ECE3849 Lab 1

Nam Tran Ngoc
Yigit Yuan
Mailbox 388

November 21, 2016

1 Introduction

During this lab experiment, we built a working implementation of a 500ksps oscilloscope on the EK-LM3S8962. The oscilloscope was able to sample the signal values from the 10-bit ADC into a circular buffer, display the values in a waveform, complete with a trigger search. The implementation was also able to do both time and voltage scale, which is adjustable using button inputs. Figure 1 shows the overview of the circuit.

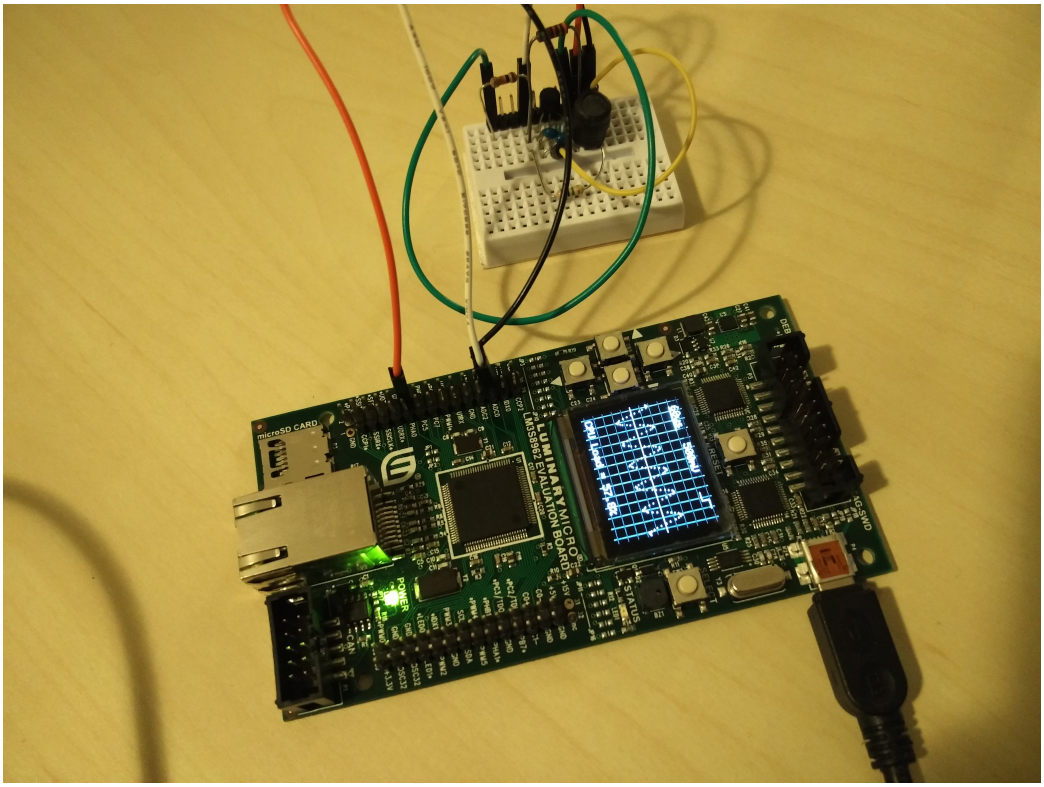

Figure 1: Overview of the circuit

2 Discussion and Results

In this part we will discuss the methodologies used during the projects, along with their results.

2.1 Building the circuit

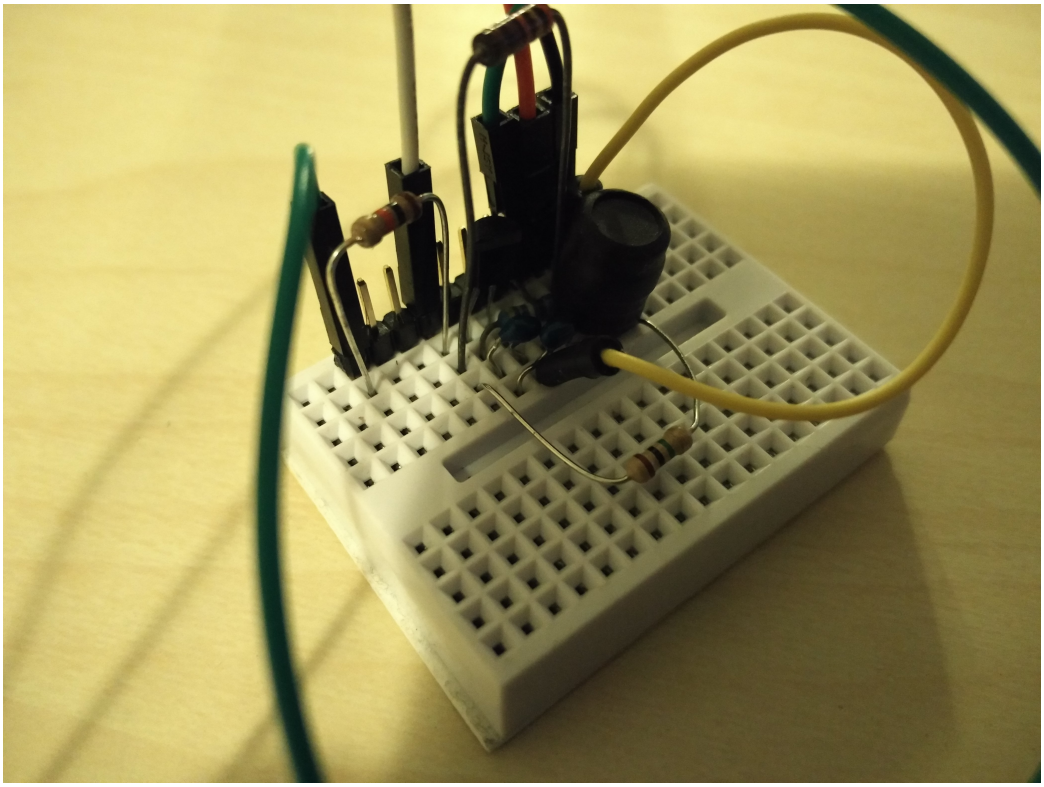

Figure 2: The Colpitts oscillator

We started off by building a Colpitts oscillator from the schematic shown in our lab requirement. The resulting circuit can be seen in Figure 2. The circuit generates a (near) sine wave signal of approximate 7.1kHz, with an amplitude of approximately 0.3V, which we were able to verify using a bench scope. The signal output is then connected to a voltage divider, which maps the voltage range from -3V - 3V to 0 - 3V.

2.2 Initializing the program

To start the program, we have to initialize different peripherals needed for the labs. Those are: system clock, display, timers (0 through 2), buttons, and ADC. We will be looking at the code for initializing the ADC below.

2.2.1 Initializing ADC

Using the functions provided in the lab requirement, we filled out the parameters using macros, and the Stellaris Peripheral Driver Library User's Guide.

```
1
2 void initializeADC()
3 {
4  SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0); //
      enable the ADC
5  SysCtlADCSpeedSet(SYSCTL_ADCSPEED_500KSPS); //
      specify 500ksps
6  ADCSequenceDisable(ADC0_BASE, 0); // choose ADC
      sequence 0; disable before configuring
7  ADCSequenceConfigure(ADC0_BASE, 0,
      ADC_TRIGGER_ALWAYS, 0); // specify the "Always"
      trigger, highest priority
8  ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
      ADC_CTL_CH0 | ADC_CTL_IE | ADC_CTL_END); // in
      the 0th step, sample channel 0
9   // enable interrupt, and make it the end of
       sequence
10  ADCIntEnable(ADC0_BASE, 0); // enable ADC
      interrupt from sequence 0
11  ADCSequenceEnable(ADC0_BASE, 0); // enable the
      sequence. it is now sampling
12  IntPrioritySet(INT_ADC0, 32); // set ADC interrupt
      priority in the interrupt controller - lower
      than main ISR
13  IntEnable(INT_ADC0); // enable ADC interrupt
14 }
```

assets/main.c

2.3 Using interrupts

In this subsection, we explore different roles the ISR take to make the program works.

2.3.1 Timer0

Timer0 is used mainly to send a pulse to blink the on-board LED, to make sure the program is running. This is mainly used for debugging/decorating purposes, and should not interfere with other main functions

2.3.2 Timer1

Timer1 is used to calculate the CPU load by calling

```
cpu_load_count()
```

every time it is called, to update the CPU load.

2.3.3 Timer2

Timer2 is used to scan the button inputs, debounce them, and put them in an array to process later. Each button press in the array is identified by a different character.

2.3.4 ADC

The ADC0 is used for this lab experiment. During the call, it checks for FIFO overflow, increment error count if that happens, and store readings to the g_psADCBuffer.

2.4 Drawing screen elements

The process of screen rendering is handled by four different functions: drawGrid, drawOverlay, drawTrigger and drawSignal. The screen is then rendered through screenDraw(), and wiped with screenClean()

2.4.1 drawGrid

drawGrid loops through the number of horizontal and vertical bars, and draw a line at the corresponding position.

2.4.2 drawOverlay

drawOverlay handles a number of different on-screen elements: time scale, voltage scale, cpu load, and trigger direction. drawOverlay also employs two helper functions, strCenter() and strWidth() to help placing and centering the text.

```
1
2 int strWidth(char* _String)
3 {
4  int offset = 0;
5  int count = 0;
6
7  while (*(_String + offset) != '\0')
8  {
9   ++count;
10   ++offset;
11  }
12  return count * 6; // Each character a width of 6
      pixels in ASCII
13 }
```

assets/main.c

```
1
2 int strCenter(int _Coordinate, char* _String)
3 {
4  int result = _Coordinate - (strWidth(_String) / 2);
5
6  if(result > 0 && result < DISPLAY_WIDTH)
7   return result;
8  else if(result > DISPLAY_WIDTH)
9   return DISPLAY_WIDTH;
10  else
11   return 0;
12 }
```

assets/main.c

2.4.3 drawTrigger

drawTrigger simply draws a horizontal line in the middle of the screen, due to the fact that our trigger is a constant for the purpose of this lab.

2.4.4 drawSignal

drawSignal turned out to be more complicated than the other drawing functions, since it takes in the inputBuffer, performs a trigger search, and generates the waveform based on found values.

We start off by declaring the pixelRange, the range to display the final pixel outputs; and the pixelWidth, with is calculated by dividing the current timeScale position by the gridWidth. This ensures that the pixels are scaled to different time scales.

First we have to disable the IRQs to avoid any time-share data problems.

```
1
2  IntMasterDisable();        // IRQs Disabled so
     PixelBuffer is filled accurately.
```

assets/main.c

Our trigger search implementation was encapsulated in an if statement, which checks for any buffer values that pass the trigger requirements: within a range of ±25 of adcZeroValue (which would be our offset), and check for rising, or falling values by comparing it to the buffer 5 indexes behind and 5 indexes ahead.

```
1
2  if (_InputBuffer[triggerIndex] < adcZeroValue + 25
     &&
3   _InputBuffer[triggerIndex] > adcZeroValue - 25 &&
4    ((triggerUp && (_InputBuffer[triggerIndex - 5] <
       _InputBuffer[triggerIndex + 5])) ||
5     (!triggerUp && (_InputBuffer[triggerIndex - 5]
       > _InputBuffer[triggerIndex + 5]))))
```

assets/main.c

If the index currently looping through was found to satisfy all requirements above, we then loop through the pixelBuffer (which is our display array), and fill it with appropriate pixels, which are calculated using pixelWidth.

```
1
2   // This while loop fills the right side of the
       buffer.
3   i = pixelRange / 2;
4   j = 0;
5   while(i < pixelRange)
6   {
7    if(triggerIndex + (int)((j * pixelWidth) / 2) <
        2048)
8     pixelBuffer[i] = _InputBuffer[triggerIndex +
         (int)((j * pixelWidth) / 2)];
9    else
10    pixelBuffer[i] = 0;     // Fill empty if trigger
         is too close to the end of sample buffer.
11   i++;
12   j++;
13  }
14
15  // This while loop fills the left side of the
       buffer.
16  i = pixelRange / 2;
17  j = 0;
18  while(i > 0)
19  {
20   if(triggerIndex - (int)((j * pixelWidth) / 2) >
        0)
21    pixelBuffer[i] = _InputBuffer[triggerIndex -
         (int)((j * pixelWidth) / 2)];
22   else
23    pixelBuffer[i] = 0;     // Fill empty if trigger
         is too close to the end of sample buffer.
24   i--;
25   j++;
26  }
```

assets/main.c

Lastly, we enable the IRQs, and draw the pixels to the RAM, scaled appropriately to the voltage.

```
1
2  IntMasterEnable();        // Values are correctly
     set. Interrupts can be enabled when drawing
     (next while)..
3
4  i = 0;
5  while(i < pixelRange)       // This while loop
     draws the pixelBuffer (Signal) on Display
     (Buffer).
6  {
7   int offsetY = gridYMin + (gridYMin + gridYMax) /
      2;
8   DrawPoint(gridXMin + i, offsetY - (pixelBuffer[i]
      - adcZeroValue) /
      (voltageNScales[voltageIndex] / 100),
      COLOR_SIGNAL);
9   i++;
10 }
```

assets/main.c

2.4.5 Scaling

Since the output waveform has to scale to different timeScale and voltageScale, one sample could not be interpreted as one pixel. For this, we implemented two arrays: timeNScales, and voltageNScales, which contain the values of the scale, and by using these, the pixel value can be extrapolated (e.g. pixelWidth variable in the drawSignal function). This method results in a continuous line of the signal, which makes it much clearer.

Figures 3 shows how different timeScales affect the same waveform.

2.5 Handling button inputs

Lastly, we go over how we process the button inputs while maintaining real-time functionality.

As we went through in the first section, we used Timer2 to scan the button inputs, and put them in an array. The processButtons() function,

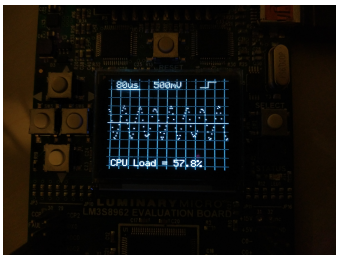

Figure 3: Timescale at 80us

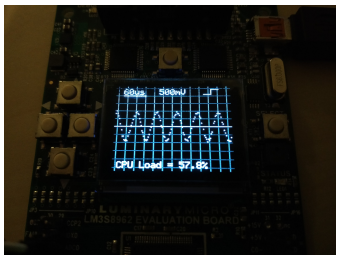

Figure 4: Timescale at 60us

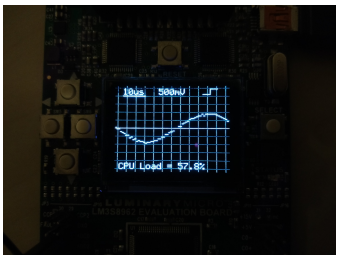

Figure 5: Timescale at 10us

which is called in the main() function, go through the array, process them, and update the user interface accordingly based on the buttonArrayIndex.

```
1
2 void processButtons()
3 {
4  if(buttonArrayIndex > 0)
5  {
6
7   char currentEntry = buttonArray[buttonArrayIndex
      - 1];
8   buttonArrayIndex--;
9
10  if(currentEntry == 'S' || currentEntry == 's')
11  {
12   if(triggerUp)
13    triggerUp = 0;
14   else
15    triggerUp = 1;
16  }
17  if(currentEntry == 'R' || currentEntry == 'r')
18  {
19   if(selectionIndex < 1)
20    selectionIndex++;
21  }
22  if(currentEntry == 'L' || currentEntry == 'l')
23  {
24   if(selectionIndex > 0)
```

```
25      selectionIndex--;
26    }
27    if(currentEntry == 'D' || currentEntry == 'd')
28    {
29     if(selectionIndex == 0)
30     {
31      if(timeIndex > 0)
32       timeIndex--;
33     }
34     else
35     {
36      if(voltageIndex > 0)
37       voltageIndex--;
38     }
39    }
40    if(currentEntry == 'U' || currentEntry == 'u')
41    {
42     if(selectionIndex == 0)
43     {
44      if(timeIndex < 9)
45       timeIndex++;
46     }
47     else
48     {
49      if(voltageIndex < 3)
50       voltageIndex++;
51     }
52    }
53   }
54 }
```

assets/main.c

3 Conclusion

In this lab, we got the chance to explore the basics of interrupt driven software architecture and programmed a TI Microcontroller as an oscilloscope.

We have built a simple Colpitts oscillator and generated sine waves to measure. We saw that this circuit is not very reliable, as the components (especially the inductor) needed very good contact with the breadboard at all times. Most of the time, it worked fine to generate a 3V +/- 0.3V sine curve, which we sampled 500.000 times a second using the analog to digital (ADC) converter on chip. While this was a thight timing requirement (leaving 2 s per sample), Cortex-M3 was up to the challenge, leaving us plenty of flops to implement other complex features as well. We have built a circullar buffer to handle samples, transfer a portion to a local buffer, map it to screen send it to the display hardware.

We have designed a responsive graphical user interface (GUI) for the display features. Overall, our oscilloscope worked very reliably, and efficient too! With only 58% CPU utilization, we are proud of our code performance.