Due: November 30, 2016

# Final Project:

# PWM Signal Generation and Monitoring System

Mathew Szymanowski V00825301
Liam Scholte V00802723

CENG 355 Lab: B05

Microprocessor-Based Systems
Dr. Daler N. Rakhmatov

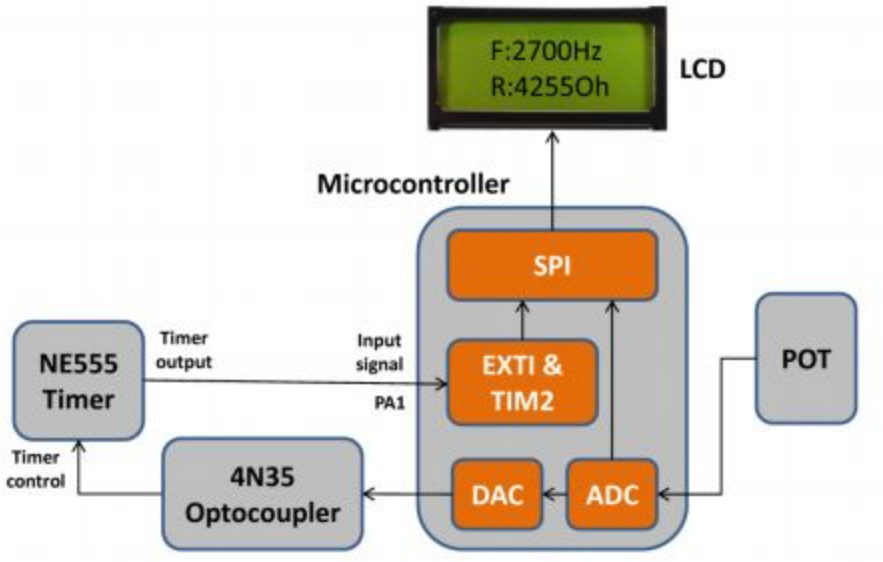
# Table of Contents

# Introduction

The purpose of this report is to discuss the design and implementation of a system that can measure the frequency of a pulse-width-modulated (PWM) signal. The frequency signal will be generated by an NE555 timer, which will be sent to an STM32F0 Discovery board, where the frequency will be measured. The measured frequency will be displayed on an LCD display that is located on a PBMCUSLK board. A potentiometer that is also housed on the PBMCUSLK board will be used to adjust the input signals to the system, resulting in different output frequencies by the NE555 timer.

# System Design

The overall system can be summarized by the following diagram:




The input to the system will come from a 5K Ohm potentiometer on the PBMCUSLK board, which will send a signal to the STM32F0 Discovery board. An analog-to-digital converter (ADC) will be used to measure the signal from the potentiometer, which will allow a software application running on the discovery board to measure the resistance of the potentiometer. This resistance will be one of two values that is displayed on the LCD.

The digital signal that is obtained from the ADC will be transformed back into an analog signal by a digital-to-analog converter (DAC). The analog signal from the DAC will be sent as input to a 4N35 octocoupler, which will be used to control the frequency of the signal generated by the NE555 timer. The signal generated by the NE555 timer will be sent back to the STM32F0 board.

The software application running on the discovery board will receive a series of interrupts at every rising edge of the signal received from the NE555 timer. Using one of the timers built into the discovery board, the period and frequency of the signal will be calculated. This frequency will be the second of two values that is displayed on the LCD.

The discovery board must be able to send the calculated frequency and resistance and display it on the PBMCULSK's LCD. This will be accomplished using the serial peripheral interface (SPI) so that the microcontroller on the discovery board may communicate with the LCD.

## Measuring the Potentiometer Resistance

The 5K Ohm potentiometer on the PBMCULSK board must first be connected to the ADC on the STM32F0 board. This can be accomplished connecting the POT pin (accessible via the J10 connector on the PBMCULSK) to the PC1 pin on the STM32F0 board. The PC1 pin will provide the analog signal to the ADC.

To configure the ADC, PC1 will need to be set to analog mode. The channel selection register for the ADC must also be set to channel 11, which will correspond to PC1 on the discovery board. In configuration register 1 for the ADC, the data alignment and data resolution must be set to right aligned and 12 bits, respectively. This will ensure that the analog signals are converted to values between 0 and 4095. Before performing and conversions, the ADC should also be calibrated to remove any initial offset error by setting the ADCAL bit in the ADC control register and waiting until it is cleared by hardware, which will indicate that the calibration is complete. Lastly, the ADC must be enabled by setting the ADEN bit in the ADC control register and waiting until the ADRDY flag is set by hardware, indicating that the ADC is ready for conversions.

Once the ADC has been initialized, the analog signal from the potentiometer can be converted to a digital signal. The resistance value of the potentiometer may change over time, so polling is used to continuously measure the signal and repeatedly calculate the resistance. This will ensure that the most up-to-date resistance value for the potentiometer is stored. This polling is performed inside an infinite while loop in the main function of the program.

Every time the loop executes, the analog signal that is sent from the potentiometer to PC1 is converted to a digital value. This is easily accomplished by calling the function shown below.

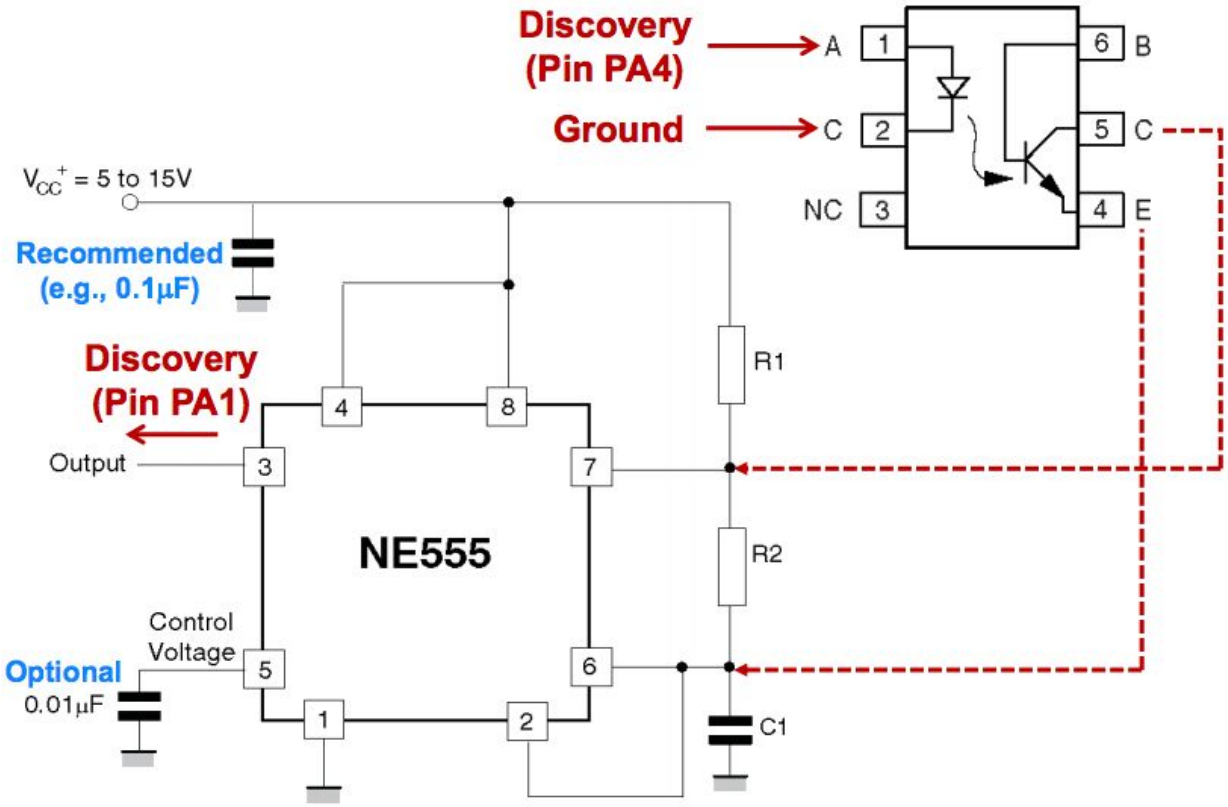
```
unsigned int convertAnalogToDigital()
{
    //Start the analog to digital conversion
    ADC1->CR |= ADC_CR_ADSTART;

    //Wait until conversion finishes
    while((ADC1->ISR & ADC_ISR_EOC) == 0);

    //Read the result of the conversion
    return ADC1->DR;
}
```

The returned value is then used to calculate the resistance of the potentiometer. The calculation for the resistance is based on the knowledge that the digital value from the ADC can be any integer between 0 and 4095 and that the potentiometer has a maximum resistance of 5000Ω. The equation for the resistance *R* is shown below. The resistance will depend on the value *d*, which is the value obtained from the ADC.

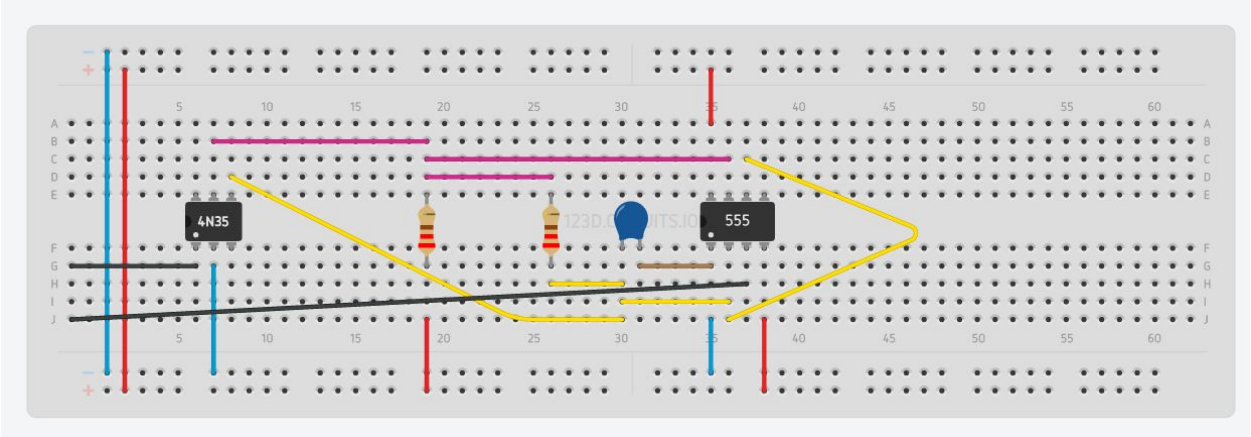
$$R = \frac{5000}{4095}d$$

## Generating the PWM Signal

The pin PA4 on the STM32F0 will be used to send the analog signal from the DAC to the 4N35 octocoupler. The octocoupler will be used to control the frequency outputted by the NE555 timer. The 4N35 and NE555 must be wired according to the circuit diagram below.



In terms of physical wiring, the above circuit can be implemented on a breadboard according to the breadboard diagram below. The negative power rails are connected to ground. The positive power rails are connected to a 5V supply. The black wires that are connected to the 4N35 and the NE555 will be connected to the DAC (pin PA4 on discovery board) and pin PA1 on the discovery board, respectively.




In order to configure the DAC, PA4 will need to be set to analog mode, and DAC channel 1 will need to be enabled. This will allow digital-to-analog conversions to take place where the output analog signal will come from PA4 on the discovery board.

Once the DAC is configured, the conversion can easily be accomplished from inside the main function while loop by calling the function below. It will take in a 12-bit value from 0 to 4095 and convert it to an analog value to be sent to the 4N35 octocoupler.

```
void convertDigitalToAnalog(unsigned int digital)
{
    //Clear the right 12 bits
    DAC->DHR12R1 &= ~DAC_DHR12R1_DACC1DHR;

    //Set the right 12 bits
    DAC->DHR12R1 |= digital;
}
```

## Measuring the PWM Signal Frequency

The signal outputted from the NE555 timer is sent to pin PA1 on the STM32F0. PA1 must be configured via software to generate interrupts at every rising edge. Additionally, the 32-bit timer TIM2 that is on the STM32F0 will need to be configured to measure the period of the PWM signal.

The number of rising edges that have been detected will be tracked. Every time an even rising edge has occurred, a timer will start, and every time an odd rising edge had occurred, the timer

will stop. The elapsed time between when the timer starts and stops will be the period of the signal, which can be used to calculate the frequency of the signal.

TIM2 is configured as a regular timer that interrupts only on overflow in case the period is longer than the timer can measure. The rising edge interrupt is configured in the external interrupt, EXTI_Init(), function. EXTI1 is mapped to the PA1 input and EXTI is set for interrupts on rising edge trigger.

```
void myEXTI_Init()
{
        /* Map EXTI1 line to PA1 */
        SYSCFG->EXTICR[0] = SYSCFG_EXTICR1_EXTI1_PA;

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

The IRQ handler for EXTI1 has a static local variable that is used for counting every even rising edge. The timer starts on an even edge and stops on an odd edge to measure the full period. After the odd edge the IRQ handler calculates the signal period along with the frequency using TIM2 count and the system core clock.

```
/* Read out the count register */
int count = TIM2->CNT;

/* Stop the timer */
TIM2->CR1 &= 0xFFFE;

// Calculate the frequency
double signalPeriod = count / (double) SystemCoreClock;
global_frequency = (int) (1 / signalPeriod);
```

## Displaying Frequency and Resistance on the LCD

The values obtained for the frequency and resistance must be displayed on the LCD. Because these values may be updating multiple times per second, this design will only send those values to the LCD at a rate of once per second. This will avoid any potential issues with overloading the LCD by updating its displayed data too rapidly, and it will also make it easier for a human to read the values on the LCD if they are refreshed at a slower rate.

The general purpose input and output ports PB3 and PB5 are configured to the SPI MOSI and the SPI shifting clock since they are the only ports with that alternate function capability. The SPI configuration function is mostly reproduced from the lecture slides and the SPI library due to the unique protocol for the stm32 microcontroller. The storage clock is configured to PB4 as a general purpose output with the lock signal being manipulated by the PB4 bit reset register. LCD configuration consisted of an algorithm to ensure that the LCD is in 4 bit mode. As a result of the LCD's latency, a delay was required between each command sent to it.

Data that is sent to the LCD must pass through the 74HC595 shift bit register. At the most basic level the HC595 received data through the SPI_SendData8() command that is in the

HC595Write() function shown below. The HC595Write function is used by the writeToLCD function that determines whether a command or a character is being sent. To send 8 bits to the LCD the writeToLCD calls the HC595Write function six times, three times for the higher 4 bits and three for the lower 4 bits.

```
void HC595Write(char data)
{
    /* Force your LCK signal to 0 */
        GPIOB->BRR = GPIO_Pin_4;

    /* Wait until SPI1 is ready (TXE = 1 or BSY = 0) */
        while((SPI1->SR & SPI_SR_BSY) != 0);

    /* Assumption: your data holds 8 bits to be sent */
    SPI_SendData8(SPI1, data);

    /* Wait until SPI1 is not busy (BSY = 0) */
        while((SPI1->SR & SPI_SR_BSY) != 0);

    /* Force your LCK signal to 1 */
        GPIOB->BSRR = GPIO_Pin_4;
}
```

TIM3 is configured to generate interrupts every second to update the display. The IRQ for TIM3 retrieves the current frequency and resistance values and stores them in two strings with the proper format for each value. The 0xF4 hexadecimal value in the code below is the character corresponding to the omega value in the LCD. The strings are then sent to the corresponding lines on LCD.

```
//Create strings holding resistance and frequency values
char frequencyStr[9];
char resistanceStr[9];
sprintf(frequencyStr, "F:%4dHz", global_frequency);
sprintf(resistanceStr, "R:%4d%c", global_resistance, 0xF4);

//Move cursor to start of first line
sendCommandToLCD(0x80);

writeStringToLCD(frequencyStr);

//Move cursor to start of second line
sendCommandToLCD(0xC0);

writeStringToLCD(resistanceStr);
```

# Testing

Testing was separated into six components that were tested independently from each other.

## Measuring Potentiometer resistance

The potentiometer resistance was tested to ensure it is working correctly by simply printing debug information (such as the digital value from the ADC or the calculated resistance) to the console. The voltage of the signal coming from the potentiometer was measured using a multimeter.

## Converting Digital Value to Analog Signal

The signal from the DAC was also measured using a multimeter. By instructing the DAC to convert fixed digital values to analog signals, the multimeter can be used to measure the voltages of the output analog signal from the DAC. This can be done without needing to wire the potentiometer to the ADC.

## Circuit for PWM Signal

The entire circuit for the 4N35 and NE555 was tested using an oscilloscope to confirm that the output of the NE555 is a PWM signal. Modifying the digital value that is sent to the DAC should cause the NE555 output frequency to change.

## Frequency Measurement

The code for measuring the frequency of a signal was tested by using a waveform generator and connecting it to PA1. The outputted frequency of the waveform generator was adjusted in order to test the frequency measuring software against a wide range of frequencies.

## Sending Data Using SPI

The SPI will be sending 8-bit words to the 74HC595 shift register located on the PBMCUSLK board. The 74HC595 is connected to the LCD which is accessible via the J9 connector next to the LCD on the PBMCUSLK. The J9 pins can be connected to LEDs that are accessible via the J10 connector. By doing so, the LEDs were used to display the values of each of the 8 bits for debugging.

## Displaying Data on LCD

Once the SPI was working correctly, the LCD was tested. It is useful to send a command to the LCD that will display the cursor and make it blink. Doing so first shows that the LCD is correctly able to interpret 4-bit commands. Second, the blinking cursor is a useful debugging tool when attempting move the cursor to a specific position on the display. Without the blinking cursor, the only way to tell that the cursor position is correct is to write a character to the display.

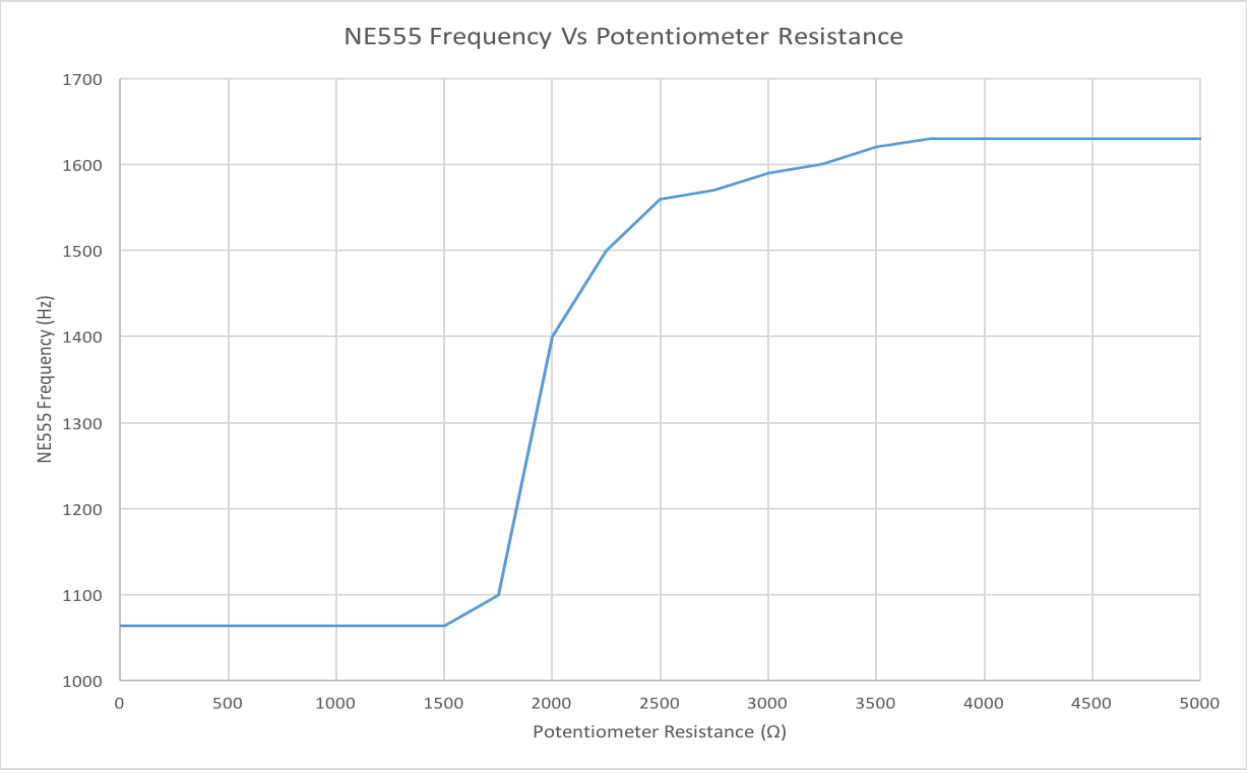
# Results

With the entire system configured and running, the LCD is able to show the most recently measured values for the frequency generated by the NE555 and the resistance of the potentiometer. The displayed values are refreshed on the LCD at a rate of once per second.

Due to limitations on the number of characters that the LCD can display, the system as it is currently designed is only capable of displaying frequencies in the range of 0Hz to 9999Hz and resistances in the range of 0Ω and 99999Ω. Since the system using a 5K Ohm potentiometer, it is only actually necessary for the LCD to display resistances between 0Ω and 5000Ω.

The software that is running on the STM32F0 has some limitations to the range of frequencies it is able to accurately measure. The microcontroller has a 48MHz clock and is using a 32-bit timer (with no pre-scaler) to measure frequencies. As such, it is only able to measure signals with a period of less than about 89.48 seconds without overflowing the timer, which corresponds to frequencies of about 0.01Hz or higher.

When testing the entire system, it was observed that low and high resistances for the potentiometer will have little effect on the the frequency outputted by the NE555 timer. Specifically, it appears that potentiometer resistances in between about 1500Ω and 3500Ω will have noticeable impacts on the frequency values. Any resistances outside that range will not cause the frequency to change. This is shown in the following graph.



# Discussion

The system is generally implemented exactly as specified in the laboratory manual and related resources. One key omission from the specification is when and how the LCD display is supposed to be refreshed, so it was decided that the LCD would be updated every second with the most recently measured frequency and resistance values. This required the use of an additional timer TIM3 to generate interrupts at 1-second intervals. Secondly, this design uses the 'Ω' character instead of 'Oh' to display the ohms unit for resistance on the LCD. This has the advantage of being less ambiguous because the letter 'O' could be misinterpreted as the number '0'. Using the character 'Ω' also has the advantage of using 1 less character on the LCD. Since the LCD can display only 8 characters per line, this could potentially allow up to 5-digit resistances to be displayed if a 10K Ohm potentiometer (or higher) were used.

It is advantageous to implement this system in a modular fashion. That is, write the code for a specific portion of the design, along with any necessary wiring, and then test it. By doing this, it is easier to understand why the system fails and how to fix it than if the trying to debug the entire system as a whole.

Furthermore, by working on specific components of the system, it is easier to budget time set delivery dates for each components. Ultimately this helps ensure that the entire system is working properly and completed on time.

# References

D. N. Rakhmatov, "Microprocessor-Based Systems," *Lab Webpage: Microprocessor-Based Systems*. [Online]. Available: http://www.ece.uvic.ca/~ceng355/lab/. [Accessed: 10-Sept-2016].

# Appendix A: The main.c File for the System

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

// ----------------------------------------------------------------------------
// School: University of Victoria, Canada.
// Course: CENG 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)

#define myTIM3_PRESCALER ((uint16_t) 0xFFFF)

#define myTIM6_PRESCALER ((uint16_t) 4)

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

void myGPIOA_Init(void);
void myGPIOB_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
void myTIM2_Init(void);
void myTIM3_Init(void);
void myTIM6_Init(void);
void myEXTI_Init(void);
void mySPI_Init(void);
void myLCD_Init(void);

void wait(int);
```

```
unsigned int convertAnalogToDigital(void);
void convertDigitalToAnalog(unsigned int);
int calculateResistance(unsigned int);

void writeToLCD(char, int);
void writeStringToLCD(char *);
void sendCommandToLCD(char);
void HC595Write(char);
void set4BitLCDMode();

//Stores values for printing to LCD display
int global_resistance = 0;
int global_frequency = 0;

int main(int argc, char* argv[])
{
    myGPIOA_Init();
    myGPIOB_Init();
    myTIM2_Init(); //Initialize timer used for measuring frequency
    myEXTI_Init();
    myADC_Init();
    myDAC_Init();
    mySPI_Init();
    myTIM3_Init(); //Initialize timer used for measuring 1 second LCD print intervals
    myTIM6_Init(); //Initialize timer used for waiting for short periods
    myLCD_Init();


    while (1)
    {
        //Convert potentiometer signal to a digital value
        unsigned int digital = convertAnalogToDigital();

        //Use the digital value to calculate resistance of pot
        global_resistance = calculateResistance(digital);

        //Convert digital signal back to analog signal
        convertDigitalToAnalog(digital);
    }

    return 0;
}


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

```
    /* Configure PA1 as input */
    // Relevant register: GPIOA->MODER
    GPIOA->MODER &= 0xFFFFFFF3;

    /* Ensure no pull-up/pull-down for PA1 */
    // Relevant register: GPIOA->PUPDR
    GPIOA->PUPDR &= 0xFFFFFFF3;
}

void myGPIOB_Init()
{
    /* Enable clock for GPIOB peripheral */
    // Relevant register: RCC->AHBENR
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;

    //Configure PB5 as alternate for MOSI
    GPIOB->MODER &= 0xFFFFFBFF;
    GPIOB->MODER |= 0x800;

    //Configure PB3 as alternate for Shifting clock SCK
    GPIOB->MODER &= 0xFFFFFFBF;
    GPIOB->MODER |= 0x80;

    //configure PB4 as output
    GPIOB->MODER &= 0xFFFFFDFF;
    GPIOB->MODER |= 0x100;
}

void myADC_Init()
{
    // Enable clock for GPIOC
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

    // Enable clock for ADC
    RCC->APB2ENR |= RCC_APB2ENR_ADCEN;

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

    // Disable pull-up/pull-down
    GPIOC->PUPDR &= 0xFFFFFFF3;

    // Set Continuous Conversion and Overrun Mode and right align
    ADC1->CFGR1 |= ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD;
    ADC1->CFGR1 &= ~ADC_CFGR1_ALIGN;

    //Select Channel 11 (which corresponds to PC1)
    ADC1->CHSELR |= ADC_CHSELR_CHSEL11;

    //Set sampling rate to 239.5
```

```
    ADC1->SMPR |= ADC_SMPR_SMP;

    //Start calibration and wait for completion
    ADC1->CR |= ADC_CR_ADCAL;
    while((ADC1->CR & ADC_CR_ADCAL) != 0);

    /* ADC is enabled */
    ADC1->CR |= ADC_CR_;

    /* ADC is ready */
    while((ADC1->ISR & ADC_ISR_ADRDY) == 0);
}

void myDAC_Init()
{
    // Enable clock for GPIOA
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    // Enable clock for DAC
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;

    // Configure PA4 analog mode
    GPIOA->MODER |= 0x00000300;

    //Disable pull-up/pull-down
    GPIOA->PUPDR &= 0xFFFFFCFF;

    //Enable DAC channel 1 (which corresponds to PA4)
    DAC->CR |= DAC_CR_EN1;
}


void myTIM2_Init()
{
    /* Enable clock for TIM2 peripheral */
    // Relevant register: RCC->APB1ENR
    RCC->APB1ENR |= RCC_APB1RSTR_TIM2RST;

    /* Configure TIM2: buffer auto-reload, count up, stop on overflow,
     * enable update events, interrupt on overflow only */
    // Relevant register: TIM2->CR1
    TIM2->CR1 &= 0xFFED;
    TIM2->CR1 |= 0x8D;

    /* 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 |= 0x1;

    /* 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;
}

void myTIM3_Init(void)
{
    /* Enable clock for TIM3 peripheral */
    // Relevant register: RCC->APB1ENR
    RCC->APB1ENR |= RCC_APB1RSTR_TIM3RST;

    /* Configure TIM3: buffer auto-reload, count down, stop on overflow,
     * enable update events, interrupt on underflow only */
    // Relevant register: TIM3->CR1
    TIM3->CR1 &= 0xFFFD;
    TIM3->CR1 |= 0x9D;

    /* Set clock prescaler value */
    //Frequency is 732Hz
    TIM3->PSC = myTIM3_PRESCALER;

    /* Set auto-reloaded delay */
    TIM3->ARR = myTIM3_PRESCALER;

    //Should correspond to a value of 1 second
    TIM3->CNT = SystemCoreClock / myTIM3_PRESCALER;

    /* Update timer registers */
    // Relevant register: TIM3->EGR
    TIM3->EGR |= 0x1;

    /* Assign TIM3 interrupt priority = 0 in NVIC */
    // Relevant register: NVIC->IP[3], or use NVIC_SetPriority
    NVIC_SetPriority(TIM3_IRQn, 0);

    /* Enable TIM3 interrupts in NVIC */
    // Relevant register: NVIC->ISER[0], or use NVIC_EnableIRQ
    NVIC_EnableIRQ(TIM3_IRQn);
```

```
    /* Enable update interrupt generation */
    // Relevant register: TIM3->DIER
    TIM3->DIER |= TIM_DIER_UIE;

    //Start timer
    TIM3->CR1 |= 0x1;
}

void myTIM6_Init()
{
    /* Enable clock for TIM6 peripheral */
    // Relevant register: RCC->APB1ENR
    RCC->APB1ENR |= RCC_APB1RSTR_TIM6RST;

    /* Configure TIM6: buffer auto-reload,
     * enable update events */
    TIM6->CR1 |= 0x84;


    /* Set clock prescaler value */
    //Frequency is 12MHz
    TIM6->PSC = myTIM6_PRESCALER;
}

void wait(int microseconds)
{
    //12 cycles with prescaler = 1 microsecond
    TIM6->ARR = 12 * microseconds;

    //Set count to 0
    TIM6->CNT = 0;

    //Start timer
    TIM6->CR1 |= 0x1;

    //Clear status register
    TIM6->SR = 0;

    //Wait until timer overflows
    while((TIM6->SR & 0x1) == 0);

    //Stop timer
    TIM6->CR1 &= 0xFFFE;
}

void myEXTI_Init()
{
    /* Map EXTI1 line to PA1 */
    //TODO: Use |= instead of =
```

```
    SYSCFG->EXTICR[0] = SYSCFG_EXTICR1_EXTI1_PA;

    /* 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[1], 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);
}

void mySPI_Init()
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);

    GPIO_PinAFConfig(GPIOB, 3, GPIO_AF_0);
    GPIO_PinAFConfig(GPIOB, 5, GPIO_AF_0);

    SPI_InitTypeDef SPI_InitStructInfo;
    SPI_InitTypeDef* SPI_InitStruct = &SPI_InitStructInfo;
    SPI_InitStruct->SPI_Direction = SPI_Direction_1Line_Tx;
    SPI_InitStruct->SPI_Mode = SPI_Mode_Master;
    SPI_InitStruct->SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStruct->SPI_CPOL = SPI_CPOL_Low;
    SPI_InitStruct->SPI_CPHA = SPI_CPHA_1Edge;
    SPI_InitStruct->SPI_NSS = SPI_NSS_Soft;
    SPI_InitStruct->SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256;
    SPI_InitStruct->SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStruct->SPI_CRCPolynomial = 7;
    SPI_Init(SPI1, SPI_InitStruct);

    SPI_Cmd(SPI1, ENABLE);
}

void myLCD_Init()
{

    set4BitLCDMode();

    //2 lines of 8 characters are displayed
    sendCommandToLCD(0x28); //DL=0, N=1, F=0
```

```
    wait(37);

    //Curser not displayed and is not blinking
    sendCommandToLCD(0x0C); //D=1, C=0, B=0
    wait(37);

    //Auto increment DDRAM address after each access
    sendCommandToLCD(0x06); //I/D=1, S=0
    wait(37);

    //Clear display
    sendCommandToLCD(0x01);
    wait(1520);
}

void set4BitLCDMode()
{
    //We have lots of waits to make sure LCD is able to be set to 4 bit mode

    HC595Write(0x3);
    wait(1520);
    HC595Write(0x3 | 0x80);
    wait(1520);
    HC595Write(0x3);

    wait(1520);
    HC595Write(0x3);
    wait(1520);
    HC595Write(0x3 | 0x80);
    wait(1520);
    HC595Write(0x3);
    wait(1520);

    HC595Write(0x3);
    wait(1520);
    HC595Write(0x3 | 0x80);
    wait(1520);
    HC595Write(0x3);
    wait(1520);

    HC595Write(0x2);
    wait(1520);
    HC595Write(0x2 | 0x80);
    wait(1520);
    HC595Write(0x2);
    wait(1520);
}

/* 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 &= 0xFFFE;

        /* Restart stopped timer */
        // Relevant register: TIM2->CR1
        TIM2->CR1 |= 0x1;
    }
}

void TIM3_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM3->SR & TIM_SR_UIF) != 0)
    {

        //Create strings holding resistance and frequency values
        char frequencyStr[9];
        char resistanceStr[9];
        sprintf(frequencyStr, "F:%4dHz", global_frequency);
        sprintf(resistanceStr, "R:%4d%c", global_resistance, 0xF4);

        //Move cursor to start of first line
        sendCommandToLCD(0x80);

        writeStringToLCD(frequencyStr);

        //Move cursor to start of second line
        sendCommandToLCD(0xC0);

        writeStringToLCD(resistanceStr);

        /* Clear update interrupt flag */
        // Relevant register: TIM2->SR
        TIM3->SR &= 0xFFFE;

        //Should correspond to a value of 1 second
        TIM3->CNT = SystemCoreClock / myTIM3_PRESCALER;

        /* Restart stopped timer */
        // Relevant register: TIM2->CR1
        TIM3->CR1 |= 0x1;
    }
}
```

```
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void EXTI0_1_IRQHandler()
{
    static int edgeCounter = 0;

    /* Check if EXTI1 interrupt pending flag is indeed set */
    if ((EXTI->PR & EXTI_PR_PR1) != 0)
    {
        //Even edge
        if(edgeCounter % 2 == 0) {

            //Clear the timer count register
            TIM2->CNT = 0;

            //Start the timer
            TIM2->CR1 |= 0x1;
        }
        else {

            //Read out the count register
            int count = TIM2->CNT;

            //Stop the timer
            TIM2->CR1 &= 0xFFFE;

            //Calculate the frequency
            double signalPeriod = count / (double) SystemCoreClock;
            global_frequency = (int) (1 / signalPeriod);

        }

        //Clear EXTI1 interrupt pending flag (EXTI->PR)
        EXTI->PR |= EXTI_PR_PR1;

        ++edgeCounter;
    }
}

unsigned int convertAnalogToDigital()
{
    //Start the analog to digital conversion
    ADC1->CR |= ADC_CR_ADSTART;

    //Wait until conversion finishes
    while((ADC1->ISR & ADC_ISR_EOC) == 0);

    //Read the result of the conversion
    return ADC1->DR;
```

```
}

void convertDigitalToAnalog(unsigned int digital)
{
    //Clear the right 12 bits
    DAC->DHR12R1 &= ~DAC_DHR12R1_DACC1DHR;

    //Set the right 12 bits
    DAC->DHR12R1 |= digital;
}

int calculateResistance(unsigned int digital)
{
    return (int)((5000.0 / 4095.0) * digital);
}


void writeToLCD( char c, int isCommand)
{
    char RS = isCommand ? 0x00 : 0x40;
    char EN = 0x80;

    char highNibble = (c & 0xF0) >> 4;
    char lowNibble = c & 0xF;

    HC595Write(highNibble | RS);
    HC595Write(highNibble | RS| EN);
    HC595Write(highNibble | RS);
    HC595Write(lowNibble | RS);
    HC595Write(lowNibble | RS | EN);
    HC595Write(lowNibble | RS);
}

void writeStringToLCD(char * s)
{
    int i;
    for(i = 0; i < strlen(s); ++i)
    {
        writeToLCD(s[i], 0);
    }
}

void sendCommandToLCD(char c)
{
    writeToLCD(c, 1);
}

void HC595Write(char data)
{
```

```
    //Based off of example from
http://www.ece.uvic.ca/~daler/courses/ceng355/interfacex.pdf

    /* Force your LCK signal to 0 */
    GPIOB->BRR = GPIO_Pin_4;

    /* Wait until SPI1 is ready (TXE = 1 or BSY = 0) */
    while((SPI1->SR & SPI_SR_BSY) != 0);

    /* Assumption: your data holds 8 bits to be sent */
    SPI_SendData8(SPI1, data);

    /* Wait until SPI1 is not busy (BSY = 0) */
    while((SPI1->SR & SPI_SR_BSY) != 0);

    /* Force your LCK signal to 1 */
    GPIOB->BSRR = GPIO_Pin_4;

}


#pragma GCC diagnostic pop

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