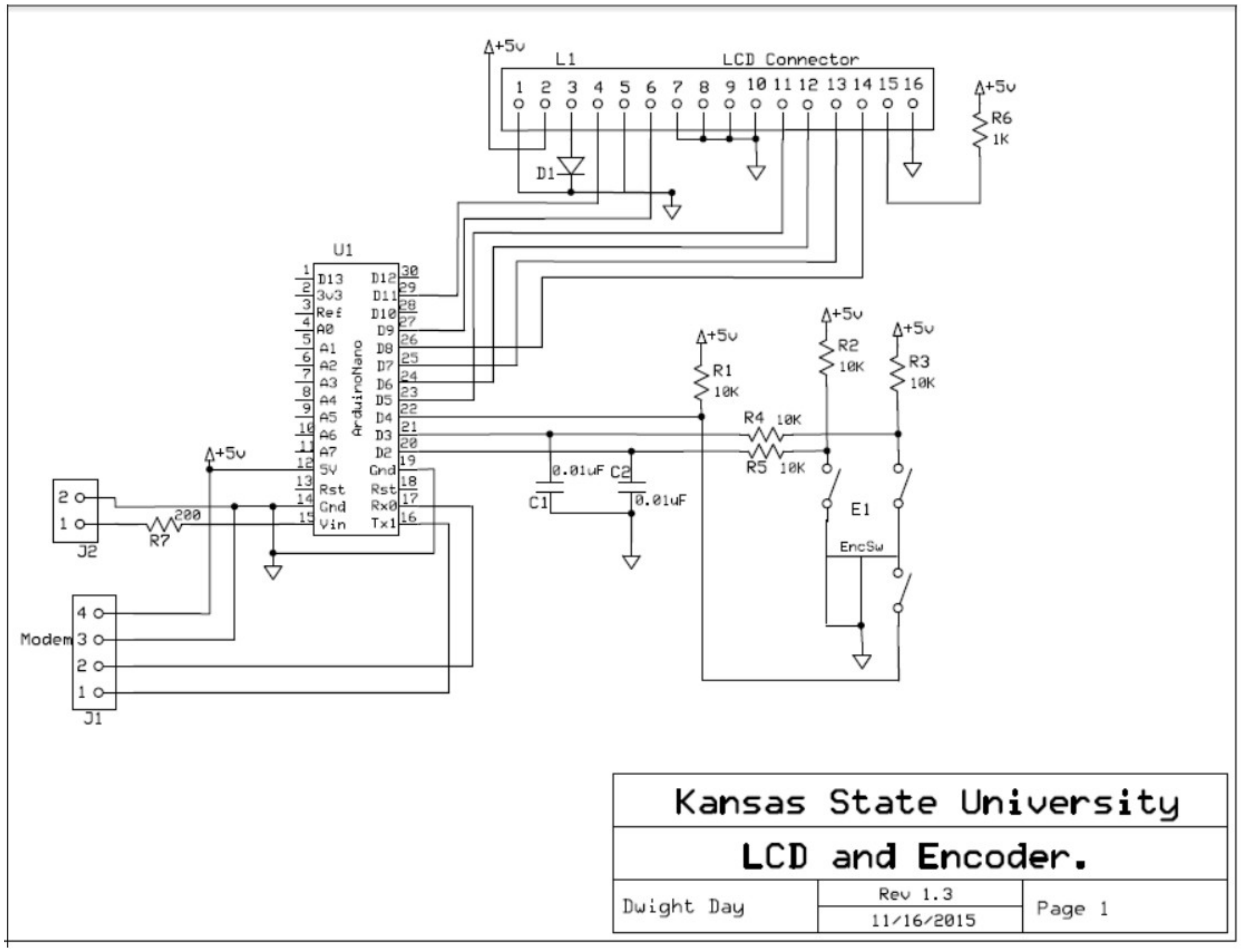
This system will have a small Liquid Crystal Display (LCD) as a means of providing information to the user. Fortunately for us the Arduino environment provides "drivers" for LCD displays. Thus we need to work at setting up the LiquidCrystal software.

The first question is how is the lcd connected to the Arduino, for this we look at the schematic, included here.



And from this and the LCD documentation we can see that LCD4 (Register Select) is connected to D11, LCD6 (LCD Enable) is connected to D9, and then LCD14 to LCD11 (DB7 to DB4) are connected to D8 to D4. Note that only four data pins are connected from the Arduino to the display. This is done so as to minimize the pins used.

Now when setting up LcdDriver, the number of arguments tells the software what type of interface you are using and which pins connect to which signals. The following is the description of how to set up the lcd driver, given in the Arduino documentation.

`LiquidCrystal()`

Description

Creates a variable of type LiquidCrystal. The display can be controlled using 4 or 8 data lines. If the former, omit the pin numbers for d0 to d3 and leave those lines unconnected. The RW pin can be tied to ground instead of connected to a pin on the Arduino; if so, omit it from this function's parameters.

Syntax

LiquidCrystal(rs, enable, d4, d5, d6, d7)

LiquidCrystal(rs, rw, enable, d4, d5, d6, d7)

LiquidCrystal(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7)

LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)

Parameters

rs: the number of the Arduino pin that is connected to the RS pin on the LCD

rw: the number of the Arduino pin that is connected to the RW pin on the LCD (optional)

enable: the number of the Arduino pin that is connected to the enable pin on the LCD

d0, d1, d2, d3, d4, d5, d6, d7: the numbers of the Arduino pins that are connected to the corresponding data pins on the LCD. d0, d1, d2, and d3 are optional; if omitted, the LCD will be controlled using only the four data lines (d4, d5, d6, d7).

Thus when setting up the software to interface to the LCD we need to create the driver object at the top of our file, making it a global variable and it would look like this

```
#include <LiquidCrystal.h>
LiquidCrystal LcdDriver(11, 9, 5, 6, 7, 8);
```

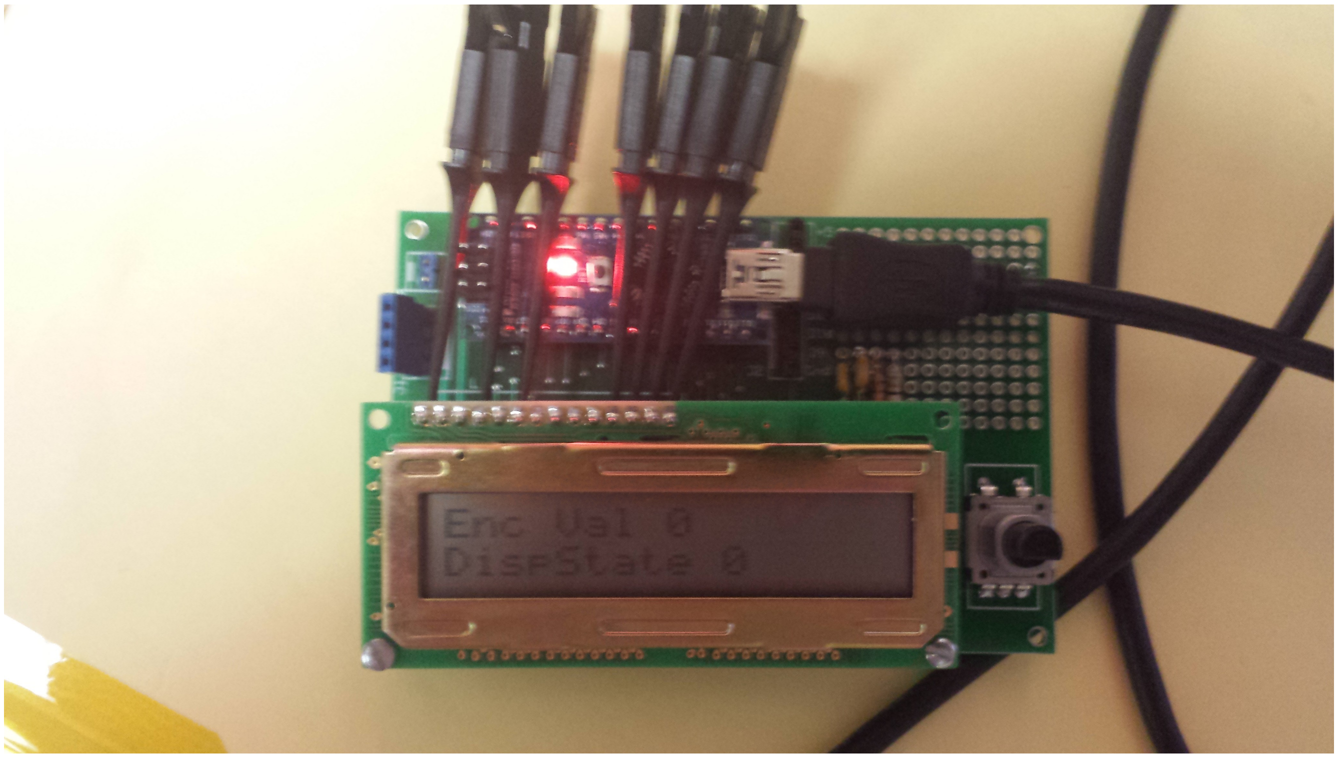
Once we have the driver established we need to tell it the size of display, so in the setup() function, we need to tell it the size of the display (16 columns and 2 rows) and start the display.

```
LcdDriver.begin(16, 2);
LcdDriver.setCursor(0, 0);
```

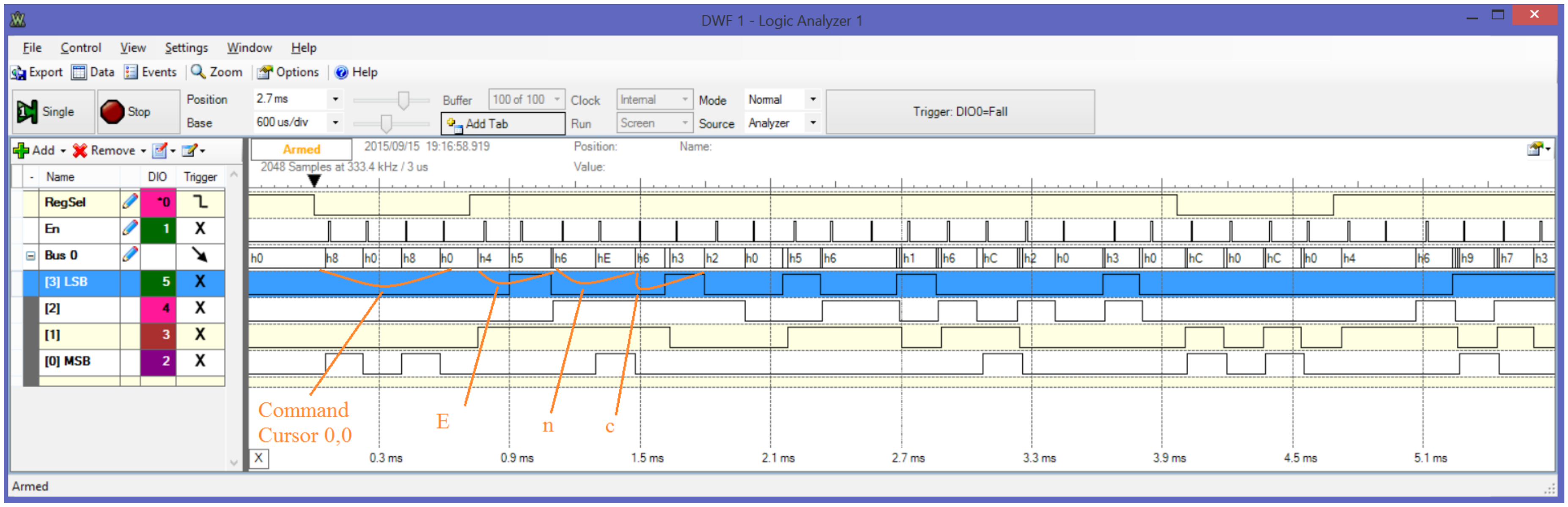
Then in the main loop, the following message was written to the

```
LcdDriver.setCursor(0,0);
LcdDriver.print( "Enc Val ");
LcdDriver.print( EncoderValue );
```

Now the test program I wrote created the display shown in the image. In this image we can see the logic analyzer is attached to the RS, E and upper data pins. Also note that the display has "Enc Val" as the first characters on the display.



The LA was used to capture the data lines as the display was being written to and the captured data is in the screen capture on the next page. In it we can see the commands being written at the start of a display cycle, where the cursor is set to location 0,0. Then the various characters come across the bus, upper four bits first and the lower four bits second, starting with 'E' (0x45).



DWF 1 - Logic Analyzer 1

Position: 2.7 ms
Base: 600 us/div
Buffer: 100 of 100
Clock: Internal
Run: Screen
Mode: Normal
Source: Analyzer
Trigger: DIO0=Fall

Armed 2015/09/15 19:16:58.919
2048 Samples at 333.4 kHz / 3 us

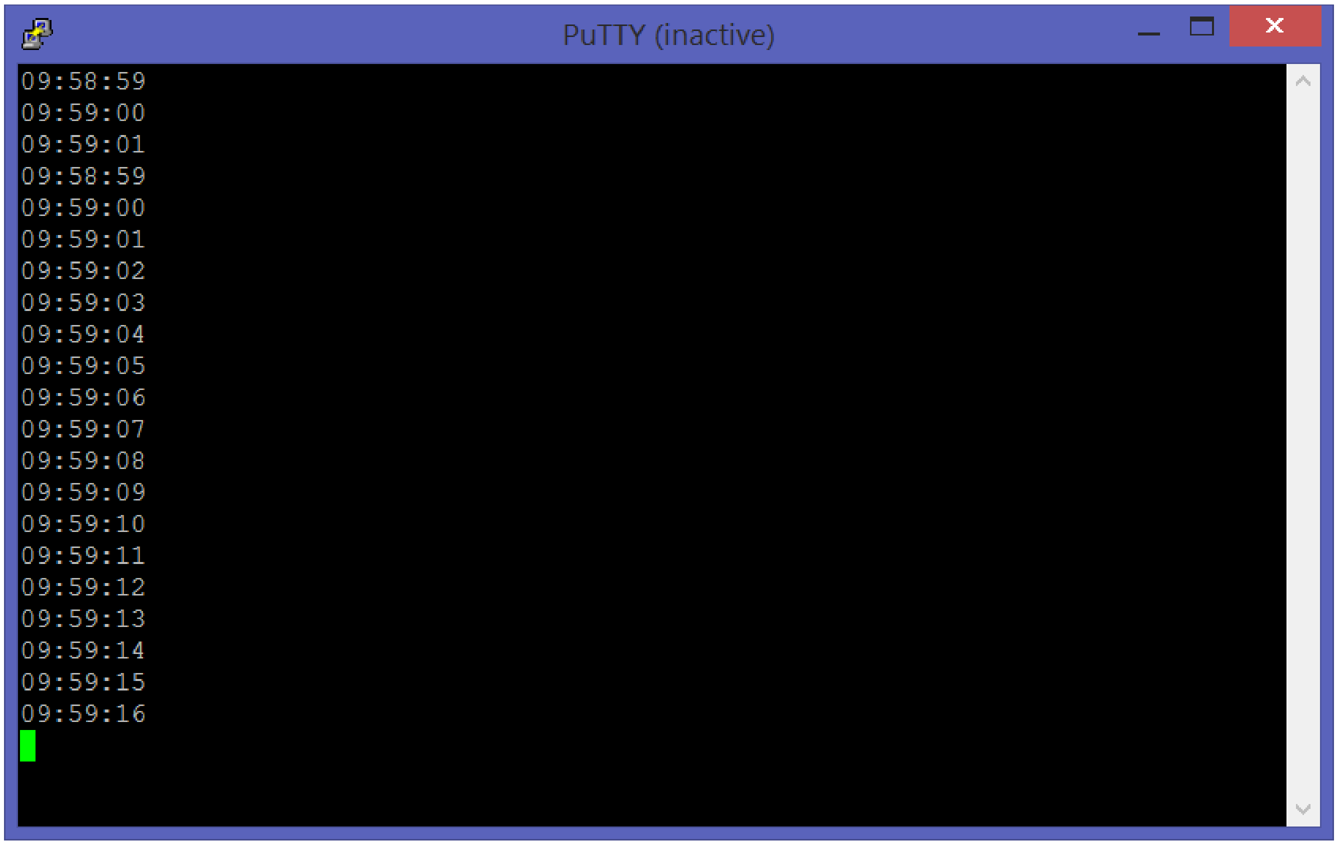
Name	DIO	Trigger
RegSel	0	⌐
En	1	X
Bus 0		↘
[3] LSB	5	X
[2]	4	X
[1]	3	X
[0] MSB	2	X

```
h0 h8 h0 h8 h0 h4 h5 h6 hE h6 h3 h2 h0 h5 h6 h1 h6 hC h2 h0 h3 h0 hC h0 hC h0 h4 h6 h9 h7 h3
```

Armed

```
// Timer for one second interval
unsigned long SecondsTimer;
#define INTERVAL 1000
// Variable used as clock settings.
int Hours, Minutes, Seconds;

// This function is to be called every second
// to update the clock represented by the
// global variables Hours, Minutes, Seconds
void UpdateClock()
{
	// Check if Seconds not at wrap point.
	if (Seconds < 59)
	{
		Seconds++; // Move seconds ahead.
	}
	else
	{
		Seconds = 0; // Reset Seconds
		// and check Minutes for wrap.
		if (Minutes < 59)
		{
			Minutes++; // Move seconds ahead.
		}
		else
		{
			Minutes = 0; // Reset Minutes
			// check Hours for wrap
			if (Hours < 23)
			{
				Hours++;// Move Hours ahead.
			}
			else
			{
				Hours = 0;// Reset Hours
			}// End of Hours test.
		} // End of Minutes test
	} // End of Seconds test
} // end of UpdateClock()

void SendClock()
{
	// Check if leading zero needs to be sent
	if (Hours < 10)
	{
		Serial.print("0");
	}
	Serial.print(Hours);  // Then send hours
	Serial.print(":");    // And separator
	// Check for leading zero on Minutes.
	if (Minutes < 10)
	{
		Serial.print("0");
	}
	Serial.print(Minutes);  // Then send Minutes
	Serial.print(":");    // And separator
	// Check for leading zero needed for Seconds.
	if (Seconds < 10)
	{
		Serial.print("0");
```

```
        }
        Serial.println(Seconds);  // Then send Seconds
        // with new line
} // End of SendClock()

// setup code, run once:
void setup()
{
        Serial.begin(9600); // Setup Serial port.
        // Initialize the timer.
        SecondsTimer = millis();
        // Initialize the clock.
        Hours = 9;
        Minutes = 58;
        Seconds = 58;
} // End of setup

// main code, ran repeatedly:
void loop()
{
        // Check timer
        if (millis() - SecondsTimer >= INTERVAL)
        {
                // Every Second this is run.
                UpdateClock();
                SendClock();
                // Update Timer.
                SecondsTimer += INTERVAL;

        }// End of Timer test
}// End of loop
```



```
09:58:59
09:59:00
09:59:01
09:58:59
09:59:00
09:59:01
09:59:02
09:59:03
09:59:04
09:59:05
09:59:06
09:59:07
09:59:08
09:59:09
09:59:10
09:59:11
09:59:12
09:59:13
09:59:14
09:59:15
09:59:16
```