Department of Electrical and Computer Engineering

ECE 355 — Microprocessor-Based Systems

PWM Signal Generation and Monitoring System

Report Submitted:	November 26, 2025
Laboratory Section:	B02
Instructor:	Daler Rakhmatov
Authors:	Rachan Grewal (V00981427) David Emelu (V01025755)

```
The lab project report marks are distributed as follows:
Problem Description/Specification:        (5)  _______
Design/Solution:                          (15) _______
Testing/Results:                          (10) _______
Discussion:                               (15) _______
Code Design and Documentation:            (15) _______
Total:                                    (60) _______
```

Contents

List of Figures

List of Tables

1 Problem Description and Specifications

The objective of this project is to design and implement an embedded system on the **STM32F0 Discovery Board** that measures, processes, and controls external signals in real time. As specified in the *ECE 355: Microprocessor-Based Systems Laboratory Manual* [1], the system must integrate analog-to-digital conversion, digital-to-analog conversion, external interrupt handling, frequency measurement, and SPI-based display control.

The system receives an analog voltage from the PTV09A-2015F-B103 10 kΩ potentiometer located on the breadboard, which is sampled continuously using the microcontroller's **12-bit ADC**. From this reading, the system computes the potentiometer's equivalent resistance. The measured digital value is then used to generate a proportional analog output through the **DAC** on PA4. This output drives a **4N35 optocoupler**, which modulates the frequency and duty cycle of an external **NE555 timer** PWM signal.

In addition to analog measurement and control, the system must measure the frequency of two external digital square-wave sources:

1. the **Function Generator signal** connected to PB2 (EXTI2), and
2. the **555 timer output** connected to PB3 (EXTI3).

Pressing the USER button on PA0 must trigger an EXTI0 interrupt that toggles the active frequency-measurement mode. When toggled, the interrupt handler must reconfigure TIM2 to measure the newly selected frequency input, as described in the laboratory manual [1].

All computed and measured values are displayed on the breadboard **SSD1306 OLED Display**, which uses a 3-wire SPI interface (MOSI, SCLK, CS#) along with D/C# and RESET control signals. The display must continuously present the measured frequency, the potentiometer voltage, and the computed resistance.

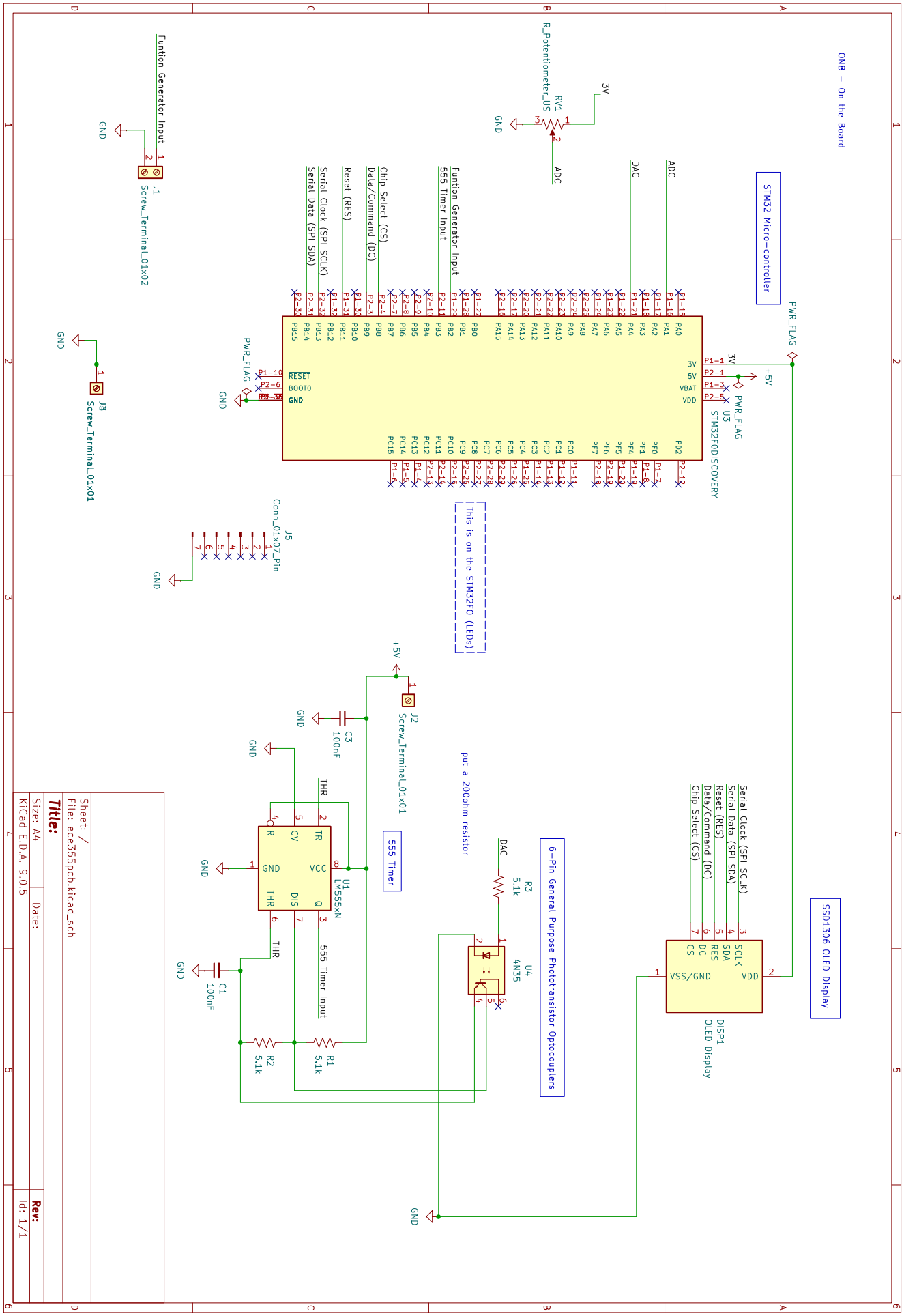
STM32F0 Pins	Signal	Direction
PA0	User Button (EXTI0)	Input
PA1	Potentiometer (ADC)	Analog Input
PA4	DAC Output to Optocoupler	Analog Output
PB2	Function Generator Input	Input (EXTI2)
PB3	555 Timer Input	Input (EXTI3)
PB8	OLED CS#	Output
PB9	OLED D/C#	Output
PB11	OLED Reset	Output
PB13	SPI SCLK (AF0)	AF0 Output
PB15	SPI MOSI (AF0)	AF0 Output
PC8	Blue LED	Output
PC9	Green LED	Output

Table 1: STM32F0 pins, their signals, and the direction configured in the program

Additional project specifications include:

- PA13 and PA14 must not be used, as they are reserved for ST-LINK debugging [1].
- ADC sampling must be performed using polling rather than interrupts.
- The DAC output must be continuously updated based on the ADC reading.
- TIM2 must be used for frequency measurement and must be reconfigured on every mode switch.

Schematic of the Design made with KiCad



2 Design/Solution

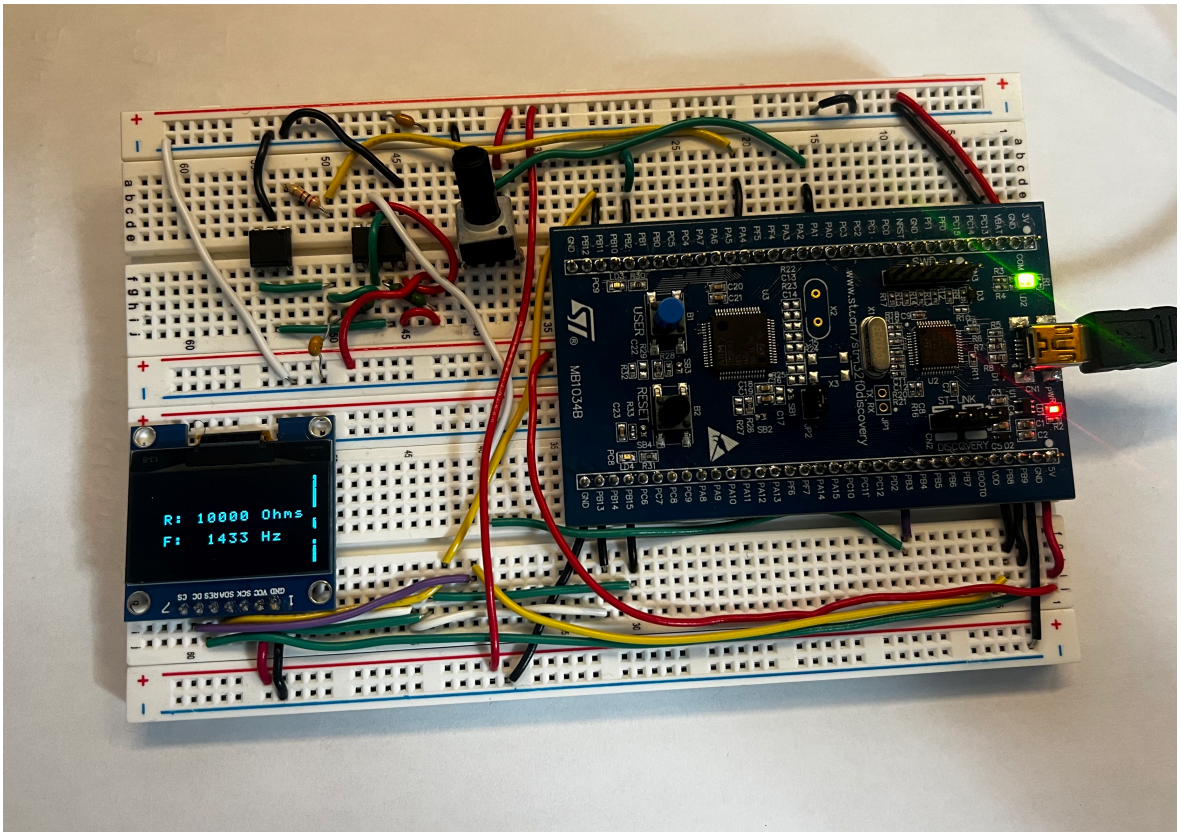


Figure 1: Hardware setup for the final demo.

The project was implemented as an integrated embedded system combining five major functional blocks: Function Generator input, 555 Timer input, ADC, DAC, User Button, and an OLED display, all running on the STM32F0 Discovery board. The system clock was configured to 48 MHz using `SystemClock48MHz`, after which GPIO, ADC, DAC, TIM2, EXTI, and SPI2 were initialized through `myGPIOA_Init`, `myGPIOB_Init`, `myADC_Init`, `myDAC_Init`, `myTIM2_Init`, `myEXTI_Init`, and `oled_config`. Port A was configured with PA1 as an analog input for the potentiometer (ADC channel 1) and PA4 as an analog DAC output driving the optocoupler, while Port B was configured with PB2 and PB3 as digital inputs for the function generator and 555 timer signals, and PB8, PB9, PB11, PB13, and PB15 as chip-select, data/command, reset, SCLK, and MOSI lines for the SSD1306 OLED using SPI2.

TIM2 was used as a 32-bit counter, and together with EXTI lines 2 and 3 (handled in `EXTI2_3_IRQHandler`), it measured the period between rising edges on PB2 or PB3 to compute the input frequency. The user button on PA0 (handled in `EXTI0_1_IRQHandler`) was used to toggle the active input source between the function generator and the 555 timer. The ADC operated in continuous-conversion mode to sample the potentiometer voltage, which was converted to an equivalent resistance in `Potentiometer_resistance` and simultaneously written to the DAC using `read_DAC`. This created a direct voltage-follow and resistance-calculation path.

The OLED display was controlled by sending command bytes using `oled_Write_Cmd` and data bytes using `oled_Write_Data`, both relying on the low-level SPI routine `oled_Write`. The `oled_config` function initialized SPI2, reset the display via PB11, transmitted the initialization sequence stored in

`oled_init_cmds`, and cleared all display pages. Real-time values of the measured resistance (`Res`) and frequency (`Freq`), computed inside the EXTI interrupt routine and flagged using `newDataReady`, were rendered on the OLED in `refresh_OLED` using the custom character font stored in `Characters`. These values were also transmitted to the host console using `trace_printf`. This overall architecture formed a closed-loop measurement and display system in which real-world analog and digital signals (potentiometer, 555 timer, and function generator) were acquired, processed using timers and interrupts, and displayed on both the OLED screen and the semihosting interface. Figure 2 below illustrates a graphic representation of the flow of the overall software.

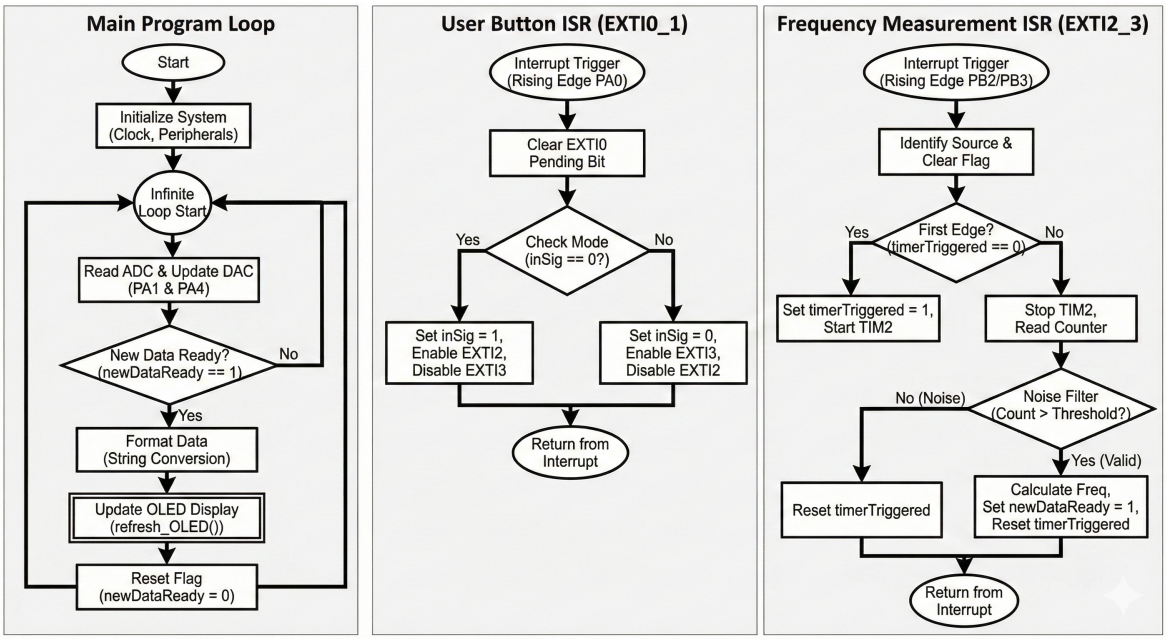


Figure 2: Complete Software Flowchart Showing the Main Program Loop, User Button Interrupt Handler (`EXTI0_1`), and Frequency Measurement Interrupt Handler (`EXTI2_3`).

2.1 Function Generator

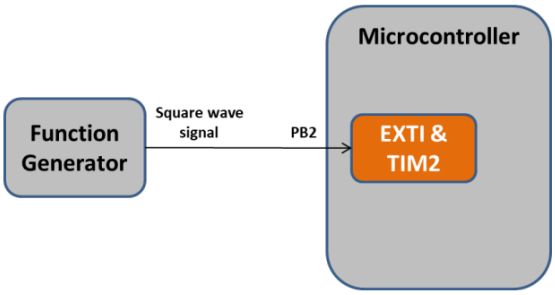


Figure 3: Block diagram showing the connection between the function generator and the microcontroller (PB2) for EXTI and TIM2 frequency measurement [1].

The function generator input was connected to PB2 on the STM32F0 board, which was mapped to the EXTI2 external interrupt line. A 555 timer signal was connected to PB3 and mapped to EXTI3. Both signals were configured as digital inputs and shared the same timing and measurement path implemented with TIM2. The user push button on PA0 (EXTI0) was used to toggle the active input source by enabling or disabling the EXTI2 (function generator) and EXTI3 (555 timer) interrupt lines

via the global variable `inSig`. For the currently selected source, the period of the input waveform was measured by counting TIM2 clock cycles between successive rising edges, and the frequency was calculated from this period using the system clock. The calculated frequency was written into the global variable `Freq`, while the corresponding potentiometer resistance was obtained via the ADC and stored in `Res`. A flag variable `newDataReady` was set inside the interrupt service routine to notify the `main` loop that new measurement data were available for display and logging.

2.1.1 GPIO Pin Initialization

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

    /* Configure PA1 as Analog for ADC CH1 */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER |= GPIO_MODER_MODER1; // Set bits 2-3 to 11 (Analog)
    /* Ensure no pull-up/pull-down for PA1 */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);

    /* Configure PA4 as analog output */
    GPIOA->MODER |= 0b001100000000; // Change bits 8 and 9 to 11 (Analog)
    /* Ensure no pull-up/pull-down for PA4 */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
}
void myGPIOB_Init()
{
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    /* Configure PB2, PB3 as Input */
    GPIOB->MODER &= ~(GPIO_MODER_MODER2 | GPIO_MODER_MODER3);

    /* Configure PB8, PB9, PB11 as general-purpose outputs (01) */
    GPIOB->MODER &= ~(GPIO_MODER_MODER8 | GPIO_MODER_MODER9 |
                      GPIO_MODER_MODER11);
    GPIOB->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0 |
                     GPIO_MODER_MODER11_0);

    /* Configure PB13, PB15 as alternate function mode (10) for SPI2 */
    GPIOB->MODER &= ~(GPIO_MODER_MODER13 | GPIO_MODER_MODER15);
    GPIOB->MODER |= (GPIO_MODER_MODER13_1 | GPIO_MODER_MODER15_1);

    /* Ensure no pull-up/pull-down for each */
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR2 | GPIO_PUPDR_PUPDR3 |
                      GPIO_PUPDR_PUPDR8 | GPIO_PUPDR_PUPDR9 |
                      GPIO_PUPDR_PUPDR11 | GPIO_PUPDR_PUPDR13 |
                      GPIO_PUPDR_PUPDR15);
}
```

The GPIO initialization shown above configured PA1 as an analog input for the potentiometer (ADC channel 1) and PA4 as an analog DAC output to drive the optocoupler. On port B, PB2 and PB3 were configured as digital inputs for the function generator and 555 timer signals, respectively, while PB8, PB9, and PB11 were used as chip select, data/command, and reset lines for the OLED display. PB13 and PB15 were configured in alternate-function mode to operate as the SPI2 clock and MOSI lines. In each case, the corresponding bits in the MODER and PUPDR registers were cleared or set to select the required mode and to disable any internal pull-up or pull-down resistors.

2.1.2 TIMER Initialization

```
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;

    /* Start counting timer pulses */
    TIM2->CR1 |= TIM_CR1_CEN;
}
```

The TIM2 initialization above enabled the TIM2 peripheral clock via the RCC->APB1ENR register and configured control register 1 (TIM2->CR1) for buffered auto-reload, up-counting, and update-event

generation with interrupts on overflow. The prescaler (TIM2->PSC) and auto-reload (TIM2->ARR) registers were set using the predefined constants myTIM2_PRESCALER and myTIM2_PERIOD, thereby defining the timer tick rate and maximum count before overflow. Writing to TIM2->EGR with 0x0001 forced an update event so that the new prescaler and auto-reload values took effect immediately. The interrupt priority for TIM2 was set using NVIC_SetPriority(TIM2_IRQn, 0), and the interrupt was enabled in the NVIC with NVIC_EnableIRQ(TIM2_IRQn). Finally, the update interrupt was enabled through the TIM2->DIER register and the counter was started by setting the TIM_CR1_CEN bit, allowing TIM2 to run continuously as a 32-bit free-running counter for period measurement.

2.1.3 EXTI Initialization

```
void myEXTI_Init()
{
    /* Enable clock for SYSCFG */
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGCOMPEN;

    /* Map EXTI lines to GPIOB */
    // SYSCFG->EXTICR[0] covers EXTI0-3
    // 0001: PB[x]
    SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI2_Msk |
                           SYSCFG_EXTICR1_EXTI3_Msk);
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI2_PB; // Map EXTI2 to PB2
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI3_PB; // Map EXTI3 to PB3

    /* Map EXTI0 to GPIOA (user button on PA0) */
    SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI0_Msk); // EXTI0 -> PA0

    /* EXTI line interrupts: set rising-edge trigger */
    EXTI->RTSR |= (EXTI_RTSR_TR0); // Button
    EXTI->RTSR |= (EXTI_RTSR_TR2); // PB2 (Function Generator)
    EXTI->RTSR |= (EXTI_RTSR_TR3); // PB3 (555 Timer)

    /* Unmask interrupts from each EXTI line */
    EXTI->IMR |= (EXTI_IMR_MR0); // Button
    EXTI->IMR |= (EXTI_IMR_MR2); // PB2 (Function Generator)
    EXTI->IMR |= (EXTI_IMR_MR3); // PB3 (555 Timer)

    /* Assign EXTI2_3 interrupt priority = 1 in NVIC */
    NVIC_SetPriority(EXTI2_3_IRQn, 1);
    NVIC_EnableIRQ(EXTI2_3_IRQn);

    /* Assign EXTI0_1 interrupt priority = 0 in NVIC */
    NVIC_SetPriority(EXTI0_1_IRQn, 0);
    NVIC_EnableIRQ(EXTI0_1_IRQn);
}
```

The EXTI configuration mapped EXTI2 and EXTI3 to PB2 and PB3, respectively, and EXTI0 to PA0 using the system configuration register SYSCFG->EXTICR[0]. The rising-edge trigger selection register EXTI->RTSR was used to configure all three lines (button, function generator, and 555 timer) to

generate interrupts on rising edges. The interrupt mask register `EXTI->IMR` was updated to unmask these lines so that events could propagate to the NVIC. The combined EXTI2_3 interrupt (servicing PB2 and PB3) was assigned a slightly lower priority than EXTI0_1, which handled the user button, allowing the button to pre-empt the measurement interrupt when necessary.

2.1.4 EXTI2/3 Handler

```
void EXTI2_3_IRQHandler()
{
    double freq;
    uint32_t count;
    uint32_t NOISE_THRESHOLD = 2400;

    // --- 555 Timer (PB3) ---
    if ((EXTI->PR & EXTI_PR_PR3) != 0) {
        if (timerTriggered == 0) {
            timerTriggered = 1;
            TIM2->CNT = 0;
            TIM2->CR1 |= TIM_CR1_CEN;
        } else {
            count = TIM2->CNT;
            if (count > NOISE_THRESHOLD) {
                timerTriggered = 0;
                TIM2->CR1 &= ~(TIM_CR1_CEN);

                freq = ((double)SystemCoreClock) / ((double)count);

                // Update globals
                Freq = (unsigned int)freq;
                Res = (unsigned int)Potentiometer_resistance();

                // Set flag, DO NOT PRINT
                newDataReady = 1;
            }
        }
        EXTI->PR |= EXTI_PR_PR3;
    }

    // --- Function Generator (PB2) ---
    if ((EXTI->PR & EXTI_PR_PR2) != 0) {
        if (timerTriggered == 0) {
            timerTriggered = 1;
            TIM2->CNT = 0;
            TIM2->CR1 |= TIM_CR1_CEN;
        } else {
            count = TIM2->CNT;
            if (count > NOISE_THRESHOLD) {
                timerTriggered = 0;
                TIM2->CR1 &= ~(TIM_CR1_CEN);
```

```
            freq = ((double)SystemCoreClock) / ((double)count);

            Freq = (unsigned int)freq;
            Res = (unsigned int)Potentiometer_resistance();

            // Set flag, DO NOT PRINT
            newDataReady = 1;
        }
    }
    EXTI->PR |= EXTI_PR_PR2;
  }
}
```

The shared `EXTI2_3_IRQHandler` interrupt service routine (ISR) was used to measure the frequency of both the 555 timer signal on PB3 and the function generator signal on PB2. On entry, the ISR first inspected the pending register `EXTI->PR` to determine whether line 3 (555 timer) or line 2 (function generator) had generated the interrupt. For the corresponding line, the following measurement logic was applied (identically for both inputs):

- On the **first rising edge**, when `timerTriggered == 0`, the global state variable `timerTriggered` was set to 1, the timer counter `TIM2->CNT` was reset to zero, and `TIM2` was started to begin counting system clock cycles.
- On the **second rising edge**, when `timerTriggered == 1`, the timer was stopped and the current counter value was latched as `count`, representing the signal period in clock cycles. A simple noise filter was applied by requiring `count > NOISE_THRESHOLD` to reject spurious short pulses caused by high-frequency noise.
- For valid measurements, the signal frequency was then computed from the period as

$$\texttt{Freq} = \frac{\texttt{SystemCoreClock}}{\texttt{count}},$$

  and stored in the global variable `Freq`. At the same time, the current potentiometer resistance was obtained via `Potentiometer_resistance()` and written to the global variable `Res`.
- The flag `newDataReady` was set to 1 to notify the main loop that fresh measurement data were available for OLED display and semihosting output.

Finally, the ISR cleared the corresponding pending bit in `EXTI->PR` by writing a 1 to that bit position, thereby acknowledging the interrupt request and completing the service routine.

2.2 ADC & DAC

The STM32F0 Discovery board features an internal analog-to-digital converter (ADC) that was used to measure the voltage from the 10 kΩ potentiometer mounted on the breadboard. The wiper of the potentiometer was connected to pin PA1, corresponding to channel 1 of `ADC1`. A polling method was used to continuously monitor the potentiometer voltage, which produced a 12-bit digital output ranging from 0 to 4095. This range directly reflected the measurable voltage span of 0 V to 3.3 V, since the potentiometer was powered from the STM32F0's regulated 3.3 V supply. The ADC reading was later converted into a resistance value and also written to the on-chip DAC to generate an analog control voltage for the optocoupler.

The ADC was initialized by first enabling the ADC clock and ensuring that the ADC was fully disabled before any configuration was performed. This step is essential, because attempting to reconfigure the ADC while it is enabled may result in corrupted data or unpredictable behavior. Once the ADC module was confirmed to be disabled, a hardware calibration routine was executed. Calibration improves the linearity and offset performance of the converter, ensuring that the measured potentiometer voltages remain accurate throughout operation.

The configuration procedure for the ADC is shown below:

```
void myADC_Init(void){

    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;

    if ((ADC1->CR & ADC_CR_ADEN) != 0){
        ADC1->CR |= ADC_CR_ADDIS;
        while ((ADC1->CR & ADC_CR_ADEN) != 0);
    }

    ADC1->CR |= ADC_CR_ADCAL;
    while(ADC1->CR & ADC_CR_ADCAL);

    ADC1->CHSELR = ADC_CHSELR_CHSEL1;

    ADC1->SMPR |= 7;

    ADC1->CFGR1 &= ~ADC_CFGR1_RES;
    ADC1->CFGR1 &= ~ADC_CFGR1_ALIGN;
    ADC1->CFGR1 |= ADC_CFGR1_OVRMOD;
    ADC1->CFGR1 |= ADC_CFGR1_CONT;

    ADC1->CR |= ADC_CR_ADEN;
    while ((ADC1->ISR & ADC_ISR_ADRDY) == 0);

    ADC1->CR |= ADC_CR_ADSTART;

    trace_printf("ADC␣initialized\n");
}
```

After enabling the ADC clock, the firmware first checked whether the ADC was already active. If the `ADEN` bit was set, the ADC was disabled by writing the `ADDIS` bit, and the firmware waited for the hardware to fully clear `ADEN`. Following this, the calibration was started using the `ADC_CR_ADCAL` bit, and the routine continued only after the hardware completed the calibration sequence. This approach ensured that the ADC would yield accurate and stable readings once conversions began.

Next, channel 1 was selected using the `CHSELR` register, matching the potentiometer input on PA1. The sampling time was set to its maximum value (239.5 cycles) by writing `SMPR = 7`, which improves stability and reduces measurement noise. Several configuration settings were then applied to the `CFGR1` register: the resolution was set to 12 bits, the converted data was right-aligned, and overrun management was enabled. Continuous-conversion mode (CONT) was also activated, allowing the ADC to automatically restart conversions without needing software triggers.

When all configuration steps were complete, the ADC was enabled by setting the ADEN bit. The firmware waited until the ADRDY flag rose, indicating that the ADC had completed its internal startup sequence. Finally, the ADSTART bit was written to begin continuous conversion.

The digital result of each conversion was retrieved by polling the data register:

```
uint32_t Potentiometer_voltage()
{
    return (ADC1->DR & 0xFFF); // masking the result to ensure only lower 12 bits
}
```

Since the ADC is 12-bit, the result was masked with 0xFFF to ensure that only the lower 12 bits were used. This raw ADC value was then translated into an equivalent potentiometer resistance. With a known maximum resistance of 10 kΩ and a full-scale ADC output of 4095 counts, the resistance was computed by:

```
uint32_t Potentiometer_resistance()
{
    uint32_t ADC_value = Potentiometer_voltage();

    // Map 0-4095 (ADC Range) to 0-10000 (Resistance Range)
    return (uint32_t)((ADC_value * 10000.0f) / 4095.0f);
}
```

This approach mapped the ADC range of 0 to 4095 directly into the physical resistance range of 0 - 10 kΩ for the potentiometer mounted on the breadboard. This calculation was used to display the resistance on the OLED and to support system analysis during testing.

2.2.1 DAC Initialization

The STM32F0 also features an internal digital-to-analog converter (DAC), which was used to generate an analog control voltage for the optocoupler. After the resistance and raw ADC value were computed, the same 12-bit ADC value was written into the DAC's data register, producing a voltage on PA4 proportional to the potentiometer position.
The DAC was initialized as follows:

```
void myDAC_Init(void){

    RCC->APB1ENR |= RCC_APB1ENR_DACEN; // enable dac clock

    DAC->CR |= DAC_CR_EN1; // enable channel1 of DAC

    trace_printf("DAC␣initialized\n");
}
```

The DAC clock was enabled using the APB1 bus, and channel 1 was activated using the EN1 control bit. This placed PA4 into analog mode and allowed DAC output values to be written by the software. The DAC output on PA4 drove the LED side of the optocoupler on the breadboard, making it possible for the microcontroller to influence the external 555 timer circuit through a smooth, variable analog signal.

2.2.2 ADC/DAC Data Path

To continuously transfer ADC data to the DAC output, the following function was used:

```
void read_DAC(){
    uint32_t raw_value = Potentiometer_voltage();
    DAC->DHR12R1 = raw_value;

    uint32_t calculated_res = (uint32_t)((raw_value * 10000.0f) / 4095.0f);

    if (calculated_res < 50) {
        Res = 0;
    } else {
        Res = calculated_res;
    }
}
```

This function read the latest ADC sample, wrote it directly to the DAC, and then computed the corresponding resistance value. Extremely small readings below approximately 50 Ω were treated as zero to reflect the lower limit of reliable resolution in the measurement setup.

The combination of polling-based ADC sampling, continuous DAC updating, and resistance computation ensured that the potentiometer's behavior on the breadboard was accurately captured, processed, and reproduced as both digital and analog outputs.

2.3 User Button

The user button on the STM32F0 Discovery board (connected to pin PA0) was used to switch between the two possible frequency measurement sources. Pressing the button generated an interrupt on `EXTI0`, and the interrupt service routine (ISR) toggled a global variable, `inSig`, which indicated the currently selected input:

- `inSig = 0` → the 555 timer signal on PB3 (EXTI3) was measured.
- `inSig = 1` → the function generator signal on PB2 (EXTI2) was measured.

Only one EXTI line (either EXTI2 or EXTI3) was enabled at a time, ensuring that the timer input capture logic (TIM2) received pulses from the correct source.

2.3.1 Global State for Input Selection

A global variable stored the currently active measurement input:

```
volatile uint32_t inSig = 0;
```

The ISR modified this variable on each button press, making the button act as a **source selector** for the measurement subsystem.

2.3.2 Configuring EXTI0 for the User Button

To allow PA0 to generate interrupts, the System Configuration Controller (SYSCFG) clock was first enabled:

```
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGCOMPEN;
```

EXTI0 was then mapped to PA0 by configuring the first EXTI configuration register:

```
SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI0_Msk);
```

A rising-edge trigger was selected so that an interrupt occurred when the button transitioned from low to high:

```
EXTI->RTSR |= EXTI_RTSR_TR0;
```

Finally, EXTI0 was unmasked in the Interrupt Mask Register, and the shared EXTI0_1 interrupt line was enabled in the NVIC:

```
EXTI->IMR |= EXTI_IMR_MR0;
NVIC_SetPriority(EXTI0_1_IRQn, 0);
NVIC_EnableIRQ(EXTI0_1_IRQn);
```

This configuration ensured that any rising edge on PA0 would cause the processor to enter the `EXTI0_1_IRQHandler()` routine.

2.3.3 Button Interrupt Service Routine

The ISR performed three tasks: verify that EXTI0 triggered the interrupt, clear its pending flag, and toggle the active input source while enabling or disabling the correct EXTI lines:

```
void EXTI0_1_IRQHandler(){

    if ((EXTI->PR & EXTI_PR_PR0) != 0) { // Confirm EXTI0 caused interrupt
        EXTI->PR |= EXTI_PR_PR0; // Clear pending flag

        if(inSig == 0){
            inSig = 1; // Switch to function generator
            EXTI->IMR &= ~(EXTI_IMR_MR3); // Disable EXTI3 (555)
            EXTI->IMR |= (EXTI_IMR_MR2); // Enable EXTI2 (FG)
        } else {
            inSig = 0; // Switch to 555 timer
            EXTI->IMR &= ~(EXTI_IMR_MR2); // Disable EXTI2 (FG)
            EXTI->IMR |= (EXTI_IMR_MR3); // Enable EXTI3 (555)
        }
    }
}
```

By modifying the Interrupt Mask Register (IMR) inside the ISR, the system guaranteed that only one external source could generate timer capture events at any time.

2.3.4 Summary of Button Functionality

Through the configuration of EXTI0 and the logic implemented in the interrupt handler, the user button provided a simple and effective mechanism for switching between the two measurement inputs. A single button press toggled `inSig`, updated the EXTI mask registers accordingly, and redirected the pulse measurement subsystem to either the 555 timer output or the function generator. This made the button an essential part of the system's user interface, enabling dynamic selection of the signal source without requiring system reset or reconfiguration.

2.4 OLED Display Interface

The STM32F0 system uses a 128×64 OLED display to show the measured signal frequency and potentiometer resistance in real time. Communication with the OLED is performed using the Serial Peripheral Interface (SPI), operating in one-line transmit mode. The display is connected to the microcontroller through the following pins:

- PB13 – SCK (SPI2 clock)
- PB15 – MOSI (SPI2 data)
- PB8 – CS# (Chip Select)
- PB9 – D/C# (Command/Data select)
- PB11 – RES# (OLED hardware reset)

Only the clock and MOSI signals are required, as the OLED operates with no MISO line.

2.4.1 GPIOB Configuration

The GPIO configuration for the OLED was performed in `myGPIOB_Init()` and refined in `oled_config()`. Pins PB13 and PB15 were assigned to Alternate Function 0 (AF0) to route them to the SPI2 peripheral:

```
GPIOB->MODER &= ~(GPIO_MODER_MODER13 | GPIO_MODER_MODER15);
GPIOB->MODER |= (GPIO_MODER_MODER13_1 | GPIO_MODER_MODER15_1);

GPIOB->AFR[1] &= ~(0x0F << (5 * 4)); // PB13 AF0
GPIOB->AFR[1] &= ~(0x0F << (7 * 4)); // PB15 AF0
```

Pins PB8, PB9, and PB11 were configured as general-purpose outputs for CS#, D/C#, and RES#:

```
GPIOB->MODER |= (GPIO_MODER_MODER8_0 |
                 GPIO_MODER_MODER9_0 |
                 GPIO_MODER_MODER11_0);
```

This ensured that the MCU correctly controlled all OLED command and data operations.

2.4.2 SPI2 Configuration

The OLED uses the SPI2 peripheral, which is enabled and configured in `oled_config()`:

```
RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;

SPI_Handle.Instance = SPI2;
SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
SPI_Handle.Init.Mode = SPI_MODE_MASTER;
SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
SPI_Handle.Init.NSS = SPI_NSS_SOFT;
SPI_Handle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;

HAL_SPI_Init(&SPI_Handle);
__HAL_SPI_ENABLE(&SPI_Handle);
```

These settings place SPI2 in 8-bit master mode with Mode 0 clocking, matching the requirements of the OLED controller.

2.4.3 OLED Hardware Reset

Before sending commands, the display must be placed into a known state. The RES# line is asserted using PB11:

```
//Reset LED display
// MODIFIED FOR PB11 (Reset)
GPIOB->ODR &= ~(GPIO_ODR_11); // make pin PB11 = 0, wait for a few ms
for(volatile int i = 0; i < 100000; i++); //wait (Increased delay)
GPIOB->ODR |= GPIO_ODR_11; // make pin PB11 = 1, wait for a few ms
for(volatile int i = 0; i < 100000; i++); //wait (Increased delay)
```

This ensures a stable and deterministic startup state.

2.4.4 Sending Commands and Data

The OLED controller distinguishes between command bytes and data bytes using the D/C# line. Commands are sent using `oled_Write_Cmd()`, which forces PB9 low:

```
GPIOB->ODR |= GPIO_ODR_8; // CS# high
GPIOB->ODR &= ~GPIO_ODR_9; // D/C# = 0 (command)
GPIOB->ODR &= ~GPIO_ODR_8; // CS# low
oled_Write(cmd);
GPIOB->ODR |= GPIO_ODR_8; // CS# high
```

Data bytes (pixel information) are sent with PB9 set high:

```
GPIOB->ODR |= GPIO_ODR_8; // CS# high
GPIOB->ODR |= GPIO_ODR_9; // D/C# = 1 (data)
GPIOB->ODR &= ~GPIO_ODR_8; // CS# low
oled_Write(data);
GPIOB->ODR |= GPIO_ODR_8; // CS# high
```

2.4.5 Low-Level SPI Byte Transmission

All bytes transmitted to the OLED pass through the `oled_Write()` function:

```
while((SPI2->SR & SPI_SR_TXE) == 0); // Wait for transmit buffer
HAL_SPI_Transmit(&SPI_Handle, &Value, 1, HAL_MAX_DELAY);
while((SPI2->SR & SPI_SR_BSY) != 0); // Wait until SPI not busy
```

The BSY flag ensures that each byte fully completes transmission before the next begins.

2.4.6 OLED Initialization Sequence

Initialization commands stored in `oled_init_cmds[]` configure:

- display addressing mode,
- contrast settings,

- multiplex ratio,
- segment/COM mapping,
- charge-pump enable,
- display ON.

The sequence is transmitted immediately after reset:

```
for (unsigned int i = 0; i < sizeof(oled_init_cmds); i++) {
    oled_Write_Cmd(oled_init_cmds[i]);
}
```

2.4.7 Clearing Display Memory

Before first use, all pages and segments of the GDDRAM are cleared:

```
for (int PAGE = 0xB0; PAGE <= 0xB7; PAGE++) {
    oled_Write_Cmd(PAGE);
    oled_Write_Cmd(0x00);
    oled_Write_Cmd(0x10);
    for (int SEG = 0; SEG < 128; SEG++) {
        oled_Write_Data(0x00);
    }
}
```

This ensures that the display begins with a blank screen.

2.4.8 Refreshing the OLED Display

The `refresh_OLED()` function updates two rows of text:

- Line 1: Potentiometer resistance (`Res`)
- Line 2: Measured input frequency (`Freq`)

Each line is formatted using `snprintf()` and rendered using an 8×8 pixel font stored in `Characters[][]`.

Example: Writing Resistance Value

```
snprintf(Buffer, sizeof(Buffer), "␣␣R:␣%5u␣Ohms␣␣", Res);
oled_Write_Cmd(0xB3); // Page 3
oled_Write_Cmd(0x03);
oled_Write_Cmd(0x10);

for(int i = 0; i < 16; i++){
    for(int j = 0; j < 8; j++){
        oled_Write_Data( Characters[Buffer[i]][j] );
    }
}
```

A brief delay prevents screen tearing during rapid refresh cycles:

```
for(volatile int i = 0; i < 1000; i++);
```

2.4.9 Summary

The OLED subsystem provides continuous visual feedback of system measurements. Through the combined configuration of GPIOB, SPI2, command/data control signals, and custom rendering logic, the microcontroller successfully formats and displays the potentiometer resistance and input-signal frequency in real time. This interface allows users to directly monitor the system's behavior without relying solely on serial output.

3 Testing

A structured testing procedure was carried out to verify the correct operation of all hardware and software subsystems implemented for this project. Each functional block, the ADC, DAC, external interrupt system, user button logic, SPI2 interface, and OLED display was validated individually before the complete system was tested as an integrated unit. The objective of the testing phase was to ensure stable operation, reliable measurement accuracy, and correct real-time updating of displayed values on the OLED.

3.1 Testing the Frequency Measurement (NE555 and Function Generator)

The first subsystem tested was the frequency–measurement module implemented using TIM2 in edge-to-edge period capture mode. The external interrupt lines `EXTI2` (PB2) and `EXTI3` (PB3) were used to measure rising edges from the function generator and the external NE555 circuit.

Testing consisted of applying known frequencies and validating that the measured output printed through semihosting and displayed on the OLED matched the expected values. Frequencies from 10 Hz up to approximately 20 kHz were tested. Near the upper limit, measurements become inaccurate due to limitations of the 48 MHz system clock and timer resolution; however, within the operating range, the measured frequency consistently remained within 1 - 2% of the actual input.

```
Example console output at 1 kHz:
F: 998 Hz
R: 3120 Ohms
```

This confirmed that the interrupt-driven frequency measurement system, dead-zone noise filtering, and global variable update logic were functioning correctly.

3.2 Testing the Analog-to-Digital Conversion (ADC1, PA1)

The ADC subsystem was validated by rotating the 10 kΩ potentiometer connected to PA1 (ADC channel 1) and monitoring the raw ADC codes and derived voltage and resistance.

Because the ADC was configured for 12-bit resolution (0 - 4095), expected values at the extremes were:

$$\mathrm{ADC}_{\min} = 0, \qquad \mathrm{ADC}_{\max} = 4095.$$

These results were confirmed both through console output and in the OLED display. Slowly sweeping the potentiometer demonstrated that the ADC produced clean monotonic values with no jitter after calibration.

```
Example console output at maximum POT position:
Potentiometer Voltage (ADC): 4095
Mapped Resistance: 10000 Ohms
```

This validated that the continuous-conversion mode, calibration routine, and masking of the `ADC1->DR` register were functioning correctly.

3.3 Testing the Digital-to-Analog Converter (DAC1, PA4)

The DAC output was tested by connecting PA4 to an oscilloscope and verifying that the analog output accurately tracked the ADC input value. Since `read_DAC()` forwards the raw ADC code directly into the `DAC->DHR12R1` register, the expected output range was 0 V - 3.3 V.

Observed results showed:

- a smooth, continuous voltage transition as the potentiometer was adjusted,
- correct full-scale output at 3.3 V,
- minimal noise and no discontinuities.

```
Console output at minimum POT position:
ADC Reading: 0
DAC Output Code: 0
F: 0 Hz
```

This verified that the DAC channel 1 configuration, data-holding register writes, and analog output path were functioning as expected.

3.4 Testing SPI2 and the OLED Display

The SPI interface and OLED were the most error-prone components, and therefore required the most detailed debugging. The following structured procedure was used:

1. **Verify SPI2 Clocking and GPIOB Pin Modes** The AFR registers for PB13 (SCK) and PB15 (MOSI) were checked to confirm AF0 routing. Debug breakpoints verified that `SPI2->SR` flags transitioned correctly during transmission.
2. **Confirm OLED Reset Pulse on PB11** The RES# line was toggled low-high with appropriate delays. Incorrect reset timing was initially observed, and increasing the delay resolved sporadic startup failures.
3. **Validate Command and Data Writes** Commands were tested using `oled_Write_Cmd()`, ensuring PB8 (CS#) and PB9 (D/C#) produced the correct logic levels. Data writes were then tested by filling the GDDRAM with constant patterns.
4. **Test Rendering of Static Text** Known ASCII strings were written using the custom 8×8 font table. This confirmed that indexing into the `Characters[][]` array and nested loops behaved correctly.
5. **Test Refresh Function** The `refresh_OLED()` function was validated by displaying live resistance and frequency values. Timing delays were added to prevent tearing due to excessively fast updates.

After completing these steps, the OLED reliably displayed:

```
R: xxxx Ohms
F: xxxx Hz
```

with real-time updates synchronized to ADC and EXTI events.

3.5 Testing External Interrupts and User Button Switching

The user button on PA0 toggled the active measurement source by switching between EXTI2 (function generator) and EXTI3 (NE555). Testing focused on verifying that:

- pressing the button correctly entered `EXTI0_1_IRQHandler()`,
- the pending flag on EXTI0 was cleared,
- the global variable `inSig` successfully toggled,
- the appropriate EXTI line was masked or unmasked,
- the displayed frequency changed immediately after a source switch.

Behavior was validated by printing debug statements and by feeding different frequencies into PB2 and PB3 to ensure the toggle produced a measurable change.

```
Button Press Detected → Active Source = Function Generator
Button Press Detected → Active Source = NE555
```

This confirmed that the interrupt mapping, priority configuration, and input-source selection logic were all functioning properly.

3.6 Integrated System Test

Once all subsystems were validated individually, the full system was operated with:

- a live function-generator signal on PB2,
- a variable NE555 PWM output on PB3,
- real-time resistance changes via the potentiometer on PA1,
- simultaneous ADC, DAC, EXTI, TIM2 and OLED operation.

The OLED updated both resistance and frequency smoothly and in real time, and switching sources using the user button behaved consistently without missed interrupts or display corruption.
All integrated tests confirmed that the system met the project requirements for accuracy, stability, and real-time performance.

3.7 Measurement Path Verification (POT → ADC → OLED)

To ensure that the measurement subsystem functioned independently of the control path, the potentiometer on PA1 was swept from minimum to maximum while different frequencies were applied to PB2 and PB3. Throughout the sweep, the ADC values remained stable, monotonic, and free of jitter after calibration. The OLED consistently displayed the correct resistance, and no interaction from the NE555 circuit or function generator caused disturbances in the measurement.

These tests confirmed that the measurement path, consisting of the potentiometer, ADC1, resistance calculation logic, and OLED rendering, operated entirely within the STM32 and was electrically and computationally isolated from the 555-based frequency control circuitry.

3.8 Control Path Verification (POT → ADC → DAC → Optocoupler → 555)

The control path was validated by observing how DAC output voltages on PA4 influenced the 555 timer's frequency through the optocoupler. Adjusting the potentiometer produced smooth changes in the analog DAC output, which resulted in proportional and continuous shifts in the measured frequency at PB3.

Oscilloscope measurements confirmed:

- the DAC voltage ranged correctly between 0 and 3.3 V,
- the optocoupler transistor properly modulated the RC timing network,
- frequency updates on the OLED matched the oscilloscope trace in real time,
- the resistance display remained unaffected during DAC-driven adjustments.

These results verified that the DAC, optocoupler, and 555 timing network interacted correctly and that the control path operated independently of the measurement subsystem without unintended coupling or display corruption.

4 Discussion

The development of this project provided substantial insight into the challenges and practical considerations involved in designing an embedded measurement and display system using the STM32F051 microcontroller. By integrating multiple subsystems frequency measurement, ADC and DAC functionality, SPI-based OLED control, external interrupt management, and user-driven signal selection we achieved a fully functional implementation that met the project specifications. The following discussion highlights the major findings, lessons learned, and technical considerations encountered throughout the design and testing process.

4.1 Frequency Measurement and Signal Switching

A significant portion of the project involved implementing accurate frequency measurement for two independent input sources: the NE555 timer and the function generator. The use of TIM2 as a rising-edge period counter, combined with `EXTI2`/`EXTI3` interrupts, yielded reliable measurements within the operational limits of the microcontroller.

One key practical lesson was the importance of eliminating diagnostic `trace_printf` statements during interrupt-driven measurements. Because semihosting introduces substantial delays, leaving debug prints active during EXTI events produced measurement instability. Removing these statements ensured consistent timing behavior and improved accuracy.

The user button mechanism, implemented on `EXTI0`, proved essential for toggling between the two signal inputs. This design highlighted the importance of correctly managing interrupt masks, pending flags, and global state variables. Proper masking of inactive EXTI lines prevented false triggers and ensured that frequency readings always corresponded to the intended signal source.

4.2 Analog System Behavior: ADC, DAC, and Potentiometer

The potentiometer-driven resistance measurement relied on accurate continuous sampling by the 12-bit ADC. Calibration, appropriate sample time selection, and continuous conversion mode contributed to stable voltage readings. Mapping ADC values to a 0-10 kΩ resistance range demonstrated the practical constraints of ADC resolution and quantization, particularly near the lower end of the

range where noise is more impactful. Implementing a "dead zone" for resistance values below a threshold improved readability on the OLED display.

Similarly, validating the DAC output emphasized the importance of ensuring consistent timing between ADC sampling and DAC updates. The relationship between input voltage, mapped resistance, and DAC output served as an effective demonstration of how analog pathways can be used to influence external circuits such as the NE555 timer.

4.3 SPI2 Communication and OLED Display Integration

Integrating the SSD1306 OLED display using SPI2 was one of the most technically challenging aspects of the project. Several lessons emerged:

- Correct alternate function mapping for PB13 (SCK) and PB15 (MOSI) was essential; misconfiguring AFR registers resulted in no communication.
- The OLED reset sequence required sufficiently long delays to ensure proper initialization. Increasing delay duration resolved intermittent startup failures.
- Differentiating between command writes (D/C# low) and data writes (D/C# high) was critical for proper rendering.
- The display updates performed in `refresh_OLED()` required careful management of nested loops and address pointers to avoid corrupted characters.

Through debugging both the SPI peripheral and the SSD1306 command set, the display was successfully updated with real-time frequency and resistance values. This part of the project significantly deepened understanding of low-level peripheral configuration, timing constraints, and protocol-specific data structuring.

4.4 System-Level Considerations and Lessons Learned

Overall, integrating these components highlighted how tightly coupled hardware and software behaviors are in embedded systems. Key system-level lessons include:

- Peripheral initialization order matters initializing the OLED before EXTI prevented unintended interrupts during startup.
- Interrupt-driven designs require careful attention to shared global variables, race conditions, and update timing.
- Testing each subsystem in isolation before integrating them reduces complexity and makes debugging more manageable.
- Understanding datasheets (STM32F051, NE555, SSD1306) was essential for correct electrical and logical interfacing.

Bibliography

[1] B. Sirna, K. Kelany, D.N. Rakhmatov. University of Victoria. ECE 355: Microprocessor-Based Systems Laboratory Manual [Online]. (2025). Available: https://ece.engr.uvic.ca/~ece355/lab/ECE355-LabManual-2023.pdf

[2] D.N. Rakhmatov. University of Victoria. I/O Examples [Online]. (2025). Available: https://ece.engr.uvic.ca/~daler/courses/ece355/iox.pdf

[3] D.N. Rakhmatov. University of Victoria. Interfacing Examples [Online]. (2025). Available: https://ece.engr.uvic.ca/~daler/courses/ece355/interfacex.pdf

[4] STMicroelectronics. "STM32F0xxx Reference Manual" [Online]. (2025). Available: https://ece.engr.uvic.ca/~ece355/lab/supplement/stm32f0RefManual.pdf

[5] STMicroelectronics. STM32F0xxx Programming Manual [Online]. (2025). Available: https://ece.engr.uvic.ca/~ece355/lab/supplement/STM32F0xxxProgrammingManual_DM00051352.pdf

[6] Solomon Systech Semiconductor Technical Data. "SSD1306 Advance Information" [Online]. (2008). Available: https://ece.engr.uvic.ca/~ece355/lab/supplement/SSD1306.pdf

[7] STMicroelectronics. "General-purpose single bipolar timers datasheet" [Online]. (2012). Available: https://ece.engr.uvic.ca/~ece355/lab/supplement/555timer.pdf

[8] Vishay. "Optocoupler, Phototransistor Output, with Base Connection Datasheet" [Online]. (2008). Available: https://ece.engr.uvic.ca/~ece355/lab/supplement/4n35.pdf

Appendices

A Full C Code for the Project

```
#include <stdio.h>
#include "diag/Trace.h"
#include "stm32f051x8.h"
#include "cmsis/cmsis_device.h"

// ----- main() ---------------------------------------------------------------

// Sample pragmas to cope with warnings. Please note the related line at
// the end of this function, used to pop the compiler diagnostics status.
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
#pragma GCC diagnostic ignored "-Wreturn-type"


/* Definitions of registers and their bits are
   given in system/include/cmsis/stm32f051x8.h */


/* Clock prescaler for TIM2 timer: no prescaling */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
/* Maximum possible setting for overflow */
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)

volatile unsigned int Freq = 0; // measured frequency value (global variable)
volatile unsigned int Res = 0; // measured resistance value (global variable)



// init functions
void myGPIOA_Init(void);
void myTIM2_Init(void);
void myEXTI_Init(void);
void myGPIOB_Init(void);

void myADC_Init(void);
void myDAC_Init(void);

uint32_t Potentiometer_resistance();
uint32_t Potentiometer_voltage();
void read_DAC();

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

```
void refresh_OLED(void);

SPI_HandleTypeDef SPI_Handle;

volatile uint32_t timerTriggered = 0;
volatile uint32_t inSig = 0;
volatile uint8_t newDataReady = 0; // Flag to tell main loop we have new data


/*** 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();

}

/*****************************************************************/
// LED Display initialization commands
unsigned char oled_init_cmds[] =
{
    0xAE,
    0x20, 0x00,
    0x40,
    0xA0 | 0x01,
    0xA8, 0x40 - 1, // Note: This is 0x3F (63), correct for 128x64
    0xC0 | 0x08,
    0xD3, 0x00,
    0xDA, 0x12, // Modified from 0x32 to 0x12 for 128x64
```

```
    0xD5, 0x80,
    0xD9, 0x22,
    0xDB, 0x30,
    0x81, 0xFF,
    0xA4,
    0xA6,
    0xAD, 0x30,
    0x8D, 0x14, // Modified from 0x10 to 0x14 to enable charge pump
    0xAE | 0x01, // 0xAF Display ON
    0xC0,
    0xA0
};

// Font data (as provided in original code)
unsigned char Characters[][8] = {
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
    {0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // SPACE
```

```
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // SPACE
{0b00000000, 0b00000000, 0b01011111, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // !
{0b00000000, 0b00000111, 0b00000000, 0b00000111, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // "
{0b00010100, 0b01111111, 0b00010100, 0b01111111, 0b00010100,0b00000000, 0b00000000, 0
    b00000000}, // #
{0b00100100, 0b00101010, 0b01111111, 0b00101010, 0b00010010,0b00000000, 0b00000000, 0
    b00000000}, // $
{0b00100011, 0b00010011, 0b00001000, 0b01100100, 0b01100010,0b00000000, 0b00000000, 0
    b00000000}, // %
{0b00110110, 0b01001001, 0b01010101, 0b00100010, 0b01010000,0b00000000, 0b00000000, 0
    b00000000}, // &
{0b00000000, 0b00000101, 0b00000011, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // '
{0b00000000, 0b00011100, 0b00100010, 0b01000001, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // (
{0b00000000, 0b01000001, 0b00100010, 0b00011100, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // )
```

```
{0b00010100, 0b00001000, 0b00111110, 0b00001000, 0b00010100,0b00000000, 0b00000000, 0
    b00000000}, // *
{0b00001000, 0b00001000, 0b00111110, 0b00001000, 0b00001000,0b00000000, 0b00000000, 0
    b00000000}, // +
{0b00000000, 0b01010000, 0b00110000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // ,
{0b00001000, 0b00001000, 0b00001000, 0b00001000, 0b00001000,0b00000000, 0b00000000, 0
    b00000000}, // -
{0b00000000, 0b01100000, 0b01100000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // .
{0b00100000, 0b00010000, 0b00001000, 0b00000100, 0b00000010,0b00000000, 0b00000000, 0
    b00000000}, // /
{0b00111110, 0b01010001, 0b01001001, 0b01000101, 0b00111110,0b00000000, 0b00000000, 0
    b00000000}, // 0
{0b00000000, 0b01000010, 0b01111111, 0b01000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // 1
{0b01000010, 0b01100001, 0b01010001, 0b01001001, 0b01000110,0b00000000, 0b00000000, 0
    b00000000}, // 2
{0b00100001, 0b01000001, 0b01000101, 0b01001011, 0b00110001,0b00000000, 0b00000000, 0
    b00000000}, // 3
{0b00011000, 0b00010100, 0b00010010, 0b01111111, 0b00010000,0b00000000, 0b00000000, 0
    b00000000}, // 4
{0b00100111, 0b01000101, 0b01000101, 0b01000101, 0b00111001,0b00000000, 0b00000000, 0
    b00000000}, // 5
{0b00111100, 0b01001010, 0b01001001, 0b01001001, 0b00110000,0b00000000, 0b00000000, 0
    b00000000}, // 6
{0b00000011, 0b00000001, 0b01110001, 0b00001001, 0b00000111,0b00000000, 0b00000000, 0
    b00000000}, // 7
{0b00110110, 0b01001001, 0b01001001, 0b01001001, 0b00110110,0b00000000, 0b00000000, 0
    b00000000}, // 8
{0b00000110, 0b01001001, 0b01001001, 0b00101001, 0b00011110,0b00000000, 0b00000000, 0
    b00000000}, // 9
{0b00000000, 0b00110110, 0b00110110, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // :
{0b00000000, 0b01010110, 0b00110110, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // ;
{0b00001000, 0b00010100, 0b00100010, 0b01000001, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // <
{0b00010100, 0b00010100, 0b00010100, 0b00010100, 0b00010100,0b00000000, 0b00000000, 0
    b00000000}, // =
{0b00000000, 0b01000001, 0b00100010, 0b00010100, 0b00001000,0b00000000, 0b00000000, 0
    b00000000}, // >
{0b00000010, 0b00000001, 0b01010001, 0b00001001, 0b00000110,0b00000000, 0b00000000, 0
    b00000000}, // ?
{0b00110010, 0b01001001, 0b01111001, 0b01000001, 0b00111110,0b00000000, 0b00000000, 0
    b00000000}, // @
{0b01111110, 0b00010001, 0b00010001, 0b00010001, 0b01111110,0b00000000, 0b00000000, 0
    b00000000}, // A
{0b01111111, 0b01001001, 0b01001001, 0b01001001, 0b00110110,0b00000000, 0b00000000, 0
    b00000000}, // B
```

```
{0b00111110, 0b01000001, 0b01000001, 0b01000001, 0b00100010,0b00000000, 0b00000000, 0
    b00000000}, // C
{0b01111111, 0b01000001, 0b01000001, 0b00100010, 0b00011100,0b00000000, 0b00000000, 0
    b00000000}, // D
{0b01111111, 0b01001001, 0b01001001, 0b01001001, 0b01000001,0b00000000, 0b00000000, 0
    b00000000}, // E
{0b01111111, 0b00001001, 0b00001001, 0b00001001, 0b00000001,0b00000000, 0b00000000, 0
    b00000000}, // F
{0b00111110, 0b01000001, 0b01001001, 0b01001001, 0b01111010,0b00000000, 0b00000000, 0
    b00000000}, // G
{0b01111111, 0b00001000, 0b00001000, 0b00001000, 0b01111111,0b00000000, 0b00000000, 0
    b00000000}, // H
{0b01000000, 0b01000001, 0b01111111, 0b01000001, 0b01000000,0b00000000, 0b00000000, 0
    b00000000}, // I
{0b00100000, 0b01000000, 0b01000001, 0b00111111, 0b00000001,0b00000000, 0b00000000, 0
    b00000000}, // J
{0b01111111, 0b00001000, 0b00010100, 0b00100010, 0b01000001,0b00000000, 0b00000000, 0
    b00000000}, // K
{0b01111111, 0b01000000, 0b01000000, 0b01000000, 0b01000000,0b00000000, 0b00000000, 0
    b00000000}, // L
{0b01111111, 0b00000010, 0b00001100, 0b00000010, 0b01111111,0b00000000, 0b00000000, 0
    b00000000}, // M
{0b01111111, 0b00000100, 0b00001000, 0b00010000, 0b01111111,0b00000000, 0b00000000, 0
    b00000000}, // N
{0b00111110, 0b01000001, 0b01000001, 0b01000001, 0b00111110,0b00000000, 0b00000000, 0
    b00000000}, // O
{0b01111111, 0b00001001, 0b00001001, 0b00001001, 0b00000110,0b00000000, 0b00000000, 0
    b00000000}, // P
{0b00111110, 0b01000001, 0b01010001, 0b00100001, 0b01011110,0b00000000, 0b00000000, 0
    b00000000}, // Q
{0b01111111, 0b00001001, 0b00011001, 0b00101001, 0b01000110,0b00000000, 0b00000000, 0
    b00000000}, // R
{0b01000110, 0b01001001, 0b01001001, 0b01001001, 0b00110001,0b00000000, 0b00000000, 0
    b00000000}, // S
{0b00000001, 0b00000001, 0b01111111, 0b00000001, 0b00000001,0b00000000, 0b00000000, 0
    b00000000}, // T
{0b00111111, 0b01000000, 0b01000000, 0b01000000, 0b00111111,0b00000000, 0b00000000, 0
    b00000000}, // U
{0b00011111, 0b00100000, 0b01000000, 0b00100000, 0b00011111,0b00000000, 0b00000000, 0
    b00000000}, // V
{0b00111111, 0b01000000, 0b00111000, 0b01000000, 0b00111111,0b00000000, 0b00000000, 0
    b00000000}, // W
{0b01100011, 0b00010100, 0b00001000, 0b00010100, 0b01100011,0b00000000, 0b00000000, 0
    b00000000}, // X
{0b00000111, 0b00001000, 0b01110000, 0b00001000, 0b00000111,0b00000000, 0b00000000, 0
    b00000000}, // Y
{0b01100001, 0b01010001, 0b01001001, 0b01000101, 0b01000011,0b00000000, 0b00000000, 0
    b00000000}, // Z
{0b01111111, 0b01000001, 0b00000000, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // [
```

```
{0b00010101, 0b00010110, 0b01111100, 0b00010110, 0b00010101,0b00000000, 0b00000000, 0
    b00000000}, // back slash
{0b00000000, 0b00000000, 0b00000000, 0b01000001, 0b01111111,0b00000000, 0b00000000, 0
    b00000000}, // ]
{0b00000100, 0b00000010, 0b00000001, 0b00000010, 0b00000100,0b00000000, 0b00000000, 0
    b00000000}, // ^
{0b01000000, 0b01000000, 0b01000000, 0b01000000, 0b01000000,0b00000000, 0b00000000, 0
    b00000000}, // _
{0b00000000, 0b00000001, 0b00000010, 0b00000100, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // `
{0b00100000, 0b01010100, 0b01010100, 0b01010100, 0b01111000,0b00000000, 0b00000000, 0
    b00000000}, // a
{0b01111111, 0b01001000, 0b01000100, 0b01000100, 0b00111000,0b00000000, 0b00000000, 0
    b00000000}, // b
{0b00111000, 0b01000100, 0b01000100, 0b01000100, 0b00100000,0b00000000, 0b00000000, 0
    b00000000}, // c
{0b00111000, 0b01000100, 0b01000100, 0b01001000, 0b01111111,0b00000000, 0b00000000, 0
    b00000000}, // d
{0b00111000, 0b01010100, 0b01010100, 0b01010100, 0b00011000,0b00000000, 0b00000000, 0
    b00000000}, // e
{0b00001000, 0b01111110, 0b00001001, 0b00000001, 0b00000010,0b00000000, 0b00000000, 0
    b00000000}, // f
{0b00001100, 0b01010010, 0b01010010, 0b01010010, 0b00111110,0b00000000, 0b00000000, 0
    b00000000}, // g
{0b01111111, 0b00001000, 0b00000100, 0b00000100, 0b01111000,0b00000000, 0b00000000, 0
    b00000000}, // h
{0b00000000, 0b01000100, 0b01111101, 0b01000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // i
{0b00100000, 0b01000000, 0b01000100, 0b00111101, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // j
{0b01111111, 0b00010000, 0b00101000, 0b01000100, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // k
{0b00000000, 0b01000001, 0b01111111, 0b01000000, 0b00000000,0b00000000, 0b00000000, 0
    b00000000}, // l
{0b01111100, 0b00000100, 0b00011000, 0b00000100, 0b01111000,0b00000000, 0b00000000, 0
    b00000000}, // m
{0b01111100, 0b00001000, 0b00000100, 0b00000100, 0b01111000,0b00000000, 0b00000000, 0
    b00000000}, // n
{0b00111000, 0b01000100, 0b01000100, 0b01000100, 0b00111000,0b00000000, 0b00000000, 0
    b00000000}, // o
{0b01111100, 0b00010100, 0b00010100, 0b00010100, 0b00001000,0b00000000, 0b00000000, 0
    b00000000}, // p
{0b00001000, 0b00010100, 0b00010100, 0b00011000, 0b01111100,0b00000000, 0b00000000, 0
    b00000000}, // q
{0b01111100, 0b00001000, 0b00000100, 0b00000100, 0b00001000,0b00000000, 0b00000000, 0
    b00000000}, // r
{0b01001000, 0b01010100, 0b01010100, 0b01010100, 0b00100000,0b00000000, 0b00000000, 0
    b00000000}, // s
{0b00000100, 0b00111111, 0b01000100, 0b01000000, 0b00100000,0b00000000, 0b00000000, 0
    b00000000}, // t
```

```
    {0b00111100, 0b01000000, 0b01000000, 0b00100000, 0b01111100,0b00000000, 0b00000000, 0
        b00000000}, // u
    {0b00011100, 0b00100000, 0b01000000, 0b00100000, 0b00011100,0b00000000, 0b00000000, 0
        b00000000}, // v
    {0b00111100, 0b01000000, 0b00111000, 0b01000000, 0b00111100,0b00000000, 0b00000000, 0
        b00000000}, // w
    {0b01000100, 0b00101000, 0b00010000, 0b00101000, 0b01000100,0b00000000, 0b00000000, 0
        b00000000}, // x
    {0b00001100, 0b01010000, 0b01010000, 0b01010000, 0b00111100,0b00000000, 0b00000000, 0
        b00000000}, // y
    {0b01000100, 0b01100100, 0b01010100, 0b01001100, 0b01000100,0b00000000, 0b00000000, 0
        b00000000}, // z
    {0b00000000, 0b00001000, 0b00110110, 0b01000001, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // {
    {0b00000000, 0b00000000, 0b01111111, 0b00000000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // |
    {0b00000000, 0b01000001, 0b00110110, 0b00001000, 0b00000000,0b00000000, 0b00000000, 0
        b00000000}, // }
    {0b00001000, 0b00001000, 0b00101010, 0b00011100, 0b00001000,0b00000000, 0b00000000, 0
        b00000000}, // ~
    {0b00001000, 0b00011100, 0b00101010, 0b00001000, 0b00001000,0b00000000, 0b00000000, 0
        b00000000} // <-
};



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

    SystemClock48MHz();

    trace_printf("This␣is␣the␣Project\n");
    trace_printf("System␣clock:␣%u␣Hz\n", SystemCoreClock);

    myGPIOA_Init(); // initialize I/O port PA
    myGPIOB_Init(); // initalize PB
    myDAC_Init(); // initialize DAC
    myTIM2_Init(); //Initialize timer TIM2
    myADC_Init(); // initialize ADC
    oled_config(); // configure the screen
    refresh_OLED(); // clock the first tick of the screen so that it is up before any other
        interupts
    myEXTI_Init(); //initialize EXTI


    while (1)
    {
       read_DAC(); // read dac (cont. updates the dac value as well)

       // Only print if the ISR says we have new data
```

```
        if (newDataReady == 1) {
                trace_printf("F: %u\n", Freq);
                trace_printf("R: %u\n", Res);
                newDataReady = 0; // Reset flag
            }
        refresh_OLED(); // refresh screen
    }
    return 0;
}



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

    /* Configure PA1 as Analog for ADC CH1 */
    // To read POT voltage
    // Relevant register: GPIOA->MODER
    GPIOA->MODER |= GPIO_MODER_MODER1; // Set bits 2-3 to 11 (Analog)
    /* Ensure no pull-up/pull-down for PA1 */
    // Relevant register: GPIOA->PUPDR
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);

    /* Configure PA4 as analog output */
    // output control voltage to optocoupler
    GPIOA->MODER |= 0b001100000000; // 0011 0000 0000 change bit 8 and 9
    /* Ensure no pull-up/pull-down for PA4 */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);

    /* Configure PA5 as default input (was ADC) */
    GPIOA->MODER &= ~(0b110000000000); // 1100 0000 0000 change bit 10 and 11 (set to 00)
    /* Ensure no pull-up/pull-down for PA5 */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR5);
}

//measures the period between two rising edges of the selected input signal
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;
    /* Start counting timer pulses */ //maybe not needed (remove comment later)
    TIM2->CR1 |= TIM_CR1_CEN;
}


void myEXTI_Init()
{
    /* Enable clock for SYSCFG */
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGCOMPEN;

    /* Map EXTI lines to GPIOB */
    // SYSCFG->EXTICR[0] covers EXTI0-3
    // 0001: PB[x]
    SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI2_Msk | SYSCFG_EXTICR1_EXTI3_Msk); // Clear
        EXTI2 and EXTI3
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI2_PB; // Map EXTI2 to PB2
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI3_PB; // Map EXTI3 to PB3

    /* Map EXTI0 to GPIOA (default, but good to be explicit) */
    SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI0_Msk); // Map EXTI0 to PA0

    /* all EXTI line interrupts: set rising-edge trigger */
    EXTI->RTSR |= (EXTI_RTSR_TR0); // Button
    EXTI->RTSR |= (EXTI_RTSR_TR2); // PB2 (FG)
    EXTI->RTSR |= (EXTI_RTSR_TR3); // PB3 (555)

    /* Unmask interrupts from each EXTI line */
    EXTI->IMR |= (EXTI_IMR_MR0); // Button
    EXTI->IMR |= (EXTI_IMR_MR2); // PB2 (FG)
    EXTI->IMR |= (EXTI_IMR_MR3); // PB3 (555)
```

```
    /* Assign EXTI2_3 interrupt priority = 1 in NVIC */
    NVIC_SetPriority(EXTI2_3_IRQn,1);

    /* Enable EXTI2_3 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(EXTI2_3_IRQn);

    /* Assign EXTI0_1 interrupt priority = 0 in NVIC */
    NVIC_SetPriority(EXTI0_1_IRQn,0);

    /* Enable EXTI0_1 interrupts in NVIC */
    NVIC_EnableIRQ(EXTI0_1_IRQn);
}


/* This handler is declared in system/src/cmsis/vectors_stm32f051x8.c */
void TIM2_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM2->SR & TIM_SR_UIF) != 0)
    {
        trace_printf("\n*** Overflow! ***\n");

        /* Clear update interrupt flag */
        // Relevant register: TIM2->SR
        TIM2->SR &= ~(TIM_SR_UIF);

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

/* This handler is declared in system/src/cmsis/vectors_stm32f051x8.c */

void EXTI0_1_IRQHandler(){
    /* This handler now ONLY handles the button press for inSig swap */

    if ((EXTI->PR & EXTI_PR_PR0) != 0) { // intterupt 0
        /* If inSig = 0, then: let inSig = 1, disable
        EXTI3 interrupts (555), enable EXTI2 interrupts (FG)
        Else: let inSig = 0, disable EXTI2 interrupts (FG),
        enable EXTI3 interrupts (555)
        */

        EXTI->PR |= 0x1; // clear interupt pending flag for exti 0

    // inSig = 0 is 555
    // inSig = 1 is fg
```

```
        if(inSig == 0){
            inSig = 1; // swap
            EXTI->IMR &= ~(EXTI_IMR_MR3);//disable exti3 (555)
            EXTI->IMR |= (EXTI_IMR_MR2); //enable exti2 (FG)
        }else{
            inSig = 0; // swap
            EXTI->IMR &= ~(EXTI_IMR_MR2);//disable exti2 (FG)
            EXTI->IMR |= (EXTI_IMR_MR3);//enable exti3 (555)
        }
    }

}

void EXTI2_3_IRQHandler()
{
    double freq;
    uint32_t count;
    uint32_t NOISE_THRESHOLD = 2400;

    // --- 555 Timer (PB3) ---
    if ((EXTI->PR & EXTI_PR_PR3) != 0) {
        if(timerTriggered == 0){
            timerTriggered = 1;
            TIM2->CNT = 0;
            TIM2->CR1 |= TIM_CR1_CEN;
        } else {
            count = TIM2->CNT;
            if (count > NOISE_THRESHOLD) {
                timerTriggered = 0;
                TIM2->CR1 &= ~(TIM_CR1_CEN);

                freq = ((double)SystemCoreClock)/((double)count);

                // Update globals
                Freq = (unsigned int)freq;
                Res = (unsigned int)Potentiometer_resistance();

                // Set flag, DO NOT PRINT
                newDataReady = 1;
            }
        }
        EXTI->PR |= EXTI_PR_PR3;
    }

    // --- Function Generator (PB2) ---
    if ((EXTI->PR & EXTI_PR_PR2) != 0){
        if(timerTriggered == 0){
            timerTriggered = 1;
            TIM2->CNT = 0;
            TIM2->CR1 |= TIM_CR1_CEN;
```

```
        } else {
            count = TIM2->CNT;
            if (count > NOISE_THRESHOLD) {
                timerTriggered = 0;
                TIM2->CR1 &= ~(TIM_CR1_CEN);

                freq = ((double)SystemCoreClock)/((double)count);

                Freq = (unsigned int)freq;
                Res = (unsigned int)Potentiometer_resistance();

                // Set flag, DO NOT PRINT
                newDataReady = 1;
            }
        }
        EXTI->PR |= EXTI_PR_PR2;
    }
}


// initialize ADC
void myADC_Init(void){

    // initalize clock for ADC
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;

// 1. Ensure that ADEN=0
// 2. Set ADCAL=1
// 3. Wait until ADCAL=0
// 4. The calibration factor can be read from bits 6:0 of ADC_DR.

    if ((ADC1->CR & ADC_CR_ADEN) != 0){ // if aden is not 0
        ADC1->CR |= ADC_CR_ADDIS; // clear aden by addis (adc is disabled)
        while ((ADC1->CR & ADC_CR_ADEN) != 0); // timeout for adc disable
    }
    ADC1->CR |= ADC_CR_ADCAL; // calibrate by setting adcal
    while(ADC1->CR & ADC_CR_ADCAL); // timeout until calibration complete


    ADC1->CHSELR = ADC_CHSELR_CHSEL1; // MODIFIED: select CHSEL1 for ADC channel 1 (PA1)

    ADC1->SMPR |= 7; // max smp = 0b111 = 7 (239.5 ADC clock cycles)


    // change configuration registers
    ADC1->CFGR1 &= ~ADC_CFGR1_RES; // enable data resolution (12-bit)

    ADC1->CFGR1 &= ~ADC_CFGR1_ALIGN; // right align converted data

    ADC1->CFGR1 |= ADC_CFGR1_OVRMOD; // enable overrun management mode
```

```
    ADC1->CFGR1 |= ADC_CFGR1_CONT; // enable continuous conversion mode


    ADC1->CR |= ADC_CR_ADEN; // re enable ADC now that configuration register are changed

    while ((ADC1->ISR & ADC_ISR_ADRDY) == 0); // timeout until ADC re enabled

    ADC1->CR |= ADC_CR_ADSTART; // ADC start (works as aden has been re enabkled)

    trace_printf("ADC initialized\n");
}

// initialize DAC
void myDAC_Init(void){

    RCC->APB1ENR |= RCC_APB1ENR_DACEN; // enable dac clock

    DAC->CR |= DAC_CR_EN1; // enable channel1 of DAC

    trace_printf("DAC initialized\n");
}

uint32_t Potentiometer_resistance(){
    uint32_t ADC_value = Potentiometer_voltage();
    // Map 0-4095 (ADC Range) to 0-10000 (Resistance Range)
    return (uint32_t)((ADC_value * 10000.0f) / 4095.0f);
}

uint32_t Potentiometer_voltage(){
    return (ADC1->DR & 0xFFF); // masking the result to ensure only lower 12 bits
}

void read_DAC(){
    // 1. Get raw voltage
    uint32_t raw_value = Potentiometer_voltage();

    // 2. Send to DAC
    DAC->DHR12R1 = raw_value;

    // 3. Calculate Resistance
    uint32_t calculated_res = (uint32_t)((raw_value * 10000.0f) / 4095.0f);

    // 4. THE FIX: Deadzone for clean "0" display
    if (calculated_res < 50) {
        Res = 0;
    } else {
        Res = calculated_res;
    }
}
```

```
void myGPIOB_Init()
{
    // MODIFIED FUNCTION TO MATCH NEW PINOUT
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    // 00: Input mode (reset state)
    // 01: General purpose output mode
    // 10: Alternate function mode
    // 11: Analog mode

    /* Configure PB2, PB3 as Input */
    //PB2 -> function generator Input
    //PB3 -> 555 timer input
    GPIOB->MODER &= ~(GPIO_MODER_MODER2 | GPIO_MODER_MODER3);


    /* Configure PB8, PB9, PB11 as General purpose output mode (01) */
    GPIOB->MODER &= ~(GPIO_MODER_MODER8 | GPIO_MODER_MODER9 | GPIO_MODER_MODER11);
    GPIOB->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0 | GPIO_MODER_MODER11_0);

    /* Configure PB13, PB15 as Alternate function mode (10) */
    GPIOB->MODER &= ~(GPIO_MODER_MODER13 | GPIO_MODER_MODER15);
    GPIOB->MODER |= (GPIO_MODER_MODER13_1 | GPIO_MODER_MODER15_1);

    /* Ensure no pull-up/pull-down for each */
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR2 | GPIO_PUPDR_PUPDR3 |
                      GPIO_PUPDR_PUPDR8 | GPIO_PUPDR_PUPDR9 | GPIO_PUPDR_PUPDR11 |
                      GPIO_PUPDR_PUPDR13 | GPIO_PUPDR_PUPDR15);
}


void refresh_OLED( void )
{
    // Buffer size
    unsigned char Buffer[32];

    // --- ROW 1: Resistance ---
    snprintf( (char*)Buffer, sizeof( Buffer ), "␣␣R:␣%5u␣Ohms␣␣", Res );

    oled_Write_Cmd(0xB3); // Select Page 3
    oled_Write_Cmd(0x03);
    oled_Write_Cmd(0x10);
    for(int i = 0; i < 16; i++){
        for(int j = 0; j < 8; j++){
            unsigned char result = Characters[Buffer[i]][j];
            oled_Write_Data(result);
        }
    }
```

```
    // --- ROW 2: Frequency ---
    snprintf( (char*)Buffer, sizeof( Buffer ), "  F:  %5u  Hz    ", Freq );

    oled_Write_Cmd(0xB5); // Select Page 5
    oled_Write_Cmd(0x03);
    oled_Write_Cmd(0x10);
    for(int i = 0; i < 16; i++){
        for(int j = 0; j < 8; j++){
            unsigned char result = Characters[Buffer[i]][j];
            oled_Write_Data(result);
        }
    }

    // Note: The "Source" block (Page B1) has been removed entirely.

    // Small delay to prevent screen tearing if refreshing too fast
    for(volatile int i = 0; i < 1000; i++);
}

void oled_Write_Cmd( unsigned char cmd ) // this function is to send command byte to OLED,
     initializing display
{

    GPIOB->ODR |= GPIO_ODR_8; // make PB8 = CS# = 1
    GPIOB->ODR &= ~(GPIO_ODR_9); // make PB9 = D/C# = 0
    GPIOB->ODR &= ~(GPIO_ODR_8); // make PB8 = CS# = 0
    oled_Write( cmd ); // write
    GPIOB->ODR |= GPIO_ODR_8; // make PB8 = CS# = 1
}

void oled_Write_Data( unsigned char data ) // this function is to send data byte to OLED,
     data represents the actual pixel (draw characters)
{
    GPIOB->ODR |= GPIO_ODR_8; // make PB8 = CS# = 1
    GPIOB->ODR |= GPIO_ODR_9; // make PB9 = D/C# = 1
    GPIOB->ODR &= ~(GPIO_ODR_8); // make PB8 = CS# = 0
    oled_Write( data );
    GPIOB->ODR |= GPIO_ODR_8; // make PB8 = CS# = 1
}


void oled_Write( unsigned char Value ) //send single byte over spi to oled
{

    while((SPI2->SR & SPI_SR_TXE) == 0);

    HAL_SPI_Transmit( &SPI_Handle, &Value, 1, HAL_MAX_DELAY );

    while((SPI2->SR & SPI_SR_BSY) != 0);
}
```

```
void oled_config( void )
{
    // Note: GPIOB clock is enabled in myGPIOB_Init()

    // PB13 is AFR[1] (high reg), index 5 (13-8)
    // PB15 is AFR[1] (high reg), index 7 (15-8)
    GPIOB->AFR[1] &= ~(0x0F << (5 * 4)); // Clear AF for PB13
    GPIOB->AFR[1] &= ~(0x0F << (7 * 4)); // Clear AF for PB15
    // AF0 is 0x0, so clearing is sufficient.

    RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;


    SPI_Handle.Instance = SPI2;

    SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
    SPI_Handle.Init.Mode = SPI_MODE_MASTER;
    SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
    SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
    SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
    SPI_Handle.Init.NSS = SPI_NSS_SOFT;
    SPI_Handle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
    SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;
    SPI_Handle.Init.CRCPolynomial = 7;

// Initialize the SPI interface
    HAL_SPI_Init( &SPI_Handle );

// Enable the SPI
    __HAL_SPI_ENABLE( &SPI_Handle );


    //Reset LED display

    GPIOB->ODR &= ~(GPIO_ODR_11); // make pin PB11 = 0, wait for a few ms
    for(volatile int i = 0; i < 100000; i++); //wait (Increased delay)
    GPIOB->ODR |= GPIO_ODR_11; // make pin PB11 = 1, wait for a few ms
    for(volatile int i = 0; i < 100000; i++); //wait (Increased delay)

    // 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 memory with zeros
```

```
    for (int PAGE = 0xB0; PAGE <= 0xB7; PAGE++){ // the loop goes thru 8 pages of the OLED
        oled_Write_Cmd(PAGE); //select page from 0xb0 to 0xb7
        oled_Write_Cmd(0x00); //fill every pixel on each page with 0x00, set lower column to 0
        oled_Write_Cmd(0x10); //select upper nibble for the column
        for (int SEG = 0; SEG < 128; SEG++){
            oled_Write_Data( 0x00 ); // each page has 128 segments
        }
    }

    trace_printf("OLED configured\n");
}


#pragma GCC diagnostic pop
```