PWM Signal Generation and Monitoring System

ECE 355: Microprocessor-Based Systems
Ryan Russell (V00873387)

Project Final Report

Report Submitted On: November 29, 2020

Report Submitted To: Khaled Kelany

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

1. Problem Description

This project aims to generate, monitor, and control a pulse-width-modulated (PWM) signal using a STM32F0 Discovery microcontroller. The signal is generated by an NE555 external timer, and a 4N35 optocoupler is used to control the frequency of the PWM signal. A potentiometer is used to control the signal voltage, which is read by the microcontroller's internal analog-to-digital (ADC) converter. From the ADC voltage, the microcontroller calculates the potentiometer resistance. This, in turn, triggers a digital-to-analog (DAC) conversion which generates an output voltage. Figure 1 displays the overall structure of the system components.

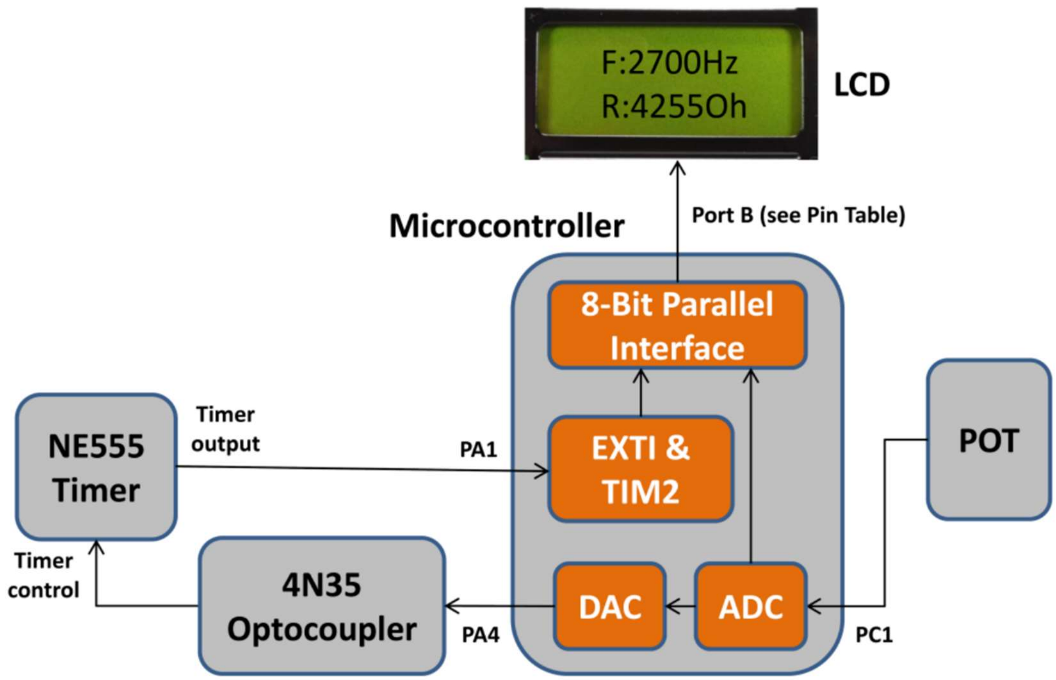

Figure 1: Overall System Diagram

The output signal is sent to the optocoupler following the generation of an output voltage. As mentioned previously, this optocoupler is used to control the PWM signal frequency. Once the output signal is sent back to microcontroller, the signal frequency is measured and passed, along with the resistance, to an external LCD via Port B. This process is continuous until the program is suspended or terminated, so the frequency and resistance are consistently updated on the LCD.

2. System Design

The design of the embedded system was separated into four subsections: frequency measurement, ADC, DAC, and LCD. Following the completion of Lab 1 Part 2, the system was already capable of reading square wave input from an external source (originally configured to be input from a function generator). Therefore, only some minor tweaks were required to effectively integrate the frequency measurement with the rest of the system.

With the frequency measurement out of the way, the next step was to transition from the usage of the function generator and set up the ADC and DAC required to calculate the potentiometer resistance. This also involved configuring the microcontroller's pins, which will be described in further detail in the following subsections. The final step was initialization of the LCD and the development of several functions capable of sending the required data to the LCD screen continuously.

Several resources were consulted when designing the embedded system for this project, including the ECE 355 Laboratory Manual, the Hitachi HD44780 LCD Controller Reference, and the STM32F0 Reference Manual [1, 2, 3].

2.1 ADC Design

The design of the ADC starts with the external potentiometer. As is shown in Figure 1, the microcontroller reads the analog voltage signal from the potentiometer by receiving input at Port C1, and then sends the input to the internal ADC. To set this up in the software, two functions were used: myGPIOC_Init() and myADC_Init(). The first of those functions enables the clock for GPIOC with the RCC->AHBENR register, configures PC1 as analog with GPIOC->MODER, and ensures that no pull-up or pull-down occurs for PC1 using GPIOC->PUPDR. Once myGPIOC_Init() has run and PC1 is initialized, the ADC can be set up using a similar initialization procedure as described in the STM32F0 Reference Manual [3]. The relevant registers to do this are RCC->APB2ENR (to enable the ADC clock), ADC1->CFGR1 (to configure the ADC for measuring continuous analog voltage in overrun mode), ADC1->CHSELR (for mapping the ADC to PC1), and ADC1->CR (which maps the input voltage range to a 12-bit resolution, the default). The code for these functions is displayed in Appendix A, along with the rest of the system's source code.

The most complex part of this initialization is the configuration of the 12-bit resolution, which involves setting a flag on the ADC's control register and waiting for that flag to be reset. Once the 12-bit resolution is set, the input voltage ranges from 0 to 4095 from the microcontroller's perspective, rather than the actual voltage range measured on the potentiometer. Because the resistance is calculated using this voltage range, it will actually represent a notional resistance, which is acceptable for the purposes of this system.

As mentioned previously, the conversion is continuous. This means that each time the ADC value is dereferenced, the ADC will initiate a new conversion. The conversion requests are actually issued from the main loop of the system repeatedly, which will eventually provide the consistent update of the resistance on the LCD. Upon each iteration of the loop, the ADC1->CR register is used to trigger the conversion (issue a conversion request) and then the program waits until the 'end of conversion' flag is set with ADC1->ISR. Finally, the voltage is retrieved from ADC1->DR and the resistance calculation can begin.

2.2 DAC Design

The DAC design is the simplest of the four system subsections. As part of the myGPIOA_Init() function, Port A4 is configured as analog output since it is the default port for the DAC (using GPIOA->MODER). In addition, setting GPIOA->PUPDR ensures no pull-up or pull-down for PA4. The myDAC_Init() function serves two main purposes: to enable the DAC clock using RCC->APB1EN and enable the default DAC channel with the DAC->CR register. In terms of DAC functionality, the system's main loop updates the DAC data register (DAC->DHR12R1) with the voltage retrieved from the ADC. This register also uses the same 12-bit resolution as the ADC data register, so no conversion is required.

2.3 LCD Design

The output resistance and frequency are written to an external LCD that is connected to the microcontroller via Port B and an 8-bit parallel interface, as can be seen in Figure 1. The LCD is initialized to display two rows of eight characters. To connect with the LCD's input pins, the program interacts with Port B pins 4 through 15. PB5 is the RS pin which can set whether you are sending a command (0) or data (1). PB6 is the R/W pin which indicates reading (1) or writing (0). PB8 through PB15 are for passing the actual data (or command). Finally, PB4 and PB7 are the ENB and DONE pins, which are used for LCD handshaking (and will be explained later). All of this pin information, including the pins for the ADC and DAC, are shown in Figure 2.

STM32F0	SIGNAL	DIRECTION
PA0	USER PUSH BUTTON	INPUT
PC8	BLUE LED	OUTPUT
PC9	GREEN LED	OUTPUT
PA1	555 TIMER	INPUT
PA2	FUNCTION GENERATOR (for Part 2 only)	INPUT
PA4	DAC	OUTPUT (Analog)
PC1	ADC	INPUT (Analog)
PB4	ENB (LCD Handshaking: "Enable")	OUTPUT
PB5	RS (0 = COMMAND, 1 = DATA)	OUTPUT
PB6	R/W (0 = WRITE, 1 = READ)	OUTPUT
PB7	DONE (LCD Handshaking: "Done")	INPUT
PB8	D0	OUTPUT
PB9	D1	OUTPUT
PB10	D2	OUTPUT
PB11	D3	OUTPUT
PB12	D4	OUTPUT
PB13	D5	OUTPUT
PB14	D6	OUTPUT
PB15	D7	OUTPUT

Figure 2: System Pin Information

The control flow for the LCD starts with the configuration of Port B in the myGPIOB_Init() function. This function enables the clock for GPIOB with the RCC->AHBENR register, configures PB4 through PB7 and PB8 through PB15 as output and PB7 as input with GPIOC->MODER, and ensures that no pull-up or pull-down occurs using GPIOC->PUPDR. Once the ports have been configured, myLCD_Init() is run.

Before the myLCD_Init() function can be explained, we need to look at the function it relies on: sendToLCD(ODR_data). This function takes, as input, a hexadecimal value ODR_data that indicates what should be written to the LCD from the microcontroller and considers the pin configuration displayed in Figure 2. Bits 0 to 15 correspond to the Port B pins defined in Figure 2, so the actual data that will be sent to the LCD is contained in the two high bytes of the four-byte hexadecimal value. Firstly, the function writes the ODR_data to Port B's output data register, GPIOB->ODR. Now, this is where the LCD handshaking comes in. The enable bit (PB4) is set using GPIOB->ODR and the program waits, continuously checking GPIOB->IDR, until the done bit is asserted by the LCD. Then, the enable bit is de-asserted and the program waits, continuously checking GPIOB->IDR, until done is de-asserted by the LCD. After that process, the desired command or data has successfully been sent to and displayed on the LCD. An example of the LCD appearance is shown in Figure 3.

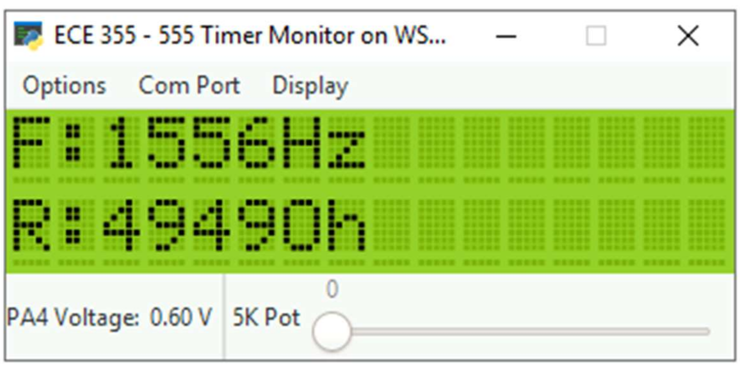

Figure 3: LCD Appearance

Now we return to the myLCD_Init() function, which has two purposes. The first of which is to initialize the LCD with the following settings: two lines of eight characters are displayed using an 8-bit interface, the display is on, the cursor is not displayed and not blinking, and the DDRAM address is auto-incremented after each access [4]. The auto-incrementation is important because it reduces the number of cursor-setting commands that need to be sent to the LCD. Secondly, the function accesses the leftmost section of the top row on the LCD and writes “F:” to indicate the frequency. Then, the leftmost section on the bottom row is accessed and “R:” is written to indicate the resistance. Each of the commands, along with the “F:” and “R:” data, is sent to the LCD using the sendLCD(ODR_data) function described above. Once this function has completed its execution, the LCD is set up and ready to receive continuous frequency and resistance updates.

After the resistance and frequency are calculated in the program’s main loop, the updateLCD(freq, resistance) function is called. This function converts the frequency and resistance to ASCII and stores them in a string before inserting the ASCII values into a hexadecimal value ODR_data that can be used with the sendLCD(ODR_data) function. Then, the program writes the frequency ASCII data followed by “Hz” and the resistance ASCII data followed by “Oh”. The final result is similar to the example shown in Figure 3.

3. Testing and Results

The testing of the embedded system for this project was done using the Eclipse IDE’s built-in debugger. The debug configuration defined in the lab manual for Lab 1 was used to ensure proper communication with the microcontroller [1]. Basic testing of the system components was performed throughout development and at the completion of both the ADC and DAC code. Debug statements were utilized to confirm the proper functionality of the ADC, DAC, and the corresponding port connections. Of course, these debug statements were also used to view the values retrieved for voltage and frequency throughout development, thus ensuring that no egregious errors made their way to future development stages. Once the LCD communication code was completed, basic visual testing was performed by using the digital potentiometer slider to adjust the resistance and viewing the adjusted values on the LCD screen.

From the basic visual testing of the LCD, it was observed that both the recorded frequency and resistance (on the LCD) increased as the potentiometer resistance increased. The recorded frequency range was roughly from 1010 Hz to 1560 Hz, while the recorded resistance range was roughly from 65 Ohms to 4910 Ohms (which lines up closely with the potentiometer resistance). One interesting observation from testing of the frequency range was the rapid frequency increase at around 1.08 V and 1800 Ohms. Here, the frequency jumped from around 1050 Hz to over 1300 Hz while the resistance continued to increase linearly. The PA4 voltage range visible on the 555 Timer Monitor, visible on the bottom left corner of Figure 3, ranged from 0.06 V to 2.22 V.

Several system and design limitations became evident during the testing process. The first limitation stems from the fact that we rely heavily on the accuracy of the 555 Timer, and we are limited by both the voltage it can produce and the usage of the potentiometer resistance slider. That is, we cannot observe a higher resistance than 5000 Ohms simply because the maximum value on the slider is 5000. Moreover, many of the limitations from Lab 1 Part 2 carry over to this project as we reuse much of the code for frequency measurement. One of these limitations is that we still rely on the SystemCoreClock, so there may exist inherent, unavoidable errors in our frequency measurement.

An additional limitation is that the Interrupt Service Routine (ISR) overhead contributes to error in the frequency measurement, muddying the result. If we were to uncomment some of the trace_printf() statements in the code, the overhead would be increased further. Finally, the way the LCD is configured we can only display four digits, so even if the resistance or frequency reached a five-digit value, this system would not be able to display the result accurately.

4. Discussion

I only had two major assumptions going into this project. My first assumption was that this project would be much more complicated and require significantly more effort than the design specification made it seem, which turned out to be correct as this project involved a significant amount of effort to complete, and there was a relatively high barrier to entry. My second assumption was that developing the code for the ADC and DAC functionality would be the most difficult part of the project. This was partly true, as I found that I had the least knowledge regarding this aspect of the project and it took me quite a while to find the resources required to set up and implement the proper functionality. However, I think the LCD communication process actually involved more work and more lines of code, although it took me less time as I found it more intuitive.

I did not implement any significant design changes from the specification. A minor alteration was the separation of each functional chunk of code into its own function to adhere to software engineering design principles. The only shortcomings of my design seem to stem from the system limitations described in Section 3. Specifically, small issues like the recorded resistance range being roughly 50 Ohms offset from the potentiometer resistance slider value. I only encountered a few errors during development, all of which were fixed with a reset of the remote lab environment. The most prevalent error was a computing environment IO exception that would block me from running and debugging my code.

As for lessons learned from this project, the most important one would be to do research in advance and lay out the materials required for each component of the project. However, this is easier said than done. Many of the available reference manuals and data sheets are hundreds of pages long. When you do not know exactly what you are looking for at the beginning, it is extremely difficult to parse through the vast walls of text and diagrams to find the pieces you need. I think providing slightly more direction towards which sections of the reference manuals are required on the lab website would be beneficial for time management in future iterations of this project. Furthermore, I learned many technical lessons pertaining to microprocessor-based systems as the development progressed, like the proper way to communicate using an 8-bit parallel interface.

Overall, the numerous challenges I faced resulted in the development of a functional embedded system that generates, monitors, and controls a PWM signal, and I learned several valuable lessons in the process.

5. References

[1] B. Sirna, K. Kelany, D. Rakhmatov, *ECE 355: Microprocessor-Based Systems Laboratory Manual*, University of Victoria, 2020.

[2] Hitachi, *HD44780U Dot Matrix Liquid Crystal Display Controller/Driver*, Version 0.0, September 1999.

[3] ST Microelectronics, *Reference Manual, STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs,* Version 5, January 2014.

[4] D. Rakhmatov, *ECE 355: Interface Examples,* University of Victoria, 2020.

Appendix A: Program Code

```
//
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//
// Extended for Lab Project By:
// Ryan Russell
// V00873387
// November 20, 2020

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

#include <stdio.h>
#include "stm32f0xx_gpio.h"
#include "stm32f0xx_rcc.h"
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
#include "assert.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 and TIM3 timers: no prescaling */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
#define myTIM3_PRESCALER ((uint16_t)0xBB7F) // 47999 in decimal (1 KHz)

/* Maximum possible setting for overflow */
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF)

// Function definitions
void myGPIOA_Init(void);
void myGPIOB_Init(void);
void myGPIOC_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
void myLCD_Init(void);
void updateLCD(unsigned int cur_freq, uint16_t cur_res);
void sendToLCD(uint16_t ODR_data);
void myTIM2_Init(void);
void myTIM3_Init(void);
void myEXTI_Init(void);
void TIM2_IRQHandler(void);
void EXTI0_1_IRQHandler(void);
```

```c
uint16_t resistance = 0;
unsigned int period_us = 0;
unsigned int freq = 0;
unsigned int voltage = 0;


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

    //trace_printf("This is Part 2 of Introductory Lab...\n");
    //trace_printf("System clock: %u Hz\n", SystemCoreClock);

    myGPIOA_Init();     /* Initialize I/O port PA */
    myGPIOB_Init();     /* Initialize I/O port PB */
    myGPIOC_Init();     /* Initialize I/O port PC */
    myTIM2_Init();      /* Initialize timer TIM2 */
    myTIM3_Init();      /* Initialize timer TIM3 */
    myEXTI_Init();      /* Initialize EXTI */
    myADC_Init();       /* Initialize ADC */
    myDAC_Init();       /* Initialize DAC */
    myLCD_Init();       /* Initialize LCD */

    while (1) {

        // Trigger ADC conversion and test EOC flag
        ADC1->CR |= 0x00000004;
        while ((ADC1->ISR & 0x00000004) == 0);

        // Calculate voltage and then set voltage from converted ADC1 data
        voltage = (unsigned int)(ADC1->DR);
        DAC->DHR12R1 = voltage;

        //trace_printf("Voltage: %d\n", (int)voltage);

        // Calculate resistance
        float res = (5000 * voltage) / 4095;
        resistance = ((uint16_t)res);

        //trace_printf("Frequency: %d\n", (int)freq);
        //trace_printf("Resistance: %d\n\n", (int)resistance);
        updateLCD(freq, resistance);
    }
    return 0;
}


void myGPIOA_Init() {

    /* Enable clock for GPIOA peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    /* Configure PA1 as input (555 Timer) */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= ~(GPIO_MODER_MODER1);

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

    /* Configure PA4 as analog (DAC Analog) */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER |= 0x00000300;

    /* Ensure no pull-up/pull-down for PA4 */
    // Relevant register: GPIOA->PUPDR
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
}
```

```c
void myGPIOB_Init() {

    /* Enable clock for GPIOB peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    /* Configure PB4->PB6 and PB8->PB15 as output */
    // Relevant register: GPIOA->MODER
    GPIOB->MODER = 0x55551500;

    /* Ensure no pull-up/pull-down for PB */
    // Relevant register: GPIOA->PUPDR
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR2);
}


void myGPIOC_Init() {

    /* Enable clock for GPIOC peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

    /* Configure PC1 as analog */
    // Relevant register: GPIOC->MODER
    GPIOC->MODER |= 0x0000000C;

    /* Ensure no pull-up/pull-down for PC1 */
    // Relevant register: GPIOC->PUPDR
    GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
}


void myADC_Init() {

    // Enable ADC clock
    RCC->APB2ENR |= 0x00000200;

    // Configure ADC to measure analog voltage signal continuously
    ADC1->CFGR1 &= ~(0x00000020);
    ADC1->CFGR1 |= 0x00002000;
    ADC1->SMPR  |= 0x00000007;

    // Maps ADC to PC1
    ADC1->CHSELR |= 0x00000800;

    // Set 12 bit resolution
    ADC1->CR |= 0x80000000;
    while ((ADC1->CR & 0x80000000) != 0);
    ADC1->CR |= 0x00000001;
    while ((ADC1->ISR & 0x00000001) == 0);
}


void myDAC_Init() {

    // Enable clock for DAC Peripheral and set GPIOC
    RCC->APB1ENR |= 0x20000000;
    GPIOC->MODER |= 0x00000300;

    DAC->CR &= 0xFFFFFCFF;

    // Set DAC Channel to known state, then set enable bit
    DAC->CR |= 0x00000000;
    DAC->CR |= 0x00000001;
}
```

```c
void myLCD_Init() {

    // Initialization command 1: DL = 1, N = 1, F = 0
    sendToLCD(0x3800);

    // Initialization command 2: D = 1, C = 0, B = 0
    sendToLCD(0x0C00);

    // Initialization command 3: I/D = 1, S = 0
    sendToLCD(0x0600);

    // Initialization command 4: clear display
    sendToLCD(0x0100);

    // Access top row of LCD (frequency row)
    sendToLCD(0x8000);

    // Set first character "F"
    sendToLCD(0x4620);

    // Set second character ":"
    sendToLCD(0x3A20);

    // Access bottom row of LCD (resistance row)
    sendToLCD(0xC000);

    // Set first character "R"
    sendToLCD(0x5220);

    // Set second character ":"
    sendToLCD(0x3A20);
}


void updateLCD(unsigned int cur_freq, uint16_t cur_res) {

    char freqString[4] = {'0', '0', '0', '0'};
    char resString[4]  = {'0', '0', '0', '0'};

    // Convert to ASCII using the offset of 0x30 ('0')
    freqString[0] = cur_freq / 1000 + '0';
    freqString[1] = ((cur_freq % 1000) / 100) + '0';
    freqString[2] = ((cur_freq % 100) / 10) + '0';
    freqString[3] = cur_freq % 10 + '0';

    resString[0] = cur_res / 1000 + '0';
    resString[1] = ((cur_res % 1000) / 100) + '0';
    resString[2] = ((cur_res % 100) / 10) + '0';
    resString[3] = cur_res % 10 + '0';

    // Access top row of LCD (frequency row)
    sendToLCD(0x8200);

    // Set all four digits of the current frequency value
    int i = 0;
    for (i = 0; i < 4; i++) {
        uint16_t ODR_data = ((freqString[i] << 8) + 0x20);
        sendToLCD(ODR_data);
    }

    // Set second last character "H"
    sendToLCD(0x4820);

    // Set last character "z"
    sendToLCD(0x7A20);

```

```c

    // Access bottom row of LCD (resistance row)
    sendToLCD(0xC200);

    // Set all four digits of the current frequency value
    int j = 0;
    for (j = 0; j < 4; j++) {
        uint16_t ODR_data = ((resString[j] << 8) + 0x20);
        sendToLCD(ODR_data);
    }

    // Set second last character "O"
    sendToLCD(0x4F20);

    // Set last character "h"
    sendToLCD(0x6820);
}


void sendToLCD(uint16_t ODR_data) {

    // Send data, RS bit, and R/W bit
    GPIOB->ODR = ODR_data;

    // Assert Enable
    GPIOB->ODR |= 0x0010;

    // Wait for Done to be asserted
    while ((GPIOB->IDR & (0x0080)) == 0);

    // De-assert Enable
    GPIOB->ODR &= ODR_data;

    // Wait for Done to be de-asserted
    while ((GPIOB->IDR & (0x0080)) != 0);
}


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;

	/* Begin timer pulsing */
	TIM2->CR1 |= TIM_CR1_CEN;
}


void myTIM3_Init(void) {

	/* Enable clock for TIM3 peripheral */
	RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;

	/* Configure TIM3: buffer autoreload, count up, stop on overflow,
	 * enable update events, interrupt on overflow only */
	TIM3->CR1 = ((uint16_t)0x008C);

	/* Set clock prescaler value: 48MHz/(47999+1) = 1 KHz */
	TIM3->PSC = myTIM3_PRESCALER;

	/* Default autoreloaded delay: 100 ms */
	TIM3->ARR = 100;

	/* Update timer registers */
	TIM3->EGR |= 0x0001;
}


void myEXTI_Init() {

	/* Map EXTI1 line to PA1 */
	// Relevant register: SYSCFG->EXTICR[0]
	SYSCFG->EXTICR[0] &= ~(SYSCFG_EXTICR1_EXTI1);

	/* EXTI1 line interrupts: set rising-edge trigger */
	// Relevant register: EXTI->RTSR
	EXTI->RTSR |= EXTI_RTSR_TR1;

	/* Unmask interrupts from EXTI1 line */
	// Relevant register: EXTI->IMR
	EXTI->IMR |= EXTI_IMR_MR1;

	/* Assign EXTI1 interrupt priority = 0 in NVIC */
	// Relevant register: NVIC->IP[2], or use NVIC_SetPriority
	NVIC_SetPriority(EXTI0_1_IRQn, 0);

	/* Enable EXTI1 interrupts in NVIC */
	// Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
	NVIC_EnableIRQ(EXTI0_1_IRQn);
}


/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.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);

```

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


	/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
	void EXTI0_1_IRQHandler() {

		// Declare/initialize your local variables here...
		uint32_t count;
		uint32_t timerTriggered;

		/* Check if EXTI1 interrupt pending flag is indeed set */
		if ((EXTI->PR & EXTI_PR_PR1) != 0) {

		timerTriggered = (TIM2->CR1 & TIM_CR1_CEN);

			if (timerTriggered) {

				// Stop the timer and retrieve the value from CNT
				EXTI->IMR &= ~(EXTI_IMR_MR2);
				TIM2->CR1 &= ~(TIM_CR1_CEN);
				count = TIM2->CNT;

				freq = (double)SystemCoreClock / (double)count;
				period_us = 1000000 / freq;

				//trace_printf("Period: %d Seconds \n", (unsigned int)period_s);
				//trace_printf("Period: %d Microseconds \n", (unsigned int)period_us);
				//trace_printf("Frequency: %d Hz\n", (unsigned int)freq);

				EXTI->IMR |= EXTI_IMR_MR2;
			} else {
				TIM2->CNT= ((uint32_t)0x0000);
				TIM2->CR1 |= TIM_CR1_CEN;
			}
		EXTI->PR |= EXTI_PR_PR1;
		}
	}


	#pragma GCC diagnostic pop

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