

Signal Generation and Monitoring System

ECE 355 B01
University of Victoria

Matthew Ebert [REDACTED]
Ian McLaren [REDACTED]

Date of Project Demo: November 22, 2022
Date of Submission: November 26, 2022

| | |
|----------------------------|-----------|
| Glossary | 3 |
| Problem Description | 4 |
| Specification | 5 |
| Design | 5 |
| System Operation Design | 6 |
| Module Description | 8 |
| ADC | 8 |
| DAC | 8 |
| Waveform Generator | 8 |
| Frequency Calculator | 9 |
| LCD Interface | 9 |
| Results | 11 |
| Discussion | 12 |
| Design Limitations | 13 |
| References | 14 |
| Appendix A | 15 |
| Main.c | 15 |
| LCD.c | 24 |
| Main.h | 31 |

Glossary

| | |
|-------------------|---|
| STM32, MCU, STMF0 | STM32F0 Discovery Microcontroller Board |
| POT | Potentiometer |
| SPI | Serial Peripheral Interface |
| ADC | Analog Digital Converter |
| DAC | Digital Analog Converter |
| LCD | Liquid Crytsal Display |
| PBMCUSLK | MCU PROJECT BOARD STUDENT LEARNING KIT |

Problem Description

The goal of this project was to create an embedded system to control and monitor a frequency modulated signal. With an ADC input from a potentiometer, the system must output an analog signal to a 555 timer circuit and measure the output frequency. This frequency and the calculated potentiometer voltage must be displayed on an LCD using an SPI interface and a shift register. The objectives of the project are summarized as

1. Read an ADC value from a potentiometer
2. Generate an analog voltage with a DAC
3. Construct a NE555 timer and 4N35 optocoupler circuit which adjusts frequency based on the applied DAC analog voltage
4. Measure the output frequency from the NE555 timer circuit
5. Use an SPI interface to write to a 74HC595 shift register to control the LCD
6. Display both the calculated frequency and potentiometer resistance for the user

To accomplish these objectives, a STM32 Discovery microcontroller board was connected to a PBMCUSLK set up which contains the POT, LCD, and a bread board to create the signal generation circuit. Figure 1 shows an overview of the system required connect.

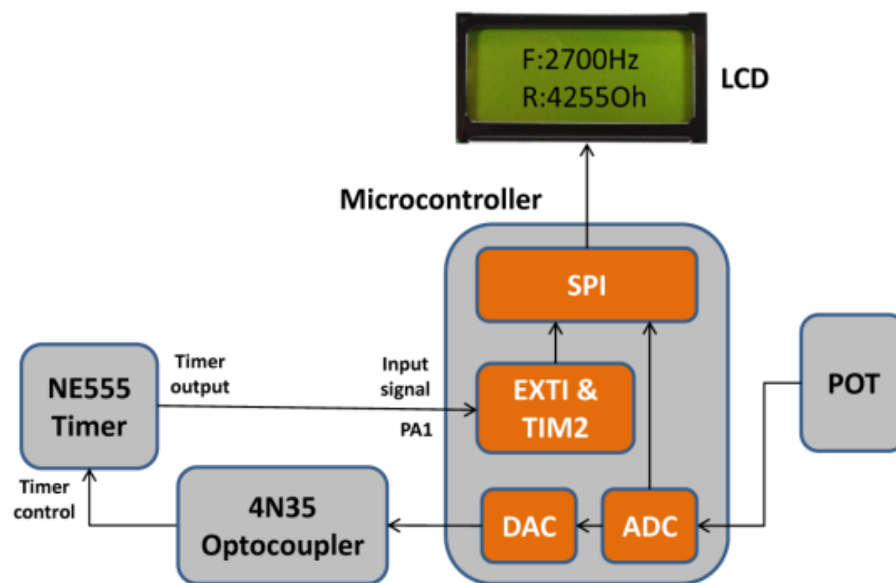


Figure 1: Overall System Diagram for Project [1]

Specification

The additional requirements of the system are shown in Table 1.

Table 1: Project requirements

| R. No | Requirement |
|-------|---|
| 1. | The system must accurately measure the POT resistance and output frequency |
| 2. | The ADC should indirectly change the output frequency through the microcontroller. (i.e. microcontroller controls frequency based on ADC input) |
| 3. | The LCD should display both the frequency and resistance value simultaneously |
| 4. | The system should not require any user input besides the potentiometer |
| 5. | The system should update in real time. |

Design

The system design consists of 4 components: The ADC; The DAC; the waveform generator; and the LCD interface. The BOM is shown in Table 2.

| Part No. | Component |
|-------------------------|---|
| STM32F0 Discovery Board | Microcontroller |
| NE555 Timer | Waveform generator |
| 4N35 Optocoupler | Waveform generator; Electrical Isolator |
| 5K POT | ADC potentiometer |
| 74HC595 shift register | LCD interface |
| Hitachi HD44780 LCD | LCD interface |

The STM32 interacts with each of these components separately and handles all interactions and data transfers. The controlling program uses interrupts and a short main loop to update in real time. The program makes use of onboard timers, external interrupts, SPI hardware, and GPIO pins.

System Operation Design

On startup, the system initializes the ADC, DAC, and SPI components on the MCU. The LCD is then reset using software commands. Next, the LCD display is initialized to 4 bit, 2 line, operation with a series of hardcoded byte commands. Finally, the external interrupt on PA1 and Timer 2 are initialized and enabled. The program then enter the main loop.

The main loop performs 3 tasks. First it waits for the ADC to finished converting and saves the value in data register (ADC_DR). It also converts the ADC value to the corresponding resistance value on the potentiometer. This is calculated by

$$R = V_{in} \cdot 5000 / 4095 \Omega \quad \text{Equation 1}$$

Secondly, it writes the saved value to the DAC control register (DAC_DHR12R1) to output a matching voltage to the waveform generator circuit. Finally, it checks if a frequency has been calculated from the external interrupt service routine on PA1 (EXTI_ISR). If it hasn't, it repeats the previous steps

If it has, it has, its writes the calculated frequency and resistance values to the LCD display. After writing to the display, the process delays for a given period which acts at the refresh rate of the system. The main loop then re-enables the EXTI on PA1 (which was disable after the frequency was calculated in the ISR) and repeats.

The frequency is calculated using the EXTI interrupt handler. When a rising edge is detected on PA1, an interrupt pending flag is set. The MCU then runs the interrupt handler.

The handler confirms the interrupt flag is set then checks if this is the first or second edge in the frequency calculation using the flag timerTrigger. If it is the first edge, the timerTrigger flag is set and TIM2 started.

If it is the second edge, the timer is immediately stopped. Next, the interrupt confirms that a write to the LCD is not in progress. If the LCD is currently being written, the interrupt is reset and control is passed back to the main loop. Otherwise, the value of the timer is converted to frequency in Hz. The WRITEFLAG is set to indicate the new frequency should be written to the LCD. Then the interrupt is disable to prevent it interrupting the CPU while writing to the LCD. The timerTriggered flag is reset and control passed back to the main loop.

A system flow chart is shown in Figure 2.

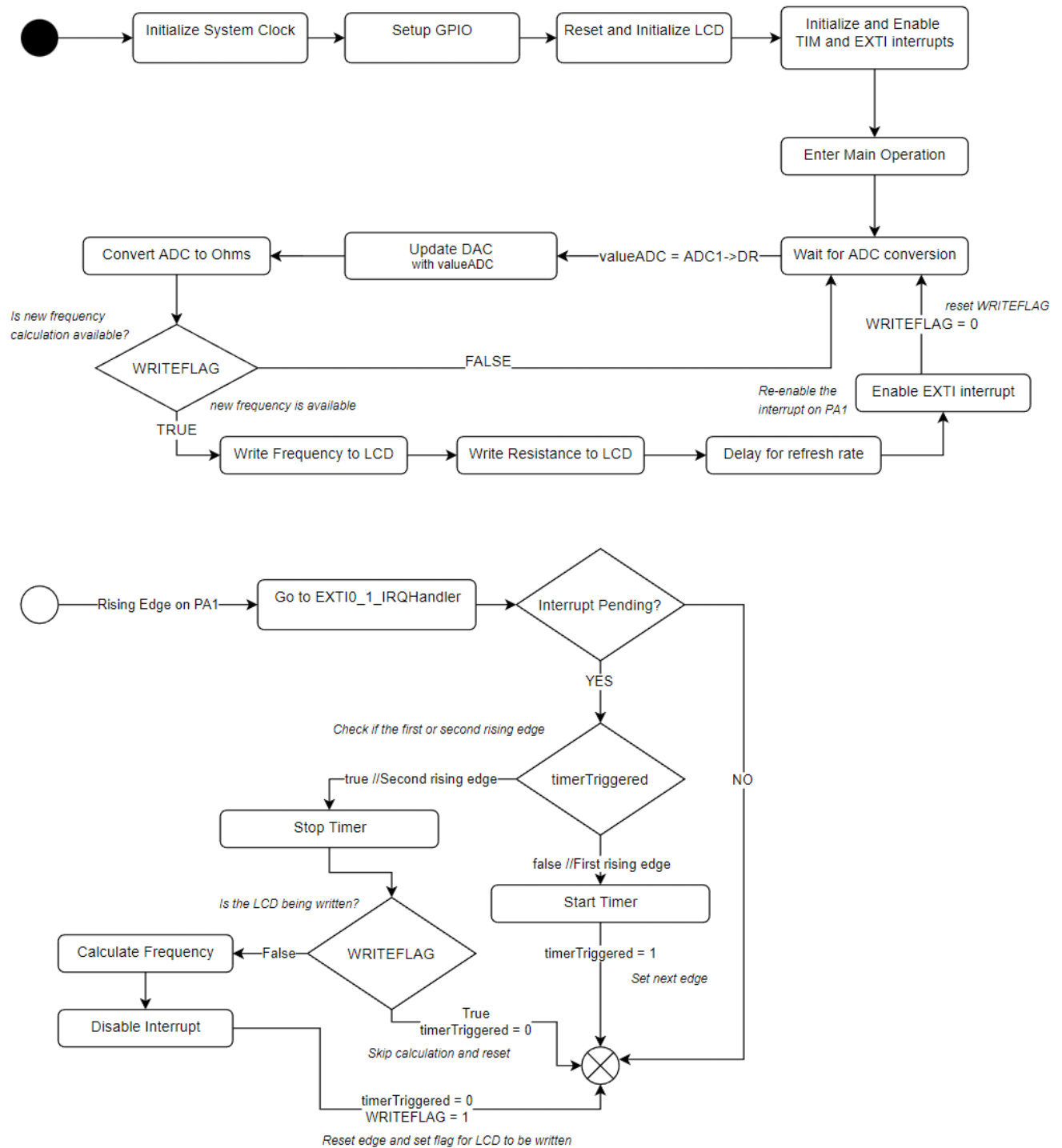


Figure 2: Flow chart for the implemented project system

Module Description

The 4 component modules are described below. Further details and the control implementation on the STM32 can be found within the code in Appendix A.

ADC

The 12 bit successive approximation ADC on the MCU was used to read an analog voltage from a user controlled potentiometer [2]. The output of the potentiometer on the PBMUSLK board was connected to pin PA5 on the MCU. The ADC was configured to read on the appropriate channel at the slowest sampling rate. When the PBMUSLK is powered a voltage between 0 and 3.3V is sent to the PA5 pin. The MCU uses the ADC to convert this value and store a bit representation in the register ADC_DR. The conversion formula between voltage and ADC bit value is

$$ADC \rightarrow DR = V_{in} * 4095 / V_{sys} = V_{in} * 1241.21 \quad \text{Equation 2}$$

The ADC was initialized for continuous conversion. On every loop of the main operation, when the ADC finished a conversion the value in ADC_DR is stored in a temporary variable to be used for other components.

DAC

The 12 bit DAC on the MCU was used to apply an analog voltage to the waveform generator. The output is controlled by the register DAC_DHR12R1[2]. The output voltage is proportional to the value stored in this register and can be calculated with

$$V_{out} = DAC * 3.3V / 4095 = DAC (8.058E - 4) V. \quad \text{Equation 3}$$

The output of the DAC in this system is set equal to the ADC temporary variable. This value is updated every time the main operating loop runs.

Waveform Generator

The waveform generator circuit consists of a NE555 timer circuit and an electrically isolating optocoupler. The circuit is shown in Figure 3. The 555 timer is in the multivibrator configuration which creates a 50% duty cycle square wave with the frequency

$$f = 1.443 / ((R1 + 2R2) * C1) \text{ Hz} \quad \text{Equation 4}$$

The 4N35 optocoupler connected across R2 acts as a current drain which controls the current through R2. This alters the voltage across R2 and changes the segment's equivalent resistance value which alters the output frequency. Since the DAC voltage controls the current through the transistor portion of the optocoupler, the output frequency of the system is controlled by the DAC voltage.

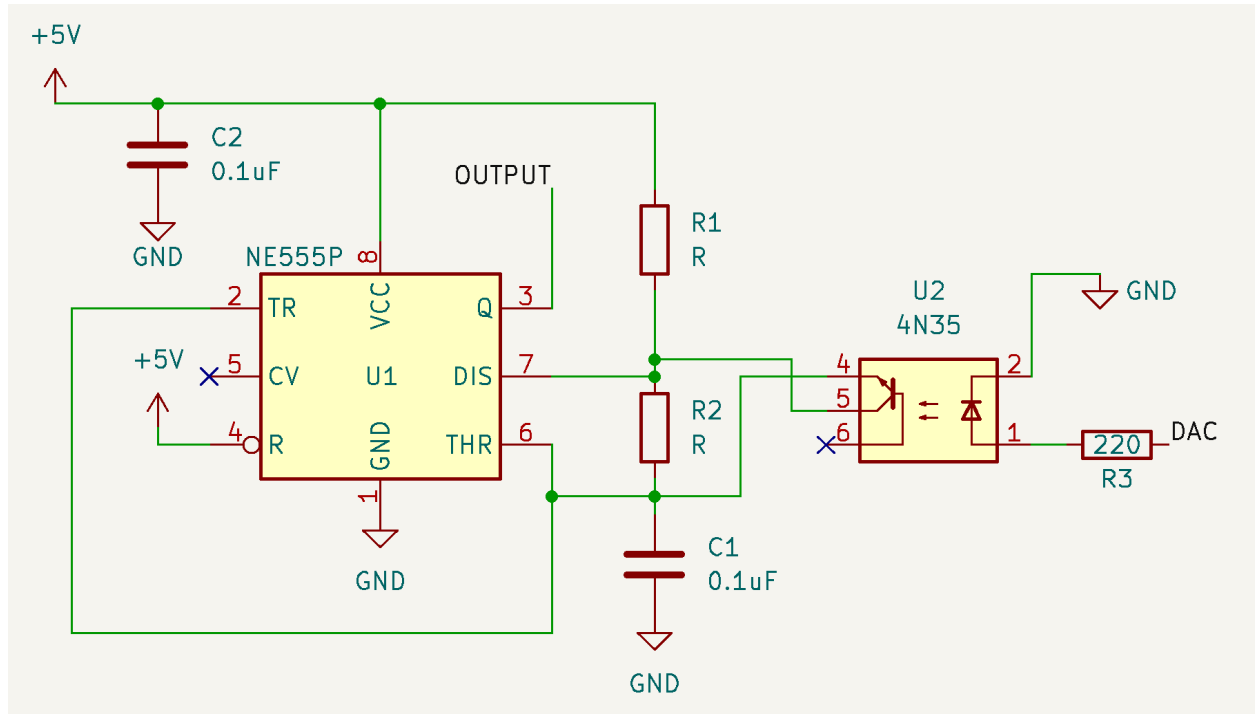


Figure 3: The function generator circuit consisting of 555 timer in multivibrator configuration and a 4N35 optocoupler electrical isolation circuit.

Frequency Calculator

The frequency calculator is an interrupt-driven software component which uses an external interrupt connected to pin PA1. The EXTI is configured to detect a rising edge. When a rising edge is detected, it starts a hardware timer on the MCU (Timer 2). When a second rising edge is detected it stops the timer and records the value stored register TIM2_CNT. Using the clock frequency, the frequency of the signal on PA1 can be calculated. At the end of this process the interrupt disables itself and sets a flag to indicate to the LCD interface that a new value is available. The process is restarted (interrupt enabled) when the LCD interface has finished writing the new value to the display.

LCD Interface

The LCD interface uses the SPI component on the MCU to write to a 74HC595 shift register which is connected to the LCD display. The SPI is configured using the HAL library for 8 bits at the lowest baud rate on pins PB3 and PB5 which are set to alternate mode. A GPIO pin (PB4) is initialized as an output and connected to the RCLK pin of the shift register (Figure 4).

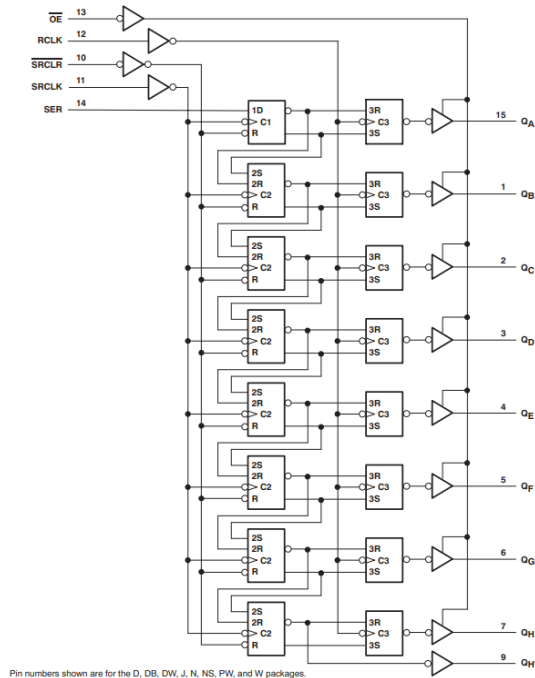




Figure 4: Block diagram for 74HC595 shift register [3]


Several software functions were implemented to control the LCD display with this system. The fundamental function of this interface is writeByte. This function sends 8 bits to the LCD with the bit order

| | | | | | | | |
|----|----|---|---|----|----|----|----|
| EN | RS | X | X | D7 | D6 | D5 | D4 |
|----|----|---|---|----|----|----|----|

After the bits are written with the SPI to the shift register, the RCLK value is pulsed with a LO-HI-LO signal from PB4 (with a delay). This shifts the written bits in parallel to the output pins which are connected to the LCD. With this method, 3 pins can write to 8 pins on the LCD. All used LCD functions are describe in Figure 5.

| | |
|---|------------------|
|  | writeByte |
| Param: The address of a byte to send over serial | |
| Desc: Using SPI hardware, sends 8 over serial to shift register | |

| | |
|---|-------------------|
|  | sendNibble |
| Param: 4 bits to send as a command to the LCD. | |
| Desc: Calls writeBytes 3 times to sends a sequence. This writes 4 bits to the LCD with an enable pulse. | |

| | |
|--|-----------------|
|  | sendByte |
| Param: 8 bits to send to the LCD as a command | |
| Desc: Calls sendNibble twice to send the upper and lower nibbles of a byte to the LCD. | |


| | |
|--|-----------------|
|  | sendLine |
| Param: - Pointer to a string to write to the LCD - length of string as int - line index to write to (0 or 1) | |
| Calls sendByte in a loop to writes a number of characters to a specified line on the LCD. It also initializes the DDRAM to the correct line. | |

Figure 5: Functions used to initialize and write to the LCD

Results

This solution proved effective for accomplishing the objectives of this project. The microcontroller successfully generated, monitored, and reported the frequency of the system on the LCD display. An oscilloscope and DMM were used to successfully verify the frequency and resistance values the system displayed.

The system could produce and measure frequencies between 1.0 kHz and 1.55 kHz and resistance between 0 and 5kOhms.

As for the results of this solution's particular components, they were also successes. Continuously polling and obtaining the ADC and DAC's values in the main

loop worked well, and the approach of using a temporary variable to store the ADC's converted value for other component's use was successful. The program for the SPI and LCD display also proved effective. Sending three consecutive nibbles with the value 0x03 successfully reset the device state, allowing for a consistent operation. The bytes sent as configuration bytes achieved the desired display parameters on the LCD display. Finally, writeByte() accomplished the needed display, allowing for the frequency and resistance to be shown.

Discussion

The results of this project proved satisfactory for the task and constraints defined in the objectives. However, a few aspects of this design could benefit from some further explanation.

Firstly, the ADC was configured to run at the slowest sampling rate. The three bits for the ADC_SMPR were set to 000, which leads to a sampling rate of 1.5 ADC clock cycles. The ADC sampling rate should be at least double the frequency being measured [4], so in this case, at least 3.0 kHz. Shown below is the calculation below, the conversion time for an ADC with a sampling rate of 1.5 clock cycles and ADC_CLK of 14Mhz [2].

$$t_{conv} = 1.5 + 12.5 = 14 \text{ ADC clock cycles} = 1 \mu s$$

This conversion time gives us a sampling frequency of 1Mhz, far exceeding the requirement of 3.0kHz. Thus, the slowest sampling rate is more than fast enough for our needs.

After the frequency calculation, the interrupt for rising edges is disabled. Only once the frequency has been written to the LCD display, is the interrupt once again enabled, and resumes waiting for the next rising edge. Disabling the interrupt prevents the interrupt wasting CPU time while the LCD is being written. Since only 1 frequency will be written to the LCD, any frequency calculation which runs during the writing period will be wasted. By pausing and resuming the interrupt, only the required frequency calculations occur during the system operation.

The SPI baud rate was configured at the slowest rate to ensure consistent communication between the stm board and the shift register and LCD. A faster baud rate may not give the slower components enough time to respond to each write which would cause data to be lost or corrupted at the receiver end. Additionally, since another wait is required for the LCD refresh rate, the slow baud rate does not have a negative performance impact on the system.

This design sends three consecutive bytes in a timed sequence with the value 0x03 to reset the LCD display. This is crucial for the success of the program, as this reset allows the code to be rerun without a hardware reset of the LCD. This reset, followed by the initialization of

the 4-bit, 2-line operation, ensures that the program is not attempting to initialize or write to the LCD display in a state in which it is not able to receive the data or is configured incorrectly. Without this reset, the frequency and resistance would not display correctly on a consistent basis.

Design Limitations

The system frequency range was limited by the signal generator circuit from the R2 and optocoupler components. The transistor portion of the optocoupler entered cutoff region at approximately 1.0V from the DAC. This state corresponds to the lowest frequency of 1.0kHz since the equivalent R2 resistance in Equation 4 is at its highest. The transistor entered saturation at a approximately 2.3V from the DAC which corresponds to a frequency of 1.55kHz when the equivalent R2 equals 0 Ohms. Thus, the controllable frequency range lies within the potentiometer range of 1.5kOhms to 3.48kOhms. To increase the frequency range of the system, different components and different capacitance and resistance values are required.

The refresh rate of the system is limited by the LCD refresh rate. Based on experimental results, a refresh rate of 10Hz produced readable results which update smoothly. This limitation is caused solely by the LCD since the next slowest component, the ADC, updates at 1MHz. Therefore, the refresh rate can be increased by acquiring a faster LCD screen.

This project also makes an assumption that the frequency and resistance calculated are aa value with no more than 4 digits. For example, had our frequency calculated been an eight-digit number, it would not have been able to display properly on the LCD display. As discussed in the Results section, a frequency this high is not possible due to our system's limitations, but was still an assumption made prior to the project's design.

References

- [1] A. Jooya et al. *ECE 355: Microprocessor-Based Systems Laboratory Manual*, (2018). Accessed: Nov 25, 2022. [Online]. Available: <https://ece.engr.uvic.ca/~ece355/lab/ECE355-LabManual-2018.pdf>
- [2] “.rm0091-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf” Accessed: Nov. 22, 2022. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0091-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf
- [3] “SN54HC595 data sheet, product information and support | TI.com.” <https://www.ti.com/product/SN54HC595> (accessed Nov. 26, 2022).
- [4] Z. Peterson. “Selecting a High Resolution or Frequency ADC.” Octopart. <https://octopart.com/blog/archives/2019/10/selecting-a-high-resolution-or-high-frequency-adc> (retrieved Nov. 25, 2022).

Appendix A

Main.c

```
/*
 * Created on: Nov. 21, 2022
 * Author: Matthew Ebert; V00884117
 * For: ECE 355; University of Victoria
 * Modified: Nov. 25, 2022
 *
 * Description: This program completes 4 tasks using an STM32F0 Discovery board.
 * 1. Read an analog voltage from a potentiometer
 * 2. Output an analog voltage via the onboard DAC
 * 3. Calculate the frequency of an input squarewave
 * 4. Display the frequency of the input wave and the resistance of the
potentiometer
 * on an LCD via a SPI interface
 *
 * This file requires the source LCD.c and main.h to run. It also uses the HAL SPI
library.
 */

#include "main.h"

volatile int timerTriggered = 0; // flag to indicate first or second rising edge
const unsigned int clockFrequency = 48000000; //board clock frequency
volatile char WRITEFLAG = 0; //flag to indicate if an LCD write is in progress
volatile int frequency=0; //the calculated frequency
volatile unsigned int signalPeriod=0; //the calculated period

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

```

{

    SystemClock48MHz(); //Set the board clock to 48MHz
    trace_puts("Hello World!");
    trace_printf("System clock: %u Hz\n", SystemCoreClock);


    myGPIOA_Init(); //enable clock for GPIOA pins
    myGPIOC_Init(); //not used


    myDAC_Init();
    myADC_Init();
    myLCD_Init(); //Set up SPI interface
    LCD_Setup(); //Reset and configure the LCD


    myEXTI_Init(); //Configure and Start EXTI on PA1
    myTIM2_Init(); //Configure onboard timer for frequency timing


    uint16_t valueADC = 0; //store last read ADC value
    float Rvalue = 0; //stores calculated resistance value
    uint16_t Res = 0; //stores Rvalue as integer
    //int count = 0; //debugging
    while(1){
        //count ++;


        //wait for adc to finish conversion
        while((ADC1->ISR & ADC_ISR_EOC) == 0){
            continue;
        }
        //save converted value from ADC
        valueADC = ADC1->DR;


        //set DAC
        DAC->DHR12R1 = valueADC;
        //calculate resistance
        Rvalue = ((float)valueADC)*(1.221);
        Res = Rvalue; //convert to uint16_t
    }
}

```



```

        /*this delay gives the interrupt time to complete frequency
        calculations if the frequency is very low (<100Hz)*/
        delay_ms(10);

        //if new frequency available
        if(WRITEFLAG){

            sendFreq(frequency);//write frequency to LCD
            sendADC(Res);//write resistance to LCD
            delay_ms(100);//delay for LCD to set refresh rate
            WRITEFLAG = 0;//indicate finished writing
            EXTI->IMR |= (1<<1);//re-enable interrupt on PA1

        }
    }
}

/*
DESCRIPTION: initialized the ADC on PA5
*/
void myADC_Init(){

    //Calibrate ADC
    //configure PA5 as Analog
    GPIOA->MODER |= (GPIO_MODER_MODER5_0 | GPIO_MODER_MODER5_1);
    //Clear ADC_CR
    ADC1->CR = 0x00000000;
    //enable clock for ADC
    RCC->APB2ENR |= RCC_APB2ENR_ADCEN;
    //enable ADC
    ADC1->CR |= ADC_CR_ADEN;
    //sample rate
    ADC1->SMPR |= ADC_SMPR_SMP_0|ADC_SMPR_SMP_1|ADC_SMPR_SMP_2;
    //Channel Select
    ADC1->CHSELR |= ADC_CHSELR_CHSEL5;
    //Configure Resolution
    ADC1->CFGR1 &= ~(ADC_CFGR1_RES_0|ADC_CFGR1_RES_1);

```

```

//Configure Alignment
ADC1->CFGR1 &= ~(ADC_CFGR1_ALIGN);
//Configure overrun Management
ADC1->CFGR1 |= (ADC_CFGR1_OVRMOD);
//Enable continuous conversion
ADC1->CFGR1 |= (ADC_CFGR1_CONT);

//wait for adc to be ready
while(((ADC1->ISR) & ADC_ISR_ADRDY) == 0) {
    continue;
}
trace_printf("ADC Configured Successfully\n");
//start ADC
ADC1->CR |= ADC_CR_ADSTART;
}

/*
DESCRIPTION: initialized the DAC on PA4
*/
void myDAC_Init(){
    //configure PA4 as Analog
    GPIOA->MODER |= (GPIO_MODER_MODER4_0 | GPIO_MODER_MODER4_1);
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
    //enable DAC clock
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;
    //enable DAC
    DAC->CR |= DAC_CR_EN1;
    //Disable high impedance
    DAC->CR &= ~(DAC_CR_BOFF1);

    //trace_printf("DAC Configured Successfully\n");
}

/*
DESCRIPTION: Enables clock for GPIOA pins
*/
void myGPIOA_Init()
{

```

```

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

}

/*
DESCRIPTION: Initilizes the onboard timer used for frequency calculations
*/
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 */
    TIM2->CR1 = ((uint16_t)0x008C);
    /* Set clock prescaler value */
    TIM2->PSC = myTIM2_PRESCALER;
    /* Set auto-reloaded delay */
    TIM2->ARR = myTIM2_PERIOD;
    /* Update timer registers */
    TIM2->EGR = ((uint16_t)0x0001);
    /* Assign TIM2 interrupt priority = 0 in NVIC */
    NVIC_SetPriority(TIM2_IRQn, 0);
    /* Enable TIM2 interrupts in NVIC */
    NVIC_EnableIRQ(TIM2_IRQn);
    /* Enable update interrupt generation */
    TIM2->DIER |= TIM_DIER_UIE;
}

void myEXTI_Init()
{
    /* Map EXTI2 line to PA1 */
    // Relevant register: SYSCFG->EXTICR[0]

```

```

SYSCFG->EXTICR[0] &= 0x0000FF0F;
/* EXTI2 line interrupts: set rising-edge trigger */
EXTI->RTSR |= (1<<1);

/* Unmask interrupts from EXTI2 line */
EXTI->IMR |= (1<<1);

/* Assign EXTI2 interrupt priority = 0 in NVIC */
NVIC_SetPriority(EXTI0_1_IRQn, 0);

/* Enable EXTI2 interrupts in NVIC */
NVIC_EnableIRQ(EXTI0_1_IRQn);
}

/*
DESCRIPTION: Runs everytime the TIM2 counter overflows
*/
void TIM2_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM2->SR & TIM_SR_UIF) != 0)
    {
        trace_printf("\n*** Overflow! ***\n");

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

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

/*
DESCRIPTION: Runs everytime a rising edge appears on PA1

```

```

*/
void EXTI0_1_IRQHandler()
{
    // Declare/initialize your local variables here...

    /* Check if EXTI2 interrupt pending flag is indeed set */
    if ((EXTI->PR & EXTI_PR_PR1) != 0)
    {
        //
        // 1. If this is the first edge:
        if (timerTriggered == 0) {
            //Clear count register (TIM2->CNT).
            TIM2->CNT = 0;
            //Start timer (TIM2->CR1).
            TIM2->CR1 |= TIM_CR1_CEN;
            timerTriggered = 1;
        }else{           // Else (this is the second edge):

            //Stop timer (TIM2->CR1).
            TIM2->CR1 &= ~(TIM_CR1_CEN);
            // - Calculate signal period and frequency.

            if(!WRITEFLAG) {
                if(TIM2->CNT < clockFrequency) {
                    //calculate values
                    frequency = clockFrequency / TIM2->CNT;
                    signalPeriod = TIM2->CNT / 48; //scaled by 1000000
                }
                if(TIM2->CNT > clockFrequency) {
                    //calculate values
                    unsigned int temp = TIM2->CNT / 1000000;
                    frequency = 4800 / temp; //scaled by 100000
                    signalPeriod = TIM2->CNT / clockFrequency;
                }
                WRITEFLAG = 1; //Indicate frequency is ready to write

                //disable interrupt to save processing time and wait for

```

```

        //LCD write.
        EXTI->IMR &= ~(1<<1);

    }

    //reset edge flag
    timerTriggered = 0;
}

//reset interrupt pending flag
EXTI->PR |= EXTI_PR_PR1;
//
}

}

/*
DESCRIPTION: Not used
*/
void myGPIOC_Init()
{
    /* Enable clock for GPIOC peripheral */
    RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

    /* Configure PC8 and PC9 as outputs */
    GPIOC->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0);
    /* Ensure push-pull mode selected for PC8 and PC9 */
    GPIOC->OTYPER &= ~(GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9);
    /* Ensure high-speed mode for PC8 and PC9 */
    GPIOC->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR8 | GPIO_OSPEEDER_OSPEEDR9);
    /* Ensure no pull-up/pull-down for PC8 and PC9 */
    GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR8 | GPIO_PUPDR_PUPDR9);
}

void SystemClock48MHz( void )
{

```

```

//
// Disable the PLL
//
RCC->CR &= ~(RCC_CR_PLLON);
//
// Wait for the PLL to unlock
//
while (( RCC->CR & RCC_CR_PLLRDY ) != 0 );
//
// Configure the PLL for a 48MHz system clock
//
RCC->CFGR = 0x00280000;

//
// Enable the PLL
//
RCC->CR |= RCC_CR_PLLON;

//
// Wait for the PLL to lock
//
while (( RCC->CR & RCC_CR_PLLRDY ) != RCC_CR_PLLRDY );

//
// Switch the processor to the PLL clock source
//
RCC->CFGR = ( RCC->CFGR & (~RCC_CFGR_SW_Msk) ) | RCC_CFGR_SW_PLL;

//
// Update the system with the new clock frequency
//
SystemCoreClockUpdate();
}

#pragma GCC diagnostic pop

// -----

```

LCD.c

```
/*
 * Created on: Nov. 21, 2022
 * Author: Matthew Ebert; V00884117
 * For: ECE 355; University of Victoria
 * Modified: Nov. 25, 2022
 *
 * Description: This file contains functions to configure and use the SPI hardware
on the
 * STM32F0 Discovery board. Additional functions are included to write specific
patterns
 * which control the LCD display on the PBMCUSLK through a 74HC595 shift register.
 *
 */

#include "main.h"

SPI_HandleTypeDef SPI_Handle; //Handle for configured SPI

volatile uint16_t delay_target = 0;

void LCD_Setup() {

    //Reset the LCD
    //This code replaces power cycling the LCD
    sendNibble(0x03);
    delay_ms(10);
    sendNibble(0x03);
    delay_ms(10);
    sendNibble(0x03);
    //Configure the LCD
    sendNibble(0x02); //set up for 4bit interface
    sendByte(0x28, 0); //two line mode
}
```



```

    sendByte(0x0C,0); //Display on
    sendByte(0x06,0); //auto increment DDRAM
    sendByte(0x01,0); //clear display
}

/*
Description: Sends an 8bit integer up to 4 digits to the
LCD in as displayed in the format F:xxxxHz
*/
void sendFreq(unsigned int freq){
    char frq[8];
    //convert int to char
    sprintf(frq, "F:%4uHz", freq);

    //send to LCD on the first line
    sendLine(frq, 8, 0);
}

/*
Description: Sends an 8bit integer up to 4 digits to the
LCD in as displayed in the format R:xxxxOh
*/
void sendADC(unsigned int adcvalue){
    char adc[8];
    //convert int to char
    sprintf(adc, "R:%4uOh", adcvalue);
    //send to LCD on the second line
    sendLine(adc, 8, 1);
}

/*
Description: Sends a number of characters to a specific
position on the LCD
*/
int sendAtPos (char *c, uint8_t len, uint8_t x, uint8_t y){
    y = (y<<6) | 0x80;

```

```

uint8_t pos = y|x;
sendByte(pos,0);//set up DDRAM

for (int i = 0; i<len; i++){
    sendByte(c[i],1);
}
return 1;
}

/*
Description: Sends a number of characters to a specific
line on the LCD
*/
int sendLine (char *c, uint8_t len, uint8_t line){
    line = (line<<6) | 0x80;
    sendByte(line,0);//set up DDRAM

    for (int i = 0; i<len; i++){
        //trace_printf("%c", c[i]);
        sendByte(c[i],1);
    }
    return 1;
}

/*
Description: Sends a complete byte to the LCD by sending
the upper and lower nibbles
*/
void sendByte (uint8_t B, uint8_t RS){
    RS = RS<<6;
    uint8_t H = (B>>4) | RS;
    uint8_t L = (0x0F & B) | RS;

    //send Upper nibble
    sendNibble(H);
    send lower nibble
    sendNibble(L);
}

```

```
/*
```

```
Description: Sends a nibble to the LCD with by writing a  
sequence of bits which creates a pulse on the EN bit
```

```
*/
```

```
void sendNibble (uint8_t nib){  
    //trace_printf("sending: %d\n", *txbuff);  
    uint8_t sequence [3];  
    sequence[0] = 0x00 | nib;  
    sequence[1] = 0x80 | nib;  
    sequence[2] = 0x00 | nib;  
    for(int i = 0; i<3;i++){  
        writeByte(&sequence[i]);  
    }  
    // trace_printf("sent\n");  
}
```

```
/*
```

```
Description: writes a bytes through the SPI interface and sends it  
to the LCD by pulsing the RCLK input (PB4)
```

```
*/
```

```
void writeByte(uint8_t *txbuff){  
    while(!__HAL_SPI_GET_FLAG(&SPI_Handle, SPI_FLAG_TXE) ){}  
  
    if(HAL_SPI_Transmit(&SPI_Handle, txbuff , 1 , HAL_MAX_DELAY) !=HAL_OK){  
        trace_printf("did not send...\n");  
    }  
    while(!__HAL_SPI_GET_FLAG(&SPI_Handle, SPI_FLAG_TXE) ){}  
  
    GPIOB->BSRR = lck_msk;  
    delay_ms(2); //delay for LCD to process  
    GPIOB->BRR = lck_msk;  
    //trace_printf("sent: %d", *txbuff);  
}
```

```
/*
```

```
Description: Configures the SPI interface
```

```

*/
void myLCD_Init(void){

    myTIM3_Init();

    //Enable clock for SPI1
    RCC->APB2ENR|= RCC_APB2ENR_SPI1EN;
    //enable clock for GPIOB
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
    //Configure PB3 and PB5 for SCLK and MOSI
    //Set as alternate function mode
    GPIOB->MODER |= GPIO_MODER_MODER3_1;
    GPIOB->MODER |= GPIO_MODER_MODER5_1;

    //Configure SPI handle
    //use SPI
    SPI_Handle.Instance = SPI1;
    //Output only
    SPI_Handle.Init.Direction = SPI_DIRECTION_1LINE;
    //set as master
    SPI_Handle.Init.Mode = SPI_MODE_MASTER;
    //8 bit transmit data size
    SPI_Handle.Init.DataSize = SPI_DATASIZE_8BIT;
    //set polarity low
    SPI_Handle.Init.CLKPolarity = SPI_POLARITY_LOW;
    //set clock phase
    SPI_Handle.Init.CLKPhase = SPI_PHASE_1EDGE;
    SPI_Handle.Init.NSS = SPI_NSS_SOFT;
    //set to lowest baud rate
    SPI_Handle.Init.BaudRatePrescaler =SPI_BAUDRATEPRESCALER_256 ;
    //set bit order
    SPI_Handle.Init.FirstBit = SPI_FIRSTBIT_MSB;
    SPI_Handle.Init.CRCPolynomial=7;

    //Initalize SPI with configuration
    HAL_SPI_MspInit(&SPI_Handle);
    if( HAL_SPI_Init(&SPI_Handle) !=HAL_OK){
        trace_printf("FAIL");
    }
}

```

```

    }
    __HAL_SPI_ENABLE(&SPI_Handle);

    myGPIOB_Init(); //setup RCLK pin
}

/*
Description: Set up PB4 as an output pin used for RCLK
*/
void myGPIOB_Init(){
    GPIOB->MODER |= (GPIO_MODER_MODER4_0);
    GPIOB->OTYPER &= ~(GPIO_OTYPER_OT_4);
    /* Ensure high-speed mode for PC8 and PC9 */
    GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR4);
    /* Ensure no pull-up/pull-down for PC8 and PC9 */
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR4);
    GPIOB->BSRR = lck_msk;
}

/*
Description: Delays a number of ms using TIM3
*/
void delay_ms(int delay){

    delay_target = 0;
    TIM3->CR1 |= TIM_CR1_CEN; //start timer
    while (delay_target < delay); //wait for delay
    TIM3->CR1 &= ~(TIM_CR1_CEN); //stop timer
}

/*
Description: Used exclusively for the delay_ms function
*/
void myTIM3_Init()
{

```

```

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

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

/* Set clock prescaler value */
TIM3->PSC = myTIM3_PRESCALER;
/* Set auto-reloaded delay */
TIM3->ARR = myTIM3_PERIOD;

/* Update timer registers */
TIM3->EGR = ((uint16_t)0x0001);

/* Assign TIM3 interrupt priority = 0 in NVIC */
NVIC_SetPriority(TIM3_IRQn, 0);
// Same as: NVIC->IP[3] = ((uint32_t)0x00FFFFFF);

/* Enable TIM3 interrupts in NVIC */
NVIC_EnableIRQ(TIM3_IRQn);
// Same as: NVIC->ISER[0] = ((uint32_t)0x00008000) */

/* Enable update interrupt generation */
TIM3->DIER |= TIM_DIER_UIE;
/* Start counting timer pulses */
TIM3->CR1 |= TIM_CR1_CEN;
}

/*
Description: Used exclusively for the delay_ms function.
when TIM3 is running this is run every 1 ms.
*/
void TIM3_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM3->SR & TIM_SR_UIF) != 0)
    {

```

```

        /* Read current PC output and isolate PC8 and PC9 bits */
        delay_target++;
        //trace_printf("%d",delay_target);
        TIM3->SR &= ~(TIM_SR_UIF); /* Clear update interrupt flag */
        TIM3->CR1 |= TIM_CR1_CEN; /* Restart stopped timer */
    }

}

//debugging
void printBin (uint8_t num){
    trace_printf("bin = ");
    for(int i = 0; i<8;i++){
        trace_printf("%ud", num>>(7-i));
    }
    trace_printf("bin = ");
}

```

Main.h

```

/*
 * Created on: Nov. 21, 2022
 * Author: Matthew Ebert; V00884117
 * For: ECE 355; University of Victoria
 * Modified: Nov. 25, 2022
 *
 * Description: Header file for main.c and LDC.c for ECE355 project
 *
 */

```

```

#ifndef MAIN_H_
#define MAIN_H_

#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"
#include "stm32f0xx_hal_spi.h"

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
#pragma GCC diagnostic ignored "-Wreturn-type"

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

/* Clock prescaler for TIM3 timer: no prescaling */
#define myTIM3_PRESCALER ((uint16_t)0x0000)
/* Delay count for TIM3 timer: 1 ms at 48 MHz */
#define myTIM3_PERIOD ((uint32_t)48000)
//msk for LCK pin for LCD

#define lck_msk 0x0010;

void SystemClock48MHz( void );

void myGPIOA_Init(void);
void myGPIOC_Init(void);

void myGPIOB_Init();
void myTIM3_Init(void);
void myLCD_Init(void);

```



```
void delay_ms(int delay);

void sendByte (uint8_t B, uint8_t RS);
void LCD_Setup();
void writeByte(uint8_t * txbuff);
void sendNibble(uint8_t nib);
void printBin (uint8_t num);
int sendLine (char *c, uint8_t len, uint8_t line);
int sendAtPos (char *c, uint8_t len, uint8_t x, uint8_t y);
void sendFreq(unsigned int freq);
void sendADC(unsigned int adc);

void myADC_Init();
void myDAC_Init();

void myTIM2_Init();
void myEXTI_Init();

void TIM2_IRQHandler();
void EXTI0_1_IRQHandler();

#endif /* MAIN_H_ */
```