This project will show how the Arduino can be used to drive an LED grid display. The display is shown file "LED_Controller.PDF" and is similar to that seen in lots of older equipment. The important thing to notice about the grid, is that the row lines are connected to multiple the LED's and the columns are also connected to multiple LED's. Thus to turn on a certain LED, we need to activate the column and the row for that LED. The implication of this is that we cannot arbitrarily turn on LEDs that aren't in the same row or column. The solution to this is to scan through the columns and depending upon which column we are on, we will turn on the rows matching the LED's we want on. Thus on a standard time basis (4 milliseconds) one of the column pairs are activated along with the appropriate rows. After each time interval, we move to the next column pair.

So to facilitate this a library was written that would hold the data for each column and rows. The library is in Appendix A, as a header file and a cpp file. We will discuss in class the format of classes and how this works, but it is not anticipated that many of you will ever have to write such code.

The other thing to observe about the system is the large number of pins that need to be driven to run the LED grid (5 lines for the columns, and 24 rows). We don't have anywhere near that many pins, and if we did we would probably not want to commit them to this one display. Thus a board was built that used hardware buffers to hold the bits which turn on the columns and rows. The hardware buffers are set up to take in data serially over and SPI bus. The schematic of this is on the second page of the file "LED_Controller.PDF" and is rather busy. I will try and point out the pertinent parts in class.

As part of this work, code was written to support an SPI transfer. Instead of using the built in hardware on the Arduino for the SPI, this code was done to allow the Analog pins to be used as the SClk and SDin signals. Also, having our own code allows us to do 32 bit transfer, instead of doing multiple 8 or 16 bit transfers.

For the timing of writes to the display, TimerOne to set to run an Interrupt Service Routine to send the data over on a 4 millisecond basis. This allows the other parts of the program to ignore this action, and if they want to have an LED appear as on, they simply set that bit in the array inside DisplayData.

In order to demonstrate the display, a clock was set up and the time displayed on the LED grid. It is shown in binary, which makes it some what NERDY and hard to read. In fact, it helps if you mask off the other parts of the display and only show the LED's that are the clock numbers.

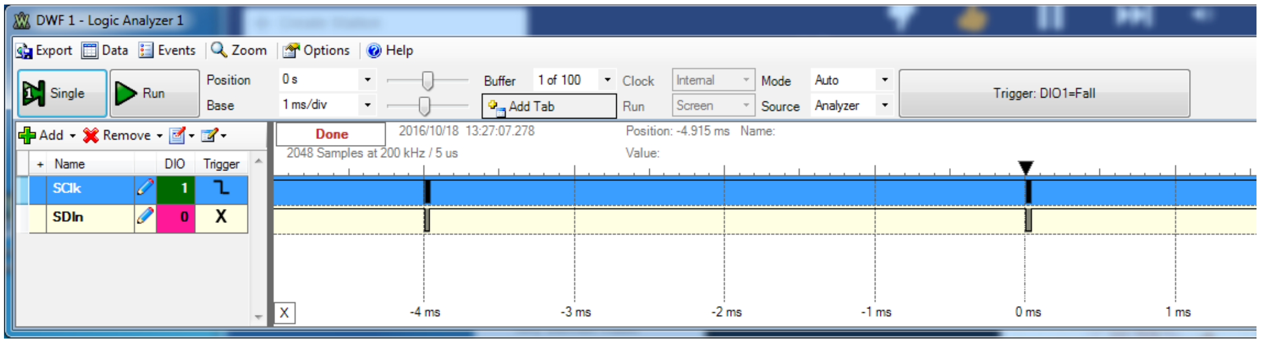
Finally, a clock that cannot be set is rather worthless. Thus a facility for setting the clock was added, employing the button and encoder. This code is also part of the Morse Code Transmitter.

Arduino Code:

```
// libraries
#include <LiquidCrystal.h>

// Defines for LCD
LiquidCrystal LcdDriver(11, 9, 5, 6, 7, 8);

#include <TimerOne.h>
#include <DisplayData.h>

DisplayData DispData;

#include "SPI_Local.h"      // Software version of SPI.
#include "ButtonDebounce.h" // Software for button debounce.
#include "EncoderMonitor.h" // Reading encoder.
#include "ClockBasics.h"    // Running the clock.

void DisplayValue(int TimeValue, int Row)
{
	int Local = TimeValue / 10;
	for (int i = 0; i < 4; i++)
	{
		if (Local & (1 << i))
			DispData.SetBit(Row, 4 - i);
		else
			DispData.ClearBit(Row, 4 - i);
	}
	Local = TimeValue % 10;
	for (int i = 0; i < 4; i++)
	{
		if (Local & (1 << i))
			DispData.SetBit(Row - 2, 4 - i);
		else
			DispData.ClearBit(Row - 2, 4 - i);
	}
} // End of DisplayValue

// Displays clock on LedMatrix
void DisplayClock()
{
	// Set Hours into rows
	DisplayValue(Hours, 10);
	// Set Minutes into row
	DisplayValue(Minutes, 6);
	// Set Seconds into row
	DisplayValue(Seconds, 2);

} // End of DisplayClock

// Interrupt Service Routine to maintain Display
void ISR_ServiceDisplay()
{
	Spi_Transmit32(DispData.NextWord());
}

// Clock Timer Setup.
#define ONE_SECOND 1000
unsigned long ClockTimer;
int EncoderTracking;

void setup()
{
	// put your setup code here, to run once:
	DispData.Initialize();
```

```
        Spi_Initialize();
        // Set up timer to service LED Display.
        Timer1.initialize(4000);  // Sets timer to fire every 4 millisec.
        Timer1.attachInterrupt(ISR_ServiceDisplay);
        // Set up clock
        ClockTimer = millis();
        Hours = 9;    // Set clock  close to rollovers.
        Minutes = 58;
        Seconds = 55;
        // The controls for setting the clock.
        ButtonInitialize(4); // connect to push button
        EncoderInitialize();   // Monitor encoder.
        EncoderTracking = EncoderValue;

        // LCD Setup.
        LcdDriver.begin(16, 2);
        LcdDriver.clear();
}
int LED_State = 0;

void loop()
{
        // Check if it is time to Update the clock.
        if (millis() - ClockTimer >= ONE_SECOND)
        {
                if (clockState == CLOCK_RUNNING)
                {       // Don't change clock when it is being set.
                        UpdateClock();
                }
                DisplayClock();
                LcdDriver.clear();
                SendClock(); // update clock on Lcd
                // place cursor to indicate if the clock is being set and
                // which item is being set.
                switch (clockState)
                {
                case CLOCK_SET_HOURS:
                        LcdDriver.setCursor(0, 0);
                        break;
                case CLOCK_SET_MINUTES:
                        LcdDriver.setCursor(3, 0);
                        break;
                case CLOCK_SET_SECONDS:
                        LcdDriver.setCursor(6, 0);
                        break;
                case CLOCK_RUNNING:
                        LcdDriver.setCursor(0, 1);
                }
                LcdDriver.blink();
                ClockTimer += ONE_SECOND;
        }
        // Watch button and encoder to set clock
        if (ButtonTest() == 1)  // Button pressed,
                SettingClock(1, 0);  // So update state.
        else if ((EncoderValue - EncoderTracking) >= 4)
        {   // The encoder has moved by one detent
                SettingClock(0, -1);  // Indicate positive travel
                EncoderTracking += 4;   // and move tracking variable.
        }
        else if ((EncoderValue - EncoderTracking) <= -4)
        {   // The encoder has moved by one detent in the opposite direction
                SettingClock(0, +1);  // Indicate negative travel
                EncoderTracking -= 4;   // and Update Tracking accordingly
        }
}
```

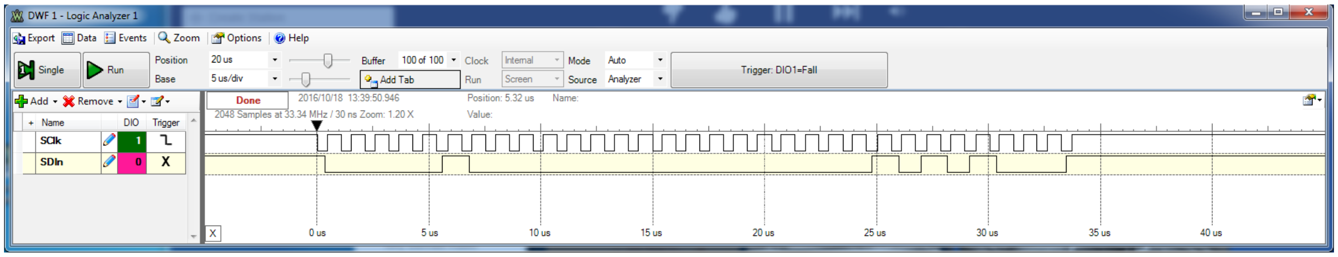
I monitored the SClk and SDin lines on the display board, using the Analog Discovery. The following screen shot shows the 4 millisecond interval, and that we are only spending a small fraction of our time servicing the display.



Position 0 s
Base 1 ms/div
Buffer 1 of 100
Clock Internal
Run Screen
Mode Auto
Source Analyzer
Trigger: DIO1=Fall
Done 2016/10/18 13:27:07.278
2048 Samples at 200 kHz / 5 us
Position: -4.915 ms Name:
Value:

Name	DIO	Trigger
SClk	1	⎸
SDIn	0	X

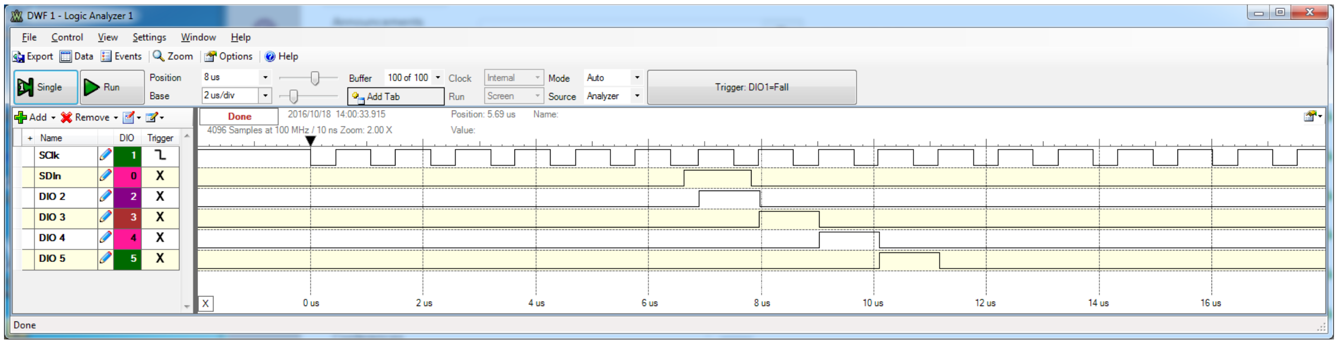
If we zoom in on the actual transfer, we can see the that it only requires less than 35 microseconds, which is less than 1% of the 4 millisecond interval.



Position 20 us
Base 5 us/div
Buffer 100 of 100
Clock Internal
Run Screen
Mode Auto
Source Analyzer
Trigger: DIO1=Fall
Done 2016/10/18 13:39:50.946
2048 Samples at 33.34 MHz / 30 ns Zoom: 1.20 X
Position: 5.32 us Name:
Value:

Name	DIO	Trigger
SClk	1	⎸
SDIn	0	X

In an effort to show how the data is shifted into the buffers, bottom four bits of the shift registers were monitored as shown here.



Position 8 us
Base 2 us/div
Buffer 100 of 100
Clock Internal
Run Screen
Mode Auto
Source Analyzer
Trigger: DIO1=Fall
Done 2016/10/18 14:00:33.915
4096 Samples at 100 MHz / 10 ns Zoom: 2.00 X
Position: 5.69 us Name:
Value:

Name	DIO	Trigger
SClk	1	⎸
SDIn	0	X
DIO 2	2	X
DIO 3	3	X
DIO 4	4	X
DIO 5	5	X

Appendix A: Library for holding display data.

Header in file DisplayData.h

```
#ifndef DisplayData_h
#define DisplayData_h

#define ColumnSets 5
#define DisplayColumns 10
#define DisplayRows    12

// Object that is used to hold data for LED display
// Controlling the writing of bits
// and sends them out to as they need to be displayed.
class DisplayData {
private: unsigned long DisplayBits[ColumnSets], CurrColumn;
              int index;
              int bit;
public:
      void Initialize();            // Sets column controls and zeros bits
      void SetBit(int r, int c);    // Turns on the LED at r,c
      int  GetBit(int r, int c);    // Returns the value of the LED at r,c
      void ClearBit(int r, int c); // Turns off the LED at r,c
      unsigned long NextWord();
}; // Advances pointer and returns next word
// to be sent.
#endif
```

C-code in file DisplayData.cpp

```
#include <DisplayData.h>

// Object for display
// Code to initialize the structure DisplayData.
void DisplayData::Initialize()
{
      DisplayBits[0] = 0x01000000; // Col 1 and 6 // Each line represents a set of
      DisplayBits[1] = 0x02000000; // Col 2 and 7 // 24 led's or 2 columns of 12 led's.
      DisplayBits[2] = 0x04000000; // Col 3 and 8 // Upper bits set the pair of columns.
      DisplayBits[3] = 0x08000000; // Col 4 and 9
      DisplayBits[4] = 0x10000000; // Col 5 and 10
      CurrColumn = 0;  // Current set of led's that are one.
} // End of Initialize

// Function to turn on an LED at the location of row and column (r,c).
void DisplayData::SetBit(int r, int c)
{
      unsigned long One = 1;
      index = c % 5;
      bit = r % 12 + 12 * ((c % 10) / 5);
      DisplayBits[index] |= (One << bit); // Mask bit to one
} // End of SetBit

// Read back the current value of a bit.
int DisplayData::GetBit(int r, int c)
{
      index = c % 5;
      bit = r % 12 + 12 * ((c % 10) / ColumnSets);
      return(DisplayBits[index] & (1 << bit));
}  // End of GetBit
```

```
// Function to turn off an LED at the location of row and column (r,c).
void DisplayData::ClearBit(int r, int c)
{
	unsigned long One = 1;
	index = c % 5;
	bit = r % 12 + 12 * ((c % 10) / ColumnSets);
	DisplayBits[index] &= ~(One << bit); // Mask bit to zero
} // End of ClearBit

// This function is called to move to the next pair of columns of LEDs
// and then return the 32 bits for the column pair, which should
// be sent over to the display via spi.
unsigned long DisplayData::NextWord()
{
	CurrColumn++; // Advance pointer
	if (CurrColumn == ColumnSets) // and wrap
		CurrColumn = 0;
	return (DisplayBits[CurrColumn]); // return current word
	                                  // Turning on column and rows.
}  // End of NextWord
```

Appendix B: SPI code for 32 bit transfer, using A5 and A4 pins.

```
#ifndef SPI_Local_h
#define SPI_Local_h 1

// SPI support on Analog pins A5 and A4
#define SpiDataBit 5
#define SpiClkBit 4

// Set up pins as outputs.
void Spi_Initialize()
{
	bitSet(DDRC, SpiDataBit);
	bitSet(DDRC, SpiClkBit);
}

// Function to transmit 32 bit number
void Spi_Transmit32(unsigned long out)
{
	int k;

	for (k = 0; k < 32; k++)
	{
		// Set Clock low
		bitClear(PORTC, SpiClkBit);

		if (out & 0x80000000) // If top bit is high,
			bitSet(PORTC, SpiDataBit);
		else  // or low
			bitClear(PORTC, SpiDataBit);

		// Set Clock High
		bitSet(PORTC, SpiClkBit);

		// Move next bit into top bit.
		out = out << 1;

	} // End of loop through bits.

	return;

} // End of Spi_Transmit32
#endif
```

Appendix C: Clock support

```
#ifndef ClockBasics_H
#define ClockBasics_H
// 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()

// Send Hours, Minutes and Seconds to a display.
void SendClock()
{
	// Check if leading zero needs to be sent
	if (Hours < 10)
	{
		LcdDriver.print("0");
	}
	LcdDriver.print(Hours); // Then send hours
	LcdDriver.print(":"); // And separator
	// Check for leading zero on Minutes.
	if (Minutes < 10)
	{
		LcdDriver.print("0");
	}
	LcdDriver.print(Minutes); // Then send Minutes
	LcdDriver.print(":"); // And separator
	// Check for leading zero needed for Seconds.
	if (Seconds < 10)
	{
		LcdDriver.print("0");
	}
	LcdDriver.print(Seconds); // Then send Seconds
} // End of SendClock()
```

```
// States for setting clock.
enum ClockStates {
       CLOCK_RUNNING, CLOCK_SET_HOURS,
       CLOCK_SET_MINUTES, CLOCK_SET_SECONDS
};
ClockStates clockState = CLOCK_RUNNING;

// Function that processes incoming characters to set the clock.
void SettingClock(int Button, int EncUpDown)
{
       // interpret input based on state
       switch (clockState)
       {
       case CLOCK_RUNNING:
              if (Button)
              {
                     clockState = CLOCK_SET_HOURS;
                     Hours = 0;   // Reset clock values.
                     Minutes = 0;
                     Seconds = 0;
              }
              break;
       case CLOCK_SET_HOURS: //
              if (EncUpDown > 0)
              {
                     Hours++;
                     if (Hours > 23)
                            Hours = 0;
              }
              else if (EncUpDown < 0)
              {
                     Hours--;
                     if (Hours < 0)
                            Hours = 23;
              }
              else if (Button)
                     clockState = CLOCK_SET_MINUTES;
              break;
       case CLOCK_SET_MINUTES: //
              if (EncUpDown > 0)
              {
                     Minutes++;
                     if (Minutes > 59)
                            Minutes = 0;
              }
              else if (EncUpDown < 0)
              {
                     Minutes--;
                     if (Minutes < 0)
                            Minutes = 59;
              }
              else if (Button)
                     clockState = CLOCK_SET_SECONDS;
              break;
       case CLOCK_SET_SECONDS: //
              if (EncUpDown > 0)
              {
                     Seconds++;
                     if (Seconds > 59)
                            Seconds = 0;
```

```
            }
            else if (EncUpDown < 0)
            {
                  Seconds--;
                  if (Seconds < 0)
                        Seconds = 59;
            }
            else if (Button)
                  clockState = CLOCK_RUNNING;
            break;

      }// End of clock mode switch.

} // End of SettingClock

#endif
```

Appendix D: Button support

```
#ifndef ButtonDebounce_H
#define ButtonDebounce_H

// Set up pin and button state.
int ButtonPin = 4;
enum buttonStates { BS_Idle, BS_Wait, BS_Low };
buttonStates buttonState = BS_Idle;
unsigned long buttonTimer;

// Initialization code, setting up pin.
void ButtonInitialize(int pin)
{
	ButtonPin = pin;
	pinMode(ButtonPin, INPUT);
} // End of ButtonInitialize

// Function called in loop to check for button release.
// Returns a 1 on the buttons release.
int ButtonTest()
{
	// Read in the buttons current value.
	int Press = digitalRead(ButtonPin);
	int ReturnValue = 0;
	switch (buttonState)
	{
	case BS_Idle: // if we are waiting for a press,
		if (Press == LOW)
		{	// Once press occurs
			buttonTimer = millis();  // record time
			buttonState = BS_Wait;	// and move to next state
		}
		break;
	case BS_Wait: // button just went low
		if (Press == HIGH) // and now goes high
		{
			buttonState = BS_Idle;  // return to 0 state.
		}
		else // if still low
		{
			// and sufficient time has passed.
			if (millis() - buttonTimer >= 10)
			{
				ReturnValue = 1; // Return 1 indicating pressed.
				buttonState = BS_Low; // move on to state two
			}
		}
		break;
	case BS_Low:
		if (Press == HIGH)
		{
			ReturnValue = 2; // Return 2 indicating release.
			buttonState = BS_Idle;
		} // End of high test.
		break;
	} // End of switch on buttonState

	return ReturnValue;

} // End of ButtonRead

#endif
```

Appendix E: Encoder support

```
#ifndef EncoderMonitor_h
#define EncoderMonitor_h

// Variable for keeping track of encoder change.
volatile int EncoderValue;

// Service Routine for Encoder line A
void EncoderMonitorPinA(void)
{
	// compare pin 3 and pin 2
	if (bitRead(PIND, 3) == bitRead(PIND, 2))
	{
		EncoderValue--;
	}
	else
	{
		EncoderValue++;
	}
}// End of EncoderMonitorPinA

// Service Routine for Encoder line B
void EncoderMonitorPinB(void)
{
	// compare pin 3 and pin 2
	if (bitRead(PIND, 3) == bitRead(PIND, 2))
	{
		EncoderValue++; // Note this is the opposite
		// of the action in PinA.
	}
	else
	{
		EncoderValue--;
	}
} // End of EncoderMonitorPinB

// Initializes the Encoder Monitoring Code.
void EncoderInitialize()
{
	// Set encoder value to zero
	EncoderValue = 0;

	// Set up interrupts to monitor inputs from encoder.
	attachInterrupt(0, EncoderMonitorPinA, CHANGE);
	attachInterrupt(1, EncoderMonitorPinB, CHANGE);

}

#endif
```