University of Victoria

Department of Electrical & Computer Engineering

Fall 2022

ECE 355 Project Report

Adam Claxton V00830677
Salem Ait Ami V00932871
Lab Section B03
November 29, 2022

The lab report marks are distributed as follows:

* Problem Description/Specifications:	(5) _____
* Design/Solution	(15) _____
* Testing/Results	(10) _____
* Discussion	(15) _____
* Code Design and Documentation:	(15) _____
* **Total**	**(60)** _____

Content

Table of Figures 3

Problem Description and Specifications 4

Design and Solutions 4

- External Trigger and Timer 2 5
- Analog to Digital Converter 8
- Digital to Analog Converter 8
- SPI and LCD 9
- GPIO 14
- Main Function 15
- Circuitry and Wiring 17

Testing and Results 18

- Extern and Timer 18
- ADC 18
- DAC 18
- SPI and LCD 18
- Circuitry 19

Discussion 19

Appendix A 21

- Code Design and Documentation 21

Table of Figures

Table of Figures 3
Problem Description and Specifications 4
Design and Solutions 4
External Trigger and Timer 2 5
Analog to Digital Converter 8
Digital to Analog Converter 8
SPI and LCD 9
GPIO 14
Main Function 15
Circuitry and Wiring 17
Testing and Results 18
Extern and Timer 18
ADC 18
DAC 18
SPI and LCD 18
Circuitry 19
Discussion 19
Appendix A 21
Code Design and Documentation 21

Problem Description and Specifications

The goal of this project is to successfully implement an embedded system using an STM32F051 microcontroller. The specifications provided require students to make use of several hardware components provided by the microcontroller along with some that are added externally. Firstly, a potentiometer is used to control the voltage provided to the optocoupler, which in turn regulates the frequency generated by the 555 timer. Prior to passing the potentiometer voltage to the optocoupler it is first passed through the ADC and DAC respectively. Though this step is redundant, it is meant to familiarize students with these hardware components and the methods required to interface with them. Lastly, students configure an SPI in order to send data to an LCD screen, where frequency and resistance are to be displayed. GPIO pins provide the interface with which the microcontroller communicates with external peripherals and the pins are configured using a C program.

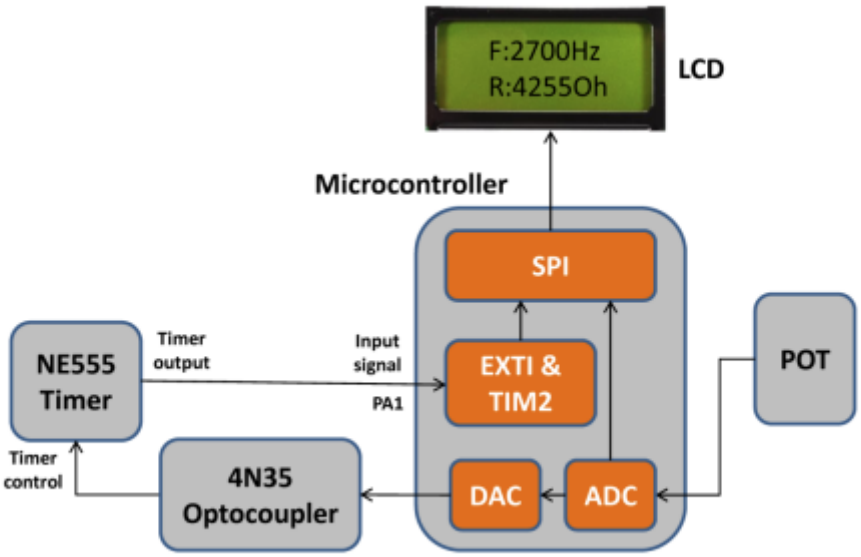

Figure 1: Project Design Specifications

Design and Solutions

The project was split up into several parts where each component was made to function individually before being put in series with the others. The first components were the EXTI (external interrupt) controller and TIM2 general-purpose timer, which were configured for the introductory labs. Next, the ADC was configured and wired to measure the value of the voltage across the potentiometer on the lab board. The next component to configure was the DAC that was used to convert the ADC value back to an analog voltage value. This voltage was then used to power an optocoupler which supplied a variable voltage to the 555 timer. The 555 timer supplied a square wave with a variable frequency based on the voltage supplied. By supplying this signal back to the external interrupt, we were able to measure the frequency of the signal created. The last part of the lab was to display the frequency of the

signal and the resistance of the potentiometer on the LCD screen provided on the lab board. The LCD required serial peripheral interface (SPI) communication messages to write the correct results and supporting information to the screen. The frequency and resistance values were calculated based on the timer 2 values and ADC values respectively. These values were then decomposed into individual digits and sent to the LCD.

External Trigger and Timer 2

The section of this project we decided to complete first was the External Trigger and Timer 2. The goal of this section is to measure the frequency of an applied square wave using a GPIO pin. The way we did this was to set up the GPIO pin as an external interrupt and use this interrupt to start and stop the Timer 2 on each rising edge of the signal. This gives us the number of clock cycles in 1 period of the signal. Knowing the number of clock cycles in one period and the system clock frequency of the STM we can calculate signal frequency by dividing system clock frequency by the number of clock cycles. This solution required setup of the Timer 2 and the EXTI external timer.

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

    //Enable timer interrupts
    TIM2->CR1 = ((uint16_t)0x008C);

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

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

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

    //Assign TIM2 interrupt priority = 0 in NVIC
    NVIC_SetPriority(TIM2_IRQn, 0);

    //Enable TIM2 interrupts in NVIC
    NVIC_EnableIRQ(TIM2_IRQn);

    //Enable update interrupt generation
    TIM2->DIER |= 0x1;
}
```

Figure 2: General-Purpose Timer 2 Initialization Code

Above is the timer 2 initialization function. This is called at the start of the program to configure the timer with the desired parameters. The first step is to enable the clock for the timer using the RCC->APB1ENR register. We then enable timer interrupts to allow the timer to interrupt the program if the maximum value is reached. Then we set the prescaler to 0 so it counts at the same rate as the system clock and set the auto reload delay to the maximum 32-bit value from the timer register. This allows us to measure the lowest possible frequencies. Then we update the registers so that the values are applied and set the priority of the TIM2_IRQn interrupt handler to 0 and enable it so that the interrupt will have the highest priority if the timer overflows.

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

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

        //Restart stopped timer
        TIM2->CR1 |= (1<<0);
    }
}
```

Figure 3: General-Purpose Timer 2 Interrupt Handler Code

In the timer 2 interrupt handler seen above we handle an overflow of the timer. If the timer is left on for long enough to reach its maximum value, it will call this interrupt that checks the interrupt flag and notifies the user that there was an overflow before clearing the flag and restarting the timer. This should only happen if we are trying to measure a frequency that is too low or the program somehow malfunctions and doesn't stop the timer correctly. This should not be called when measuring frequencies in the correct range.

```
void myEXTI_Init(){
    //Map EXTI1 line to PA1
    SYSCFG->EXTICR[0] |= 0x80;

    //EXTI1 line interrupts: set rising-edge trigger
    EXTI->RTSR |= (1<<1) ;

    //Unmask interrupts from EXTI1 line
    EXTI->IMR |= (1<<1);

    //Assign EXTI1 interrupt priority = 0 in NVIC
    NVIC_SetPriority(EXTI0_1_IRQn, 0);

    //Enable EXTI1 interrupts in NVIC
    NVIC_EnableIRQ(EXTI0_1_IRQn);
}
```

Figure 4: External Interrupt Initialization Code

The next part of this section is the EXTI external interrupt. The EXTI initialization function above shows how the interrupt was configured. Using the SYSCFG->EXTICR[0] register we set the external interrupt to GPIO pin PA1. THe RTSR register allows us to choose to trigger on the rising or falling edge of the input signal. Using the IMR register we can choose to mask or unmask the interrupt. This allows us to mute new interrupts while we are handling the previous interrupt. To initialize we unmask the interrupt. We then set the interrupt handler priority to 0 so it has the highest priority and enable it.

```
void EXTI0_1_IRQHandler(){
    //Check if EXTI1 interrupt pending flag is set
    if ((EXTI->PR & EXTI_PR_PR1) != 0)
    {
        //First timer rising edge; start count
        if(timerTriggered == 0){
            //Mask interrupt
            EXTI->IMR &= ~(1<<1);

            //Restart timer
            TIM2->CR1 |= (1<<0);
            //Reset count
            TIM2->CNT = 0;

            //Unmask interrupt
            EXTI->IMR |= (1<<1);
            //Count rising edge count
            timerTriggered = 1;
        }
        //Second timer rising edge; stop count
        else {
            //Mask interrupt
            EXTI->IMR &= ~(1<<1);

            //Stop timer
            TIM2->CR1 &= ~(1<<0);

            //Read count; calculate frequency
            unsigned long int count = TIM2->CNT;
            freq = fs/count;

            //Un-mask interrupt
            EXTI->IMR |= (1<<1);
            //Reset rising edge count
            timerTriggered = 0;
        }

        //Reset interrupt flag
        EXTI -> PR |= (1<<1);
    }
}
```

Figure 5: External Interrupt Handler Code

Lastly, we have the external interrupt handler seen above. When the signal has a rising-edge the EXTI-PR flag is set, and the interrupt handler is called. The handler first checks that the flag is indeed set. Then using the global variable timerTriggered we can determine if it is the first or second rising-edge. If it is the first edge, then the timerTriggered variable is 0. We then mask the interrupt, restart the timer 2 and set the timer 2 count back to zero. Then unmask the interrupt, set the timerTriggered to 1, reset the interrupt flag and wait for the second rising edge to call the handler again. On the second rising edge we mask the interrupt and stop the timer. Now we have the number of clock cycles in one period stored in TIM2->CNT. Dividing the 48MHz clock frequency by the timer count we can find the signal frequency which is then stored in a global variable for access later. Lastly, we unmask the

interrupt, set the timerTriggered back to 0 and reset the interrupt flag so the process can repeat.

Analog to Digital Converter

The analog to digital converter is built into the STM board. The onboard ADC is a 12 bit converter which means it has a resolution of 4095. The ADC is used to measure the voltage across a potentiometer that ranges from 0V to 3.3V. This voltage is converted to a digital value from 0 to 4095. This value is later used to calculate the resistance of the 5kΩ potentiometer. This requires the initialization of the onboard ADC and then the conversions are done using a polling approach in the main loop.

```
void ADC_Init(){
    ADC1->CR |= ADC_CR_ADEN; //Enable ADC
    ADC1->CR &= ~ADC_CR_ADSTART; //Disable start conversions
    ADC1->CFGR1 |= ADC_CFGR1_CONT; //Continuous conversion mode
    ADC1->CHSELR = ADC_CHSELR_CHSEL0; //Select channel 0 (disable others)
    ADC1->CFGR2 = ADC_CFGR2_CKMODE; //ADC Asynchronous clock mode
    ADC1->SMPR = ADC_SMPR_SMP_0; //Sampling time selection
    ADC1->CFGR1 &= ~ADC_CFGR1_AWDEN; //Disable watchdog
    ADC1->CFGR1 &= ~ADC_CFGR1_SCANDIR; //Scan sequence direction (upward scan)
    ADC1->CFGR1 &= ~ADC_CFGR1_ALIGN; //Data alignment (right)
    ADC1->CFGR1 &= ~ADC_CFGR1_EXTEN; //Disable external trigger conversion mode
    ADC1->CFGR1 &= ~ADC_CFGR1_OVRMOD; //Overrun management mode
    ADC1->CFGR1 &= ~ADC_CFGR1_WAIT; //Disable wait conversion mode
    ADC1->CFGR1 &= ~ADC_CFGR1_DISCEN; //Disable discontinuous mode
    ADC1->IER |= ADC_IER_ADRDYIE; //ADC Ready interrupt (enable)
    ADC1->IER |= ADC_IER_EOCIE; //End of conversion interrupt (enable)
    ADC1->IER |= ADC_IER_EOSMPIE; //End of sampling interrupt (enable)
    ADC->CCR |=  ADC_CCR_VREFEN; //Vref internal (enable)
    ADC1->CFGR1 &= ~ADC_CFGR1_RES; //Set resolution (12)
}
```

Figure 6: ADC Initialization Code

We initialized the ADC by setting all the options of all the relevant registers we could find in the reference manual. Most of the registers seen in the ADC initialization function seen above are explicitly set to their default value. The first step is to set the ADC1->CR enable to enable the ADC. The next step is to set ADC1->CFGR1 to continuous conversion mode. The next step is to set ADC1-CFGR1 to 12-bit resolution. The rest of the steps are not required as they are default or don't affect the performance in our application. As this was the first part, we configured it without assistance and didn't realize the default values didn't need to be set until we already had it working. The actual conversions done with the ADC can be seen in the main function section.

Digital to Analog Converter

The Digital to Analog Converter is an internal component of the STM board that converts digital values to analog output voltages. The onboard DAC is 12 bits, so the resolution is the same as the ADC at 4095. In this project we use the DAC to convert the digital value of the ADC from 0 to 4095 to an analog voltage ranging from 0V to 3.3V. The output voltage is then used to drive the optocoupler and 555 timer to create the square wave signal. Using the DAC required an initialization function and then the conversions were used in the main function.

```
void DAC_Init()
{
    //Enable DAC clock
    RCC->APB1ENR |= 0x20000000;
    //Enable the DAC Channel 1
    DAC->CR |= 0x1;
    //Set the trigger to software
    DAC->CR |= DAC_CR_TSEL1;
}
```

Figure 7: DAC Initialization Code

Learning from the ADC initialization function the DAC initialization is much smaller as we only configured the required parameters. The first thing we do is to enable the DAC clock using RCC -> APB1ENR. Next, we use DAC->CR to enable the DAC channel 1. This sets the DAC output to pin A4. Lastly, we set DAC->CR to a software trigger so that we can start the conversion using the software.

SPI and LCD

The last part of the project was to set up the SPI communication and LCD control. The project requires the LCD screen on the project board to display updated frequency of the 555 timer output signal and the resistance of the potentiometer. The LCD is controlled by providing it with 8-bit parallel messages. To avoid using 8 GPIO pins on the STM we use the Serial Peripheral Interface communication protocol to serially send 8 bits to an 8-bit shift register that then converts the serial message to a parallel message used by the LCD. This section first required SPI to be set up in an initialization function. Then, using this the SPI transmit function we setup 3 functions to control the LCD and an LCD initialization.

```
void SPI_Init(){
    //Enable clock for SPI1
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
    //Configure SPI1
    SPI_Handle.Instance = SPI1;
    SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
    SPI_Handle.Init.Mode = SPI_MODE_MASTER;
    SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
    SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
    SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
    SPI_Handle.Init.NSS = SPI_NSS_SOFT;
    SPI_Handle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
    SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;
    SPI_Handle.Init.CRCPolynomial = 7;

    //Initialize SPI instance with desired configuration
    HAL_SPI_Init(&SPI_Handle);
    //Enable SPI
    __HAL_SPI_ENABLE(&SPI_Handle);
}
```

Figure 8: SPI Initialization Code

The first part of this section is to set up the SPI communication protocol. The setup for the SPI1 register is provided in the lecture notes on Brightspace [1]. We enable the SPI clock

and set the rest of the parameters according to the instructions. It is set to 1 way communication, so we need 3 lines to communicate effectively. These lines are a system clock line which is set to PB3, a latch line which is set to PB4 and a data line that is set to PB5. The data is set to 8 bits so it will send an 8-bit message that can be used by the LCD. Setting the prescaller to 256 allowed for the slowest transmission rate to avoid overwhelming the LCD. Lastly, we enable the Hal SPI functions to allow us to use the built in SPI_Transmit function.

```
void LCD_SendBits(uint8_t Bits_8){
    //Set latch bit for PB4 low
    GPIOB->BSRR |= (uint32_t)0x100000;

    //Transmit 8 bits of data
    HAL_SPI_Transmit(&SPI_Handle, (uint8_t*)&Bits_8, sizeof(Bits_8), HAL_MAX_DELAY);

    //Wait for SPI buffer to be empty
    while((SPI1->SR & (uint16_t)0x2) == 0 ){};

    //Set latch bit for PB4 high
    GPIOB->BSRR |= (uint32_t)0x10;
}
```

Figure 9: LCD Send Bits Code

The first of the LCD functions is SendBits. This function sends 8 bits of data to the LCD via SPI. First, we set the latch bit low using the BSRR to allow for transmission. Then we use the built in HAL_SPI_Transmit function to send the 8 bits over SPI. Then we monitor SPI->SR to check when the TXE flag is cleared meaning the SPI buffer is empty. Lastly, we set the latch bit high to stop transmission and clock the message through the shift register to the LCD.

```
void LCD_Send_Message(uint16_t message){
    //Isolate RS bit
    uint8_t RS = (message & 0x0200)>>9;
    //Isolate High bits [7:4]
    uint8_t High = (message & 0x00F0)>> 4;
    //Isolate Low bits [3:0]
    uint8_t Low = (message & 0x000F);

    //Create first high message (EN = 0)
    uint8_t High0 = ((RS << 6 ) | High);
    //Create second high message (EN = 1)
    uint8_t High1 = ((1 << 7) | (RS << 6) | High);

    //Create first low message (EN = 0)
    uint8_t Low0 = ((RS << 6 ) | Low);
    //Create second low message (EN = 1)
    uint8_t Low1  = ((1 << 7) | (RS << 6) | Low);

    //Send 3 high messages
    LCD_SendBits(High0);
    LCD_SendBits(High1);
    LCD_SendBits(High0);

    //Send 3 low messages
    LCD_SendBits(Low0);
    LCD_SendBits(Low1);
    LCD_SendBits(Low0);
}
```

Figure 10: LCD Send Message Code

The next part is to condition the 8-bit LCD commands and 8-bit ascii characters into messages that can be used by the LCD. The overall message has an RS bit and an Enable bit in front of the 8-bit message for a total of 10 bits. Since we can only send 8 bits to the LCD at a time, we need to split the 8-bit message into 2 messages of 4 bits each. This is done by splitting them into a high bit's message or the upper 4 bits and a low bit's message or the lower 4 bits. Then they are appended to EN, RS and two zeros to create the messages sent to the LCD. For the LCD to collect the messages in 4-bit mode, each message must be sent 3 times. First with EN low then with EN high and lastly with EN low again. The RS bit determines if the LCD is receiving a command or a character.

```
void LCD_Init(){
    // Set to 4 bit mode; No low sent
    LCD_SendBits(0x02);
    LCD_SendBits(0x82);
    LCD_SendBits(0x02);

    //Set LCD to use 2 lines of characters
    LCD_Send_Message(0x0028);

    //Turn of cursor; turn on LCD
    LCD_Send_Message(0x000C);
    //Display shift off; auto-increment on
    LCD_Send_Message(0x0006);
    //Clear display
    LCD_Send_Message(0x0001);
}
```

Figure 11: LCD Initialization Code

The LCD requires its own initialization function to set it up with the correct parameters. We send the LCD a series of commands via SPI to get the desired functionality. The first initialization command sets the LCD to 4-bit mode. This is done by manually sending 8 bits at a time because the second half of the command is all zeros and doesn't require transmission. The next 4 messages sent to the LCD set 2 character lines, turn on the LCD and turn off the cursor, turn off display shift and auto-increment on and lastly to clear the display. These commands are found in the lecture notes on Brightspace [1] and the datasheet for the LCD [11].

```
void LCD_Write(uint16_t resistance, uint16_t frequency){
    // delay to allow for smooth update
    for(int i = 0; i < 500000; i++){}
    //Limit data to 4 digits for LCD space provided
    resistance = resistance % 10000;
    frequency = frequency % 10000;

    //Break data down into individual digits
    uint16_t res4 = resistance % 10;
    uint16_t res3 = ((resistance - res4) % 100)/10;
    uint16_t res2 = ((resistance - res3 - res4) % 1000)/100;
    uint16_t res1 = resistance/1000;
    uint16_t freq4 = frequency % 10;
    uint16_t freq3 = ((frequency - freq4) % 100)/10;
    uint16_t freq2 = ((frequency - freq3 - freq4) % 1000)/100;
    uint16_t freq1 = frequency/1000;

    // delay to allow for smooth update
    for(int i = 0; i < 500000; i++){}

    //Select first line of LCD
    LCD_Send_Message(0x0080);
    //Write 'F:' to LCD
    LCD_Send_Message(0x0200 | 0x0046);
    LCD_Send_Message(0x0200 | 0x003A);
    //Write frequency digits to LCD
    LCD_Send_Message(0x0200 | 0x0030 | freq1);
    LCD_Send_Message(0x0200 | 0x0030 | freq2);
    LCD_Send_Message(0x0200 | 0x0030 | freq3);
    LCD_Send_Message(0x0200 | 0x0030 | freq4);
    //Write 'Hz' to LCD
    LCD_Send_Message(0x0200 | 0x0048);
    LCD_Send_Message(0x0200 | 0x007A);

    //Select second line of LCD
    LCD_Send_Message(0x00C0);
    //Write 'R:' to LCD
    LCD_Send_Message(0x0200 | 0x0052);
    LCD_Send_Message(0x0200 | 0x003A);
    //Write resistance digits to LCD
    LCD_Send_Message(0x0200 | 0x0030 | res1);
    LCD_Send_Message(0x0200 | 0x0030 | res2);
    LCD_Send_Message(0x0200 | 0x0030 | res3);
    LCD_Send_Message(0x0200 | 0x0030 | res4);
    //Write ohm symbol to LCD
    LCD_Send_Message(0x0200 | 0x00F4);
}
```

Figure 12: LCD Write Code

The last step in the LCD control is to write the frequency and resistance values to the LCD. This is done using the LCD write function seen above. This function gets passed the frequency and resistance values as unsigned 16 bit integers. First, we use a for loop to delay the LCD in order to avoid overwhelming it. Then we limit the frequency and resistance values to 4 digits, so they don't over run their allotted space on the screen. Next, we break the values down into individual digits using a combination of subtraction, division and modulo

to get each digit. Then we write the frequency to the LCD by setting the LCD address to the first spot on the first line. Then we print ‘F:####Hz’ on the top line one character at a time using LCD_Send_Message. Then we set the LCD address to the first spot on the second line and print the resistance value in the same method. We print this one in the format of ‘R:####Ω’. This function prints the current frequency and resistance values to the LCD using a polling method and is called from the main loop.

GPIO

GPIO pins were the interface used for communicating with the board. GPIOA was configured to handle communications involving the potentiometer and 555 timer as input, along with the DAC as output. Using the alternate function modes, GPIOB was configured to handle communications with the SPI, which in turn is used to interface with the LCD.

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

    //Enable ADC1 clock
    RCC->APB2ENR |= (1<<9);

    //PA0 as input for pot
    GPIOA->MODER &= ~(GPIO_MODER_MODER0);

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

    //PA4 as general-purpose output mode for DAC output
    GPIOA->MODER  |= GPIO_MODER_MODER4;

    //Ensure no pull-up/pull-down
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
}
```

Figure 13: GPIOA Initialization Code

In order to use any GPIO interface it must first be enabled using the RCC->AHBENR register. Next, the clock for the ADC is enabled using the RCC->AHB2ENR register. Though this step could’ve been carried out in the ADC_Init() function, we elected to do it here. Next, PA0 is initialized as an input pin along with PA1 by setting their respective bits to 00 in the GPIOA->MODER register. We then connect PA0 with the potentiometer and PA1 with the 555 timer by wiring them together. Then, PA4 is initialized to output mode by using the same GPIOA->MODER register so that it can be used as the output of the DAC. Lastly, using the GPIOA->PUPDR we ensure there is no pull-up or pull-down on any of the pins we have initialized.

```
void myGPIOB_Init(){
    //Enable GPIOB
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    //Set PB3 to alternate function mode
    //PB3 as SPI clock
    GPIOB->MODER |= GPIO_MODER_MODER3_1;

    //PB4 as general-purpose output mode
    //PB4 as SPI latch
    GPIOB->MODER |= GPIO_MODER_MODER4_0;

    //Set PB5 to alternate function mode
    //PB5 as MOSI
    GPIOB->MODER |= GPIO_MODER_MODER5_1;

    //Ensure no pull-up/pull-down
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR5);
}
```

Figure 14: GPIOB Initialization Code

Like GPIOA, the first step is to enable GPIOB through the RCC->AHBENR register. However, unlike with GPIOA we make use of the alternate function mode of PB3 and PB5. Using the alternate functions table provided for GPIOB we set PB3 and PB5 to AF0, which corresponds to SPI1_SCK (SPI clock) and SPI1_MOSI (SPI Master-out Slave-in) respectively. PB4 is then initialized as an output pin which corresponds to the SPI latch. Lastly, we ensure no pull-up or pull-down for any of the initialized pins.

Main Function

The main function is where all the initialization functions are called before the main loop is implemented. The main loop is where the ADC and DAC conversions take place and where the LCD_Write function is called. These functions use a polling method to step through each step and get the current values. Using a while(1) infinite loop we cause the STM to repeatedly check these values forever.

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

    SystemClock48MHz();
    trace_printf("System clock: %u Hz\n", SystemCoreClock);

    myGPIOA_Init();     /* Initialize I/O port PA */
    myGPIOB_Init();     /* Initialize I/O port PB */
    myTIM2_Init();      /* Initialize timer TIM2 */
    myEXTI_Init();      /* Initialize EXTI */
    ADC_Init();         /*Initialize ADC*/
    DAC_Init();         /*Initialize DAC*/
    SPI_Init();         /*Initialize SPI*/
    LCD_Init();         /*Initialize LCD*/

    //Variables for resistance and frequency for LCD screen
    uint16_t resistance = 0;
    float frequency = 0;

    while (1)
    {
        //Start conversions
        ADC1->CR |= ADC_CR_ADSTART;
        //Wait for EOC flag
        while(!(ADC1->ISR & ADC_ISR_EOC));

        //Calculate resistance from ADC value
        resistance = (ADC1->DR*5000)/4095;
        //Frequency placeholder for LCD
        frequency = freq;

        //Load DAC buffer with ADC data
        DAC->DHR12R1 = ADC1->DR;
        //Trigger software trigger to load DAC
        DAC->SWTRIGR = DAC_SWTRIGR_SWTRIG1;

        //Wait for DAC conversion
        while(DAC->SWTRIGR);
        //Write resistance and frequency to LCD
        LCD_Write(resistance, (uint16_t)frequency);
    }

    return 0;
}
```

Figure 15: Main Function Code

As we can see from the main function above, the first thing we do is update the system clock frequency to 48MHz. Then we call all the initialization functions to configure each part of the project correctly. Once everything is initialized the program enters the main loop where the ADC conversions are started and the ISR is monitored to see when the conversions are done. Once the ADC conversion is complete the resistance value is calculated for the potentiometer and stored in the resistance value. The frequency value is calculated in the EXTI interrupt handler and placed in a global variable. Here we set the local variable equal to the global variable to avoid any race conditions while in the middle of an LCD write. Next, we load the DAC buffer with the ADC value and trigger the software trigger to trigger the

DAC conversion. We then wait for the DAC conversion to finish and then write the updated frequency and resistance values to the LCD screen.

Circuitry and Wiring

The circuit was wired according to the circuit diagram provided in the lecture notes on Brightspace [1].

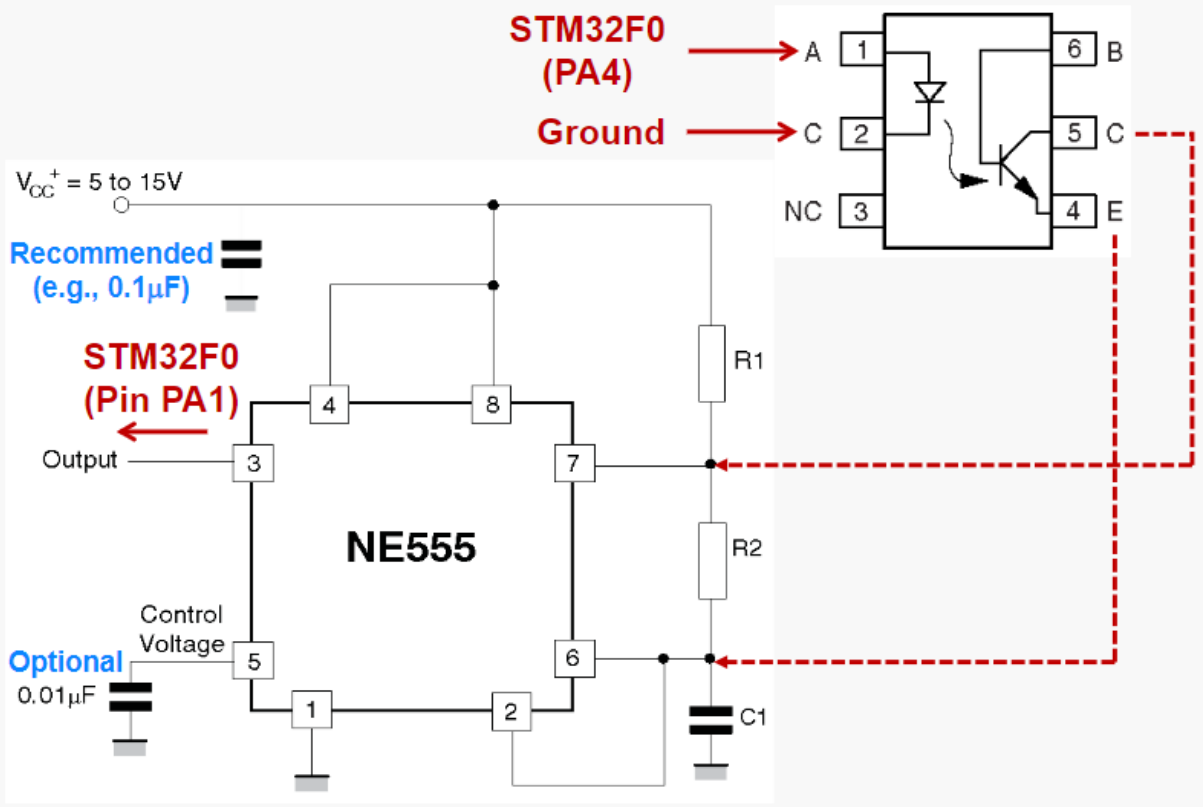

Figure 16: 555 Timer Wiring Diagram

The circuit has the output voltage from the DAC on pin PA4 enter the optocoupler to isolate the 3.3V of the microcontroller from the 5V of the 555 timer. The output of the optocoupler is then attached to the input of the 555 timer. Using the configuration above the 555 timer produces a square wave that has variable frequency depending on the voltage provided by the optocoupler. The output signal of the 555 timer is then input in the pin PA1 as the external interrupt to measure the frequency. There is also a 5KΩ potentiometer that is connect from 3.3V to ground with the wiper pin connected to pin PA0 to be read by the ADC to determine resistance and be converted by the DAC.

Testing and Results

Extern and Timer

The testing of the External Trigger and Timer 2 setup to measure frequency was verified using a function generator to create the square wave and an oscilloscope to measure the frequency. Theoretically we should be able to measure any frequency that has at least 1 clock cycle in each period and less than 4,294,967,296 clock cycles in each period so that the timer can count the correct number of clock cycles. This would be a frequency range of 10mHz to 48MHz. Unfortunately, the realistic values we found were about .012Hz and roughly 350Mhz. The upper bound is not a good as we hoped because the calculation of the values caused a delay in the count of the clock cycles. The lower end was almost what we expected but we had to account for the overflow of the timer so we couldn't get quite as low as we wanted. In the final design the circuit only produced frequencies from 900 Hz to 1400 Hz so these constraints were not a concern. We did notice that the higher the frequency the less accurate the reading. This is because the delay causes a certain number of clock cycles to be missed from the period. The higher the frequency to lower the clock cycles so the missed clock cycles account for a larger percent error.

ADC

The ADC was tested by printing the values stored in its data register in the main function loop. Since we initialized the ADC with a 12-bit data register and we know the reference voltage is 3.3 V, calculating the ADC voltage resolution gives us 0.805 mV. This tells us that every integer increment corresponds to an increase of 0.805 mV to the potentiometer voltage. However, since the ADC is only capable of converting voltage to a 12-bit number some information is lost on the original voltage.

DAC

The DAC was tested by calculating the desired voltage based on the digital value from the ADC and then measuring the actual voltage with the digital multimeter in the lab. We found that the output voltage matched the calculated voltage. One downfall to the DAC is that it only has a 12-bit output and the reference voltage is 0.805mV. This means that it can only produce voltages in steps of 0.805mV and will lose accuracy compared to taking the voltage directly from the potentiometer.

SPI and LCD

The SPI was tested by configuring the SPI hardware based on the configuration provided in the lecture notes [1]. It was tested for proper functionality by monitoring the output signals of the latch and clock lines against the system clock line using an oscilloscope. By sending a known message and monitoring the outputs of each line against the clock we could verify that we were sending the correct 8-bit message. From there we had to follow the lecture notes and data sheet to send the proper initialization messages to the LCD. This was tested by sending the initialization and then printing a known character to the screen. Once we could reliably print, we found that printing the frequency and resistance values was quite

easy. We did have to add a few delays into the LCD update function because the display would change so fast that the numbers were not readable.

Circuitry

The circuitry was very easy to test. We tested by wiring based on the provided circuit diagram and then providing the voltage from the potentiometer directly to the input of the optocoupler and measuring the output with the oscilloscope. There was nothing to design for this portion as the diagram was provided but the circuit does have limitations. The main limitation is that the 555 timer only provides a frequency range of 900 Hz to 1400 Hz given the supply voltage and capacitors in the circuit. This is much less than the measurable range of the software. The other limitation is that the optocoupler has a diode inside that doesn't turn on until 0.7V. While this device isolates the two supply voltages is also limits the range of voltages provided to the 555 timer.

Discussion

The project shows that the design specifications can be met in a reasonable amount of time with the hardware provided. A lot of research and reading is required in order to make use of every component and becoming familiar with the reference manual is a necessity. The trace_printf() function was very helpful for debugging purposes, but it's important to note that adding too many print statements will negatively affect performance. It is also important to note that frequency for the 555 timer doesn't change between 0 V - 0.7 V. This is due to the optocoupler's forward voltage which causes the 555 timer to interpret all voltages in that range as 0V.

The debugging process for this embedded system was very instructive as it required more "outside the box" thinking than a regular programming project. In many cases simply printing a variable to the console was not sufficient for ensuring a component's functionality. For example, when configuring the DAC, the first step was ensuring that the DAC->DOR1 (Data Output Register) register was properly loaded with the right values. This was easy enough as printing it to the console was enough to inspire confidence. The next part involved ensuring that a voltage was outputted and that it matched our specifications correctly. Since it is not feasible to "print" a voltage, other tools needed to be used, such as a digital multimeter, in order to give us confidence in our design.

In conclusion, this project was an instructive introduction into the world of embedded programming. It gave us insight and experience into how hardware and software work in tangent and helped us get familiar with the nomenclature and conventions of embedded systems.

References

[1] D. Capson and S. Shuja, “Lecture Notes,” *Brightspace ECE 355 Home Page*, 2022. [Online]. Available: https://bright.uvic.ca/d2l/home/218760. [Accessed: 25-Nov-2022].

[2] W. El-Kharashi and D. Capson, “Electrical and Computer Engineering - University of Victoria,” *ECE 355 Lab Information*, 2022. [Online]. Available: https://ece.engr.uvic.ca/~ece355/lab/. [Accessed: 25-Nov-2022].

[3] Admin, “DMA with ADC using registers in STM32 " controllerstech,” *ControllersTech*, 10-Oct-2022. [Online]. Available: https://controllerstech.com/dma-with-adc-using-registers-in-stm32/. [Accessed: 25-Nov-2022].

[4] P. Dhaker, “Introduction to SPI interface,” *Introduction to SPI Interface | Analog Devices*, Sep-2018. [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html. [Accessed: 25-Nov-2022].

[5] STMicroelectronics, “PM0215 Programming manual,” STM32F0xxx Cortex-M0 programming manual, 2012.

[6] STMicroelectronics, “RM0091 Reference manual,” STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs, 2014.

[7] STMicroelectronics, “STM32F051xx,” ARM-based 32-bit MCU, 16 to 64-KB Flash, timers, ADC, DAC and communication interfaces, 2.0-3.6 V , 2014.

[8] STMicroelectronics, “NE555 SA555 - SE555,” General-purpose single bipolar timers, 2012.

[9] Vishay Semiconductors, “4N35, 4N36, 4N37,” Optocoupler, Phototransistor Output, with Base Connection, 2012 [02-Oct-12].

[10] Texas Instruments, “SN54HC595 SN74HC595,” 8-BIT SHIFT REGISTERS WITH 3-STATE OUTPUT REGISTERS, 2014.

[11] Hitachi, “HD44780U (LCD-II),” Dot Matrix Liquid Crystal Display Controller/Driver, 1998.

Appendix A

Code Design and Documentation

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

// ----------------------------------------------------------------------
// School: University of Victoria, Canada.
// Course: ECE 355 "Microprocessor-Based Systems".
// This is template code for Part 2 of Introductory Lab.
//
// See "system/include/cmsis/stm32f0xx.h" for register/bit definitions.
// See "system/src/cmsis/vectors_stm32f0xx.c" for handler declarations.
// ----------------------------------------------------------------------

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

// ----------------------------------------------------------------------
//
// STM32F0 empty sample (trace via $(trace)).
//
// Trace support is enabled by adding the TRACE macro definition.
// By default the trace messages are forwarded to the $(trace) output,
// but can be rerouted to any device or completely suppressed, by
// changing the definitions required in system/src/diag/trace_impl.c
// (currently OS_USE_TRACE_ITM, OS_USE_TRACE_SEMIHOSTING_DEBUG/_STDOUT).
//

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

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


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

```
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)

//Function prototypes
void myGPIOA_Init(void);
void myGPIOB_Init(void);
void myTIM2_Init(void);
void myEXTI_Init(void);
void ADC_Init(void);
void DAC_Init(void);
void SPI_Init(void);
void LCD_Init(void);
void LCD_SendBits(uint8_t);
void LCD_Send_Message(uint16_t);
void LCD_Write(uint16_t, uint16_t);

//Global variables
int timerTriggered = 0;
float fs = 48000000;
float freq = 0;
SPI_HandleTypeDef SPI_Handle;

void SystemClock48MHz( void )
{
//
// Disable the PLL
//
  RCC->CR &= ~(RCC_CR_PLLON);
//
// Wait for the PLL to unlock
//
  while (( RCC->CR & RCC_CR_PLLRDY ) != 0 );
//
// Configure the PLL for a 48MHz 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();

}

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

        SystemClock48MHz();
        trace_printf("System clock: %u Hz\n", SystemCoreClock);

        myGPIOA_Init();                 /* Initialize I/O port PA */
        myGPIOB_Init();                 /* Initialize I/O port PB */
        myTIM2_Init();          /* Initialize timer TIM2 */
        myEXTI_Init();          /* Initialize EXTI */
        ADC_Init();                     /*Initialize ADC*/
        DAC_Init();                     /*Initialize DAC*/
        SPI_Init();                     /*Initialize SPI*/
        LCD_Init();                     /*Initialize LCD*/

        //Variables for resistance and frequency for LCD screen
        uint16_t resistance = 0;
        float frequency = 0;

        while (1)
        {
                //Start conversions
                ADC1->CR |= ADC_CR_ADSTART;
                //Wait for EOC flag
                while(!(ADC1->ISR & ADC_ISR_EOC));

                //Calculate resistance from ADC value
                resistance = (ADC1->DR*5000)/4095;
                //Frequency placeholder for LCD
                frequency = freq;

                //Load DAC buffer with ADC data
                DAC->DHR12R1 = ADC1->DR;
                //Trigger software trigger to load DAC
```

```
			DAC->SWTRIGR = DAC_SWTRIGR_SWTRIG1;

			//Wait for DAC conversion
			while(DAC->SWTRIGR);
			//Write resistance and frequency to LCD
			LCD_Write(resistance, (uint16_t)frequency);
		}

		return 0;
}

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

		//Enable ADC1 clock
		RCC->APB2ENR |= (1<<9);

		//PA0 as input for pot
		GPIOA->MODER &= ~(GPIO_MODER_MODER0);

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

		//PA4 as general-purpose output mode for DAC output
		GPIOA->MODER  |= GPIO_MODER_MODER4;

		//Ensure no pull-up/pull-down
		GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);
		GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
		GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
}

void myGPIOB_Init(){
		//Enable GPIOB
		RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

		//Set PB3 to alternate function mode
		//PB3 as SPI clock
		GPIOB->MODER |= GPIO_MODER_MODER3_1;

		//PB4 as general-purpose output mode
		//PB4 as SPI latch
		GPIOB->MODER |= GPIO_MODER_MODER4_0;

		//Set PB5 to alternate function mode
```

```
	//PB5 as MOSI
	GPIOB->MODER |= GPIO_MODER_MODER5_1;

	//Ensure no pull-up/pull-down
	GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR3);
	GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
	GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR5);
}

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

	//Enable timer interrupts
	TIM2->CR1 = ((uint16_t)0x008C);

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

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

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

	//Assign TIM2 interrupt priority = 0 in NVIC
	NVIC_SetPriority(TIM2_IRQn, 0);

	//Enable TIM2 interrupts in NVIC
	NVIC_EnableIRQ(TIM2_IRQn);

	//Enable update interrupt generation
	TIM2->DIER |= 0x1;
}

void TIM2_IRQHandler(){
	//Check if update interrupt flag is set
	if ((TIM2->SR & TIM_SR_UIF) != 0){
		trace_printf("\n*** Overflow! ***\n");

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

		//Restart stopped timer
		TIM2->CR1 |= (1<<0);
	}
```

```
}

void myEXTI_Init(){
	//Map EXTI1 line to PA1
	SYSCFG->EXTICR[0] |= 0x80;

	//EXTI1 line interrupts: set rising-edge trigger
	EXTI->RTSR |= (1<<1) ;

	//Unmask interrupts from EXTI1 line
	EXTI->IMR |= (1<<1);

	//Assign EXTI1 interrupt priority = 0 in NVIC
	NVIC_SetPriority(EXTI0_1_IRQn, 0);

	//Enable EXTI1 interrupts in NVIC
	NVIC_EnableIRQ(EXTI0_1_IRQn);
}



void EXTI0_1_IRQHandler(){
	//Check if EXTI1 interrupt pending flag is set
	if ((EXTI->PR & EXTI_PR_PR1) != 0)
	{
		//First timer rising edge; start count
		if(timerTriggered == 0){
			//Mask interrupt
			EXTI->IMR &= ~(1<<1);

			//Restart timer
			TIM2->CR1 |= (1<<0);
			//Reset count
			TIM2->CNT = 0;

			//Unmask interrupt
			EXTI->IMR |= (1<<1);
			//Count rising edge count
			timerTriggered = 1;
		}
		//Second timer rising edge; stop count
		else {
			//Mask interrupt
			EXTI->IMR &= ~(1<<1);

			//Stop timer
```

```
                TIM2->CR1 &= ~(1<<0);

                //Read count; calculate frequency
                unsigned long int count = TIM2->CNT;
                freq = fs/count;

                //Un-mask interrupt
                EXTI->IMR |= (1<<1);
                //Reset rising edge count
                timerTriggered = 0;
            }

            //Reset interrupt flag
            EXTI -> PR |= (1<<1);
        }
}

void ADC_Init(){
        ADC1->CR |= ADC_CR_ADEN; //Enable ADC
        ADC1->CR &= ~ADC_CR_ADSTART; //Disable start conversions
        ADC1->CFGR1 |= ADC_CFGR1_CONT; //Continuous conversion mode
        ADC1->CHSELR = ADC_CHSELR_CHSEL0; //Select channel 0 (disable others)
        ADC1->CFGR2 = ADC_CFGR2_CKMODE; //ADC Asynchronous clock mode
        ADC1->SMPR = ADC_SMPR_SMP_0; //Sampling time selection
        ADC1->CFGR1 &= ~ADC_CFGR1_AWDEN; //Disable watchdog
        ADC1->CFGR1 &= ~ADC_CFGR1_SCANDIR; //Scan sequence direction (upward scan)
        ADC1->CFGR1 &= ~ADC_CFGR1_ALIGN; //Data alignment (right)
        ADC1->CFGR1 &= ~ADC_CFGR1_EXTEN; //Disable external trigger conversion mode
        ADC1->CFGR1 &= ~ADC_CFGR1_OVRMOD; //Overrun management mode
        ADC1->CFGR1 &= ~ADC_CFGR1_WAIT; //Disable wait conversion mode
        ADC1->CFGR1 &= ~ADC_CFGR1_DISCEN; //Disable discontinuous mode
        ADC1->IER |= ADC_IER_ADRDYIE; //ADC Ready interrupt (enable)
        ADC1->IER |= ADC_IER_EOCIE; //End of conversion interrupt (enable)
        ADC1->IER |= ADC_IER_EOSMPIE; //End of sampling interrupt (enable)
        ADC->CCR |=  ADC_CCR_VREFEN; //Vref internal (enable)
        ADC1->CFGR1 &= ~ADC_CFGR1_RES; //Set resolution (12)
}

void DAC_Init()
{
        //Enable DAC clock
        RCC->APB1ENR |= 0x20000000;
        //Enable the DAC Channel 1
        DAC->CR |= 0x1;
        //Set the trigger to software
        DAC->CR |= DAC_CR_TSEL1;
```

```
}

void SPI_Init(){
	//Enable clock for SPI1
	RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
	//Configure SPI1
	SPI_Handle.Instance = SPI1;
	SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
	SPI_Handle.Init.Mode = SPI_MODE_MASTER;
	SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
	SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
	SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
	SPI_Handle.Init.NSS = SPI_NSS_SOFT;
	SPI_Handle.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
	SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;
	SPI_Handle.Init.CRCPolynomial = 7;

	//Initialize SPI instance with desired configuration
	HAL_SPI_Init(&SPI_Handle);
	//Enable SPI
	__HAL_SPI_ENABLE(&SPI_Handle);
}

void LCD_SendBits(uint8_t Bits_8){
	//Set latch bit for PB4 low
	GPIOB->BSRR |= (uint32_t)0x100000;

	//Transmit 8 bits of data
	HAL_SPI_Transmit(&SPI_Handle, (uint8_t*)&Bits_8, sizeof(Bits_8),
HAL_MAX_DELAY);

	//Wait for SPI buffer to be empty
	while((SPI1->SR & (uint16_t)0x2) == 0 ){};

	//Set latch bit for PB4 high
	GPIOB->BSRR |= (uint32_t)0x10;
}

void LCD_Send_Message(uint16_t message){
	//Isolate RS bit
	uint8_t RS = (message & 0x0200)>>9;
	//Isolate High bits [7:4]
	uint8_t High = (message & 0x00F0)>> 4;
	//Isolate Low bits [3:0]
	uint8_t Low = (message & 0x000F);
```

```
	//Create first high message (EN = 0)
	uint8_t High0 = ((RS << 6 ) | High);
	//Create second high message (EN = 1)
	uint8_t High1 = ((1 << 7) | (RS << 6) | High);

	//Create first low message (EN = 0)
	uint8_t Low0 = ((RS << 6 ) | Low);
	//Create second low message (EN = 1)
	uint8_t Low1  = ((1 << 7) | (RS << 6) | Low);

	//Send 3 high messages
	LCD_SendBits(High0);
	LCD_SendBits(High1);
	LCD_SendBits(High0);

	//Send 3 low messages
	LCD_SendBits(Low0);
	LCD_SendBits(Low1);
	LCD_SendBits(Low0);
}

void LCD_Init(){
	// Set to 4 bit mode; No low sent
	LCD_SendBits(0x02);
   LCD_SendBits(0x82);
	LCD_SendBits(0x02);

	//Set LCD to use 2 lines of characters
	LCD_Send_Message(0x0028);

	//Turn of cursor; turn on LCD
	LCD_Send_Message(0x000C);
	//Display shift off; auto-increment on
	LCD_Send_Message(0x0006);
	//Clear display
	LCD_Send_Message(0x0001);
}

void LCD_Write(uint16_t resistance, uint16_t frequency){
	// delay to allow for smooth update
	for(int i = 0; i < 500000; i++){}
	//Limit data to 4 digits for LCD space provided
	resistance = resistance % 10000;
	frequency = frequency % 10000;

	//Break data down into individual digits
```

```
        uint16_t res4 = resistance % 10;
        uint16_t res3 = ((resistance - res4) % 100)/10;
        uint16_t res2 = ((resistance - res3 - res4) % 1000)/100;
        uint16_t res1 = resistance/1000;
        uint16_t freq4 = frequency % 10;
        uint16_t freq3 = ((frequency - freq4) % 100)/10;
        uint16_t freq2 = ((frequency - freq3 - freq4) % 1000)/100;
        uint16_t freq1 = frequency/1000;

        // delay to allow for smooth update
        for(int i = 0; i < 500000; i++){}

        //Select first line of LCD
        LCD_Send_Message(0x0080);
        //Write 'F:' to LCD
        LCD_Send_Message(0x0200 | 0x0046);
        LCD_Send_Message(0x0200 | 0x003A);
        //Write frequency digits to LCD
        LCD_Send_Message(0x0200 | 0x0030 | freq1);
        LCD_Send_Message(0x0200 | 0x0030 | freq2);
        LCD_Send_Message(0x0200 | 0x0030 | freq3);
        LCD_Send_Message(0x0200 | 0x0030 | freq4);
        //Write 'Hz' to LCD
        LCD_Send_Message(0x0200 | 0x0048);
        LCD_Send_Message(0x0200 | 0x007A);

        //Select second line of LCD
        LCD_Send_Message(0x00C0);
        //Write 'R:' to LCD
        LCD_Send_Message(0x0200 | 0x0052);
        LCD_Send_Message(0x0200 | 0x003A);
        //Write resistance digits to LCD
        LCD_Send_Message(0x0200 | 0x0030 | res1);
        LCD_Send_Message(0x0200 | 0x0030 | res2);
        LCD_Send_Message(0x0200 | 0x0030 | res3);
        LCD_Send_Message(0x0200 | 0x0030 | res4);
        //Write ohm symbol to LCD
        LCD_Send_Message(0x0200 | 0x00F4);
}



#pragma GCC diagnostic pop

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