

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	3
1 Problem Description and Technical Specifications	4
1.1 Objectives	4
1.2 Technical Specifications	5
2 Design and Solution	7
2.1 System Overview	7
2.1.1 Signal Processing Components	7
2.1.2 Operational Modes	7
2.1.3 System Integration	7
2.2 Hardware Design	8
2.2.1 Block Diagram	8
2.2.2 Hardware Components [3]	9
2.2.3 Pin Configuration [3]	9
2.2.4 Power Distribution	10
3 Software Design	11
3.1 Initialization Functions	11
3.1.1 System Clock	11
3.1.2 GPIO Initialization	11
3.1.3 ADC and DAC Initialization	12
3.1.4 Timer (TIM2) Initialization	13
3.1.5 External Interrupt Initialization	13
3.2 Core Logic	14
3.2.1 Signal Measurement and Reconstruction	14
3.2.2 Frequency and Resistance Computation	14
3.2.3 Mode Switching Logic	15
3.2.4 Button Press Handling [5]	15
3.3 Utilities	16
3.3.1 Timer Interrupt Handling [5]	16
3.3.2 Value Computation	16
3.3.3 External Interrupt Handlers	16
3.4 Key Features	17
3.4.1 Signal Measurement and Generation	17
3.4.2 Signal Generation	17
3.4.3 Mode Switching	18
3.4.4 Synchronization	18
3.4.5 OLED Screen Operation	18
3.5 Peripheral Justification	20

4	Testing and Results	21
4.1	Testing Procedure	21
4.1.1	System Initialization and Peripheral Verification	21
4.1.2	Frequency Measurement Validation	21
4.1.3	Resistance Calculation Validation	22
4.1.4	Mode Switching Functionality	22
4.1.5	Signal Monitoring and DAC Control	22
4.1.6	Noise and Stability Testing	22
4.2	Results	23
4.2.1	Frequency Measurement Performance	23
4.2.2	Resistance Measurement Performance	25
4.2.3	NE555 Timer Frequencies against Potentiometer Readings	25
4.2.4	Mode Switching Reliability	26
4.2.5	Noise Mitigation and Stability	26
4.2.6	System Limitations	26
5	Discussion and Conclusion	27
5.1	Challenges	27
5.2	Future Work	27
5.3	Conclusion	28
6	References	29
7	Appendices	30
7.1	Main File	30
7.1.1	<code>main.c</code>	30
7.2	OLED Screen Header	42
7.2.1	<code>oled_screen.h</code>	42
7.3	OLED Screen Source	43
7.3.1	<code>oled_screen.c</code>	43

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

[Opt] 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.

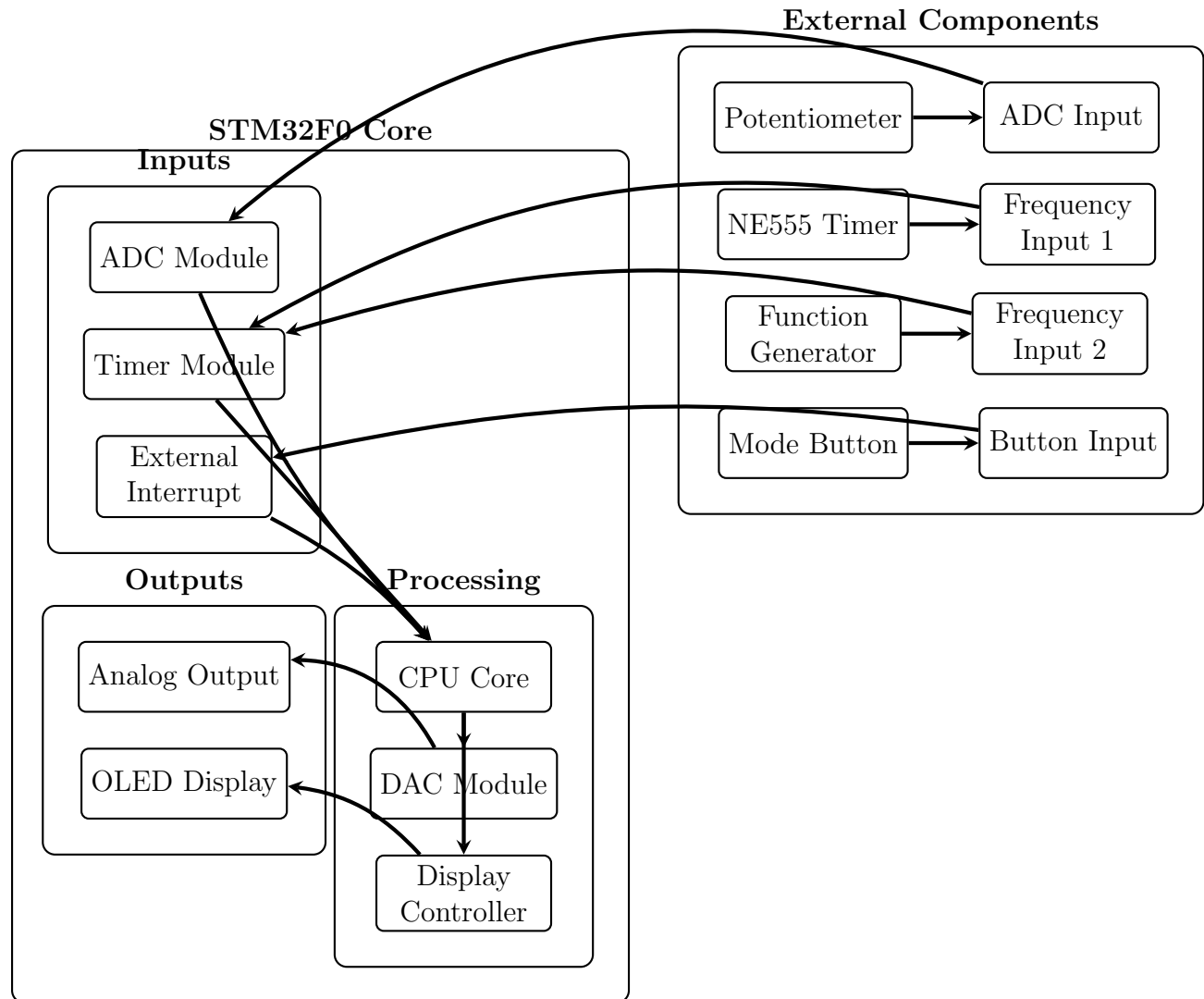


Figure 1: STM32F0 Microcontroller System Block Diagram

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)

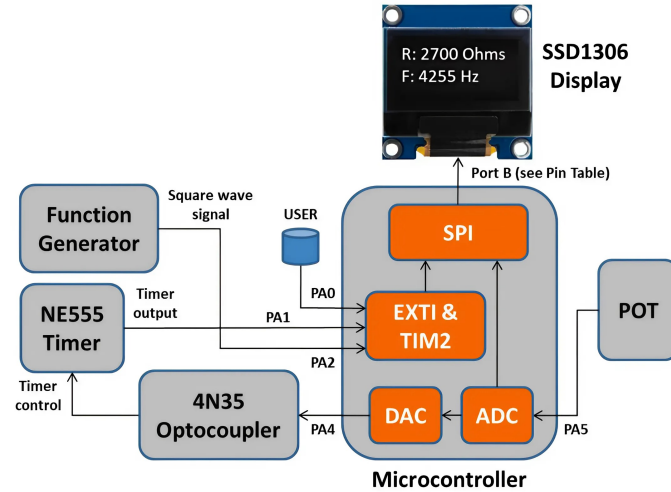


Figure 2: System architecture showing the interaction between major components of the frequency measurement and signal generation system [3]

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

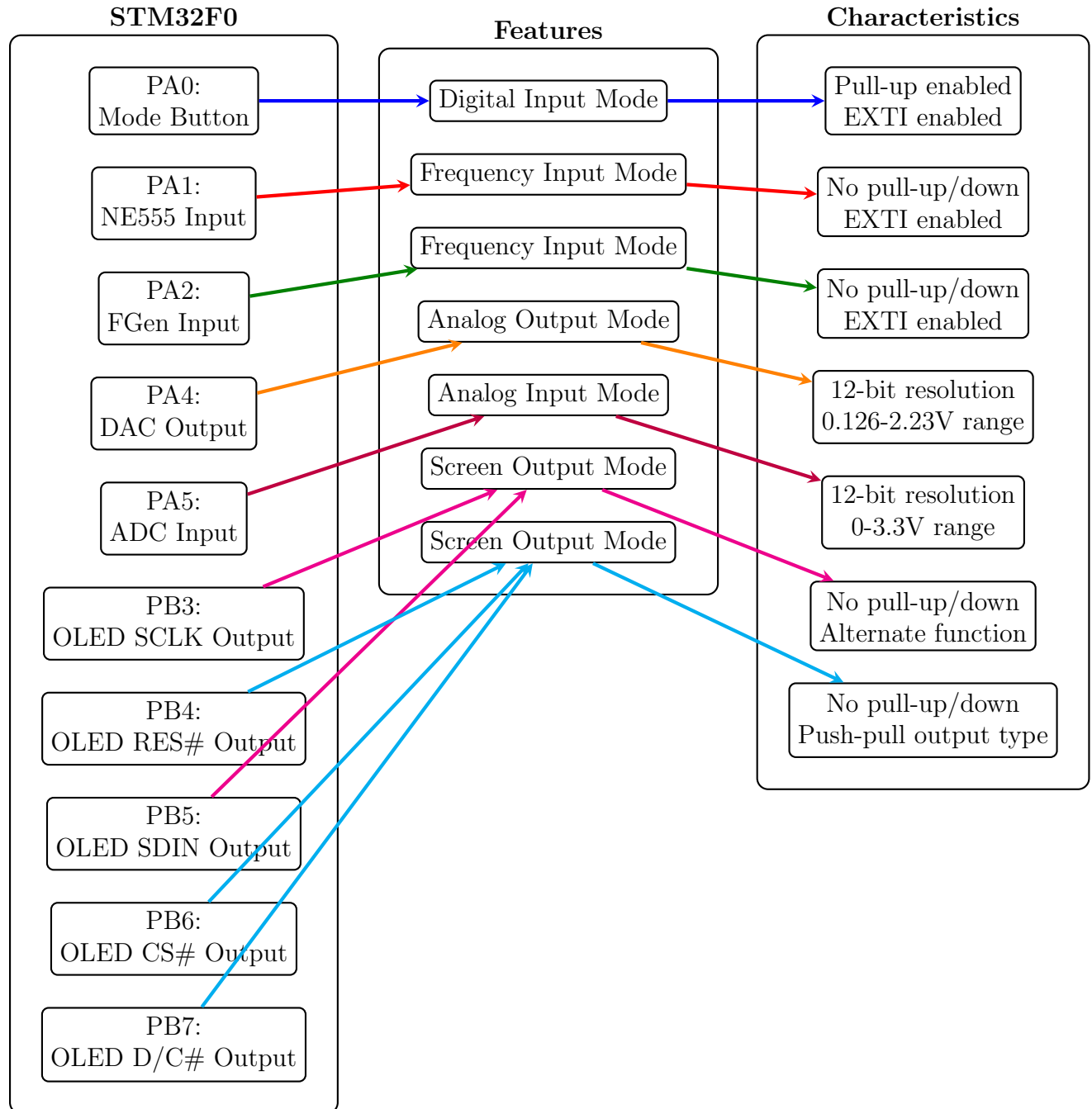


Figure 3: Pin Configurations of the GPIO Pins [3]

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

```
1 void SystemClock48MHz(void) {
2     // Disable the PLL
3     RCC->CR &= ~(RCC_CR_PLLON);
4     // Wait for the PLL to unlock
5     while ((RCC->CR & RCC_CR_PLLRDY) != 0);
6     // Configure the PLL for a 48 MHz system clock
7     RCC->CFGR = 0x00280000;
8     // Enable the PLL
9     RCC->CR |= RCC_CR_PLLON;
10    // Wait for the PLL to lock
11    while ((RCC->CR & RCC_CR_PLLRDY) != RCC_CR_PLLRDY);
12    // Switch to the PLL as the clock source
13    RCC->CFGR = (RCC->CFGR & (~RCC_CFGR_SW_Msk)) | RCC_CFGR_SW_PLL;
14    // Update the system clock variable
15    SystemCoreClockUpdate();
16 }
```

Listing 1: System Clock Initialization Function [5]

3.1.2 GPIO Initialization

```
1 void myGPIOA_Init(void) {
2     // Enable GPIOA clock
3     RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
4
5     // Configure PA0 (button) as input
6     GPIOA->MODER &= ~(GPIO_MODER_MODER0);
7
8     // Configure PA1 (555 timer) as input
9     GPIOA->MODER &= ~(GPIO_MODER_MODER1);
10
11    // Configure PA2 (function generator) as input
```

```
12     GPIOA->MODER &= ~(GPIO_MODER_MODER2);
13
14     // Configure PA4 and PA5 as analog mode
15     GPIOA->MODER |= GPIO_MODER_MODER4;
16     GPIOA->MODER |= GPIO_MODER_MODER5;
17
18     // Ensure no pull-up/pull-down for PA1 and PA2
19     GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1 | GPIO_PUPDR_PUPDR2);
20 }
```

Listing 2: GPIO Port A Initialization Function [5]

3.1.3 ADC and DAC Initialization

```
1 void myADC_Init(void) {
2     // Enable ADC clock
3     RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;
4
5     // Configure ADC settings
6     ADC1->SMPR = 0x7; // Maximum sampling time
7     ADC1->CHSELR = ADC_CHSELR_CHSEL5; // Select channel 5
8
9     // Calibrate ADC if enabled
10    if (ENABLE_CAL) {
11        ADC1->CR = ADC_CR_ADCAL;
12        while (ADC1->CR == ADC_CR_ADCAL);
13    }
14
15    // Enable ADC and wait for ready
16    ADC1->CR |= ADC_CR_ADEN;
17    while (!(ADC1->ISR & ADC_ISR_ADRDY));
18
19    // Configure continuous conversion mode
20    ADC1->CFGR1 |= (ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD);
21 }
22
23 void myDAC_init(void) {
24     // Enable DAC Clock
25     RCC->APB1ENR |= RCC_APB1ENR_DACEN;
26
27     // Clear and configure DAC control register
28     DAC->CR &= ~(0x7);
29     DAC->CR |= DAC_CR_EN1;
30 }
```

Listing 3: ADC and DAC Initialization Function [6]

3.1.4 Timer (TIM2) Initialization

```
1 void myTIM2_Init(void) {
2     // Enable clock for TIM2
3     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
4
5     // Configure TIM2
6     TIM2->CR1 = ((uint16_t)0x008C);
7     TIM2->PSC = myTIM2_PRESCALER;
8     TIM2->ARR = myTIM2_PERIOD;
9
10    // Update timer registers
11    TIM2->EGR |= ((uint16_t)0x0001);
12
13    // Configure and enable interrupts
14    NVIC_SetPriority(TIM2_IRQn, 0);
15    NVIC_EnableIRQ(TIM2_IRQn);
16    TIM2->DIER |= TIM_DIER_UIE;
17 }
```

Listing 4: Timer 2 Initialization Function [5]

3.1.5 External Interrupt Initialization

```
1 void EXTI_Init(void) {
2     // Map EXTI2 and EXTIO lines to PA2 and PA0 respectively
3     SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTIO |
4         SYSCFG_EXTICR1_EXTI1 | SYSCFG_EXTICR1_EXTI2);
5     SYSCFG->EXTICR[0] |= (SYSCFG_EXTICR1_EXTIO_PA |
6         SYSCFG_EXTICR1_EXTI1_PA | SYSCFG_EXTICR1_EXTI2_PA);
7
8     // Set rising-edge trigger for EXTI2 and EXTIO lines
9     EXTI->RTSR |= (EXTI_RTSR_TR0 | EXTI_RTSR_TR1 | EXTI_RTSR_TR2);
10
11    // Unmask interrupts from EXTI2 and EXTIO lines
12    EXTI->IMR |= (EXTI_IMR_IM0 | EXTI_IMR_IM1);
13
14    // Configure interrupt priorities and enable in NVIC
15    NVIC_SetPriority(EXTIO_1_IRQn, 0);
16    NVIC_EnableIRQ(EXTIO_1_IRQn);
17
18    NVIC_SetPriority(EXTI2_3_IRQn, 1);
19    NVIC_EnableIRQ(EXTI2_3_IRQn);
20 }
```

Listing 5: EXTI Initialization Function [5]

3.2 Core Logic

3.2.1 Signal Measurement and Reconstruction

```

1  // ADC Reading (Measurement)
2  uint32_t readADC(void) {
3      // Start ADC conversion
4      ADC1->CR |= ADC_CR_ADSTART;
5
6      // Wait for conversion completion
7      while (!(ADC1->ISR & ADC_ISR_EOC));
8
9      // Return the ADC result
10     return ADC1->DR;
11 }
12
13 // Write to DAC (Reconstruction)
14 void writeDAC(uint32_t adc_val)
15 {
16     // DHR12R1 is the 12-bit right-aligned data
17     DAC->DHR12R1 = adc_val;
18 }

```

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

3.2.2 Frequency and Resistance Computation

```

1  void measure_frequency(unsigned int bit_number, unsigned int*
   var_address) {
2      // Assign some necessary variables
3      unsigned int count = 0;
4      float period = 0;
5      float frequency = 0;
6      // Compute the register mask by bit number
7      uint32_t register_mask = EXTI_PR_PRO << bit_number;
8
9      /* Check if EXTI2 interrupt pending flag is indeed set */
10     if ((EXTI->PR & register_mask) != 0) {
11         // If this is the first 'rising' edge:
12         if ((TIM2->CR1 & TIM_CR1_CEN) == 0) {
13             TIM2->CNT = 0;    // Clear count register
14             TIM2->CR1 |= TIM_CR1_CEN;    // Start the timer
15         } else {    // Otherwise, this is the second edge
16             TIM2->CR1 &= ~(TIM_CR1_CEN);    // Stop timer
17             // Read out count register
18             count = TIM2->CNT;
19             // Calculate the frequency

```

```

20         period = (float)count / (float)SystemCoreClock;
21         frequency = 1 / period;
22         // 'var_address' - variable where the frequency is
           stored
23         // Update the 'var_address'
24         *var_address = (unsigned int)(frequency);
25     }
26     EXTI->PR |= register_mask;
27 }
28 }

```

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.

```

1 void toggle_mode(void) {
2     // Toggle the mode
3     funcGen_mode = !funcGen_mode;
4
5     // Enable or disable interrupts based on the mode
6     if (!funcGen_mode) { // NE555 timer mode
7         EXTI->IMR &= ~(EXTI_IMR_IM2);
8         EXTI->IMR |= EXTI_IMR_IM1;
9     } else { // Function generator mode
10        EXTI->IMR &= ~(EXTI_IMR_IM1);
11        EXTI->IMR |= EXTI_IMR_IM2;
12    }
13
14    // Debug output (optional)
15    if (TOGGLE_DEBUG) {
16        trace_printf(funcGen_mode ? "<<<< FUNCTION GENERATOR >>>>\n"
17                       : "<<<< NE555 TIMER >>>>\n");
18    }
19 }

```

Listing 8: Toggle Mode Function

3.2.4 Button Press Handling [5]

```

1 void button_push(void) {
2     // Check for a pending interrupt on PA0
3     if ((EXTI->PR & EXTI_PR_PRO) != 0) {
4         if ((GPIOA->IDR & GPIO_IDR_0) != 0) {
5             // Wait for button release

```



```

6         while ((GPIOA->IDR & GPIO_IDR_0) != 0) {}
7
8         // Trigger the mode toggle
9         toggle_mode();
10    }
11    // Clear the pending interrupt flag
12    EXTI->PR |= EXTI_PR_PRO;
13 }
14 }

```

Listing 9: Button Push Handler Function

3.3 Utilities

3.3.1 Timer Interrupt Handling [5]

```

1 void TIM2_IRQHandler(void) {
2     if ((TIM2->SR & TIM_SR_UIF) != 0) {
3         // Handle timer overflow
4         trace_printf("\n*** Overflow in TIM2! ***\n");
5
6         // Clear interrupt flag and restart the timer
7         TIM2->SR &= ~TIM_SR_UIF;
8         TIM2->CR1 |= TIM_CR1_CEN;
9     }
10 }

```

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:

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

```

1 unsigned int toOhms(uint32_t adc_val) {
2     return (unsigned int)((float)adc_val/4095.0) * 5000.0;
3 }

```

Listing 11: Resistance Calculation Function

3.3.3 External Interrupt Handlers

```

1 void EXTI0_1_IRQHandler(void) {
2     // Handle button press
3     button_push();

```

```

4
5 // Measure frequency from PA1 if in NE555 timer mode
6 if (!funcGen_mode) {
7     measure_frequency(1, &ne555_frequency);
8 }
9 }
10
11 void EXTI2_3_IRQHandler(void) {
12     // Measure frequency from PA2 if in function generator mode
13     if (funcGen_mode) {
14         measure_frequency(2, &fgen_frequency);
15     }
16 }

```

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:

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

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

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

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 $\frac{4800}{48000 \text{ Hz}} = 100 \text{ ms}$ [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 k Ω).
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:

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

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

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

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.

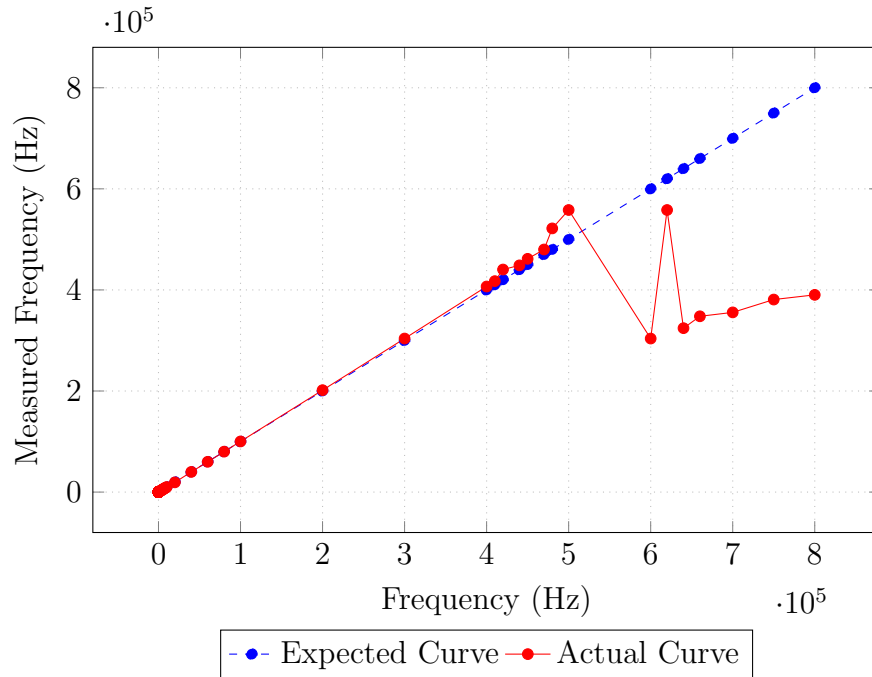


Figure 5: Graph of Function Generator Frequency vs. Measured Frequency from the MCU

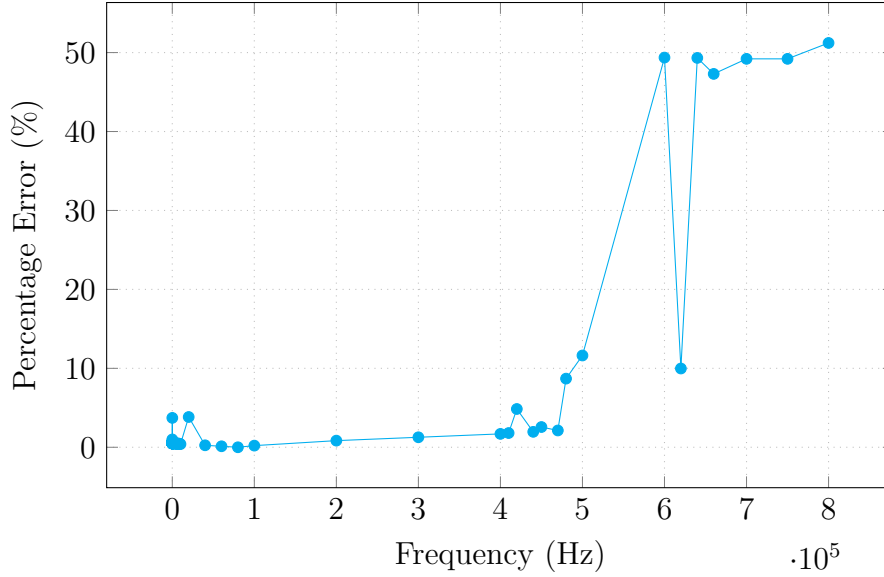


Figure 6: Graph of Percentage Error against Inputted Frequency

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:

$$N = T \times f_{\text{clock}} \quad (1)$$

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:

$$\begin{aligned} \text{Frequency error} &= \frac{f_{\text{clock}}}{N} - \frac{f_{\text{clock}}}{N-1} \\ \text{Error} &= \frac{48,000,000}{80} - \frac{48,000,000}{79} \approx 600,000 - 607,595 = 7,595 \text{ Hz} \end{aligned}$$

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 succeeded 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 k Ω).
- **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.)

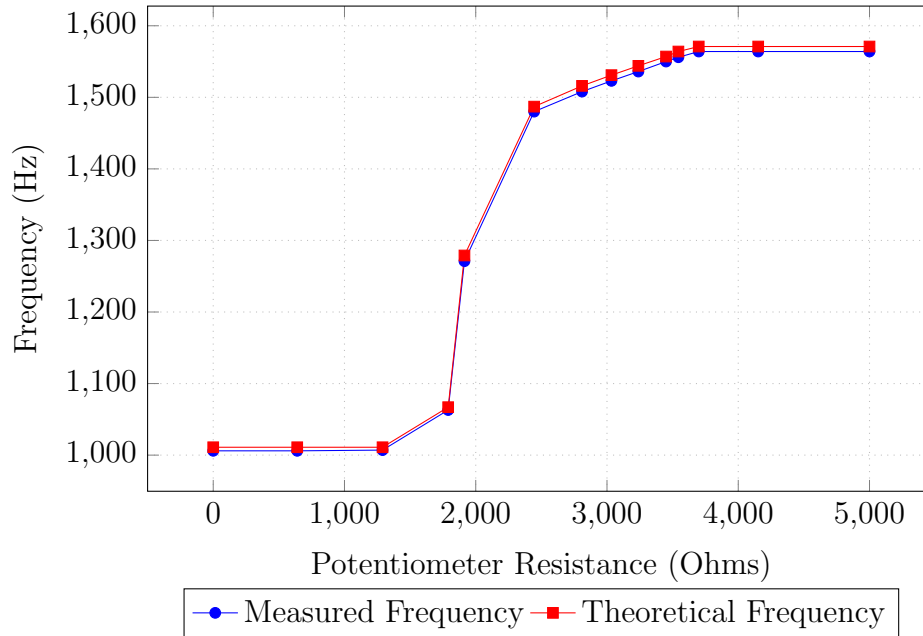


Figure 7: Graph of 555 Timer Frequency against Potentiometer Resistance

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

This project successfully developed a modular, reliable system 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. Despite challenges such as noise and synchronization complexities, the system performed well under testing and offers a strong platform for future development. This work demonstrates the potential of embedded systems in real-time signal processing and lays the groundwork for advanced 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

```
1  //
2  // This file is part of the GNU ARM Eclipse distribution.
3  // Copyright (c) 2014 Liviu Ionescu.
4  //
5
6  /*****
7  // School: University of Victoria, Canada.
8  // Course: ECE 355 "Microprocessor-Based Systems".
9  // This is template code for Part 2 of Introductory Lab.
10 //
11 // See "system/include/cmsis/stm32f051x8.h" for register/bit
    definitions.
12 // See "system/src/cmsis/vectors_stm32f051x8.c" for handler
    declarations.
13 /*****/
14
15 #include <stdio.h>
16 #include "diag/Trace.h"
17 #include "cmsis/cmsis_device.h"
18 #include "oled_screen.h"
19
20 /*****/
21 //
22 // STM32F0 empty sample (trace via $(trace)).
23 //
24 // Trace support is enabled by adding the TRACE macro definition.
25 // By default the trace messages are forwarded to the $(trace)
    output,
26 // but can be rerouted to any device or completely suppressed, by
27 // changing the definitions required in system/src/diag/trace_impl.c
28 // (currently OS_USE_TRACE_ITM, OS_USE_TRACE_SEMIHOSTING_DEBUG/
    _STDOUT).
29
30 /*****/
31 /**                                PRAGMA                                **/
32 /*****/
33
34 // Sample pragmas to cope with warnings. Please note the related
    line at
35 // the end of this function, used to pop the compiler diagnostics
    status.
```

```

36 #pragma GCC diagnostic push
37 #pragma GCC diagnostic ignored "-Wunused-parameter"
38 #pragma GCC diagnostic ignored "-Wmissing-declarations"
39 #pragma GCC diagnostic ignored "-Wreturn-type"
40
41 /*****
42 /**                                DEFINES                                **/
43 *****/
44
45 /* Definitions of registers and their bits are
46    given in system/include/cmsis/stm32f051x8.h */
47
48 /* Clock prescaler for TIM2 timer: no prescaling */
49 #define myTIM2_PRESCALER ((uint16_t)0x0000)
50 /* Maximum possible setting for overflow */ // Free running timer
51 #define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)
52
53 /* TEST PRINTS (FOR DEBUGGING PURPOSES) */
54 #define MAIN_DEBUG 1
55 #define ADC_DEBUG 0
56 #define FREQ_DEBUG 0
57 #define ENABLE_CAL 1 // allow calibration
58 #define TOGGLE_DEBUG 0
59 #define OUTPUT_DEBUG 0
60
61 /*****
62 /**                                TYPEDEFS                                **/
63 *****/
64
65 /*
66  * These object-oriented structs are intended to be used as
67  * singletons
68  * to wrap ADC, DAC, and SPI functionality.
69  * This will make the code cleaner
70  * This pattern is used in lots of system implementations,
71  * for example the linux kernel
72  */
73
74 /*****
75 /**                                FUNCTION PROTOTYPES                                **/
76 *****/
77
78 /****** Under-the-hood functions *****/
79
80 void myGPIOA_Init(void);
81 void myTIM2_Init(void);
82 void EXTI_Init(void);

```



```

81
82 void toggle_mode(void);
83 void button_push(void);
84 void measure_frequency(unsigned int bit_number, unsigned int *
    var_address);
85
86 ***** ADC Prototypes *****
    */
87 void calibrate_ADC(void);
88 void myADC_Init(void);
89 uint32_t readADC(void);
90 unsigned int to0hms(uint32_t adc_val);
91
92 ***** DAC Prototypes *****
    */
93 void myDAC_init(void);
94 void writeDAC(uint32_t adc_val);
95
96 // Declare/initialize your global variables here...
97 // NOTE: You'll need at least one global variable
98 // (say, timerTriggered = 0 or 1) to indicate
99 // whether TIM2 has started counting or not.
100
101 *** Call this function to boost the STM32F0xx clock to 48 MHz ***
102
103 void SystemClock48MHz(void)
104 {
105     // Disable the PLL
106     RCC->CR &= ~(RCC_CR_PLLON);
107     // Wait for the PLL to unlock
108     while ((RCC->CR & RCC_CR_PLLRDY) != 0)
109         ;
110     // Configure the PLL for 48-MHz system clock
111     RCC->CFGR = 0x00280000;
112     // Enable the PLL
113     RCC->CR |= RCC_CR_PLLON;
114     // Wait for the PLL to lock
115     while ((RCC->CR & RCC_CR_PLLRDY) != RCC_CR_PLLRDY)
116         ;
117     // Switch the processor to the PLL clock source
118     RCC->CFGR = (RCC->CFGR & (~RCC_CFGR_SW_Msk)) | RCC_CFGR_SW_PLL;
119     // Update the system with the new clock frequency
120     SystemCoreClockUpdate();
121 }
122
123 *****
124 **                               Global Variables                               **

```

```

125  *****
126
127  volatile int funcGen_mode = 0; // 0 = NEC555 frequency; 1 = Function
      generator frequency
128  volatile uint32_t adc_value;
129
130  // Measured values
131  unsigned int resistance;
132  unsigned int ne555_frequency;
133  unsigned int fgen_frequency;
134
135  *****
136  /**                                MAIN                                **/
137  *****
138
139  int main(int argc, char *argv[])
140  {
141
142      SystemClock48MHz();
143      if (MAIN_DEBUG)
144      {
145          trace_printf("Arfaz and Aly's ECE355 Final Project\n");
146          trace_printf("System clock: %u Hz\n\n", SystemCoreClock);
147      }
148
149      myGPIOA_Init(); // Initialize I/O port PA
150      myTIM2_Init(); // Initialize timer TIM2
151      EXTI_Init(); // Initialize EXTI
152
153      myADC_Init(); // Initialize ADC
154      myDAC_init(); // Initialize DAC
155
156      oled_config();
157
158      while (1)
159      {
160          adc_value = readADC(); // Read from the
              potentiometer
161          resistance = to0hms(adc_value); // Convert the ADC value to
              resistance and updates it regularly
162          writeDAC(adc_value); // Writes the value
163
164          // Display values to the LED screen
165          if (!funcGen_mode)
166          {
167              // 1 - NE555 Timer
168              refresh_OLED(resistance, ne555_frequency, 1);

```

```

169     }
170     else
171     {
172         // 2 - NE555 Timer
173         refresh_OLED(resistance, fgen_frequency, 2);
174     }
175 }
176
177 return 0;
178 }
179
180 void myGPIOA_Init()
181 {
182     /* Enable clock for GPIOA peripheral */
183     // Relevant register: RCC->AHBENR
184     RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
185
186     // MODER:
187     //
188
189     /* Configure PA0 (button) as input from the function generator
190     */
191     // Relevant register: GPIOA->MODER
192     GPIOA->MODER &= ~(GPIO_MODER_MODER0); // Clear bits PA0
193
194     /* Configure PA1 (555 timer) as input from the function
195     generator */
196     // Relevant register: GPIOA->MODER
197     GPIOA->MODER &= ~(GPIO_MODER_MODER1); // Set the PA1 bits to 00
198     (where 00 - input)
199
200     /* Configure PA2 (function generator) as input from the function
201     generator */
202     // Relevant register: GPIOA->MODER
203     GPIOA->MODER &= ~(GPIO_MODER_MODER2); // Set the PA2 bits to 00
204     (where 00 - input)
205
206     // Set GPIO PA5 and PA4 to Analog Mode, (Or I can use 0x3 << 10)
207     // 11 - Analog
208     GPIOA->MODER |= GPIO_MODER_MODER4;
209     GPIOA->MODER |= GPIO_MODER_MODER5;
210     // GPIOA->MODER |= 0xC00; // Set GPIO Pin A to Analog Mode, (
211     // Or I can use 0x3 << 10)
212     // GPIOA->MODER |= 0x300; // (or 0x3 << 8)
213
214     /*Ensure no pull-up/pull-down for PA0*/
215     // GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);

```

```
209
210     /* Ensure no pull-up/pull-down for PA1 and PA2 */
211     // Relevant register: GPIOA->PUPDR
212     GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1 | GPIO_PUPDR_PUPDR2);
213 }
214
215 void myTIM2_Init()
216 {
217     /* Enable clock for TIM2 peripheral */
218     // Relevant register: RCC->APB1ENR
219     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
220     /* Configure TIM2: buffer auto-reload, count up, stop on
221        overflow,
222        * enable update events, interrupt on overflow only */
223     // Relevant register: TIM2->CR1
224     TIM2->CR1 = ((uint16_t)0x008C);
225     /* Set clock prescaler value */
226     TIM2->PSC = myTIM2_PRESCALER;
227     /* Set auto-reloaded delay */
228     TIM2->ARR = myTIM2_PERIOD;
229     /* Update timer registers */
230     // Relevant register: TIM2->EGR
231     TIM2->EGR |= ((uint16_t)0x0001);
232     /* Assign TIM2 interrupt priority = 0 in NVIC */
233     // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
234     NVIC_SetPriority(TIM2_IRQn, 0);
235     /* Enable TIM2 interrupts in NVIC */
236     // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
237     NVIC_EnableIRQ(TIM2_IRQn);
238     /* Enable update interrupt generation */
239     // Relevant register: TIM2->DIER
240     TIM2->DIER |= TIM_DIER_UIE;
241 }
242
243 void EXTI_Init()
244 {
245     /* Map EXTI2 and EXTI0 line to PA2 and PA0 respectively */
246     // Relevant register: SYSCFG -> EXTICR[0]
247     SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTIO |
248        SYSCFG_EXTICR1_EXTI1 | SYSCFG_EXTICR1_EXTI2);
249     SYSCFG->EXTICR[0] |= (SYSCFG_EXTICR1_EXTIO_PA |
250        SYSCFG_EXTICR1_EXTI1_PA | SYSCFG_EXTICR1_EXTI2_PA);
251
252     // SYSCFG->EXTICR[0] &= 0xFF0F;
253
254     /* EXTI2 and EXTI0 line interrupts: set rising-edge trigger */
255     // Relevant register: EXTI->RTSR
```

```

253     EXTI->RTSR |= (EXTI_RTSR_TR0 | EXTI_RTSR_TR1 | EXTI_RTSR_TR2);
254
255     /* Unmask interrupts from EXTI2 and EXTI0 line */
256     // Relevant register: EXTI->IMR
257     EXTI->IMR |= (EXTI_IMR_IM0 | EXTI_IMR_IM1);
258
259     /* Assign EXTI2 interrupt priority = 0 in NVIC */
260     // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
261     NVIC_SetPriority(EXTIO_1_IRQn, 0);
262     /* Enable EXTI2 interrupts in NVIC */
263     // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
264     NVIC_EnableIRQ(EXTIO_1_IRQn);
265
266     /* Assign EXTI2 interrupt priority = 0 in NVIC */
267     // Relevant register: NVIC->IP[2], or use NVIC_SetPriority
268     NVIC_SetPriority(EXTI2_3_IRQn, 1);
269     /* Enable EXTI2 interrupts in NVIC */
270     // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
271     NVIC_EnableIRQ(EXTI2_3_IRQn);
272 }
273
274 /* This handler is declared in system/src/cmsis/vectors_stm32f051x8.
   c */
275 void TIM2_IRQHandler()
276 {
277     /* Check if update interrupt flag is indeed set */
278     if ((TIM2->SR & TIM_SR_UIF) != 0)
279     {
280         trace_printf("\n*** Overflow in TIM2! ***\n");
281
282         TIM2->SR &= ~TIM_SR_UIF; // Clear update interrupt flag
283         TIM2->CR1 |= TIM_CR1_CEN; // Restart stopped timer
284     }
285 }
286
287 /* Toggles between the mode for NE555 and the function generator */
288 void toggle_mode()
289 {
290     // Simply flip the boolean
291     funcGen_mode = !funcGen_mode;
292
293     // Disable one of the interrupts
294     if (!funcGen_mode)
295     { // If using 555 timer
296         EXTI->IMR &= ~(EXTI_IMR_IM2);
297         EXTI->IMR |= EXTI_IMR_IM1;
298     }

```

```
299     else
300     {
301         EXTI->IMR &= ~(EXTI_IMR_IM1);
302         EXTI->IMR |= EXTI_IMR_IM2;
303     }
304
305     // Prints out to the console
306     if (TOGGLE_DEBUG)
307     {
308         if (!funcGen_mode)
309         {
310             trace_printf("<<<< NEC555 TIMER >>>>\n");
311         }
312         else
313         {
314             trace_printf("<<<< FUNCTION GENERATOR >>>>\n");
315         }
316     }
317
318     if (OUTPUT_DEBUG)
319     {
320         if (!funcGen_mode)
321         {
322             trace_printf("Resistance: %u\n", resistance);
323             trace_printf("Frequency 1: %u\n\n", ne555_frequency);
324         }
325         else
326         {
327             trace_printf("Frequency 2: %u\n\n", fgen_frequency);
328         }
329     }
330 }
331
332 void button_push()
333 {
334     // There is some change in the voltage value
335     if ((EXTI->PR & EXTI_PR_PRO) != 0)
336     {
337
338         if ((GPIOA->IDR & GPIO_IDR_0) != 0)
339         {
340             // Wait for button to be released (PA0 = 0)
341             while ((GPIOA->IDR & GPIO_IDR_0) != 0)
342             {
343             }
344         }
```

```

345         // Trigger a function or a block of code here
           *****
346         toggle_mode();
347         //
           *****

348     }
349     // Clear pending interrupt flag
350     EXTI->PR |= EXTI_PR_PRO;
351 }
352 }
353
354 /* Measures the frequency and stores the value */
355 void measure_frequency(unsigned int bit_number, unsigned int *
    var_address)
356 {
357
358     // Declare/initialize your local variables here...
359     unsigned int count = 0;
360     float period = 0;
361     float frequency = 0;
362
363     uint32_t register_mask = EXTI_PR_PRO << bit_number;
364
365     /* Check if EXTI2 interrupt pending flag is indeed set */
366     if ((EXTI->PR & register_mask) != 0)
367     {
368         //
369         // 1. If this is the first edge:
370         // - Clear count register (TIM2->CNT).
371         // - Start timer (TIM2->CR1).
372         // Else (this is the second edge):
373         // - Stop timer (TIM2->CR1).
374         // - Read out count register (TIM2->CNT).
375         // - Calculate signal period and frequency.
376         // - Print calculated values to the console.
377         // NOTE: Function trace_printf does not work
378         // with floating-point numbers: you must use
379         // "unsigned int" type to print your signal
380         // period and frequency.
381         //
382
383         // Start the TIM2 timer
384         if ((TIM2->CR1 & TIM_CR1_CEN) == 0)
385         {
386             TIM2->CNT = 0;
387             TIM2->CR1 |= TIM_CR1_CEN;

```

```
388     }
389     else
390     {
391         TIM2->CR1 &= ~(TIM_CR1_CEN);
392         count = TIM2->CNT;
393         period = (float)count / (float)SystemCoreClock;
394         frequency = 1 / period;
395
396         //          trace_printf("Resistance: %u\n",
397             resistance);
398         *var_address = (unsigned int)(frequency);
399
400         // Check if the frequency value is saved
401         if (FREQ_DEBUG)
402         {
403             if (bit_number == 1)
404             {
405                 trace_printf("Frequency: %u\n", ne555_frequency);
406             }
407             else if (bit_number == 2)
408             {
409                 trace_printf("Frequency: %u\n", fgen_frequency);
410             }
411         }
412
413         // 2. Clear EXTI2 interrupt pending flag (EXTI->PR).
414         // NOTE: A pending register (PR) bit is cleared
415         // by writing 1 to it.
416         //
417         EXTI->PR |= register_mask; // Clear interrupt flag for the
418             given bit number
419     }
420
421 void EXTI0_1_IRQHandler()
422 {
423     // processes the button push
424     button_push();
425
426     if (!funcGen_mode)
427     { // If in 555 timer mode
428         // Measure frequency from PA1 (555 timer)
429         measure_frequency(1, &ne555_frequency);
430     }
431 }
```



```

432
433 /* This handler is declared in system/src/cmsis/vectors_stm32f051x8.
    c */
434 void EXTI2_3_IRQHandler()
435 {
436     if (funcGen_mode)
437     { // If in Function generator mode
438         // Measure frequency from PA1 (555 timer)
439         measure_frequency(2, &fgen_frequency);
440     }
441 }
442
443 *****
444 /**                               UTILITIES                               **/
445 *****
446
447 *** Initializing Analog to Digital Conversion ***
448
449 /** Calibrates the ADC **/
450 void calibrate_ADC(void)
451 {
452     if (ADC_DEBUG)
453     {
454         trace_printf("Start ADC Calibration\n");
455     }
456     ADC1->CR = ADC_CR_ADCAL; // Start ADC self-calibration process
457     while (ADC1->CR == ADC_CR_ADCAL)
458         ; // Wait until ADC calibration completes
459     if (ADC_DEBUG)
460     {
461         trace_printf("Finished ADC calibration\n");
462     }
463 }
464
465 /* Initializes the ADC to read values from a Potentiometer in Line 5
    */
466 void myADC_Init()
467 {
468     RCC->APB2ENR |= RCC_APB2ENR_ADC1EN; // Enabling ADC1 clock
469
470     ADC1->SMPR = 0x7; // Set the sampling time to a
        maximum clock cycle (239.5 cycles)
471     ADC1->CHSELR = ADC_CHSELR_CHSEL5; // Select channel 5 for ADC
        conversion
472
473     // Calibrate the ADC
474     if (ENABLE_CAL)

```

```
475     {
476         calibrate_ADC();
477     }
478
479     if (ADC_DEBUG)
480     {
481         trace_printf("Start Enabling ADC, waiting for acknowledgment
482                     ...\\n");
483     }
484
485     ADC1->CR |= ADC_CR_ADEN; // Enable ADC by setting the ADEN bit
486                             // high
487     while (!(ADC1->ISR & ADC_ISR_ADRDY))
488         ; // Wait until ADC is ready for conversion
489     if (ADC_DEBUG)
490     {
491         trace_printf("ADC Enabled\\n");
492     }
493
494     /// Set ADC to continuous conversion mode and overwrite old data
495     // on overrun
496     ADC1->CFGR1 |= (ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD);
497 }
498
499 // convert ADC reading to resistance
500 unsigned int toOhms(uint32_t adc_val)
501 {
502     // Rescaled the ADC Value to the resistance of the potentiometer
503     // 4096 is the maximum 12-bit value from the ADC
504     // Maximum resistance of the potentiometer = 5000 Ohm
505     return (unsigned int)(((float)adc_val / 4095.0) * 5000.0);
506 }
507
508 /* Converts and reads the ADC value */
509 uint32_t readADC()
510 {
511     /// Start the conversion process of ADC Control Register
512     ADC1->CR |= ADC_CR_ADSTART; // ADC group regular conversion
513     // start
514
515     /// Wait until the channel sampling is complete
516     // while (!(ADC1->ISR & ADC_ISR_EOSMP)); (Not necessary.)
517
518     /// After sampling, wait until the ADC1's end-of-conversion (EOC
519     // ) flag is set.
520     // Hardware sets this bit at the end of each conversion of a
521     // channel when a new result is available in ADC_DR
```

```

517 // Hardware clears this bit when ADC_DR is read
518 while (!(ADC1->ISR & ADC_ISR_EOC))
519     ;
520
521 /// Read the ADC result from DR
522 // Retrieve the ADC value from the register; DR = Data Register
523 return ADC1->DR;
524 }
525
526 /* Initializes the DAC to read values output from PA4 */
527 void myDAC_init()
528 {
529     RCC->APB1ENR |= RCC_APB1ENR_DACEN; // Enable DAC Clock
530     // DAC->CR &= 0xFFFFFFFF8;           // Clear unwanted bits
531     // in the CR (TEN1 and BOFF1)
532
533     // Enabling BOFF1 and TEN1 will prevent the frequency from
534     // changing
535     DAC->CR &= ~(0x7); // So, clear all the bits in the CR first
536     // (TEN1, EN1 and BOFF1)
537     DAC->CR |= DAC_CR_EN1; // Switch the DAC enable bit and the
538     // trigger bit to 1
539 }
540
541 // Write to DAC
542 void writeDAC(uint32_t adc_val)
543 {
544     // DHR12R1 is the 12-bit right-aligned data
545     DAC->DHR12R1 = adc_val;
546 }
547
548 #pragma GCC diagnostic pop
549
550 /*****

```

Listing 13: Content of main.c

7.2 OLED Screen Header

7.2.1 oled_screen.h

```

1 #ifndef OLED_SCREEN
2 #define OLED_SCREEN
3
4 #include <stdio.h>
5 #include "diag/Trace.h"
6 #include <string.h>

```

```
7
8 #include "cmsis/cmsis_device.h"
9
10 // Constants
11 #define myTIM3_PRESCALER ((uint16_t)48000U)
12 #define myTIMx_PERIOD (0xFFFFFFFF)
13 #define STARTING_COL (uint8_t)1U
14 #define REFRESH_PERIOD ((uint16_t)100U)
15
16 // Function Prototypes
17 void myGPIOB_Init(void);
18 void mySPI_Init(void);
19 void myTIM3_Init();
20
21
22 void oled_Write(unsigned char);
23 void oled_Write_Cmd(unsigned char);
24 void oled_Write_Data(unsigned char);
25
26 void oled_config(void);
27 void set_Page(uint8_t page);
28 void set_Segment(uint8_t seg);
29
30 void refresh_OLED( unsigned int res, unsigned int freq, uint8_t
    freq_number );
31 void refresh_OLED_test(void);
32 void TIM3_delay(uint8_t milliseconds);
33
34
35 #endif
```

Listing 14: Content of oled_screen.h

7.3 OLED Screen Source

7.3.1 oled_screen.c

```
1 //
2 // This file is part of the GNU ARM Eclipse distribution.
3 // Copyright (c) 2014 Liviu Ionescu.
4 //
5
6 #include "oled_screen.h"
7
8 SPI_HandleTypeDef SPI_Handle;
9
10 //
```

44

45

```

71 {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // SPACE
72 {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // SPACE
73 {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // SPACE
74 {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // SPACE
75 {0b00000000, 0b00000000, 0b01011111, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // !
76 {0b00000000, 0b00000111, 0b00000000, 0b00000111, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // "
77 {0b00010100, 0b01111111, 0b00010100, 0b01111111, 0b00010100, 0
    b00000000, 0b00000000, 0b00000000}, // #
78 {0b00100100, 0b00101010, 0b01111111, 0b00101010, 0b00010010, 0
    b00000000, 0b00000000, 0b00000000}, // $
79 {0b00100011, 0b00010011, 0b00001000, 0b01100100, 0b01100010, 0
    b00000000, 0b00000000, 0b00000000}, // %
80 {0b00110110, 0b01001001, 0b01010101, 0b00100010, 0b01010000, 0
    b00000000, 0b00000000, 0b00000000}, // &
81 {0b00000000, 0b00000101, 0b00000011, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // '
82 {0b00000000, 0b00011100, 0b00100010, 0b01000001, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // (
83 {0b00000000, 0b01000001, 0b00100010, 0b00011100, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // )
84 {0b00010100, 0b00001000, 0b00111110, 0b00001000, 0b00010100, 0
    b00000000, 0b00000000, 0b00000000}, // *
85 {0b00001000, 0b00001000, 0b00111110, 0b00001000, 0b00001000, 0
    b00000000, 0b00000000, 0b00000000}, // +
86 {0b00000000, 0b01010000, 0b00110000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // ,
87 {0b00001000, 0b00001000, 0b00001000, 0b00001000, 0b00001000, 0
    b00000000, 0b00000000, 0b00000000}, // -
88 {0b00000000, 0b01100000, 0b01100000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // .
89 {0b00100000, 0b00010000, 0b00001000, 0b00000100, 0b00000010, 0
    b00000000, 0b00000000, 0b00000000}, // /
90 {0b00111110, 0b01010001, 0b01001001, 0b01000101, 0b00111110, 0
    b00000000, 0b00000000, 0b00000000}, // 0
91 {0b00000000, 0b01000010, 0b01111111, 0b01000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // 1
92 {0b01000010, 0b01100001, 0b01010001, 0b01001001, 0b01000110, 0
    b00000000, 0b00000000, 0b00000000}, // 2
93 {0b00100001, 0b01000001, 0b01000101, 0b01001011, 0b00110001, 0
    b00000000, 0b00000000, 0b00000000}, // 3

```

```

94 {0b00011000, 0b00010100, 0b00010010, 0b01111111, 0b00010000, 0
    b00000000, 0b00000000, 0b00000000}, // 4
95 {0b00100111, 0b01000101, 0b01000101, 0b01000101, 0b00111001, 0
    b00000000, 0b00000000, 0b00000000}, // 5
96 {0b00111100, 0b01001010, 0b01001001, 0b01001001, 0b00110000, 0
    b00000000, 0b00000000, 0b00000000}, // 6
97 {0b00000011, 0b00000001, 0b01110001, 0b00001001, 0b00000111, 0
    b00000000, 0b00000000, 0b00000000}, // 7
98 {0b00110110, 0b01001001, 0b01001001, 0b01001001, 0b00110110, 0
    b00000000, 0b00000000, 0b00000000}, // 8
99 {0b00000110, 0b01001001, 0b01001001, 0b00101001, 0b00011110, 0
    b00000000, 0b00000000, 0b00000000}, // 9
100 {0b00000000, 0b00110110, 0b00110110, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // :
101 {0b00000000, 0b01010110, 0b00110110, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // ;
102 {0b00001000, 0b00010100, 0b00100010, 0b01000001, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // <
103 {0b00010100, 0b00010100, 0b00010100, 0b00010100, 0b00010100, 0
    b00000000, 0b00000000, 0b00000000}, // =
104 {0b00000000, 0b01000001, 0b00100010, 0b00010100, 0b00001000, 0
    b00000000, 0b00000000, 0b00000000}, // >
105 {0b00000010, 0b00000001, 0b01010001, 0b00001001, 0b00000110, 0
    b00000000, 0b00000000, 0b00000000}, // ?
106 {0b00110010, 0b01001001, 0b01111001, 0b01000001, 0b00111110, 0
    b00000000, 0b00000000, 0b00000000}, // @
107 {0b01111110, 0b00010001, 0b00010001, 0b00010001, 0b01111110, 0
    b00000000, 0b00000000, 0b00000000}, // A
108 {0b01111111, 0b01001001, 0b01001001, 0b01001001, 0b00110110, 0
    b00000000, 0b00000000, 0b00000000}, // B
109 {0b00111110, 0b01000001, 0b01000001, 0b01000001, 0b00100010, 0
    b00000000, 0b00000000, 0b00000000}, // C
110 {0b01111111, 0b01000001, 0b01000001, 0b00100010, 0b00011100, 0
    b00000000, 0b00000000, 0b00000000}, // D
111 {0b01111111, 0b01001001, 0b01001001, 0b01001001, 0b01000001, 0
    b00000000, 0b00000000, 0b00000000}, // E
112 {0b01111111, 0b00001001, 0b00001001, 0b00001001, 0b00000001, 0
    b00000000, 0b00000000, 0b00000000}, // F
113 {0b00111110, 0b01000001, 0b01001001, 0b01001001, 0b01111010, 0
    b00000000, 0b00000000, 0b00000000}, // G
114 {0b01111111, 0b00001000, 0b00001000, 0b00001000, 0b01111111, 0
    b00000000, 0b00000000, 0b00000000}, // H
115 {0b01000000, 0b01000001, 0b01111111, 0b01000001, 0b01000000, 0
    b00000000, 0b00000000, 0b00000000}, // I
116 {0b00100000, 0b01000000, 0b01000001, 0b00111111, 0b00000001, 0
    b00000000, 0b00000000, 0b00000000}, // J

```



```

117 {0b01111111, 0b00001000, 0b00010100, 0b00100010, 0b01000001, 0
    b00000000, 0b00000000, 0b00000000}, // K
118 {0b01111111, 0b01000000, 0b01000000, 0b01000000, 0b01000000, 0
    b00000000, 0b00000000, 0b00000000}, // L
119 {0b01111111, 0b00000010, 0b00001100, 0b00000010, 0b01111111, 0
    b00000000, 0b00000000, 0b00000000}, // M
120 {0b01111111, 0b00000100, 0b00001000, 0b00010000, 0b01111111, 0
    b00000000, 0b00000000, 0b00000000}, // N
121 {0b00111110, 0b01000001, 0b01000001, 0b01000001, 0b00111110, 0
    b00000000, 0b00000000, 0b00000000}, // O
122 {0b01111111, 0b00001001, 0b00001001, 0b00001001, 0b00000110, 0
    b00000000, 0b00000000, 0b00000000}, // P
123 {0b00111110, 0b01000001, 0b01010001, 0b00100001, 0b01011110, 0
    b00000000, 0b00000000, 0b00000000}, // Q
124 {0b01111111, 0b00001001, 0b00011001, 0b00101001, 0b01000110, 0
    b00000000, 0b00000000, 0b00000000}, // R
125 {0b01000110, 0b01001001, 0b01001001, 0b01001001, 0b00110001, 0
    b00000000, 0b00000000, 0b00000000}, // S
126 {0b00000001, 0b00000001, 0b01111111, 0b00000001, 0b00000001, 0
    b00000000, 0b00000000, 0b00000000}, // T
127 {0b00111111, 0b01000000, 0b01000000, 0b01000000, 0b00111111, 0
    b00000000, 0b00000000, 0b00000000}, // U
128 {0b00011111, 0b00100000, 0b01000000, 0b00100000, 0b00011111, 0
    b00000000, 0b00000000, 0b00000000}, // V
129 {0b00111111, 0b01000000, 0b00111000, 0b01000000, 0b00111111, 0
    b00000000, 0b00000000, 0b00000000}, // W
130 {0b01100011, 0b00010100, 0b00001000, 0b00010100, 0b01100011, 0
    b00000000, 0b00000000, 0b00000000}, // X
131 {0b00000111, 0b00001000, 0b01110000, 0b00001000, 0b00000111, 0
    b00000000, 0b00000000, 0b00000000}, // Y
132 {0b01100001, 0b01010001, 0b01001001, 0b01000101, 0b01000011, 0
    b00000000, 0b00000000, 0b00000000}, // Z
133 {0b01111111, 0b01000001, 0b00000000, 0b00000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // [
134 {0b00010101, 0b00010110, 0b01111100, 0b00010110, 0b00010101, 0
    b00000000, 0b00000000, 0b00000000}, // back slash
135 {0b00000000, 0b00000000, 0b00000000, 0b01000001, 0b01111111, 0
    b00000000, 0b00000000, 0b00000000}, // ]
136 {0b00000100, 0b00000010, 0b00000001, 0b00000010, 0b00000100, 0
    b00000000, 0b00000000, 0b00000000}, // ^
137 {0b01000000, 0b01000000, 0b01000000, 0b01000000, 0b01000000, 0
    b00000000, 0b00000000, 0b00000000}, // _
138 {0b00000000, 0b00000001, 0b00000010, 0b00000100, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // '
139 {0b00100000, 0b01010100, 0b01010100, 0b01010100, 0b01111000, 0
    b00000000, 0b00000000, 0b00000000}, // a

```

```

140 {0b01111111, 0b01001000, 0b01000100, 0b01000100, 0b00111000, 0
    b00000000, 0b00000000, 0b00000000}, // b
141 {0b00111000, 0b01000100, 0b01000100, 0b01000100, 0b00100000, 0
    b00000000, 0b00000000, 0b00000000}, // c
142 {0b00111000, 0b01000100, 0b01000100, 0b01001000, 0b01111111, 0
    b00000000, 0b00000000, 0b00000000}, // d
143 {0b00111000, 0b01010100, 0b01010100, 0b01010100, 0b00011000, 0
    b00000000, 0b00000000, 0b00000000}, // e
144 {0b00001000, 0b01111110, 0b00001001, 0b00000001, 0b00000010, 0
    b00000000, 0b00000000, 0b00000000}, // f
145 {0b00001100, 0b01010010, 0b01010010, 0b01010010, 0b00111110, 0
    b00000000, 0b00000000, 0b00000000}, // g
146 {0b01111111, 0b00001000, 0b00000100, 0b00000100, 0b01111000, 0
    b00000000, 0b00000000, 0b00000000}, // h
147 {0b00000000, 0b01000100, 0b01111101, 0b01000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // i
148 {0b00100000, 0b01000000, 0b01000100, 0b00111101, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // j
149 {0b01111111, 0b00010000, 0b00101000, 0b01000100, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // k
150 {0b00000000, 0b01000001, 0b01111111, 0b01000000, 0b00000000, 0
    b00000000, 0b00000000, 0b00000000}, // l
151 {0b01111100, 0b00000100, 0b00011000, 0b00000100, 0b01111000, 0
    b00000000, 0b00000000, 0b00000000}, // m
152 {0b01111100, 0b00000100, 0b00000100, 0b00000100, 0b01111000, 0
    b00000000, 0b00000000, 0b00000000}, // n
153 {0b00111000, 0b01000100, 0b01000100, 0b01000100, 0b00111000, 0
    b00000000, 0b00000000, 0b00000000}, // o
154 {0b01111100, 0b00010100, 0b00010100, 0b00010100, 0b00001000, 0
    b00000000, 0b00000000, 0b00000000}, // p
155 {0b00001000, 0b00010100, 0b00010100, 0b00011000, 0b01111100, 0
    b00000000, 0b00000000, 0b00000000}, // q
156 {0b01111100, 0b00000100, 0b00000100, 0b00000100, 0b00001000, 0
    b00000000, 0b00000000, 0b00000000}, // r
157 {0b01001000, 0b01010100, 0b01010100, 0b01010100, 0b00100000, 0
    b00000000, 0b00000000, 0b00000000}, // s
158 {0b00000100, 0b00111111, 0b01000100, 0b01000000, 0b00100000, 0
    b00000000, 0b00000000, 0b00000000}, // t
159 {0b00111100, 0b01000000, 0b01000000, 0b00100000, 0b01111100, 0
    b00000000, 0b00000000, 0b00000000}, // u
160 {0b00011100, 0b00100000, 0b01000000, 0b00100000, 0b00011100, 0
    b00000000, 0b00000000, 0b00000000}, // v
161 {0b00111100, 0b01000000, 0b00111000, 0b01000000, 0b00111100, 0
    b00000000, 0b00000000, 0b00000000}, // w
162 {0b01000100, 0b00101000, 0b00010000, 0b00101000, 0b01000100, 0
    b00000000, 0b00000000, 0b00000000}, // x

```

```

163     {0b00001100, 0b01010000, 0b01010000, 0b01010000, 0b00111100, 0
        b00000000, 0b00000000, 0b00000000}, // y
164     {0b01000100, 0b01100100, 0b01010100, 0b01001100, 0b01000100, 0
        b00000000, 0b00000000, 0b00000000}, // z
165     {0b00000000, 0b00001000, 0b00110110, 0b01000001, 0b00000000, 0
        b00000000, 0b00000000, 0b00000000}, // {
166     {0b00000000, 0b00000000, 0b01111111, 0b00000000, 0b00000000, 0
        b00000000, 0b00000000, 0b00000000}, // |
167     {0b00000000, 0b01000001, 0b00110110, 0b00001000, 0b00000000, 0
        b00000000, 0b00000000, 0b00000000}, // }
168     {0b00001000, 0b00001000, 0b00101010, 0b00011100, 0b00001000, 0
        b00000000, 0b00000000, 0b00000000}, // ~
169     {0b00001000, 0b00011100, 0b00101010, 0b00001000, 0b00001000, 0
        b00000000, 0b00000000, 0b00000000} // <-
170 };
171
172 // Initialize GPIOB pins
173 void myGPIOB_Init()
174 {
175
176     // Enable the clock for the GPIOB peripheral
177     RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
178
179     // Configure PB4, PB6 and PB7 as output
180     GPIOB->MODER &= ~(GPIO_MODER_MODER4 | GPIO_MODER_MODER6 |
        GPIO_MODER_MODER7); // Clear the bits
181     GPIOB->MODER |= (GPIO_MODER_MODER4_0 | GPIO_MODER_MODER6_0 |
        GPIO_MODER_MODER7_0); // Set bits to output (01 = output)
182     GPIOB->OTYPER &= ~(GPIO_OTYPER_OT_4 | GPIO_OTYPER_OT_6 |
        GPIO_OTYPER_OT_7); // Set the output type as push-pull
183     // Configure high-speed mode (11 - high-speed)
184     GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR4 |
        GPIO_OSPEEDER_OSPEEDR6 | GPIO_OSPEEDER_OSPEEDR7);
185     GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4 | GPIO_PUPDR_PUPDR6 |
        GPIO_PUPDR_PUPDR7); // Ensure no pull-up/pull-down
186
187     // Configure AF0 for PB3
188     GPIOB->MODER &= ~(GPIO_MODER_MODER3); // Clear PB3 bits
189     GPIOB->MODER |= GPIO_MODER_MODER3_1; // Set PB5 bits as
        alternate function (10 = AF0)
190     GPIOB->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR3; // Set at high-speed
191     GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3); // Ensure no pull-up/
        pull-down
192
193     // Configure AF0 for PB5
194     GPIOB->MODER &= ~(GPIO_MODER_MODER5); // Clear PB5 bits

```

```
195     GPIOB->MODER |= GPIO_MODER_MODER5_1;        // Set PB5 bits as
        alternate function (10 = AF0)
196     GPIOB->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR5;    // Set at high-speed
197     GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR5);        // Ensure no pull-up/
        pull-down
198 }
199
200 // Initialize SPI1
201 void mySPI_Init(void)
202 {
203
204     // Enable the SPI1 clock
205     RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
206
207     SPI_Handle.Instance = SPI1;
208
209     SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
210     SPI_Handle.Init.Mode = SPI_MODE_MASTER;
211     SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
212     SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
213     SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
214     SPI_Handle.Init.NSS = SPI_NSS_SOFT;
215     SPI_Handle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
216     SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;
217     SPI_Handle.Init.CRCPolynomial = 7;
218
219     // Initialize the SPI interface
220     HAL_SPI_Init(&SPI_Handle);
221
222     // Enable the SPI
223     __HAL_SPI_ENABLE(&SPI_Handle);
224 }
225
226 // Initialize TIM3
227 void myTIM3_Init()
228 {
229     /* Enable clock for TIM2 peripheral */
230     // Relevant register: RCC->APB1ENR
231     RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
232     /* Configure TIM2: buffer auto-reload, count up, stop on
        overflow,
233     * enable update events, interrupt on overflow only */
234     // Relevant register: TIM2->CR1
235     TIM3->CR1 = ((uint16_t)0x008C);
236     /* Set clock prescaler value */
237     TIM3->PSC = myTIM3_PRESCALER - 1;
238     /* Set auto-reloaded delay */
```

```
239 TIM3->ARR = myTIMx_PERIOD;
240 /* Update timer registers */
241 // Relevant register: TIM2->EGR
242 TIM3->EGR |= ((uint16_t)0x0001);
243 /* Assign TIM2 interrupt priority = 0 in NVIC */
244 // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
245 NVIC_SetPriority(TIM3_IRQn, 1);
246 /* Enable TIM2 interrupts in NVIC */
247 // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
248 NVIC_EnableIRQ(TIM3_IRQn);
249 /* Enable update interrupt generation */
250 // Relevant register: TIM2->DIER
251 TIM3->DIER |= TIM_DIER_UIE;
252
253 // Start the timer in TIM3
254 if ((TIM3->CR1 & TIM_CR1_CEN) == 0)
255 {
256     TIM3->CNT = 0;
257     TIM3->CR1 |= TIM_CR1_CEN;
258 }
259 }
260
261 void TIM3_IRQHandler()
262 {
263     /* Check if update interrupt flag is indeed set */
264     if ((TIM3->SR & TIM_SR_UIF) != 0)
265     {
266         trace_printf("\n*** Overflow in TIM3! ***\n");
267
268         TIM3->SR &= ~TIM_SR_UIF; // Clear update interrupt flag
269         TIM3->CR1 |= TIM_CR1_CEN; // Restart stopped timer
270     }
271 }
272
273 // Delay for about a couple of milliseconds
274 void TIM3_delay(uint8_t milliseconds)
275 {
276
277     // Reset the timer
278     TIM3->CNT = 0;
279
280     // Start the timer in TIM3, if not running
281     if ((TIM3->CR1 & TIM_CR1_CEN) == 0)
282     {
283         TIM3->CR1 |= TIM_CR1_CEN;
284     }
285 }
```

```

286 // Wait for the timer to reach the given amount of time
287 while (TIM3->CNT < milliseconds)
288     ;
289     TIM3->CNT = 0;
290 }
291
292 //
293 // LED Display Functions
294 //
295
296 // Prints a string/buffer to the LED Screen
297 // Params: buffer string, page, starting_column (each column
           occupies 8 segments to store a character)
298 void oled_print(unsigned char buffer[17], uint8_t page, uint8_t
           starting_column)
299 {
300     /* The buffer contains your character ASCII codes for LED
           Display */
301
302     // Initialize the ascii_code and the segment, starting from the
           'starting_column'
303     uint8_t segment = starting_column * 8;
304     uint8_t ascii_code;
305
306     // - select PAGE (LED Display line)
307     set_Page(page);
308
309     // Iterate through each character from the buffer
310     for (uint8_t char_index = 0; char_index < strlen((const char *)
           buffer); char_index++)
311     {
312
313         // Grab the ASCII code from the buffer
314         ascii_code = (uint8_t)(buffer[char_index]);
315
316         /*
317          - for each c = ASCII code = Buffer[0], Buffer[1], ...,
318          send 8 bytes in Characters[c][0-7] to LED Display
319          */
320
321         // Iterate through each byte from the Characters array
322         for (uint8_t byte_index = 0; byte_index < 8; byte_index++)
323         {
324             set_Segment(segment++); //
           Don't forget to set the segment
325             oled_Write_Data(Characters[ascii_code][byte_index]); //
           Write the data to the SDDRAM memory of the OLED

```

```
326     }
327 }
328 }
329
330 // PARAMS: Resistance, frequency, and the frequency number
331 // The frequency number is to reference a device which generates
    square waves
332 // In this project, 1 - NE555 timer, 2 - Function generator
333 void refresh_OLED(unsigned int res, unsigned int freq, uint8_t
    freq_number)
334 {
335     // Buffer size = at most 16 characters per PAGE + terminating
        '\0'
336     unsigned char Buffer[17];
337
338     if (TIM3->CNT >= REFRESH_PERIOD - 1)
339     {
340         // Update the buffer and then print it out
341         snprintf((char *)Buffer, sizeof(Buffer), "R: %6u Ohms", res)
            ;
342         oled_print(Buffer, 2, STARTING_COL);
343
344         // Do it again, but this time, for the frequency
345         snprintf((char *)Buffer, sizeof(Buffer), "F%u: %5u Hz  ",
            freq_number, freq);
346         oled_print(Buffer, 4, STARTING_COL);
347
348         TIM3->CNT = 0;
349     }
350 }
351
352 // Used for testing purposes
353 void refresh_OLED_test(void)
354 {
355     unsigned char Buffer[17];
356     unsigned int count = 0;
357
358     while (count <= 200)
359     {
360         if (TIM3->CNT >= 500 - 1)
361         {
362             // Update the buffer and then print it out
363             snprintf((char *)Buffer, sizeof(Buffer), "PEEKABOO!");
364             oled_print(Buffer, 1, STARTING_COL);
365
366             snprintf((char *)Buffer, sizeof(Buffer), "I see you!");
367             oled_print(Buffer, 3, STARTING_COL);
```

```

368
369         snprintf((char *)Buffer, sizeof(Buffer), "Count: %d",
370                 count);
371         oled_print(Buffer, 5, STARTING_COL);
372
373         TIM3->CNT = 0;
374         count++;
375     }
376     TIM3->CR1 &= ~(TIM_CR1_CEN);
377 }
378
379 // Sets a page (row) for the data to be written
380 void set_Page(uint8_t page)
381 {
382     oled_Write_Cmd(0xB0 | (page & 0x7)); // Select PAGE
383 }
384
385 // Selects a segment (column) for the data to be written
386 void set_Segment(uint8_t seg)
387 {
388     oled_Write_Cmd(0x00 | (seg & 0xF)); // Take the lower
389         half of the SEG
390     oled_Write_Cmd(0x10 | ((seg >> 4) & 0xF)); // Take the upper
391         half of the SEG
392 }
393
394 void oled_Write_Cmd(unsigned char cmd)
395 {
396     GPIOB->ODR |= GPIO_ODR_6; // make PB6 = CS# = 1
397     GPIOB->ODR &= ~(GPIO_ODR_7); // make PB7 = D/C# = 0 <== That
398         's where the difference is
399     GPIOB->ODR &= ~(GPIO_ODR_6); // make PB6 = CS# = 0
400     oled_Write(cmd);
401     GPIOB->ODR |= GPIO_ODR_6; // make PB6 = CS# = 1
402 }
403
404 void oled_Write_Data(unsigned char data)
405 {
406     GPIOB->ODR |= GPIO_ODR_6; // make PB6 = CS# = 1
407     GPIOB->ODR |= GPIO_ODR_7; // make PB7 = D/C# = 1 <== That's
408         where the difference is
409     GPIOB->ODR &= ~(GPIO_ODR_6); // make PB6 = CS# = 0
410     oled_Write(data);
411     GPIOB->ODR |= GPIO_ODR_6; // make PB6 = CS# = 1
412 }

```



```
410 void oled_Write(unsigned char Value)
411 {
412     /* Wait until SPI1 is ready for writing (TXE = 1 in SPI1_SR) */
413     while ((SPI1->SR & SPI_SR_TXE) == 0)
414         ;
415
416     /* Send one 8-bit character:
417        - This function also sets BIDIOE = 1 in SPI1_CR1
418     */
419     HAL_SPI_Transmit(&SPI_Handle, &Value, 1, HAL_MAX_DELAY);
420
421     /* Wait until transmission is complete (TXE = 1 in SPI1_SR) */
422     while ((SPI1->SR & SPI_SR_TXE) == 0)
423         ;
424 }
425
426 void oled_config(void)
427 {
428     // Initialize GPIOB pins
429     myGPIOB_Init();
430
431     // Initialize SPI
432     mySPI_Init();
433
434     // Initialize TIM3
435     myTIM3_Init();
436
437     // Set PB4 to low (0) to initiate the reset
438     GPIOB->BSRR = GPIO_BSRR_BR_4; // Clear bit 4
439                                     //      GPIOB->ODR &= ~(GPIO_ODR_4);
440                                     //      // Clear bit 4 (PB4 = 0)
441     TIM3_delay(10);                // Wait for 10 milliseconds
442
443     GPIOB->BSRR = GPIO_BSRR_BS_4; // Set bit 4 back to HIGH
444                                     //      GPIOB->ODR |= GPIO_ODR_4;
445                                     //      // Set bit 4 back to 1
446     TIM3_delay(10);                // Wait for 10 milliseconds
447
448     //
449     // Send initialization commands to LED Display
450     //
451     for (unsigned int i = 0; i < sizeof(oled_init_cmds); i++)
452     {
453         oled_Write_Cmd(oled_init_cmds[i]);
454     }
```

```
455  /* Fill LED Display data memory (GDDRAM) with zeros:
456  - for each PAGE = 0, 1, ..., 7
457      set starting SEG = 0
458      call oled_Write_Data( 0x00 ) 128 times
459  */
460  for (uint8_t page = 0; page < 8; page++)
461  {
462      set_Page(page); // Set the page / row
463
464      // set starting SEG = 0, and call oled_Write_Data( 0x00 )
465      128 times
466      for (uint8_t seg = 0; seg < 128; seg++)
467      {
468          set_Segment(seg);
469          oled_Write_Data(0x00);
470      }
471  }
472
473  #pragma GCC diagnostic pop
474
475  /****************************************************************
```

Listing 15: Content of oled.screen.c