ECE 355

Microprocessor – Based Systems

University of Victoria

Final Project Report

November 28th, 2020

Treavor Gagne

V00890643

Software Engineering

treavorgagne@uvic.ca

Table of Contents

Table of Figures

1.0 Problem Definition

The objective of this project was to observe and manipulate a pulse width modulated (PWM) signal using the STM32F051R8T6 microcontroller with some external circuitry on a STM32F0 Discovery board. The PWM signal was produced by an external 555 timer (NE555 Timer) and the frequency of the signal is manipulated by the external optocoupler (4N35 Optocoupler) attached to the microcontroller. Before the frequency can be manipulated, the voltage must be measured across the potentiometer attached to the microcontroller in order to calculate the resistance. To be specific, a built-in analogue-to-digital converter (ADC) is used to read an analogue voltage value from the potentiometer continuously. The potentiometer is simulated using an emulator designed for the ECE 355 project. As such, the value from the potentiometer is relayed to the octopolar from the microcontroller to form the signal which is measured by the microcontroller. Specifically, a digital-to-analog converter (DAC) is used to relay the analogue voltage value to the external circuitry containing the 555 timers to manipulate the frequency. In addition, the frequency of the PWM signal is measured using the EXIT and TIM2 on the microcontroller modified from a previous lab which monitors falling and rising edge of a signal [1]. Overall, the goal of the system is to measure the voltage of the system while displaying the frequency of the system and the resistance calculated to form the frequency and voltage to an LCD board emulated for the ECE 355 project. Particularly, the LCD is updated via 8-bit parallel interface using different controller signals and 8 data signals to send instructions or data to the LCD. The system described can be seen below in Figure 1.

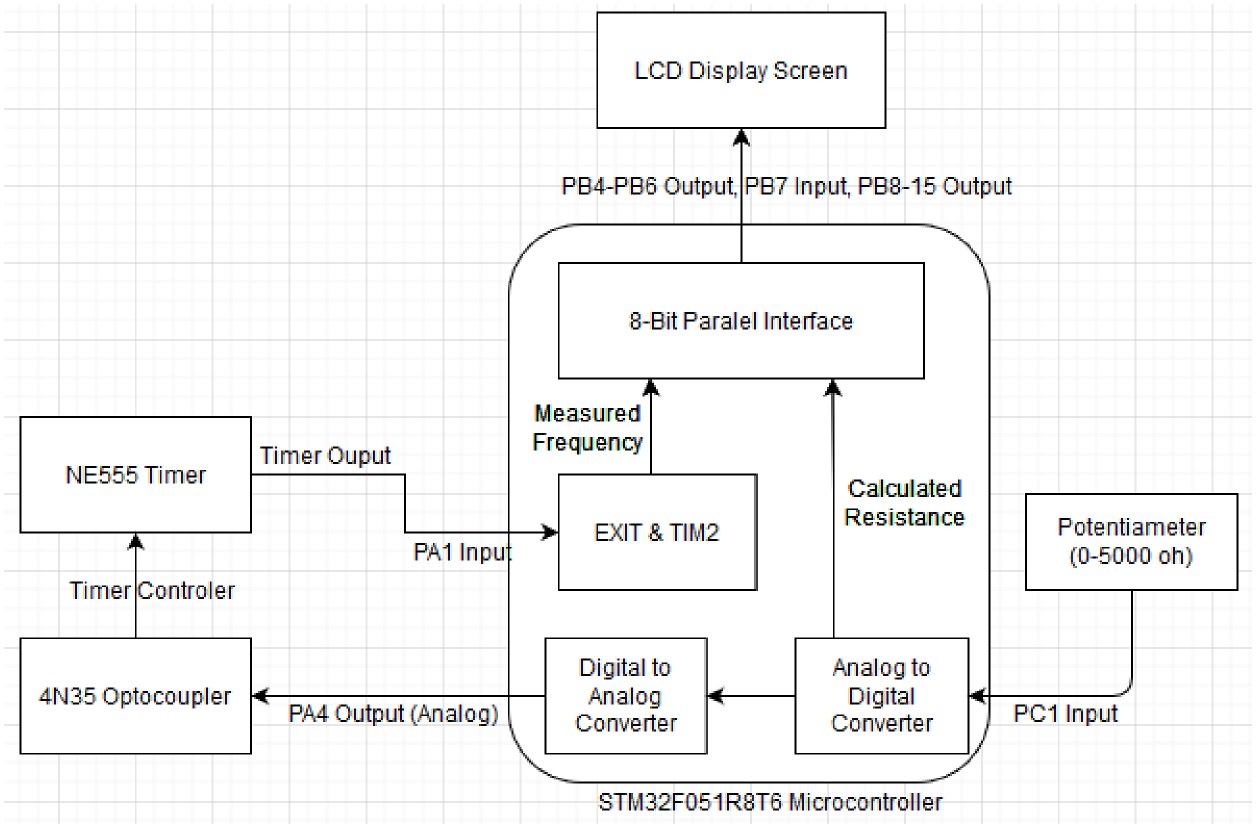

Figure 1: Block Diagram Overview of Microcontroller System

2.0 Design Solution

The design process for the project followed no particular method, but each component of the project was developed in what at the time seemed most logical. For example, the LCD was developed last since completing it first would be would difficult to test without having frequency or resistance values to send to it. As such, the order to in which the solution to the project was developed in this order:

1. Port Pins Initializations
2. ADC Initialization Function and Code
3. Calculating Resistance
4. DAC Initialization Function and Code

5. Modifying EXIT1 and TIM2 for Frequency Measurements
6. LCD Initialization Function and Code

The project solution was developed in the order, as each of these elements depend on the previous element. Together, the all these elements function together to successfully manipulate the PWM signal of the system.

2.1 Port Pins initializations

The first step taken in developing a design solution was to initialize all the port pins for the various components connected to the microcontroller. The pins that need to be configured can be seen in Figure 1, and are detailed in the lab manual [1]. The pins configuration is spread about ports A, B, and C. Each of these ports and their respective pins are configured with a three different initialization functions called in the beginning of the main function. These three functions for the initialization are called myGPIOA_init(), myGPIOB_init(), and myGPIOC_init(), as seen in the code in the Appendix A. Each of these initialization functions follow a similar structure. That is, at the beginning each function the `RCC->AHBENR` is set to set using the macro `RCC_AHBENR_GPIOXEN` from the header file containing all the macros for the microcontroller where is X is A, B, or C [2]. These macros when set to the `RCC->AHBENR` register will enable the clock for the various ports which is needed for input or output on any of the pins. After which, the `GPIOX->MODER` register is used where X is A, B, or C to set the mode (input, output, or both) of the port pin using the macro `GPIO_MODER_MODERX_Y` where the X is the pin in which you want to set for the port and Y is the with 1 for input, 0 for output, or nothing for both. As such, the various port pins are set depending on the specification of the lab manual, as seen in Figure 1 [1]. Lastly, the `GPIOX->PUPDR` register, where X is either A, B, or C, is set using the `GPIO_PUPDR_PUPDRY` where Y is the pin number to ensure that the pin specified does not pull up or pull down, as this could negatively affect the results of the systems. It should be noted that all the ports handle input or output of different parts of the system, such as port B which handles the input and output of LCD. Furthermore, the initialization of port B does not use macro values since it uses 12 pins and thus require 12 macros used together, as such the macros are combined as single hexadecimal value to set the needed registers.

2.2 ADC Initialization Function and Code

As needed for the system, the ADC is used to measure the analog voltage across the potentiometer via port C pin 1 configured to input and convert the analog value into a digital value to calculate resistance value, as per the Figure 2 and the lab manual [1]. To perform ADC an initialization function in the beginning of the main loop is called to enable the conversion, such that during the while loop in the main function the measured value across the potentiometer will continuously be converted. The function for initialization the ADC conversion is myADC_init() and can be reference in Appendix A. Firstly, this clears the control register (`ADC1->CR`) by setting all its bits to zero. This is done in case any bits are set from the previous use of the microcontroller which may not necessarily have been this project. After which, the clock register (`RCC->APB2ENR`) for ADC is enable with the `RCC_APB2ENR_ADCEN` macro found in the header file [2]. Then, ADC channel 11 is enabled for port c, as it is connected to port C pin 1 which is connected to the potentiometer, by setting the `ADC1->CHSELR` register to the macro value `ADC_CHSELR_CHSEL11` which enables channel 11. The system must be set to continuous mode by setting `ADC1->CFGR1` with the `ADC_CFGR1_CONT` macro as the ADC converter will run continuously, as opposed to synchronously according to the lab manual [1]. At this point, all that is need to be done is enable the ADC control register (`ADC1->CR`) with the `ADC_CR_ADEN` macro. Then initializations waits for the ready flag to be set by the microcontroller for the ADC interrupt register with a while loop (`while((ADC1->ISR & ADC_ISR_ADRDY) == 0);`), and then starts the ADC by setting the start bit on the ADC control register (`ADC1->CR`) with the `ADC_CR_ADSTART` macro.

After initializing the required ADC registers on the microcontroller, the system can continuously convert the analog voltages across the port C pin 1 from the potentiometer. The system performs this conversion in the main function inside a while to continuously pole the ADC. This is done firstly by waiting until end of conversion flag is has been set to 1 by the hardware with `while((ADC1->ISR & ADC_ISR_EOC) == 0);` and storing the converted value analog voltage value into an integer variable from the ADC data register which stores the value after the conversion with `acd_value = ADC1->DR;`. As such, the `acd_value` can now be sent used be sent to the DAC or be used to calculate the current resistance of the potentiometer.

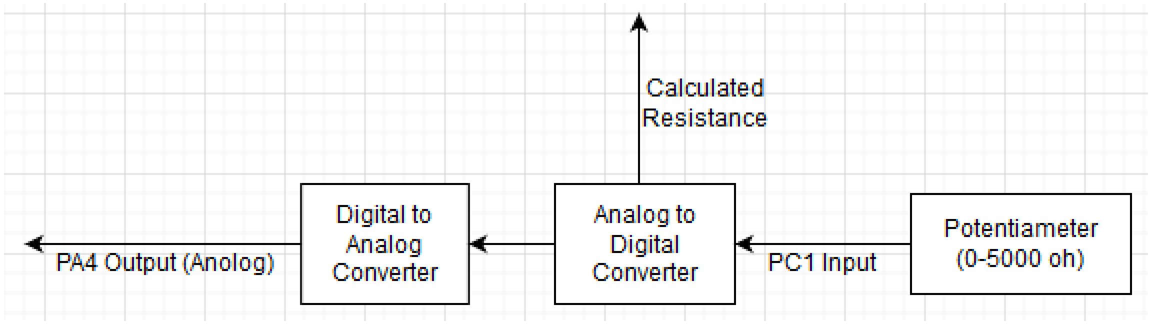
Calculated Resistance
PA4 Output (Anolog)
Digital to Analog Converter
Analog to Digital Converter
PC1 Input
Potentiameter (0-5000 oh)

Figure 2: ADC and DAC Block Diagram

2.3 Calculating Resistance

Once the ADC is initialized and can successfully convert the analog voltage across the port C pin 1 from the potentiometer the resistance calculation is trivial given some background knowledge. That is, given that ADC is a 12-bit value ranging from 0 to 4095 needed to normalize the `acd_value`, as per the reference manual [3]. Furthermore, the board is powered by 3.3V, and the maximum resistance is 5000Oh (max potentiometer value), as per the lab manual [1]. Based on this information, the resistance of the system can be calculated using the `acd_value`. Therefore, the resistance is calculated by `res = (float) (((acd_value * 3.3)/4095)/ 0.00066);` where 0.0066 Amps is the maximum current with the voltage of the system (3.3V) divided by the maximum possible resistance value of 5000Oh. This formulation provides the proper units of resistance of being volts over amps. Note, the calculated resistance value is casted float as the result is not of this type and the global variable `res` is declared as a volatile unsigned float for the most precise value possible.

2.4 DAC Initialization Function and Code

Logically, the DAC is the next component of the system to develop given that the ADC is complete. The DAC uses port A pin 4 which is configured for output. Therefore, all that is need to do is initialize the DAC and relay the `acd_value` from the ADC to the DAC, as in seen in Figure 2. To no surprise the DAC initialization function `myDAC_Init()` performed in the beginning of the main function is very simple, as working with digital signals is much easier than analog signals. Therefore, all that is required to initialize the DAC is to clear the DAC control register (`DAC->CR`) by resetting all its bits to 0, enabling the DAC clock register (`RCC->APB1ENR`) with the `RCC_APB1ENR_DACEN` macro, and finally by setting the enable bit on the DAC control register (`DAC->CR`) with the `DAC_CR_EN1` macro. With the DAC initialized all do to is set the DAC channel 1 12-bit right-aligned data holding register (`DAC->DHR12R1`) to a casted `(unsigned int) acd_value` to perform the DAC conversions, as per the microcontroller reference manual [3]. This DAC not only performs the conversion, but also relays the value to the external circuitry need to manipulate the frequency of the PWM signal.

2.5 Modifying EXIT1 and TIM2 for Frequency Measurements

As the external circuitry is already complete, the logically component to complete the EXTI1 and TIM2 interrupts. This component can be seen in Figure 3 and is modified from a lab completed previously to monitor the rising and falling edge of a signal from a function generator to calculate the frequency of the signal. While this previous implementation used PA2 with EXTI2 the current implementation for the system uses PA1 and EXTI1, as described in the lab manual [1]. As such, the details of the initializations and interrupt functions are very similar to the previous implementation and will be listed in a condensed form with a general explanation along with it. It should be noted if more detail of the registers and macros is required the code is commented extensively in Appendix A.

The initialization functions of the both the EXTI and TIM2 are performed at the beginning of the main function. These functions are called `myEXTI_Init()` and `myTIM2_Init()` respectively. Firstly, it should be noted that theses interrupts function differently from one another. That is, the EXTI interrupt is triggered by input from port A pin 1 for rising and falling edges of functions by design while the TIM2 interrupt is timer based on pre-set value decided during initialization. Therefore, the `myEXTI_Init()` initialization function is as followed:

- Map EXTI interrupt to the port A pin 1
- Set rising-edge trigger to EXTI line interrupt
- Unmask interrupts from EXTI1 line
- Assign interrupt priority to 0 (highest) using NVIC_SetPriority function
- Enable EXTI1 interrupt using NVIC_EnableIRQ function

It should be noted that the changes from the previous implementation to the current is only in the macros used, as they now reflect EXTI1 instead of previous implementation. Thus, the `myTIM2_Init()` initialization is as followed:

- Enable clock for TIM2
- Configure TIM2 for buffer auto-reload, count up, stop on overflow, enable update events, and set the interrupt to trigger when the counter overs only
- Set clock pre scalar value (0xFFFFFFFF) (roughly 82 seconds for this implementation)
- Set auto-reload delay (set to 0 seconds)
- Update timer registers
- Assign interrupt priority to 0 (highest) using NVIC_SetPriority function
- Enable EXTI1 interrupt using NVIC_EnableIRQ function
- Enable update interrupt generation
- Start counting timer pulses

This implementation did not change from the previous to current implementation and therefore acts as an overflow for the EXTI interrupt if it is not triggered from the elapsed pre-scalar value which is roughly 82 seconds. Although, the timer for the TIM2 interrupt only starts counting once the rising edge in the EXTI interrupt handler function is triggered. That is, when the EXTI1 interrupt is triggered by the falling and rising edge of the frequency of the PWM pulse the `EXTI0_1_IRQHandler()` is executed by the microcontroller. This interrupt triggers TIM2 counter register to (`TIM2->CNT)` start counting on the rising edge of the function and stops counting when the interrupt is triggered on the falling edge. Therefore, the frequency can be calculated by taking the defined value of the SystemCoreClock and dividing it by the TIM2 counter. This calculates the frequency which than gets stored into a volatile unsigned float global variable. It should be noted that this interrupt handler function verifies that the `(EXTI->PR)` that the bit in in the EXIT pending register has been set to 0 using this macro `EXTI_PR_PR1.` This ensures that the register has been indeed triggered, and as such once the interrupt is complete the bit must be reset manually back to 1. As for, the TIM2_IRQHandler() interrupt function it is very simple. It checks to make sure that the TIM2 status flag has been set 0, much like the pending register for EXIT1, and then clears the status flag to reset it in addition to restarting the overflowed (stopped) TIM2 clock by setting the control register bit (`TIM2->CR1`) with the `TIM_CR1_CEN` macro. Ultimately, the TIM2 interrupt is only triggered if the period of the frequency is 82 seconds or longer since that is the roughly the time in which for the TIM2 timer to overflow given the pre-scalar value used in the initialization. Overall, these two-interrupt work in tandem to measure the frequency of the system, in order to relay it to the LCD to be displayed.

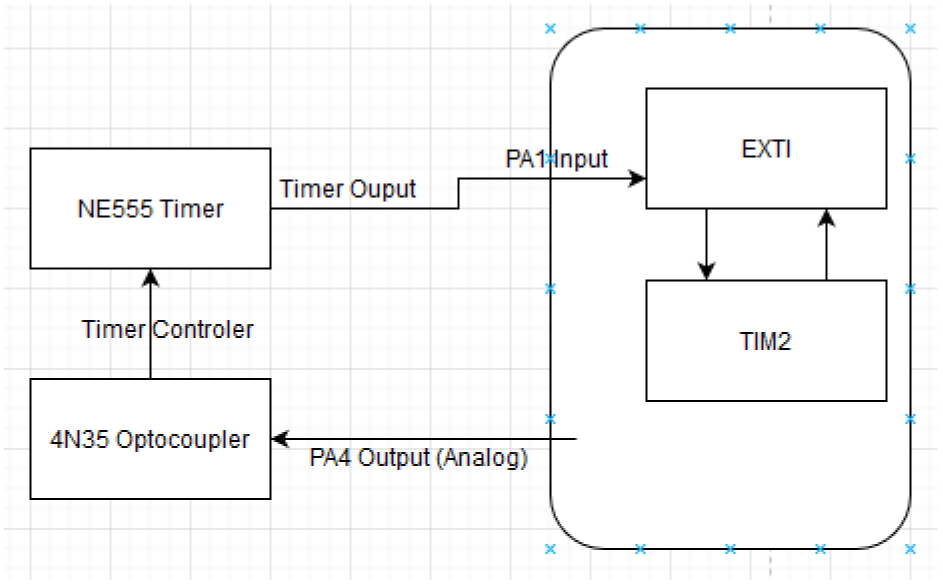

Figure 3: External Circuitry, EXIT1, and TIM2 Block Diagram

2.6 LCD Initialization Function and Code

The final element to develop is the LCD display to calculated frequency and resistance of the system, as all the other components are complete. For the development it should be noted that the LCD display that it requires 12 pins from port B, as according to the lab manual [1]. Furthermore, 11 of these pins are output pins and 1 is for input. A total of 8 of the 11 output pins from port B pin 8 to pin 15 are used as data signals to write data or commands to the LCD. While, the other 3 output pins on port B pin 4, 5, and 6 are used as control signals to enable the LCD handshake, RS (command=0 or data=1), and the R/W (read=1 or write=0). Theses 3 control signal must be set every time, such that the LCD handshake bit is used to enable the data transfer process, the RS signal is set to either command or data depending if LCD needs to take a command or display data, and the R/W signal is always set to write since the LCD cannot read [2]. Lastly, the only input pin on port B pin 7 is used to determine when the LCD handshake is complete, such that the system can perform other tasks.

With all port pins configured the LCD display needs to be initialized, but before that the LCD handshake needs to established write commands or data. The handshake `void lcd(uint32_t x)` function take an unsigned 32 bit int as an variable. This variable contains the 8-bit data along with the RS and R/W instruction for write command or data. Therefore, the variable can be broken into 32 bits where bits 8-15 are reserved for data and the bits 5 and 6 are used for RS and R/W with the rest of the bits serving to particular purpose of importance except for pins 4 and 7. As such, the handshake code follows resetting the register with `GPIOB->BRR = 0xFFFF;`, passing the 32 bit variable with the data and write command or data with `GPIOB->BSRR = x` and enabling the handshake bit (bit 4) on the `GPIOB->BSRR` register. After which, the code waits for the input pin on port B pin 7 with `while((GPIOB-`

`>IDR & GPIO_IDR_7) == 0)` for the handshake to be finished, as the macro corresponds to the bit 7 for handshake done [1]. Lastly, the handshake is assert is reset or de-asserted with `GPIOB->BRR = 0x10` and then waits for the de-assertion to finish with `while((GPIOB->IDR & GPIO_IDR_7)! = 0).` Therefore, to update the LCD or initialize the LCD all is that is needed to do is call the handshake function. Thus for initialization, the `myLCD_Init()` function is called at the beginning of the main function which contains 4 write commands calling `lcd()`, as seen in the code in Appendix A. The initialization starts with the `lcd(0x3800)` command written to the LCD to set it to 2 lines (up and down) of 8 characters with DDRAM access to the 8-bit interface, as according to the interface slides [4]. Another write command (`lcd(0x0C00)`) is then used to disable blinking or a cursor on the display for clarity purposes. Then the `lcd(0x0600)` command is used configure the LCD to auto increment its position on the display. Lastly, the `lcd(0x0100)` write command is used to clear the LCD display incase anything is displayed from previous use.

With the LCD handshake initialized all that is needed to do is get the frequency and resistance calculation which are stored globally as volatile unsigned floats and update the LDC. The `update_lcd()` function updates the LCD and performs the calculations to the resistance and frequency variables to convert number in each position of the variables to its character equivalent, since the LCD needs hexadecimal values sent which are converted to 32 bits for the handshake. This is done with the help of an array with contains the hexadecimal for values 0 to 10. As such, the other information is sent to the LCD handshake, such as the command to write to the top of bottom or row of the LCD display along with some data write text to describe frequency or resistance. Lastly, the update LCD function is triggered by the TIM3 interrupt which is configured similar to the TIM2 interrupt except that it triggered many times a second to update the LCD often. The interrupt can be seen integrated in Figure 4.

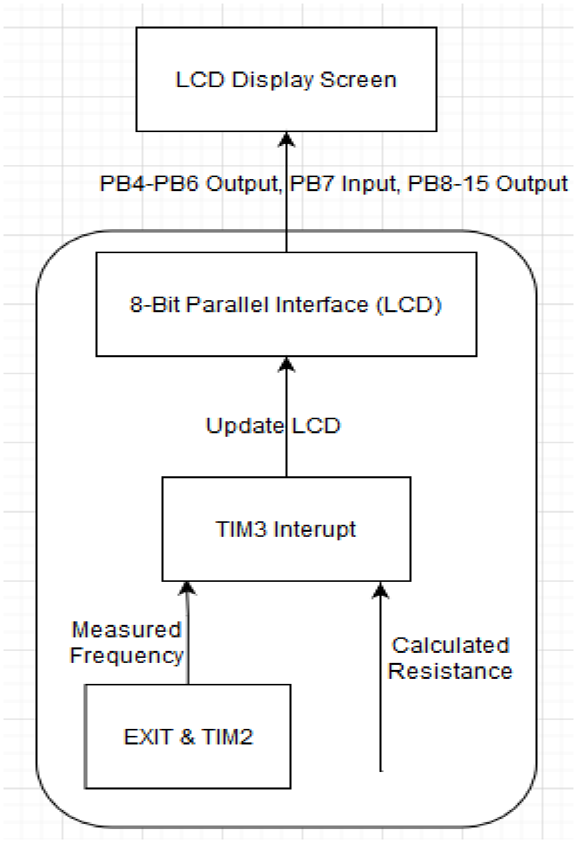

Figure 4: LCD Update Block Diagram

3.0 Testing and Results

The testing purpose, acceptance condition(s), and procedure for important components all detailed below along with the frequency, resistance, and voltage results compared to the expected results [5].

3.1 Testing

In developing the design solution three components were tested in various ways to ensure the results were the desired outcome. These components were the frequency, resistance, and LCD display, as they involve all other components in order to produce their results.

3.1.1 Resistance

The first tested component of the system was the resistance calculation, as this component requires the ADC to be functioning.

Purpose

To determine if the resistance is being calculated correctly. In addition, to verifying that the ADC is functioning as intended.

Acceptance

The resistance was deemed acceptable if the resistance shown on the console was congruent with the resistance set on the potentiometer with a 10-20% variation [5].

Procedure

1. Build the program in debug mode.
2. Ensure trace_printf() is configured in the settings to accept floating point values.
3. Insert a trace_printf() statement after in the while loop in the main function to output the resistance calculation to the console.
4. Run the program for the design solution.
5. Vary the resistance on the potentiometer.
6. Monitor the changes of the resistance and see if they match the resistance set by the potentiometer.

3.1.2 Frequency

After the resistance was calculated and determined that the ADC was functioning properly the frequency was tested. This test also checks to see if the DAC and the modified EXTI and TIM2 are function as intended.

Purpose

To determine if the frequency is being measured correctly. In addition, to verifying that the DAC, EXTI and TIM2, along with the external circuitry are functioning as intended.

Acceptance

The frequency is deemed acceptable if the resistance shown on the console is within range of the values in which Professor Rakhmatov provided in an email with 10-20% variation [5].

Procedure

1. Finish Testing the Resistance Calculation (frequency depends on ADC)
2. Build the program in debug mode.
3. Ensure trace_printf() is configured in the settings to accept floating point values.
4. Insert a trace_printf() statement inside the else statement for the falling edge of the EXTI interrupt to output the measured frequency to the console.
5. Vary the resistance on the potentiometer.
6. Monitor the changes of the frequency measurement, verify that the range of the frequency reaches the min and max detailed in the email from Professor Rakhmatov when changing the potentiometer [5].

3.1.3 LCD Display

The LCD display test is an amalgamation of all the components working together. A successful test of the LCD display determines the completion of the project.

Purpose

To determine if the LCD handshake is functioning properly to display the correct LCD output.

Acceptance

The LCD display is updated regularly and displays the correct frequency measurement and resistance calculation from both trace_printf() with a possible 10-20% variation [5].

Procedure

1. Finish all previous testing (LCD depends on all)
2. Build the program in debug mode.
3. Ensure trace_printf() is configured in the settings to accept floating point values.
4. Insert a trace_printf() statement inside the else statement for the falling edge of the EXTI interrupt to output the measured frequency to the console.
5. Insert a trace_printf() statement after in the while loop in the main function to output the resistance calculation to the console.
6. Run the program for the design solution.
7. Vary the resistance on the potentiometer.
8. Monitor the changes of the LCD display to ensure that the values outputted to the display are congruent with both trace_printf() statements for the frequency measurement and resistance calculation.

3.2 Results

The results of the design solution were within the acceptable range based on the email sent by Professor Rakhmatov [5]. The min and max result of the design system are compared to the expected values in the table below, as seen in Figure 5.

	Min Result	**Max Result**	**Min Expected**	**Max Expected**
Frequency	1015Hz	1567Hz	800Hz	1600Hz
Resistance	59Oh	4904Oh	0Oh	4900Oh
Voltage	0.06V	2.22V	0.06V	2.22V

Figure 5: Result Table

From the results table it can be observed that design solution results are very concise with the expected values for the frequency, resistance, and voltage of the system. The only result which is slightly out of range is the min frequency which doesn't quiet observe lower frequencies. This could be a timing issue with interrupt or possibly a measurement error connected to the specific lab machine. Ultimately, since the result are very concise to the expected results there is no design limitations, but the system does have some limitations. That is, the potentiometer can only range from 0 to 5000Oh, so the maximum current is limited to 0.00666Amps. Furthermore, the voltage will never exceed 3.3V since that is all the volts the systems can handle, and the reason the maximum voltage is 2.2V is because there is a protection resistor present to keep the voltage in check.

4.0 Discussion

Overall, the design solution developed for this project executes exactly as intended to a very high degree, as reflected by the results. As such, this code performs very well and has no major short comings which impact the code in a negative way. Although, the assumption that there are no errors in the code is naïve as there could possibly errors or even flaws in the hardware or C libraires used. Ignoring all possible errors, the design solution executes almost identically to the expected output [5].

Ultimately, there is no major differences between this design solution and the stated design described in the lab manual [1]. While, this design could be programmed differently given proper research and a different code implementation it would be more difficult to do so. As such, a design solution stays closest to the stated design in order make the development as easy as possible, for such a complex project. Therefore, there is no added code to the design in order to prevent any unwanted code functionality. Furthermore, there is no added benefit to adding extra features to the project and doing so would be hard without access to the hardware.

Over the course of this project a number of different skills have been developed. Firstly, priceless knowledge of how to program microcontrollers was gained. In addition, the knowledge of how to search through documentation for the proper macros needed given registers to complete initializations were gained. Some information in which would have been useful in the beginning of the developing the project was to how to trouble shoot and the advice of fully understanding the project lab manual. That is, in the beginning of the project a lot of time was wasted as the specifications did not make entire sense and troubleshooting virtually unaware of anything of the system was very unproductive. Therefore, it is advised to prepare and carefully examine if the design specification and documentation when developing a project of this scope and niche.

References

[1] Jooya, K. Jones, D. Rakhmatov, B. Sirna. ECE 355: Microprocessor-Based Systems Laboratory Manual. University of Victoria, 2018

[2] stm32f0xx.h; CMSIS Cortex-M0 Device Peripheral Access Layer Header File. STMicroelectronics, 2014.

[3] Reference Manual; STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs. STMicroelectronics, 2014.

[4] D. Rakhmatov. Interface Examples. University of Victoria. 2019.

[5] D. Rakhmatov. Expected Result Range Email. University of Victoria. 2019.

Appendix A Project Code

Below is the copy of the finished code used on the microcontroller to produces the results discussed above.

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

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

void myGPIOA_Init(); // init for port TIMER and Optocoupler
void myGPIOB_Init(); // init for port for LCD
void myGPIOC_Init(); // init for port for potentiometer
void myTIM2_Init(); // timer interupt init for rising and falling edge for freq
void myTIM3_Init(); // time interrupt init for lcd
void myEXTI_Init(); // init for exit interrupt for freq
void myADC_Init();  // init for ADC conversion
void myDAC_Init();  // init for DAC conversion
void myLCD_Init();  // init for LCD
void lcd(uint32_t x); // function for writing a single character to LCD

volatile unsigned int edge = 0;
volatile float res = 0;
volatile float freq = 0;

int main(int argc, char* argv[])
{
    myGPIOA_Init();     /* Initialize I/O port PA */
    myGPIOB_Init();     /* Initialize I/O port PB */
    myLCD_Init();       /* Initialize LCD */
    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 */

    int acd_value = 0;

    while (1)
    {
        // wait until the end of the conversion
        while((ADC1->ISR & ADC_ISR_EOC) == 0);
        acd_value = ADC1->DR; // get adc value
        // calculate resistance
        res = (float) (((acd_value * 3.3)/4095)/ 0.00066);
        // send acd_value to dac register
        DAC->DHR12R1 = (unsigned int) acd_value;
    }

    return 0;
}
```

```
void myGPIOA_Init()
{
    /* Enable clock for GPIOA peripheral */
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
    GPIOA->MODER &= ~(GPIO_MODER_MODER1_1); /* Configure PA1 as input */
    GPIOA->MODER &= ~(GPIO_MODER_MODER4_0); /* Configure PA4 as output */
    /* Ensure no pull-up/pull-down for PA1 */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
    /* Ensure no pull-up/pull-down for PA4 */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
}

void myGPIOB_Init()
{
    /* Enable clock for GPIOB peripheral */
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
    GPIOB->MODER &= ~(GPIO_MODER_MODER7_1); // config PB7 as input
    // config port B 4 to 15 bits involved in lcd to input(PB7) & output
    GPIOB->MODER |= 0x55551500;
    GPIOB->PUPDR &= ~(0xFFFF3F00); // ensure no pull-up/pull-down for port B bits
}

void myGPIOC_Init()
{
    /* Enable clock for GPIOC peripheral */
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;
    // configure PC1 for input analog
    GPIOC->MODER |= GPIO_MODER_MODER1_1;
    /* Ensure no pull-up/pull-down for PC1 */
    GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR1);
}

void myTIM2_Init()
{
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;/* Enable clock for TIM2 peripheral */
    /* Configure TIM2: buffer auto-reload, count up, stop on overflow,
     * enable update events, interrupt on overflow only */
    TIM2->CR1 = ((uint16_t)0x008C);
    TIM2->PSC = myTIM2_PRESCALER;/* Set clock prescaler value */
    TIM2->ARR = myTIM2_PERIOD;/* Set auto-reloaded delay */
    TIM2->EGR = TIM_EGR_UG;/* Update timer registers */
    /* Assign TIM2 interrupt priority = 0 in NVIC */
    NVIC_SetPriority(TIM2_IRQn, 0);
    NVIC_EnableIRQ(TIM2_IRQn);/* Enable TIM2 interrupts in NVIC */
    TIM2->DIER |= TIM_DIER_UIE;/* Enable update interrupt generation */
    TIM2->CR1 |= TIM_CR1_CEN;/* Start counting timer pulses */
}

void myTIM3_Init()
{
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN; /* Enable clock for TIM3 peripheral */
    /* Configure TIM3: buffer auto-reload, count up, stop on overflow,
         * enable update events, interrupt on overflow only */
    TIM3->CR1 = (uint16_t)0x008C;
    TIM3->PSC = (uint16_t)0xFF00; /* Set clock value delay for LCD */
    TIM3->ARR = (uint16_t)0x0032; /* Set auto-reloaded delay 50ms */
    TIM3->EGR = TIM_EGR_UG; /* Update timer registers */
    /* Assign TIM3 interrupt priority = 0 in NVIC */
    NVIC_SetPriority(TIM3_IRQn, 0);
    NVIC_EnableIRQ(TIM3_IRQn);/* Enable TIM3 interrupts in NVIC */
    TIM3->DIER |= TIM_DIER_UIE;/* Enable update interrupt generation */
    TIM3->CR1 |= TIM_CR1_CEN;/* Start counting timer pulses */
}

void myEXTI_Init()
{
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PA; /* Map EXTI1 line to PA1 */
    /* EXTI1 line interrupts: set rising-edge trigger */
    EXTI->RTSR |= EXTI_RTSR_TR1;
    EXTI->IMR |= EXTI_IMR_MR1;/* Unmask interrupts from EXTI1 line */
    /* Assign EXTI1 interrupt priority = 0 in NVIC */
    NVIC_SetPriority(EXTI0_1_IRQn, 0);
    NVIC_EnableIRQ(EXTI0_1_IRQn);/* Enable EXTI1 interrupts in NVIC */
}

void myADC_Init()
{
    ADC1->CR = (uint32_t)0x00000000; // reset/clear ADC control register
    RCC->APB2ENR |= RCC_APB2ENR_ADCEN; // enable ADC clock
    ADC1->CHSELR |= ADC_CHSELR_CHSEL11; // Enable channel 11 connected to port c
    ADC1->CFGR1 |= ADC_CFGR1_CONT; // set continuous mode
    ADC1->CR |= ADC_CR_ADEN; // ADC enable control
    while((ADC1->ISR & ADC_ISR_ADRDY) == 0); // wait for ready flag
    ADC1->CR |= ADC_CR_ADSTART; //start ADC
}

void myDAC_Init()
{
    DAC->CR = (uint32_t)0x00000000; // reset/clear DAC control register
    RCC->APB1ENR |= RCC_APB1ENR_DACEN; // enable DAC clock
    DAC->CR = DAC_CR_EN1; // enable DAC
}

void myLCD_Init() // code follows slide 5 in interface.ptx
{
    // set lcd to 2 lines of 8 chars with DDRAM access to 8-bit interface
    lcd(0x3800);
    lcd(0x0C00); // display on with no blinking or cursor
    lcd(0x0600); // auto increment DDRAM address
    lcd(0x0100); // clear display
}

void lcd(uint32_t x)// code follows slide 9 in interface.ptx based on handshake
{
    GPIOB->BRR = 0xFFFF; // reset bits at register
    GPIOB->BSRR = x; //send var to LDC
    GPIOB->BSRR |= 0x10; //enable handshake/assert
    while((GPIOB->IDR & GPIO_IDR_7) == 0); // wait for handshake to finish
    GPIOB->BRR = 0x10; //deassert enable
    while((GPIOB->IDR & GPIO_IDR_7)! = 0); //wait for deasserted
}

void update_lcd(){

    char numToChar[10] = {'0','1','2','3','4','5','6','7','8','9'};
    int temp = (int) res;

    // get individual values of resitance
    int var4 = ((temp/1000)%10);
    int var3 = ((temp/100)%10);
    int var2 = ((temp/10)%10);
    int var1 = (temp%10);

    lcd(0xC000);    // write to bottom bar
    lcd(0x5220);    // write "R"
    lcd(0x3A20);    // write ":"

    lcd(numToChar[var4] << 8 | 0x0020); // write var4
    lcd(numToChar[var3] << 8 | 0x0020); // write var3
    lcd(numToChar[var2] << 8 | 0x0020); // write var2
    lcd(numToChar[var1] << 8 | 0x0020); // write var1

    lcd(0x4f20);    // write "O"
    lcd(0x6820);    // write "h"
    temp = (int) freq;

    // get individual values of frequency
    var4 = ((temp/1000)%10);
    var3 = ((temp/100)%10);
    var2 = ((temp/10)%10);
    var1 = (temp%10);

    lcd(0x8000);    // write to top bar
    lcd(0x4620);    // write "F"
    lcd(0x3A20);    // write ":"

    lcd(numToChar[var4] << 8 | 0x0020); // write var4
    lcd(numToChar[var3] << 8 | 0x0020); // write var3
    lcd(numToChar[var2] << 8 | 0x0020); // write var2
    lcd(numToChar[var1] << 8 | 0x0020); // write var1

    lcd(0x4820);    // write "H"
    lcd(0x7A20);    // write "z
}

void TIM2_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM2->SR & TIM_SR_UIF)!= 0)
    {
        TIM2->SR &= ~(TIM_SR_UIF); // clear update interrupt flag
        TIM2->CR1 |= TIM_CR1_CEN; // restart stopped timer
    }
}

void TIM3_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM3->SR & TIM_SR_UIF) != 0)
    {
        update_lcd(); //updates lcd
        TIM3->SR &= ~(TIM_SR_UIF); // clear update interrupt flag
        TIM3->CR1 |= TIM_CR1_CEN; // restart stopped timer
    }
}

void EXTI0_1_IRQHandler()
{
    float counter = 0;
    /* Check if EXTI1 interrupt pending flag is indeed set */
    if ((EXTI->PR & EXTI_PR_PR1) != 0)
    {
        if(edge == 0){
            TIM2->CNT = 0x00000000; // clear count register
            TIM2->CR1 |= TIM_CR1_CEN; // start counter
            edge = 1; // set edge variable
        }
        else{
            TIM2->CR1 &= ~(TIM_CR1_CEN); // stop count register
            counter = TIM2->CNT; // get count value
            freq = SystemCoreClock/counter; // calculate freqency
            edge = 0; // set edge variable
        }

        EXTI->PR = EXTI_PR_PR1; // clear interupt pending flag by wrting 1 to it
    }
}

#pragma GCC diagnostic pop

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