Department of Electrical and Computer Engineering
University of Victoria
CENG 355 - Microprocessor-Based Systems

Project Report

Report submitted on:	7 December, 2016
To:	Jiachang Guo
Names:	T. Pimlott (V00800717)
	T. Stephen (V00812021)

Problem description / Specifications	_____/5
Design / Solution	_____/15
Testing / Results	_____/10
Discussion	_____/15
Code design and documentation	_____/15
Total	**_____/60**

1 Problem description

This project uses a STM32F0 Discovery microcontroller (hereafter referred to as "the microcontroller") and a PBMCUSLK project board ("the project board") to both generate and monitor a pulse width modulated (PWM) signal. A potentiometer on the project board controls a signal voltage that is read by the analog-to-digital (ADC) converter on the microcontroller. Using the ADC voltage, the microcontroller calculates the potentiometer resistance value and generates an output voltage via digital-to-analog (DAC) conversion. The output signal is isolated with a 4N35 optocoupler and used to control the frequency of a NE555 timer. The square wave timer output is fed back into the microcontroller, which measures the timer output signal frequency. The microcontroller uses its serial peripheral interface (SPI) to pass data to an LCD on the project board. The timer frequency and potentiometer resistance are displayed on the LCD and updated on an ongoing basis. Figure 1 shows the interactions of the major system components.

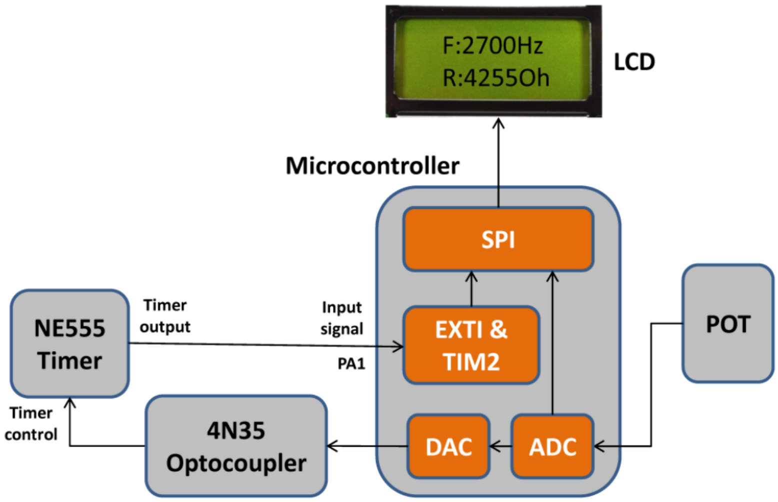


Figure 1: Block diagram for microcontroller-based resistance and frequency measurement system [1]

The lab manual [1] specifies some constraints on development, for pedagogical purposes:

- Potentiometer voltage values must be obtained through polling,
- STM CMSIS functions are to be avoided; except,
- CMSIS may be used for SPI initialization and control.

2 Design solution

At the conclusion of the second lab session we had source code capable of reading an input square wave from an external source and displaying the frequency on the console. Starting development on the LCD interface is a natural first step because the timer control functionality can be abstracted by a function generator. Furthermore, using a function generator eliminates ambiguity around the accuracy of the timer control circuit and allows testing over a wider range of frequencies.

Once the LCD interface is complete we can configure the ADC to measure an input voltage. This allows the LCD to display both changing values: the frequency from the function generator, and potentiometer voltage and equivalent resistance. Next, the timer circuit will be constructed and tested with external control voltages to determine its expected performance range and guarantee its accuracy. Finally, the microcontroller DAC will be configured to act as a control signal for the timer circuit. This development progression allows for each new system to be tested in integration with the previous systems.

The external connections between the project board, microcontroller and timer circuit (on an external breadboard) are summarized in Figure 2.

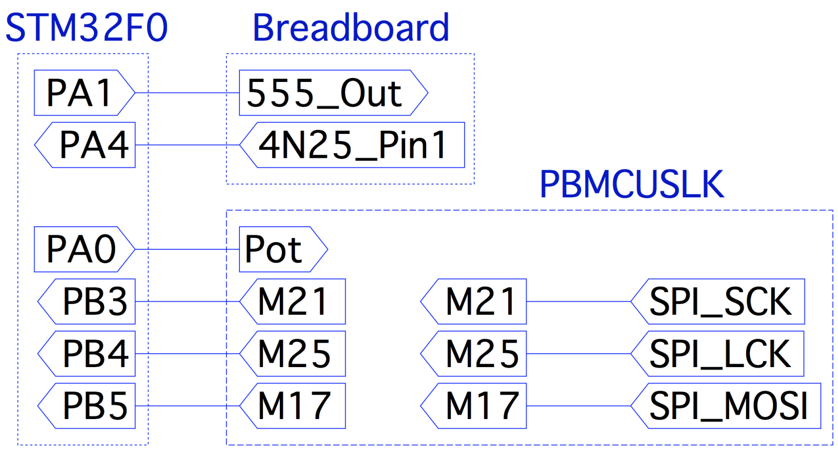


Figure 2: External device connection summary

2.1 LCD

The project board has an eight character by two row LCD for text display. The LCD has a standard HD44780 control circuit that provides fourteen inputs. Only six of the inputs - data lines 4 to 7, the RS pin and EN pin - are controllable by the user. The hard-wired values for the driver are shown in Figure 3.

A down/up transition of the EN pin causes the LCD to accept and execute the current command or data on its data line. The RS pin sets the interpretation of the value on the data pins. If RS is low the data is a *command* which can change the operating state of the LCD. RS high indicates a *data* word, such as a character to display. This value is saved in the LCD's internal DDRAM [2].

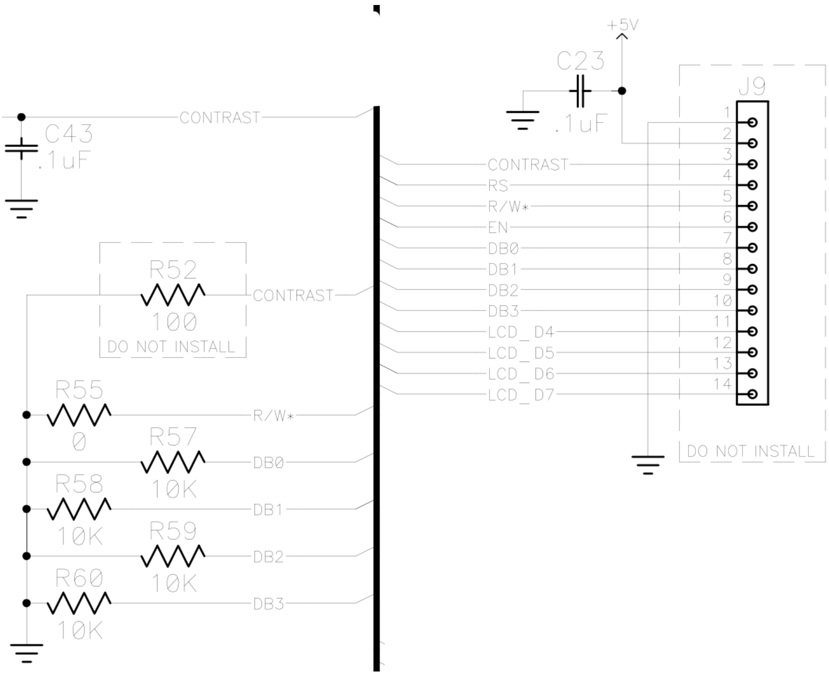


Figure 3: The HD44780 LCD driver only exposes the upper four data lines, RS and EN pins to the user [3, p. 3]

The user-controllable pins from the HD44780 are connected to an HC595 8-bit shift register. This is a common practice because data can be loaded in to the shift register sequentially and then exposed to each line of the controller in parallel. This reduces the number of connections from the board to the LCD driver at the cost of a slower transfer rate. The connections between the project board, shift register and LCD driver are shown in Figure 4.

The MOSI line transfers signals from the microcontroller to the shift register input, A. A down/up transition on the SCK line will move the data in registers QA through QG down one position and move the value on line A into QA. If the LCK line is low the Qx outputs will hold their output values while their internal registers take on new values. When LCK is set high the Qx outputs update to match their internal register values [4].

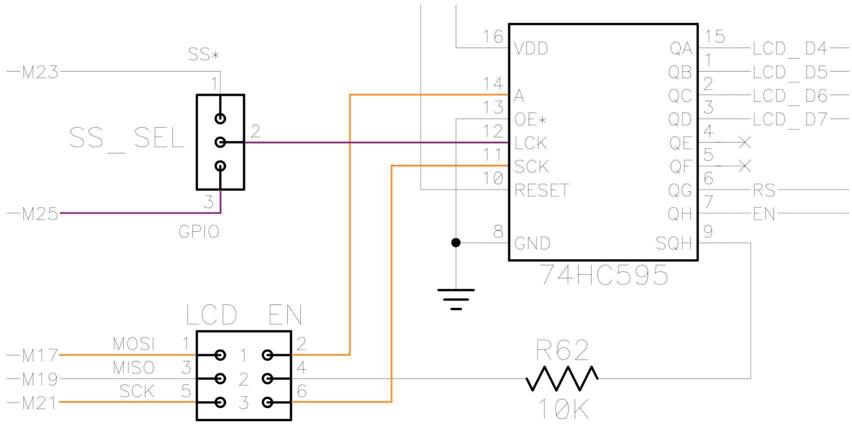


Figure 4: Detail of the SPI communication lines (orange) and shift register control line (purple) [3, p. 3]

All of the LCD control functionality is included in `lcd.c`, a reusable library of LCD interface functions. Software configuration for SPI communication begins in `myGPIOB_Init()` with setting GPIO pins PB3 and PB5 to operate in alternate function mode. The F0 reference manual indicates that PB5 will be used for MOSI communication and PB3 for SCK control [5]. PB4 is configured as a regular digital out and will be used to control LCK, the shift register output update pin.

`stm32f0xx_spi.h` provides library functions for using the built-in SPI capabilities of the microcontroller. The `SPI_InitStruct` has a few notable values: SPI will be unidirectional with the microcontroller acting as master; data size is 8 bits, to match the size of the shift register, and; the baud rate prescaler is 256, the slowest possible transfer rate. Normally, the $R/\overline{W}$ bit of the HD44780 can be set to read. This exposes a flag that indicates if the LCD is ready to receive more data. However, the project board grounds this pin, preventing the LCD from entering read mode (see Figure 3). The ready status of the LCD is unknowable. Since the microcontroller has a much faster clock speed than the HD44780 we assume the bottleneck will be on the receiver's end.

The microcontroller must follow a precise sequence of events to correctly expose data to the shift register and then the LCD controller. Before sending data to the shift register the LCK pin is set low to ensure only valid words are exposed to the LCD controller. Next, the SPI is checked to see if it can accept more data in its transmission queue. When the queue is clear the library function `SPI_SendData8(SPIx, word)` will send an 8-bit word into the shift register by controlling the MOSI and SCK values. SPI flags are checked again to wait for transmission to finish. When finished, LCK is set high and the new 8-bit word is exposed to the LCD controller.

The communication between the shift register and the LCD controller is less straight-forward because the LCD controller has two operating modes: 8-bit mode and 4-bit mode. In 8-bit mode the controller reads values from *all* data lines, including the grounded lines 0 to 3. In 4-bit mode the controller ignores lines 0 to 3 and reads lines 4 to 7 as successive half-words, starting with the upper half. Once the lower half is received the corresponding command is executed.

To send an 8-bit word to an LCD in 4-bit mode the word must be broken into upper and lower half-words. As shown in Figure 4, the data lines correspond to the lower half of the shift register and the RS and EN lines correspond to the upper half. Data is sent to the shift register in the following order:

1. Disable LCD, send high data
2. Enable LCD, send high data
3. Disable LCD, send high data
4. Disable LCD, send low data
5. Enable LCD, send low data
6. Disable LCD, send low data

Steps 1-3 load the upper half-word into the LCD and steps 4-6 send the lower half. Toggling the LCD from Disable $\rightarrow$ Enable is what causes the word to be loaded. Additionally, the RS bit is constant for all steps and indicates whether the data corresponds to a command or data word.

When the LCD boots from power off it loads its default configuration which, unfortunately, sets it in 8-bit mode [2]. By sending the word `0x2` to the LCD we set it to 4-bit mode and can continue with further configuration. (It's not always the case that the LCD is configured from a default state. Consider debugging, where the LCD might already be in 4-bit mode when it tries to execute the initial configuration. Section 4 outlines a procedure for configuring the LCD from an unknown initial state.) Subsequent configuration commands enable two line display, hide the cursor and set the LCD to increment to the next character after one has been written. Finally, the clear screen command is issued.

Some LCD commands, such as clear screen, require over 1.5 ms to complete. This is longer than the minimum SPI baud rate. Sending a command immediately after a slow command like clear screen will result in undefined behavior from the LCD. As such, a 2 ms delays is inserted after the clear command. Timer 3 is configured as a non-blocking timer. When a delay is called the timer is polled until its internal counter expires. This allows other time critical functions, such as input wave edge measurements, to interrupt the delay.

To display characters on the LCD the output character must be mapped to the corresponding entry on the LCD character table [2, p. 17]. Thankfully, 8-bit representation of `char *` values for A-Z,a-Z map directly to the LCD table so explicit casting will give the correct value. Digits 0-9 must be mapped to values starting at `0x30`, where `0x30` → 0, `0x31` → 1, and so on.

The last consideration for designing the LCD interface is its update interval. Attempting to update the LCD too quickly, such as in the main program loop, quickly overflows the LCD's internal buffers and leads to undefined behavior. A third timer, `TIM16` is configured to interrupt every 250 ms. The interrupt handler writes new values for resistance and frequency to the LCD. Since writing to the LCD involves sending data to the shift register six times for each *character*, each with two polling loops, it is the slowest operation on the microcontroller. It's also the least reliant on real-time execution since it's not trying to measure a continuously changing signal. 250 ms is infrequent enough to not hog the majority of CPU cycles but quick enough to seem responsive to a human user.

2.2 ADC

The project board has a 5 kΩ potentiometer (Figure 5) that is connected to an analog input on the microcontroller. As is common with potentiometers, the lower resistance value is significantly above 0 Ω. With minimum resistance, the measured voltage on the potentiometer center lead is ≈ 1.1 V, indicating a minimum resistance of ≈ 1.67 kΩ.

PA0 is one of sixteen available analog inputs with ADC converters on the microcontroller [5, ch. 13]. During initialization, `myGPIOA_Init()` configures PA0 as an analog input. `myADC_Init()` executes the ADC specific initialization procedure outlined in the f0 reference manual [5, ch. 13]. After the reference clock is enabled for the ADC, the `ADC_CR_ADCAL` flag is set in the ADC control register. This flag begins the built-in ADC calibration procedure which maps the input voltage range, 1.1 V to 3.3 V, to the default 12-bit resolution, values 0 to 4095. Hence, when the

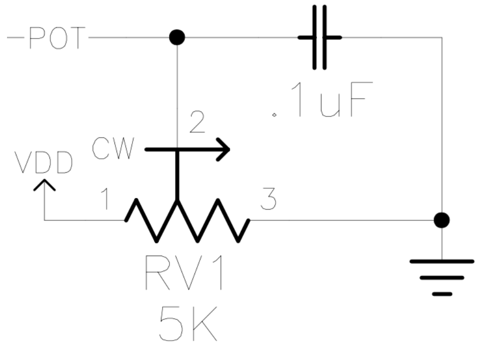


Figure 5: Schematic of the project board potentiometer, $V_{DD} = 3.3\,\mathrm{V}$ [6].

ADC voltage is converted to a resistance it represents a *notional* resistance value rather than the *actually existing* resistance value of the potentiometer.

When the calibration completes the ADC is set to operate in continuous conversion and overrun modes. Continuous conversion will automatically start a new conversion once the ADC value is dereferenced. Overrun mode sets the ADC to overwrite its buffer when full. If this isn't set, a full ADC buffer will cause an overrun error and crash both the ADC *and* DMA controllers without proper handling.

Finally, the enable flag is send to the ADC control register. If the configuration is valid the `ADC_ISR_ADRDY` flag is set, indicating that the ADC is ready to accept conversion requests.

Conversion requests are issued from the main loop of the program. Each loop cycle, the conversion request is sent with `ADC_CR_ADSTART`. The program polls until the end of conversion flag, `ADC_ISR_EOC`, is set. When the conversion is complete, the program resets the end of conversion flag and reads the 12-bit converted value from the ADC data register `ADC1->DR`. This sequence is encapsulated in `getPotADCValue()`.

2.3 555 Timer circuit

The analog output voltage generated by the microcontroller is from a pulse wave modulated (PWM) signal. This signal contains an AC component that would result in unstable operation operation of the 555 timer. To eliminate the AC component, the PWM signal is connected to an optocoupler. The input signal modulates the brightness of an internal LED. The LED brightness controls the gain of an internal NPN BJT. Since the brightness of the LED is not affected by the AC component the result is a DC signal on the BJT. Figure 6 shows the connection of the optocoupler to a 555 timer in astable operation.

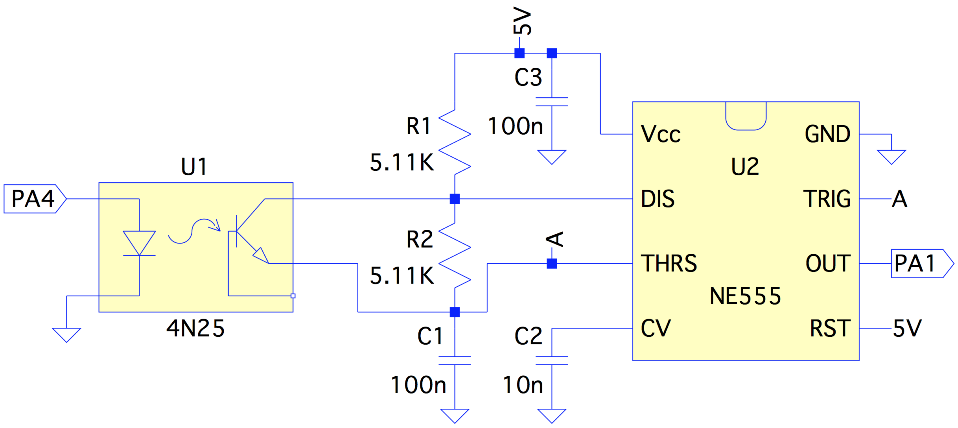


Figure 6: An optocoupler allows a PWM signal to control the output frequency of a 555 timer in astable operation

With this astable configuration, the output signal is a square wave with the frequency determined by [7]:

$$f = \frac{1.44}{(R_1 + 2R_2)\,C_1}. \tag{1}$$

Changing the gain of the NPN BJT in the optocoupler changes the voltages on either side of R_2, thus altering R_2's value in (1).

This circuit was constructed and tested in isolation using the potentiometer voltage as an input (instead of PA4) and an oscilloscope as an output (instead of PA1). Testing confirmed the relationship expected in (1), that $f \propto R_2^{-1}$. See Section 3.2 for more information on the testing procedure.

Testing the timer operation in isolation also reveal a range of input voltages that did not affect the timer output. For input voltages 0 V to 1.07 V there was no change in timer frequency. This is because the optocoupler has two internal diodes: one in the LED and another in the BJT. This is consistent with the forward voltage listed in the 4N35 datasheet: 0.9 V to 1.7 V with a typical value of 1.3 V [8]. As a result of this deadband voltage range the DAC will only output values above 1.07 V so that all values of the potentiometer generate a unique timer voltage.

2.4 DAC

Compared to the LCD controller and ADC, the DAC is simple to initialize and use. PA4 is the default connection for DAC output so it's configured as an analog output in `myGPIOA_Init()`.

The DAC is initialized by enabling its associated clock and setting the enable bit in its control register. The main loop sets the DAC output voltage by writing a 12-bit value to `DAC->DHR12R1`, the DAC data register which is 12-bit and right aligned. Conveniently, this is the same alignment as the ADC value.

The function `applyOptoOffsetToDAC()` shifts the 12-bit value read from the ADC to output voltages 1.07 V to 3.3 V, which is the range of voltages that change the 555 timer frequency.

3 Test procedure and results

3.1 Frequency measurement

3.1.1 Lower limit

The timer used for frequency measurement has a 32-bit counter. Operating at 48 MHz, the time before counter overflow is:

$$\frac{2^{32}-1}{48\,\mathrm{MHz}} \approx 89.48\,\mathrm{s}.$$

This corresponds to a minimum frequency of 0.0112 Hz. This value was confirmed by using a function generator to create a square pulse with a 90 s period and observing the "Timer period overflow" message on the console.

3.1.2 Upper limit

We were able to determine the ISR overhead by sending high frequency signals to the microcontroller and checking the elapsed clock cycles between interrupts. It was not possible to exit and re-enter the ISR in fewer than 61 clock cycles. At 48 MHz this corresponds to 1.27 µs of overhead or 787 kHz. Indeed, input frequencies lower than 787 kHz had more than 61 clock cycles of overhead, confirming that 61 is the minimum overhead.

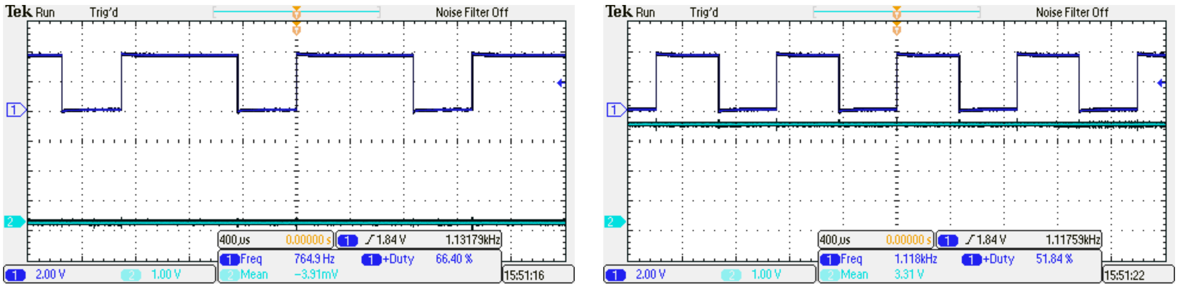
The error in the measured frequency can be expressed as:

$$\frac{f \cdot 61}{48\,\mathrm{MHz}} \times 100\% \tag{2}$$

Using the minimum and maximum 555 timer frequency values from Section 3.2 in (2) gives the range of error on the measured signal as 0.0972 % to 0.142 %.

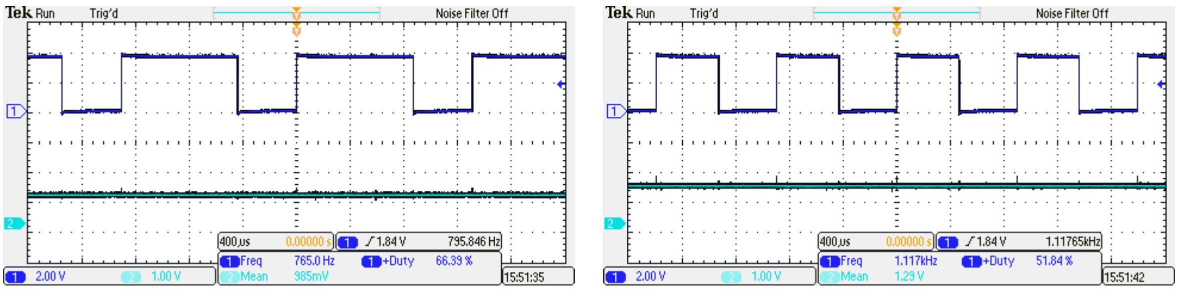
The Nyquist sampling limit, $f_s = \frac{1}{2} f_{max} = 24\,\text{MHz}$, is not relevant because the upper limit from the overhead applies well before the sampling limit.

3.2 Timer output



(a) Potentiometer at $0\,\Omega$ (b) Potentiometer at $5\,\text{k}\Omega$

Figure 7: Timer output (top, PA1) and potentiometer voltage (bottom, PA0)



(a) Potentiometer at $0\,\Omega$ (b) Potentiometer at $5\,\text{k}\Omega$

Figure 8: Timer output (top, PA1) and optocoupler voltage (bottom, PA4)

The values in Figs. 7 and 8 match the output on the LCD exactly. This is expected based on the low error from ISR overhead at this frequency range, as explained in Section 3.1.

Using these frequencies with (1) we can determine the effective value for R_2.

$$R_2 = \frac{1}{2}\left(\frac{1.44}{fC_1} - R_1\right)$$

$$f_{low} = 765\,\text{Hz} \implies R_2 = 6.86\,\text{k}\Omega \quad \text{and} \quad f_{high} = 1118\,\text{Hz} \implies R_2 = 3.890\,\text{kHz}$$

This suggests that the BJT in the 4N35 optocoupler is not in saturation when 1.29 V is supplied by the microcontroller.

4 Discussion

A number of features were added to make development easier and offer a better user experience and interface.

Agnostic LCD configuration routine When the project board powers on it will load the LCD controller into its default state which includes 8-bit operation. However, debugging the microcontroller will cause the LCD initialization code to be re-run on an LCD controller that is not in its default state. This is a particular problem if the LCD initialization assumes that the controller is in 8-bit operating mode because commands will be interpreted differently (perhaps in an undefined manner) in 4-bit mode.

This problem is solved by sending `0x02` as the initial configuration command. Recall that the command is sent in two parts: high then low. In 8-bit mode the high half-word `0x0` is interpreted as `0x00` which has no effect on the controller. The low half-word `0x2` is interpreted as `0x20` which sets the controller to 4-bit mode. If `0x02` is sent in 4-bit mode it is interpreted correctly as `0x02` which maps to the command "send cursor home." In both initial states, 8-bit and 4-bit, the controller is operating in 4-bit mode and the rest of the configuration can continue with that assumption.

Custom unit symbols Instead of displaying the units for resistance as "Oh" and frequency as "Hz" we use Ω and a single column character for "Hz". Displaying Ω was simple because the symbol is pre-loaded in the default 5×7 character table (it has address `0xF4`).

Creating a Hertz symbol was more involved because it's a custom symbol that had to be written into an empty space in the character table. The HD44780 has 8 spaces in RAM (see Figure 9) that can can be written to by the programmer.

After specifying a CGRAM address, the next 8 words received by the HD44780 will be interpreted as rows of the custom symbol where 1 indicates a dark pixel and 0 a light pixel. For example, the first row of the custom Hertz symbol in Fig. 10 is sent as a data command with word `0b10001`.

The custom symbol is displayed by issuing a word command with the address of the symbol on the character table. This is the same procedure for displaying other pre-loaded symbols on the

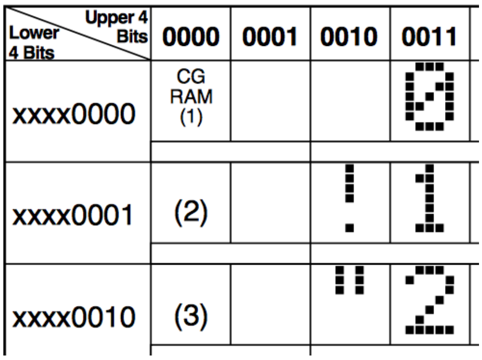


Figure 9: The first 8 addresses (`0x00` → `0x07`) in the HD44780 character table are user-writable CGRAM [2, p. 17]

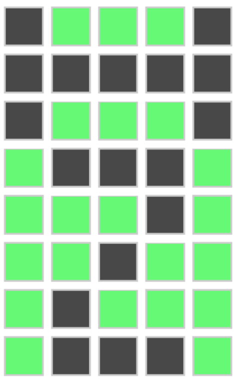


Figure 10: Custom Hertz symbol

table.

Unit auto-ranging Resistance and frequency units will adjust so that four significant figures are always shown. The display precision progression is shown in Table 1. This is implemented

Table 1: Auto-ranging display will always show four significant figures

Measured	Displayed
1.234 Hz	`1.234 Hz`
12.34 Hz	`12.34 Hz`
123.4 Hz	`123.4 Hz`
1234 Hz	`1.234kHz`

with a custom data structure, `metricFloat`, that holds the scaled value and its associated metric prefix. `sprintf` truncates and rounds the scaled value to a fixed precision and creates a `char[]` that is printed with `LCD_SendText()`. Metric values between 1 and 999,999 can be represented with the appropriate metric prefix and precision.

References

[1] A. Jooya, K. Jones, D. Rakhmatov, and B. Sirna, *CENG 355 Laboratory manual*, University of Victoria, 2016.

[2] Hitachi, *Dot matrix liquid crystal display controller/driver*, HD44780, version 0.0, Sep. 1999.

[3] Axiom Manufacturing, *PBMCUSLK AXM-0392*, version D, Mar. 17, 2007.

[4] Texas Instruments, *8-bit shift registers with 3-state output registers*, HC595, [Revised Nov. 2009], Dec. 1982.

[5] ST Microelectronics, *Reference manual, STM32F0x1, STM32F0x2, STM32F0x8 advanced ARM-based 32-bit MCUs*, version 5, Jan. 2014.

[6] Freescale Semiconductor, *MCU Project board student learning kit, Prototyping board with microcontroller interface*, version 1, Jul. 2007.

[7] ST Microelectronics, *General-purpose single bipolar timers*, NE555, version 6, Jan. 2012.

[8] Vishay Semiconductors, *Optocoupler, phototransistor output, with base connection*, 4N35, version 1.2, Jan. 7, 2010.

Appendix A Source code

```c
//
// 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"
#include "assert.h"
#include "lcd.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"


// ----------------------------------------------------------------------------
//                          DEFINES
// ----------------------------------------------------------------------------

#ifndef VERBOSE
#define VERBOSE 0
#endif // VERBOSE

#ifdef USE_FULL_ASSERT
#define assert_param(expr) ((expr) ? (void)0 : assert_failed((uint8_t *)__FILE__,
    __LINE__))
void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif // USE_FULL_ASSERT

#define EPSILON ((float)0.00001)
```

```c
#define float_eq(f1, f2) (((f1 - f2) < EPSILON) || ((f2 - f1) < EPSILON))
#define float_geq(f1, f2) ((f1 > f2) || float_eq(f1, f2))
#define float_leq(f1, f2) ((f1 < f2) || float_eq(f1, f2))

/* Clock prescalers */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
#define ONE_MS_PER_TICK_PRESCALER ((uint16_t)((SystemCoreClock - 1) / 1000))
/* Clock periods */
#define myTIM2_PERIOD ((uint32_t)0xFFFFFFFF) // max value before overflow
#if VERBOSE
#define LCD_UPDATE_PERIOD_MS ((uint32_t)2500)
#else
#define LCD_UPDATE_PERIOD_MS ((uint32_t)250)
#endif
/* Circuit tuning params */
#define RESISTANCE_MAX_VALUE (5000.0) // PBMCUSLK uses a 5k pot
#define ADC_MAX_VALUE ((float)(0xFFF)) // ADC bit resolution (12 by default)
#define DAC_MAX_VALUE ((float)(0xFFF)) // DAC bit resolution (12 by default)
#define DAC_MAX_VOLTAGE (2.95) // measured output from PA4 when
                               // DAC->DHR12R1 = DAC_MAX_VALUE
#define OPTO_DEADBAND_END_VOLTAGE (1.0) // voltage needed to overcome diode
                                        // drops in opto

// ----------------------------------------------------------------------------
//                         PROTOTYPES
// ----------------------------------------------------------------------------

void myGPIOA_Init(void);
void myTIM2_Init(void);
void myTIM16_Init(void);
void myEXTI_Init(void);
void myADC_Init(void);
void myDAC_Init(void);
uint32_t getPotADCValue(void);
uint32_t applyOptoOffsetToDAC(uint32_t rawDAC);

// ----------------------------------------------------------------------------
//                         GLOBAL VARIABLES
// ----------------------------------------------------------------------------

float gbl_sigFreq = 0.0;
float gbl_resistance = 0.0;

// ----------------------------------------------------------------------------
//                         IMPLEMENTATION
// ----------------------------------------------------------------------------

int
main(int argc, char* argv[])
{
    trace_printf("Welcome to the final project.\n");
    if (VERBOSE) trace_printf("System clock: %u Hz\n", SystemCoreClock);

    myGPIOA_Init();  /* Init I/O port PA for input sig, ADC, DAC */
    myADC_Init();    /* Init ADC for continuous measurement */
    myDAC_Init();    /* Init DAC to output timer control voltage */
    myTIM2_Init();   /* Init timer for input sig period measurement */
    myEXTI_Init();   /* Init EXTI to trigger on input sig waveform edge */
    LCD_Init();      /* Init LCD control through SPI & write placeholder */
    myTIM16_Init();  /* Init & start LCD update timer */

    while (1) {
        uint32_t potADCValue = getPotADCValue();
```

```

        // Use the ADC value for DAC but offset it to avoid output voltages that
        // lie in the deadband for the optocoupler (around 0->1V). This will
        // allow all values of the pot to correspond to a change in timer freq.
        uint32_t timerControlDACValue = applyOptoOffsetToDAC(potADCValue);

        // Update the DAC value
        // DHR12R1: "Data Holding Register, 12b, Right aligned, Channel 1"
        DAC->DHR12R1 = timerControlDACValue;

        // Convert to resistance range
        float normalizedPotADC = (((float)potADCValue) / ADC_MAX_VALUE);
        gbl_resistance = normalizedPotADC * RESISTANCE_MAX_VALUE;

        if (VERBOSE) trace_printf("ADC Value: %d\n", potADCValue);
        if (VERBOSE) trace_printf("Resistance: %f\n", gbl_resistance);
    }

    return 0;

}

void myGPIOA_Init()
{
    // Configure:
    //   PA0 --ana-> Analog in to ADC for resistance
    //   PA1 --in--> Read 555 timer edge transitions
    //   PA4 --ana-> Analog out from DAC for timer control voltage

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

    /* Configure PA0 */
    GPIOA->MODER &= ~(GPIO_MODER_MODER0);  /* Analog input */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);  /* No pull up/down */

    /* Configure PA1 */
    GPIOA->MODER &= ~(GPIO_MODER_MODER1);  /* Input */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR1);  /* No pull up/down */

    /* Configure PA4 */
    GPIOA->MODER &= ~(GPIO_MODER_MODER4);  /* Analog output */
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR4);  /* No pull up/down */
}

void myTIM2_Init()
{
    /* Enable clock for TIM2 peripheral */
    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 myTIM16_Init()
{
    /* Enable clock for TIM16 peripheral */
    RCC->APB2ENR |= RCC_APB2ENR_TIM16EN;

    /* Configure TIM16: buffer auto-reload, count up, stop on overflow,
     * enable update events, interrupt on overflow only */
    TIM16->CR1 = ((uint16_t) 0x008C);

    /* Set clock prescaler value */
    TIM16->PSC = ONE_MS_PER_TICK_PRESCALER;
    /* Set auto-reloaded delay */
    TIM16->ARR = LCD_UPDATE_PERIOD_MS;

    /* Update timer registers */
    TIM16->EGR = ((uint16_t) 0x0001);

    /* Assign TIM16 interrupt priority = 1 in NVIC */
    /* Will be interrupted by P0 tasks, like edge measurements */
    NVIC_SetPriority(TIM16_IRQn, 1);

    /* Enable TIM16 interrupts in NVIC */
    NVIC_EnableIRQ(TIM16_IRQn);

    /* Enable update interrupt generation */
    TIM16->DIER |= TIM_DIER_UIE;

    /* Activate timer! */
    TIM16->CR1 |= TIM_CR1_CEN;
}

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;

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

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

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

void myADC_Init(){
    /* Enable clock for ADC */
    RCC->APB2ENR |= RCC_APB2ENR_ADCEN;

    /* Tell ADC to begin self-calibration and wait for it to finish */
```

```c
    if (VERBOSE) trace_printf("Start ADC calibration...\n");
    ADC1->CR = ADC_CR_ADCAL;
    while (ADC1->CR == ADC_CR_ADCAL) {};
    if (VERBOSE) trace_printf("ADC calibration finished!\n\n");

    /* ADC Configuration:
     *    - Continuous conversion
     *    - Overrun mode
     */
    ADC1->CFGR1 |= (ADC_CFGR1_CONT | ADC_CFGR1_OVRMOD);

    /* Select channel, PA0 needs Channel 0 */
    ADC1->CHSELR = ADC_CHSELR_CHSEL0;

    /* Enable ADC and wait for it to have ready status */
    if (VERBOSE) trace_printf("Start ADC enable...\n");
    ADC1->CR |= ADC_CR_ADEN;
    while (!(ADC1->ISR & ADC_ISR_ADRDY)) {};
    if (VERBOSE) trace_printf("ADC enable finished!\n\n");
}

void myDAC_Init() {
    /* Enable clock for DAC */
    RCC->APB1ENR |= RCC_APB1ENR_DACEN;

    /* Enable DAC */
    DAC->CR |= DAC_CR_EN1;
}

/* 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*** 555 Timer Input Period Overflow ***\n");

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

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

void TIM16_IRQHandler()
{
    /* Check if update interrupt flag is indeed set */
    if ((TIM16->SR & TIM_SR_UIF) != 0) {
        if (VERBOSE) trace_printf("\nUpdating LCD with freq: %f Hz\n", gbl_sigFreq);
        LCD_UpdateFreq(gbl_sigFreq);

        if (VERBOSE) trace_printf("Updating LCD with resistance: $f Ohm\n\n");
        LCD_UpdateResistance(gbl_resistance);

        /* Clear update interrupt flag */
        TIM16->SR &= ~(TIM_SR_UIF);

        /* Restart stopped timer */
        TIM16->CR1 |= TIM_CR1_CEN;
    }
}
```

```
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void EXTI0_1_IRQHandler()
{
    /* Check if EXTI1 interrupt pending flag is indeed set */
    if ((EXTI->PR & EXTI_PR_PR1) != 0)
    {
        // timer will be disabled for first edge and enabled on second
        uint16_t isTimerEnabled = (TIM2->CR1 & TIM_CR1_CEN);

        if (isTimerEnabled) {
            // stop timer and get count
            TIM2->CR1 &= ~(TIM_CR1_CEN);
            uint32_t count = TIM2->CNT;
            gbl_sigFreq = ((float)SystemCoreClock) / count;
            float sigPeriod = 1.0 / gbl_sigFreq;

            if (VERBOSE)trace_printf("Signal Freq:   %f Hz\n", gbl_sigFreq);
            if (VERBOSE)trace_printf("Signal Period: %f s\n\n", sigPeriod);

        } else {
            // reset & start timer
            TIM2->CNT = (uint32_t)0x0;
            TIM2->CR1 |= TIM_CR1_CEN;
        }

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

uint32_t getPotADCValue(){
    // Start ADC conversion
    ADC1->CR |= ADC_CR_ADSTART;

    // Wait for End Of Conversion flag to be set
    while (!(ADC1->ISR & ADC_ISR_EOC)) {
        /* loop until conversion is complete */
    };

    // Reset End Of Conversion flag
    ADC1->ISR &= ~(ADC_ISR_EOC);

    // Apply the data mask to the data register
    return ((ADC1->DR) & ADC_DR_DATA);
}

uint32_t applyOptoOffsetToDAC(uint32_t rawDAC){
    // map the DAC value -> voltage range (opto deadband to max output)
    float normalizedDAC = rawDAC / DAC_MAX_VALUE;
    float outputVoltageRange = DAC_MAX_VOLTAGE - OPTO_DEADBAND_END_VOLTAGE;
    float outputVoltage = (normalizedDAC * outputVoltageRange) +
        OPTO_DEADBAND_END_VOLTAGE;

    // check that output mapping function produced valid output
    assert_param(float_geq(outputVoltage, 0.0));
    assert_param(float_geq(outputVoltage, OPTO_DEADBAND_END_VOLTAGE));
    assert_param(float_leq(outputVoltage, DAC_MAX_VOLTAGE));

    if (VERBOSE) trace_printf("\nOutput voltage: %f", outputVoltage);

    // convert the voltage value back to a DAC level
    float normalizedOutputVoltage = outputVoltage / DAC_MAX_VOLTAGE;
    float outputDACValue = normalizedOutputVoltage * DAC_MAX_VALUE;
```

```c

    return ((uint32_t)outputDACValue);
}

#pragma GCC diagnostic pop

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

Listing 1: main.c

```c
#ifndef __LCD_H
#define __LCD_H

// ----------------------------------------------------------------------------
//                          USER CONFIGURATION
// ----------------------------------------------------------------------------

// Pin configuration, must be on GPIOB
#define LCD_LCK_PIN (GPIO_Pin_4)

// Maps info to LCD rows [1,2]
#define LCD_FREQ_ROW (1)
#define LCD_RESISTANCE_ROW (2)
// digits that can be displayed on LCD, not including decimal
#define LCD_MAX_DIGIT_COLS (4)

// ----------------------------------------------------------------------------
//                          PROTOTYPES
// ----------------------------------------------------------------------------

// Initializes the LCD from a fresh power-on state on the PBMCUSLK.
// It's a 2x8 character LCD with its input buffered by a shift register.
// The LCD is set to auto-increment and use 4bit data mode.
// Configures:
//    - GPIOB
//    - SPI
//    - TIM3
void LCD_Init(void);

// Turns on GPIOB so it can be used by SPI.
//  PB3 -> MOSI (Master Out Slave In)
//  PB4 -> LCK  (Load clock)
//  PB5 -> SCK  (Shift clock)
void myGPIOB_Init(void);

// Use SPI in master mode, 8b data out.
void mySPI_Init(void);

// Configures TIM3 for use as a delay timer
void DELAY_Init(void);

// Send data to the shift register via SPI
void HC595_Write(uint8_t word);

// Write 8b word to an LCD configured for 4b input using shift register
// @param type: Can be LCD_COMMAND or LCD_DATA
void LCD_SendWord(uint8_t type, uint8_t word);

// Convenience variant of LCD_SendWord that accepts ASCII words
// @param character: A single character e.g. "H"
void LCD_SendASCIIChar(const char* character);

// Convenience variant of LCD_SendWord that prints single digits 0:9
```

```c
// @param digit: An int, 0:9
void LCD_SendInteger(uint8_t digit);

// Prints all values in the text to the LCD
// @param text: a null terminated string
void LCD_SendText(char* text);

// Prints all values in the text to the LCD
// @param text: a null terminated string
void LCD_SendText(char* text);

// Clears the LCD and injects a 2ms delay to allow the LCD to finish the operation
void LCD_Clear(void);

// Execute empty loop for specified time in ms
void DELAY_Set(uint32_t milliseconds);

// True if SPI can accept data to send
// @return: False / True as 0 / 1
uint8_t SPI_ReadyToSend(void);

// True if the SPI is not in use
// @return: False / True as 0 / 1
uint8_t SPI_DoneSending(void);

// Positions the LCD cursor at row x [1,2] col y [1:8]
void LCD_MoveCursor(uint8_t row, uint8_t col);

// Write a new frequency value to the LCD
void LCD_UpdateFreq(float freq);

// Write a new resistance to the LCD;
void LCD_UpdateResistance(float resistance);

// Write a new value to the specified row, leaving the unit symbol untouched
void LCD_UpdateRow(uint8_t row, float val);

// Writes custom 5x8 symbols to user-writable CGRAM slots
void LCD_LoadCustomSymbols(void);

#endif /* __LCD_H */
```

Listing 2: lcd.h

```c
#include <stdio.h>
#include "stm32f0xx_gpio.h"
#include "stm32f0xx_rcc.h"
#include "stm32f0xx_spi.h"
#include "cmsis/cmsis_device.h"
#include "diag/Trace.h"
#include "assert.h"
#include "lcd.h"

// ----------------------------------------------------------------------------
//                          STRUCTS
// ----------------------------------------------------------------------------
typedef struct metricFloat {
    float value;
    uint32_t multiplier;
    char* prefix;
} metricFloat;

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

```c
//                          DEFINES
// ----------------------------------------------------------------------------

#ifndef VERBOSE
#define VERBOSE 0
#endif // VERBOSE

#ifdef USE_FULL_ASSERT
#define assert_param(expr) ((expr) ? (void)0 : assert_failed((uint8_t *)__FILE__, __LINE__))
void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif // USE_FULL_ASSERT

// Maps to EN and RS bits for LCD
#define LCD_ENABLE (0x80)
#define LCD_DISABLE (0x0)
#define LCD_COMMAND (0x0)
#define LCD_DATA (0x40)

// LCD Config commands
#define LCD_CURSOR_ON (0x2)
#define LCD_MOVE_CURSOR_CMD (0x80)
#define LCD_CLEAR_CMD (0x1)
#define LCD_ROW_MIN (1)
#define LCD_ROW_MAX (2)
#define LCD_COL_MIN (1)
#define LCD_COL_MAX (8)
#define LCD_FIRST_ROW_OFFSET (0x0)
#define LCD_SECOND_ROW_OFFSET (0x40)

// Positions on the LCD character table
#define SYMBOL_OMEGA (0xF4)
#define SYMBOL_HZ (0x00)

#define DELAY_PRESCALER_1KHZ (47999) /* 48 MHz / (47999 + 1) = 1 kHz */
#define DELAY_PERIOD_DEFAULT (100) /* 100 ms */

// ----------------------------------------------------------------------------
//                          IMPLEMENTATION
// ----------------------------------------------------------------------------

void LCD_Init(void){
    // Configure GPIOB to control LCD via SPI
    myGPIOB_Init();
    mySPI_Init();

    // We'll need the delay timer to wait for some operations (clear) to
    // complete on the LCD. The sane way to do this is to read the LCD status
    // but the PBMCUSLK hard-wires the LCD to write mode :(
    DELAY_Init();

    // Configure the internal LCD controls
    //
    // PBMCUSLK has a shift register connected to the LCD like:
    //
    //     ______________HC 595______________
    //     | Q7  Q6  Q5  Q4  Q3  Q2  Q1  Q0 |  0   0   0   0
    //     |=|===|===|===|===|===|===|===|==         |         |         |         |
    //     | EN  RS  NC  NC  D7  D6  D5  D4    D3  D2  D1  D0 |
```

```
    //                                              H D 4 4 7 8 0  LCD C o n t r o l l e r
    //
    // On initialization, the LCD may be in 4b or 8b mode. We need to change it
    // to 4b mode to use LCD_SendWord, which assumes 4b mode and sends high and
    // low half-words.
    //
    // If the LCD is in 8b mode:
    //   1. Send 0x0, interpreted as command 0x00 which has no effect
    //   2. Send 0x2, interpreted as command 0x20 which changes to 4b mode
    //
    // If the LCD is in 4b mode:
    //   1. Send 0x0, interpreted as high half-word
    //   2. Send 0x2, interpreted as lower half-word, command 0x02 is "send
    //      cursor home"
    // Since we entered in 4b mode we can continue with the rest of the
    // initialization.
    // Note, "cursor home" has a 1.52 ms execution time so we add a 2ms delay to
    // ensure the LCD is ready to receive the rest of the config.
    LCD_SendWord(LCD_COMMAND, 0x2);
    DELAY_Set(2);
    // Now we're in 4b mode we can do the rest of the LCD config
    // https://en.wikipedia.org/wiki/Hitachi_HD44780_LCD_controller#Instruction_set
    // 1. set 4b, 2 line, 5x7 font
    // 2. display on, blink off, cursor?
    // 3. cursor auto-increment, no display shift
    uint8_t showCursorState = VERBOSE ? LCD_CURSOR_ON : 0x0;
    LCD_SendWord(LCD_COMMAND, 0x28);
    LCD_SendWord(LCD_COMMAND, 0x0C | showCursorState);
    LCD_SendWord(LCD_COMMAND, 0x06);
    LCD_Clear();

    LCD_LoadCustomSymbols();

    // Write the initial units and resistance / freq placeholders manually
    //   "  ???  H"
    //   "  ???    "
    LCD_MoveCursor(LCD_FREQ_ROW, 3);
    LCD_SendText("???");
    LCD_MoveCursor(LCD_FREQ_ROW, 8);
    LCD_SendWord(LCD_DATA, SYMBOL_HZ);

    LCD_MoveCursor(LCD_RESISTANCE_ROW, 3);
    LCD_SendText("???");
    LCD_MoveCursor(LCD_RESISTANCE_ROW, 8);
    LCD_SendWord(LCD_DATA, SYMBOL_OMEGA);
}

void myGPIOB_Init(void){
    // Turn on the GPIOB clock
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE);

    // PB3 --AF0-> SPI MOSI
    // PB5 --AF0-> SPI SCK
    GPIO_InitTypeDef GPIO_InitStruct;
    GPIO_InitStruct.GPIO_Pin   = GPIO_Pin_3 | GPIO_Pin_5;
    GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOB, &GPIO_InitStruct);

    // Configure the LCK pin for "manual" control in HC595_Write
```

```
    GPIO_InitStruct.GPIO_Pin   = LCD_LCK_PIN;
    GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOB, &GPIO_InitStruct);
}

void LCD_LoadCustomSymbols(void){
  char hertzPixels[8] = {
    0b10001,
    0b11111,
    0b10001,
    0b01110,
    0b00010,
    0b00100,
    0b01000,
    0b01110
  };
  // Sets CG RAM address
  LCD_SendWord(LCD_COMMAND, (0x40 + SYMBOL_HZ));
  // all subsequent data is written and auto incremented to CG RAM address
  //each byte (represeting 1 line of the custom char) is written to byte locations
  //64 to 71 of the CG RAM (Custom Generator Ram)
  for(int a = 0; a < 8; a++){
    LCD_SendWord(LCD_DATA, hertzPixels[a]);
  }
}

void mySPI_Init(void){
    // Turn on the SPI clock
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

    SPI_InitTypeDef SPI_InitStruct;
    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 HC595_Write(uint8_t word){
    // We only want to expose the register values to the LCD once the new word
    // has loaded completely. We do this by toggling LCK, which controls whether
    // or not the register output tracks its current contents

    // Don't update the register output; LCK = 0
    GPIOB->BRR = LCD_LCK_PIN;

    // Poll SPI until its ready to receive more data
    while (!SPI_ReadyToSend()) {
        /* polling... */
    };

    SPI_SendData8(SPI1, word);

```

```
    // Poll SPI to determine when it's finished transmitting
    while (!SPI_DoneSending()) {
        /* polling... */
    };

    // Update the output; LCK = 1
    GPIOB->BSRR = LCD_LCK_PIN;
}

uint8_t SPI_ReadyToSend(void){
    // SPI can accept more data into its TX queue if TXE = 1 OR BSY = 0
    return ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == SET)
            || (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_BSY) == RESET));
}

uint8_t SPI_DoneSending(void){
    // SPI is done sending when BSY = 0
    return (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_BSY) == RESET);
}

void LCD_SendWord(uint8_t type, uint8_t word){
    // The high half of the register output is always reserved for EN and RS.
    // To send an 8b target word we have to send it as two sequential 4b
    // half-words, with the target half-words in the lower half of the word.

    uint8_t high = ((word & 0xF0) >> 4);
    uint8_t low = word & 0x0F;

    HC595_Write(LCD_DISABLE | type | high);
    HC595_Write(LCD_ENABLE  | type | high);
    HC595_Write(LCD_DISABLE | type | high);

    HC595_Write(LCD_DISABLE | type | low);
    HC595_Write(LCD_ENABLE  | type | low);
    HC595_Write(LCD_DISABLE | type | low);
}

void LCD_SendASCIIChar(const char* character){
    LCD_SendWord(LCD_DATA, (uint8_t)(*character));
}

void LCD_SendText(char* text){
    const char* ch = text;
    while(*ch){
        LCD_SendASCIIChar(ch++);
    }
}

void LCD_SendInteger(uint8_t digit){
    // Enforce range of 0:9
    // No check needed for >= 0 since digit is unsigned
    assert_param(digit <= 9);

    // Digits on the ASCII table are mapped like:
    //   0x30 -> 0
    //   0x31 -> 1
    //   ...
    //   0x39 -> 9
    uint8_t asciiDigit = 0x30 + digit;

    LCD_SendWord(LCD_DATA, asciiDigit);
}

```

```c
void LCD_MoveCursor(uint8_t row, uint8_t col){
    // Check for valid row selection and assign offset
    assert_param(row >= LCD_ROW_MIN);
    assert_param(row <= LCD_ROW_MAX);

    uint8_t rowOffset = 0x0;
    switch (row) {
        case 1:
            rowOffset = LCD_FIRST_ROW_OFFSET;
            break;
        case 2:
            rowOffset = LCD_SECOND_ROW_OFFSET;
            break;
        default:
            break;
    }

    // Similarly, constrain allowed column input values and then shift for
    // 0-indexing on the LCD
    assert_param(col >= LCD_COL_MIN);
    assert_param(col <= LCD_COL_MAX);

    uint8_t colOffset = col - 1;

    uint8_t moveCursorCommand = LCD_MOVE_CURSOR_CMD | rowOffset | colOffset;

    LCD_SendWord(LCD_COMMAND, moveCursorCommand);
}

void LCD_Clear(void){
    LCD_SendWord(LCD_COMMAND, LCD_CLEAR_CMD);
    DELAY_Set(2);
}

void DELAY_Init()
{
    // Enable timer clock
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;

    // Set timer to:
    //   Auto reload buffer
    //   Stop on overflow
    //   Enable update events
    //   Interrupt on overflow only
    TIM3->CR1 = ((uint16_t) 0x8C);

    TIM3->PSC = DELAY_PRESCALER_1KHZ;
    TIM3->ARR = DELAY_PERIOD_DEFAULT;

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

void DELAY_Set(uint32_t milliseconds){
    // Clear timer
    TIM3->CNT |= 0x0;

    // Set timeout
    TIM3->ARR = milliseconds;

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

```

```
    // Start the timer
    TIM3->CR1 |= TIM_CR1_CEN;

    // Loop until interrupt flag is set by timer expiry
    while (!(TIM3->SR & TIM_SR_UIF)) {
        /* polling... */
    }

    // Stop the timer
    TIM3 -> CR1 &= ~(TIM_CR1_CEN);

    // Reset the interrupt flag
    TIM3->SR &= ~(TIM_SR_UIF);
}

void LCD_UpdateFreq(float freq){
    if (freq < 1) {
        LCD_MoveCursor(LCD_FREQ_ROW, 1);
        LCD_SendText("<1.000 ");
    } else if (freq > 999999999.9) {
        LCD_MoveCursor(LCD_FREQ_ROW, 1);
        LCD_SendText(">999.9M");
    } else {
        LCD_UpdateRow(LCD_FREQ_ROW, freq);
    }
}

void LCD_UpdateResistance(float resistance){
    if (resistance < 1) {
        LCD_MoveCursor(LCD_RESISTANCE_ROW, 1);
        LCD_SendText("<1.000 ");
    } else {
        LCD_UpdateRow(LCD_RESISTANCE_ROW, resistance);
    }
}

void LCD_UpdateRow(uint8_t row, float val){
    // Determine the metric prefix to use
    metricFloat metricVal;

    // We assume val >= 1.0
    if (val < 999.9) {
        metricVal.prefix = " ";
        metricVal.multiplier = 1;
    } else if (val < 999999.9) {
        metricVal.prefix = "k";
        metricVal.multiplier = 1000;
    } else if (val < 999999999.9) {
        metricVal.prefix = "M";
        metricVal.multiplier = 1000 * 1000;
    }

    // Scale val for metric representation
    metricVal.value = val / metricVal.multiplier;

    // Determine number of decimal places to display by removing the part of
    // the number greater than 0 from the total number of available columns.
    // The remainder will go to the decimal;
    // TODO: Dynamically determine precision range from LCD_MAX_DIGIT_COLS
    char* floatPrecisionFormat;
    if (metricVal.value < 10) {
        floatPrecisionFormat = "%.3f"; // 1.234
    } else if (metricVal.value < 100) {
```

```
        floatPrecisionFormat = "%.2f"; // 12.34
    } else {
        floatPrecisionFormat = "%.1f"; // 123.4
    }

    // Convert our float to a printable string
    // Allocate size for all digits + decimal
    char printableVal[LCD_MAX_DIGIT_COLS + 1];
    sprintf(printableVal, floatPrecisionFormat, metricVal.value);

    LCD_MoveCursor(row, 1);
    LCD_SendText(" "); // overwrite over/under range symbol
    LCD_SendText(printableVal);
    LCD_SendText(metricVal.prefix);
}
```

Listing 3: lcd.c