook.pdf.

7 Appendices

7.1 Main File

7.1.1 main.c

```c
//
// 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 "oled_screen.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).

/*****************************************************************/
/**                           PRAGMA                            **/
/*****************************************************************/

// 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"

/*****************************************************************/
/**                          DEFINES                            **/
/*****************************************************************/

/* 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 */ // Free running timer
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)

/* TEST PRINTS (FOR DEBUGGING PURPOSES) */
#define MAIN_DEBUG 1
#define ADC_DEBUG 0
#define FREQ_DEBUG 0
#define ENABLE_CAL 1 // allow calibration
#define TOGGLE_DEBUG 0
#define OUTPUT_DEBUG 0

/*****************************************************************/
/**                          TYPEDEFS                           **/
/*****************************************************************/

/*
 * These object-oriented structs are intended to be used as singletons
 * to wrap ADC, DAC, and SPI functionality.
 * This will make the code cleaner
 * This pattern is used in lots of system implementations,
 * for example the linux kernel
 */

/*****************************************************************/
/**                    FUNCTION PROTOTYPES                      **/
/*****************************************************************/

/********************* Under-the-hood functions ***************** */
void myGPIOA_Init(void);
void myTIM2_Init(void);
void EXTI_Init(void);
```

```

void toggle_mode(void);
void button_push(void);
void measure_frequency(unsigned int bit_number, unsigned int *
    var_address);

/************************* ADC Prototypes ************************
    */
void calibrate_ADC(void);
void myADC_Init(void);
uint32_t readADC(void);
unsigned int toOhms(uint32_t adc_val);

/************************* DAC Prototypes ************************
    */
void myDAC_init(void);
void writeDAC(uint32_t adc_val);

// 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.

/*** Call this function to boost the STM32F0xx clock to 48 MHz ***/

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();
}

/*****************************************************************/
/**                          Global Variables                    **/
```

```
/*****************************************************************/

volatile int funcGen_mode = 0; // 0 = NEC555 frequency; 1 = Function generator frequency
volatile uint32_t adc_value;

// Measured values
unsigned int resistance;
unsigned int ne555_frequency;
unsigned int fgen_frequency;

/*****************************************************************/
/**                            MAIN                             **/
/*****************************************************************/

int main(int argc, char *argv[])
{

    SystemClock48MHz();
    if (MAIN_DEBUG)
    {
        trace_printf("Arfaz and Aly's ECE355 Final Project\n");
        trace_printf("System clock: %u Hz\n\n", SystemCoreClock);
    }

    myGPIOA_Init(); // Initialize I/O port PA
    myTIM2_Init();  // Initialize timer TIM2
    EXTI_Init();    // Initialize EXTI

    myADC_Init(); // Initialize ADC
    myDAC_init(); // Initialize DAC

    oled_config();

    while (1)
    {
        adc_value = readADC();          // Read from the potentiometer
        resistance = toOhms(adc_value); // Convert the ADC value to resistance and updates it regularly
        writeDAC(adc_value);            // Writes the value

        // Display values to the LED screen
        if (!funcGen_mode)
        {
            // 1 - NE555 Timer
            refresh_OLED(resistance, ne555_frequency, 1);
```

```
        }
        else
        {
            // 2 - NE555 Timer
            refresh_OLED(resistance, fgen_frequency, 2);
        }
    }

    return 0;
}

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

    // MODER:
    //

    /* Configure PA0 (button) as input from the function generator
       */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= ~(GPIO_MODER_MODER0); // Clear bits PA0

    /* Configure PA1 (555 timer) as input from the function
       generator */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= ~(GPIO_MODER_MODER1); // Set the PA1 bits to 00
       (where 00 - input)

    /* Configure PA2 (function generator) as input from the function
        generator */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= ~(GPIO_MODER_MODER2); // Set the PA2 bits to 00
       (where 00 - input)

    // Set GPIO PA5 and PA4 to Analog Mode, (Or I can use 0x3 << 10)
        // 11 - Analog
    GPIOA->MODER |= GPIO_MODER_MODER4;
    GPIOA->MODER |= GPIO_MODER_MODER5;
    //  GPIOA->MODER |= 0xC00;  // Set GPIO Pin A to Analog Mode, (
       Or I can use 0x3 << 10)
    //  GPIOA->MODER |= 0x300;  // (or 0x3 << 8)

    /*Ensure no pull-up/pull-down for PA0*/
    //  GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);
```

```

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

void myTIM2_Init()
{
    /* Enable clock for TIM2 peripheral */
    // Relevant register: RCC->APB1ENR
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
    /* Configure TIM2: buffer auto-reload, count up, stop on overflow,
     * enable update events, interrupt on overflow only */
    // Relevant register: TIM2->CR1
    TIM2->CR1 = ((uint16_t)0x008C);
    /* Set clock prescaler value */
    TIM2->PSC = myTIM2_PRESCALER;
    /* Set auto-reloaded delay */
    TIM2->ARR = myTIM2_PERIOD;
    /* Update timer registers */
    // Relevant register: TIM2->EGR
    TIM2->EGR |= ((uint16_t)0x0001);
    /* Assign TIM2 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
    NVIC_SetPriority(TIM2_IRQn, 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;
}

void EXTI_Init()
{
    /* Map EXTI2 and EXTI0 line to PA2 and PA0 respectively */
    // Relevant register: SYSCFG -> EXTICR[0]
    SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI0 | SYSCFG_EXTICR1_EXTI1 | SYSCFG_EXTICR1_EXTI2);
    SYSCFG->EXTICR[0] |= (SYSCFG_EXTICR1_EXTI0_PA | SYSCFG_EXTICR1_EXTI1_PA | SYSCFG_EXTICR1_EXTI2_PA);

    // SYSCFG->EXTICR[0] &= 0xFF0F;

    /* EXTI2 and EXTI0 line interrupts: set rising-edge trigger */
    // Relevant register: EXTI->RTSR
```

```
    EXTI->RTSR |= (EXTI_RTSR_TR0 | EXTI_RTSR_TR1 | EXTI_RTSR_TR2);

    /* Unmask interrupts from EXTI2 and EXTI0 line */
    // Relevant register: EXTI->IMR
    EXTI->IMR |= (EXTI_IMR_IM0 | EXTI_IMR_IM1);

    /* Assign EXTI2 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
    NVIC_SetPriority(EXTI0_1_IRQn, 0);
    /* Enable EXTI2 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(EXTI0_1_IRQn);

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

/* 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 in TIM2! ***\n");

        TIM2->SR &= ~TIM_SR_UIF;  // Clear update interrupt flag
        TIM2->CR1 |= TIM_CR1_CEN; // Restart stopped timer
    }
}

/* Toggles between the mode for NE555 and the function generator */
void toggle_mode()
{
    // Simply flip the boolean
    funcGen_mode = !funcGen_mode;

    // Disable one of the interrupts
    if (!funcGen_mode)
    { // If using 555 timer
        EXTI->IMR &= ~(EXTI_IMR_IM2);
        EXTI->IMR |= EXTI_IMR_IM1;
    }
```

```c
    else
    {
        EXTI->IMR &= ~(EXTI_IMR_IM1);
        EXTI->IMR |= EXTI_IMR_IM2;
    }

    // Prints out to the console
    if (TOGGLE_DEBUG)
    {
        if (!funcGen_mode)
        {
            trace_printf("<<<< NEC555 TIMER >>>>\n");
        }
        else
        {
            trace_printf("<<<< FUNCTION GENERATOR >>>>\n");
        }
    }

    if (OUTPUT_DEBUG)
    {
        if (!funcGen_mode)
        {
            trace_printf("Resistance: %u\n", resistance);
            trace_printf("Frequency 1: %u\n\n", ne555_frequency);
        }
        else
        {
            trace_printf("Frequency 2: %u\n\n", fgen_frequency);
        }
    }
}

void button_push()
{
    // There is some change in the voltage value
    if ((EXTI->PR & EXTI_PR_PR0) != 0)
    {

        if ((GPIOA->IDR & GPIO_IDR_0) != 0)
        {
            // Wait for button to be released (PA0 = 0)
            while ((GPIOA->IDR & GPIO_IDR_0) != 0)
            {
            }

```

```
            // Trigger a function or a block of code here
                *************
            toggle_mode();
            //
                ********************************************************

        }
        // Clear pending interrupt flag
        EXTI->PR |= EXTI_PR_PR0;
    }
}

/* Measures the frequency and stores the value */
void measure_frequency(unsigned int bit_number, unsigned int *
   var_address)
{

    // Declare/initialize your local variables here...
    unsigned int count = 0;
    float period = 0;
    float frequency = 0;

    uint32_t register_mask = EXTI_PR_PR0 << bit_number;

    /* Check if EXTI2 interrupt pending flag is indeed set */
    if ((EXTI->PR & register_mask) != 0)
    {
        //
        // 1. If this is the first edge:
        //  - Clear count register (TIM2->CNT).
        //  - Start timer (TIM2->CR1).
        //    Else (this is the second edge):
        //  - Stop timer (TIM2->CR1).
        //  - Read out count register (TIM2->CNT).
        //  - Calculate signal period and frequency.
        //  - Print calculated values to the console.
        //    NOTE: Function trace_printf does not work
        //    with floating-point numbers: you must use
        //    "unsigned int" type to print your signal
        //    period and frequency.
        //

        // Start the TIM2 timer
        if ((TIM2->CR1 & TIM_CR1_CEN) == 0)
        {
            TIM2->CNT = 0;
            TIM2->CR1 |= TIM_CR1_CEN;
```

```c
        }
        else
        {
            TIM2->CR1 &= ~(TIM_CR1_CEN);
            count = TIM2->CNT;
            period = (float)count / (float)SystemCoreClock;
            frequency = 1 / period;

            //              trace_printf("Resistance: %u\n", resistance);
            *var_address = (unsigned int)(frequency);

            // Check if the frequency value is saved
            if (FREQ_DEBUG)
            {
                if (bit_number == 1)
                {
                    trace_printf("Frequency: %u\n", ne555_frequency);
                }
                else if (bit_number == 2)
                {
                    trace_printf("Frequency: %u\n", fgen_frequency);
                }
            }
        }

        // 2. Clear EXTI2 interrupt pending flag (EXTI->PR).
        // NOTE: A pending register (PR) bit is cleared
        // by writing 1 to it.
        //
        EXTI->PR |= register_mask; // Clear interrupt flag for the given bit number
    }
}

void EXTI0_1_IRQHandler()
{
    // processes the button push
    button_push();

    if (!funcGen_mode)
    { // If in 555 timer mode
        // Measure frequency from PA1 (555 timer)
        measure_frequency(1, &ne555_frequency);
    }
}
```

```c

/* This handler is declared in system/src/cmsis/vectors_stm32f051x8.
    c */
void EXTI2_3_IRQHandler()
{
    if (funcGen_mode)
    { // If in Function generator mode
        // Measure frequency from PA1 (555 timer)
        measure_frequency(2, &fgen_frequency);
    }
}

/*******************************************************************/
/**                           UTILITIES                           **/
/*******************************************************************/

/*** Initializing Analog to Digital Conversion ***/

/** Calibrates the ADC **/
void calibrate_ADC(void)
{
    if (ADC_DEBUG)
    {
        trace_printf("Start ADC Calibration\n");
    }
    ADC1->CR = ADC_CR_ADCAL; // Start ADC self-calibration process
    while (ADC1->CR == ADC_CR_ADCAL)
        ; // Wait until ADC calibration completes
    if (ADC_DEBUG)
    {
        trace_printf("Finished ADC calibration\n");
    }
}

/* Initializes the ADC to read values from a Potentiometer in Line 5
    */
void myADC_Init()
{
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; // Enabling ADC1 clock

    ADC1->SMPR = 0x7;                  // Set the sampling time to a
        maximum clock cycle (239.5 cycles)
    ADC1->CHSELR = ADC_CHSELR_CHSEL5; // Select channel 5 for ADC
        conversion

    // Calibrate the ADC
    if (ENABLE_CAL)
```

```
    {
        calibrate_ADC();
    }

    if (ADC_DEBUG)
    {
        trace_printf("Start Enabling ADC, waiting for acknowledgment
           ...\n");
    }

    ADC1->CR |= ADC_CR_ADEN; // Enable ADC by setting the ADEN bit
       high
    while (!(ADC1->ISR & ADC_ISR_ADRDY))
        ; // Wait until ADC is ready for conversion
    if (ADC_DEBUG)
    {
        trace_printf("ADC Enabled\n");
    }

    /// Set ADC to continuous conversion mode and overwrite old data
        on overrun
    ADC1->CFGR1 |= (ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD);
}

// convert ADC reading to resistance
unsigned int toOhms(uint32_t adc_val)
{
    // Rescaled the ADC Value to the resistance of the potentiometer
    // 4096 is the maximum 12-bit value from the ADC
    // Maximum resistance of the potentiometer = 5000 Ohm
    return (unsigned int)(((float)adc_val / 4095.0) * 5000.0);
}

/* Converts and reads the ADC value */
uint32_t readADC()
{
    /// Start the conversion process of ADC Control Register
    ADC1->CR |= ADC_CR_ADSTART; // ADC group regular conversion
       start

    /// Wait until the channel sampling is complete
    //  while (!(ADC1->ISR & ADC_ISR_EOSMP)); (Not necessary.)

    /// After sampling, wait until the ADC1's end-of-conversion (EOC
       ) flag is set.
    // Hardware sets this bit at the end of each conversion of a
    // channel when a new result is available in ADC_DR
```

```c
    // Hardware clears this bit when ADC_DR is read
    while (!(ADC1->ISR & ADC_ISR_EOC))
        ;

    /// Read the ADC result from DR
    // Retrieve the ADC value from the register; DR = Data Register
    return ADC1->DR;
}

/* Initializes the DAC to read values output from PA4 */
void myDAC_init()
{
    RCC->APB1ENR |= RCC_APB1ENR_DACEN; // Enable DAC Clock
    // DAC->CR &= 0xFFFFFFF8;               // Clear unwanted bits in the CR (TEN1 and BOFF1)

    // Enabling BOFF1 and TEN1 will prevent the frequency from changing
    DAC->CR &= ~(0x7);      // So, clear all the bits in the CR first (TEN1, EN1 and BOFF1)
    DAC->CR |= DAC_CR_EN1; // Switch the DAC enable bit and the trigger bit to 1
}

// Write to DAC
void writeDAC(uint32_t adc_val)
{
    // DHR12R1 is the 12-bit right-aligned data
    DAC->DHR12R1 = adc_val;
}

#pragma GCC diagnostic pop

/*****************************************************************/
```

Listing 13: Content of `main.c`

7.2 OLED Screen Header

7.2.1 oled_screen.h

```c
#ifndef OLED_SCREEN
#define OLED_SCREEN

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

```

#include "cmsis/cmsis_device.h"

// Constants
#define myTIM3_PRESCALER ((uint16_t)48000U)
#define myTIMx_PERIOD (0xFFFFFFFF)
#define STARTING_COL (uint8_t)1U
#define REFRESH_PERIOD ((uint16_t)100U)

// Function Prototypes
void myGPIOB_Init(void);
void mySPI_Init(void);
void myTIM3_Init();


void oled_Write(unsigned char);
void oled_Write_Cmd(unsigned char);
void oled_Write_Data(unsigned char);

void oled_config(void);
void set_Page(uint8_t page);
void set_Segment(uint8_t seg);

void refresh_OLED( unsigned int res, unsigned int freq, uint8_t
    freq_number );
void refresh_OLED_test(void);
void TIM3_delay(uint8_t milliseconds);


#endif
```

Listing 14: Content of `oled_screen.h`

7.3 OLED Screen Source

7.3.1 oled_screen.c

```
//
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//

#include "oled_screen.h"

SPI_HandleTypeDef SPI_Handle;

//
```

```c
// LED Display initialization commands
//
static unsigned char oled_init_cmds[] =
    {
        0xAE,
        0x20, 0x00,
        0x40,
        0xA0 | 0x01,
        0xA8, 0x40 - 1,
        0xC0 | 0x08,
        0xD3, 0x00,
        0xDA, 0x32,
        0xD5, 0x80,
        0xD9, 0x22,
        0xDB, 0x30,
        0x81, 0xFF,
        0xA4,
        0xA6,
        0xAD, 0x30,
        0x8D, 0x10,
        0xAE | 0x01,
        0xC0,
        0xA0};

//
// Character specifications for LED Display (1 row = 8 bytes = 1 ASCII character)
// Example: to display '4', retrieve 8 data bytes stored in Characters[52][X] row
//          (where X = 0, 1, ..., 7) and send them one by one to LED Display.
// Row number = character ASCII code (e.g., ASCII code of '4' is 0x34 = 52)
//
static 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, 0
       b00000000, 0b00000000, 0b00000000}, // y
    {0b01000100, 0b01100100, 0b01010100, 0b01001100, 0b01000100, 0
       b00000000, 0b00000000, 0b00000000}, // z
    {0b00000000, 0b00001000, 0b00110110, 0b01000001, 0b00000000, 0
       b00000000, 0b00000000, 0b00000000}, // {
    {0b00000000, 0b00000000, 0b01111111, 0b00000000, 0b00000000, 0
       b00000000, 0b00000000, 0b00000000}, // |
    {0b00000000, 0b01000001, 0b00110110, 0b00001000, 0b00000000, 0
       b00000000, 0b00000000, 0b00000000}, // }
    {0b00001000, 0b00001000, 0b00101010, 0b00011100, 0b00001000, 0
       b00000000, 0b00000000, 0b00000000}, // ~
    {0b00001000, 0b00011100, 0b00101010, 0b00001000, 0b00001000, 0
       b00000000, 0b00000000, 0b00000000}  // <-
};

// Initialize GPIOB pins
void myGPIOB_Init()
{

    // Enable the clock for the GPIOB peripheral
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    // Configure PB4, PB6 and PB7 as output
    GPIOB->MODER &= ~(GPIO_MODER_MODER4 | GPIO_MODER_MODER6 |
       GPIO_MODER_MODER7);      // Clear the bits
    GPIOB->MODER |= (GPIO_MODER_MODER4_0 | GPIO_MODER_MODER6_0 |
       GPIO_MODER_MODER7_0); // Set bits to output (01 = output)
    GPIOB->OTYPER &= ~(GPIO_OTYPER_OT_4 | GPIO_OTYPER_OT_6 |
       GPIO_OTYPER_OT_7);        // Set the output type as push-pull
    // Configure high-speed mode (11 - high-speed)
    GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR4 |
       GPIO_OSPEEDER_OSPEEDR6 | GPIO_OSPEEDER_OSPEEDR7);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4 | GPIO_PUPDR_PUPDR6 |
       GPIO_PUPDR_PUPDR7); // Ensure no pull-up/pull-down

    // Configure AF0 for PB3
    GPIOB->MODER &= ~(GPIO_MODER_MODER3);     // Clear PB3 bits
    GPIOB->MODER |= GPIO_MODER_MODER3_1;      // Set PB5 bits as
       alternate function (10 = AF0)
    GPIOB->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR3; // Set at high-speed
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3);     // Ensure no pull-up/
       pull-down

    // Configure AF0 for PB5
    GPIOB->MODER &= ~(GPIO_MODER_MODER5);     // Clear PB5 bits
```

```c
    GPIOB->MODER |= GPIO_MODER_MODER5_1;        // Set PB5 bits as alternate function (10 = AF0)
    GPIOB->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR5; // Set at high-speed
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR5);       // Ensure no pull-up/pull-down
}

// Initialize SPI1
void mySPI_Init(void)
{

    // Enable the SPI1 clock
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;

    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);
}

// Initialize TIM3
void myTIM3_Init()
{
    /* Enable clock for TIM2 peripheral */
    // Relevant register: RCC->APB1ENR
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
    /* Configure TIM2: buffer auto-reload, count up, stop on overflow,
     * enable update events, interrupt on overflow only */
    // Relevant register: TIM2->CR1
    TIM3->CR1 = ((uint16_t)0x008C);
    /* Set clock prescaler value */
    TIM3->PSC = myTIM3_PRESCALER - 1;
    /* Set auto-reloaded delay */
```

```
    TIM3->ARR = myTIMx_PERIOD;
    /* Update timer registers */
    // Relevant register: TIM2->EGR
    TIM3->EGR |= ((uint16_t)0x0001);
    /* Assign TIM2 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
    NVIC_SetPriority(TIM3_IRQn, 1);
    /* Enable TIM2 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(TIM3_IRQn);
    /* Enable update interrupt generation */
    // Relevant register: TIM2->DIER
    TIM3->DIER |= TIM_DIER_UIE;

    // Start the timer in TIM3
    if ((TIM3->CR1 & TIM_CR1_CEN) == 0)
    {
        TIM3->CNT = 0;
        TIM3->CR1 |= TIM_CR1_CEN;
    }
}

void TIM3_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM3->SR & TIM_SR_UIF) != 0)
    {
        trace_printf("\n*** Overflow in TIM3! ***\n");

        TIM3->SR &= ~TIM_SR_UIF;  // Clear update interrupt flag
        TIM3->CR1 |= TIM_CR1_CEN; // Restart stopped timer
    }
}

// Delay for about a couple of milliseconds
void TIM3_delay(uint8_t milliseconds)
{

    // Reset the timer
    TIM3->CNT = 0;

    // Start the timer in TIM3, if not running
    if ((TIM3->CR1 & TIM_CR1_CEN) == 0)
    {
        TIM3->CR1 |= TIM_CR1_CEN;
    }

```

```
    // Wait for the timer to reach the given amount of time
    while (TIM3->CNT < milliseconds)
        ;
    TIM3->CNT = 0;
}

//
// LED Display Functions
//

// Prints a string/buffer to the LED Screen
// Params: buffer string, page, starting_column (each column occupies 8 segments to store a character)
void oled_print(unsigned char buffer[17], uint8_t page, uint8_t starting_column)
{
    /* The buffer contains your character ASCII codes for LED Display */

    // Initialize the ascii_code and the segment, starting from the 'starting_column'
    uint8_t segment = starting_column * 8;
    uint8_t ascii_code;

    // - select PAGE (LED Display line)
    set_Page(page);

    // Iterate through each character from the buffer
    for (uint8_t char_index = 0; char_index < strlen((const char *)buffer); char_index++)
    {

        // Grab the ASCII code from the buffer
        ascii_code = (uint8_t)(buffer[char_index]);

        /*
            - for each c = ASCII code = Buffer[0], Buffer[1], ...,
            send 8 bytes in Characters[c][0-7] to LED Display
        */

        // Iterate through each byte from the Characters array
        for (uint8_t byte_index = 0; byte_index < 8; byte_index++)
        {
            set_Segment(segment++);                              // Don't forget to set the segment
            oled_Write_Data(Characters[ascii_code][byte_index]); // Write the data to the SDDRAM memory of the OLED
```

```c
        }
    }
}

// PARAMS: Resistance, frequency, and the frequency number
// The frequency number is to reference a device which generates square waves
// In this project, 1 - NE555 timer, 2 - Function generator
void refresh_OLED(unsigned int res, unsigned int freq, uint8_t freq_number)
{
    // Buffer size = at most 16 characters per PAGE + terminating '\0'
    unsigned char Buffer[17];

    if (TIM3->CNT >= REFRESH_PERIOD - 1)
    {
        // Update the buffer and then print it out
        snprintf((char *)Buffer, sizeof(Buffer), "R: %6u Ohms", res);
        oled_print(Buffer, 2, STARTING_COL);

        // Do it again, but this time, for the frequency
        snprintf((char *)Buffer, sizeof(Buffer), "F%u: %5u Hz  ", freq_number, freq);
        oled_print(Buffer, 4, STARTING_COL);

        TIM3->CNT = 0;
    }
}

// Used for testing purposes
void refresh_OLED_test(void)
{
    unsigned char Buffer[17];
    unsigned int count = 0;

    while (count <= 200)
    {
        if (TIM3->CNT >= 500 - 1)
        {
            // Update the buffer and then print it out
            snprintf((char *)Buffer, sizeof(Buffer), "PEEKABOO!");
            oled_print(Buffer, 1, STARTING_COL);

            snprintf((char *)Buffer, sizeof(Buffer), "I see you!");
            oled_print(Buffer, 3, STARTING_COL);
```

```

            snprintf((char *)Buffer, sizeof(Buffer), "Count: %d", count);
            oled_print(Buffer, 5, STARTING_COL);

            TIM3->CNT = 0;
            count++;
        }
    }
    TIM3->CR1 &= ~(TIM_CR1_CEN);
}

// Sets a page (row) for the data to be written
void set_Page(uint8_t page)
{
    oled_Write_Cmd(0xB0 | (page & 0x7)); // Select PAGE
}

// Selects a segment (column) for the data to be written
void set_Segment(uint8_t seg)
{
    oled_Write_Cmd(0x00 | (seg & 0xF));        // Take the lower half of the SEG
    oled_Write_Cmd(0x10 | ((seg >> 4) & 0xF)); // Take the upper half of the SEG
}

void oled_Write_Cmd(unsigned char cmd)
{
    GPIOB->ODR |= GPIO_ODR_6;    // make PB6 = CS# = 1
    GPIOB->ODR &= ~(GPIO_ODR_7); // make PB7 = D/C# = 0      <== That's where the difference is
    GPIOB->ODR &= ~(GPIO_ODR_6); // make PB6 = CS# = 0
    oled_Write(cmd);
    GPIOB->ODR |= GPIO_ODR_6; // make PB6 = CS# = 1
}

void oled_Write_Data(unsigned char data)
{
    GPIOB->ODR |= GPIO_ODR_6;    // make PB6 = CS# = 1
    GPIOB->ODR |= GPIO_ODR_7;    // make PB7 = D/C# = 1  <== That's where the difference is
    GPIOB->ODR &= ~(GPIO_ODR_6); // make PB6 = CS# = 0
    oled_Write(data);
    GPIOB->ODR |= GPIO_ODR_6; // make PB6 = CS# = 1
}

```

```c
void oled_Write(unsigned char Value)
{

    /* Wait until SPI1 is ready for writing (TXE = 1 in SPI1_SR) */
    while ((SPI1->SR & SPI_SR_TXE) == 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) */
    while ((SPI1->SR & SPI_SR_TXE) == 0)
        ;
}

void oled_config(void)
{
    // Initialize GPIOB pins
    myGPIOB_Init();

    // Initialize SPI
    mySPI_Init();

    // Initialize TIM3
    myTIM3_Init();

    // Set PB4 to low (0) to initiate the reset
    GPIOB->BSRR = GPIO_BSRR_BR_4; // Clear bit 4
                                  //    GPIOB->ODR &= ~(GPIO_ODR_4);
                                       // Clear bit 4 (PB4 = 0)
    TIM3_delay(10);               // Wait for 10 milliseconds

    GPIOB->BSRR = GPIO_BSRR_BS_4; // Set bit 4 back to HIGH
                                  //    GPIOB->ODR |= GPIO_ODR_4;
                                         // Set bit 4 back to 1
    TIM3_delay(10);               // Wait for 10 milliseconds

    //
    // 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 (uint8_t page = 0; page < 8; page++)
    {
        set_Page(page); // Set the page / row

        // set starting SEG = 0, and call oled_Write_Data( 0x00 )
           128 times
        for (uint8_t seg = 0; seg < 128; seg++)
        {
            set_Segment(seg);
            oled_Write_Data(0x00);
        }
    }
}

#pragma GCC diagnostic pop

/*****************************************************************/
```

Listing 15: Content of `oled_screen.c`