ECE 3849 B2022

Real-Time Embedded Systems

Lab 1: Digital Oscilloscope Report

Professor Stander

William Tyrrell

11/22/22

Introduction:

In this lab, we will be making a 1 Msps digital oscilloscope using the ECE 3849 lab kit. The goals of the lab are to mimic the sample display shown in figure 1 and display a square wave on the screen. This requires me to implement a CPU load measurement, a trigger slope indicator and ability to change the rising and falling edge of the wave, and for extra credit a variable voltage scale and time scale. I successfully implemented the trigger slope, CPU utilization, and the voltage scale in my lab.

Discussion and Results:

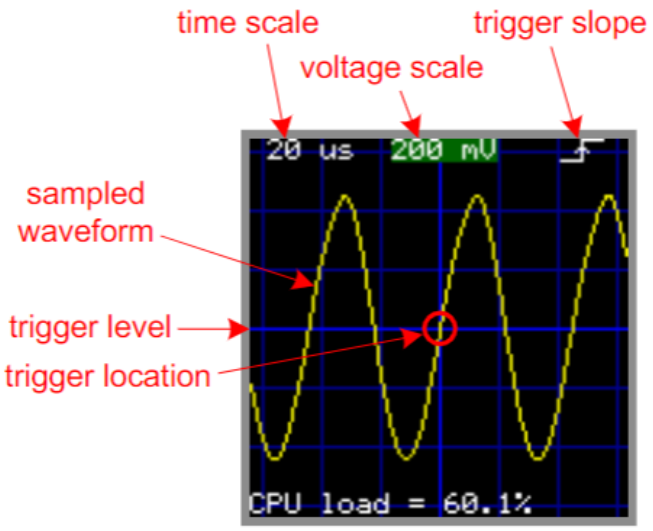

Figure 1: Sample Screen for Oscilloscope

.

5

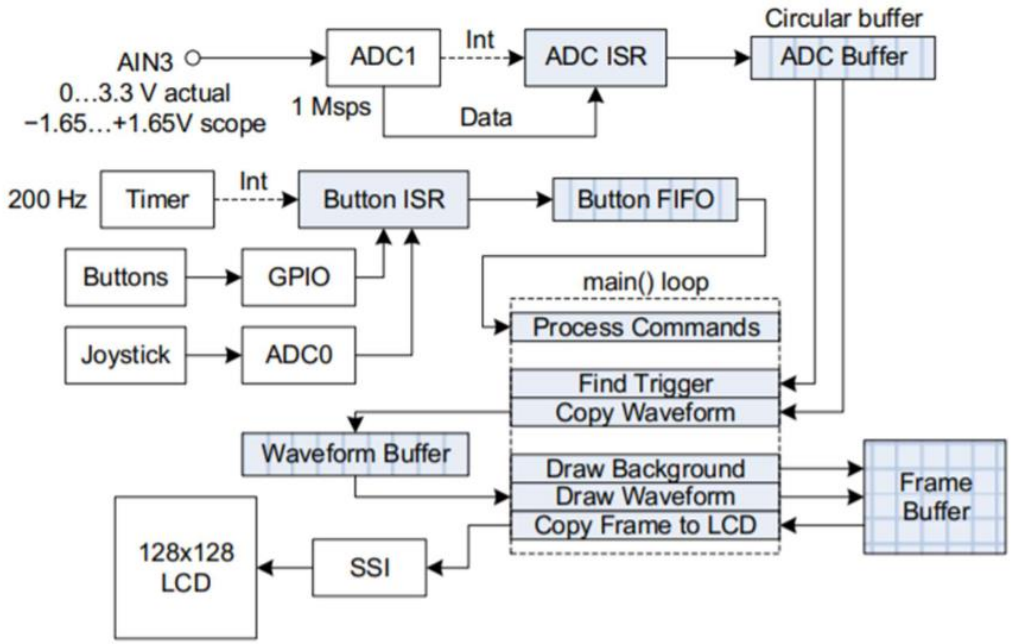

Figure 2: Software Block Diagram

Step 1: Setting-up the input signal source to the oscilloscope:

The fist thing that I did to start the lab was copy over the files from Lab 0 as they will be needed to complete this lab. The first task of the lab was copying some given define and include statements into my main file. I then used the given code shown in figure 4, that produces a 20 kHz PWN square wave and placed it in my main file. This signal will be fed into an Analog to Digital Converter (ADC) to sketch the waveform. This will range from 0 to 3.3V and it will be centered in the screen so the actual range will be -1.65V to +1.65V. I then connected a jumper wire to the pins shown in figure 3.

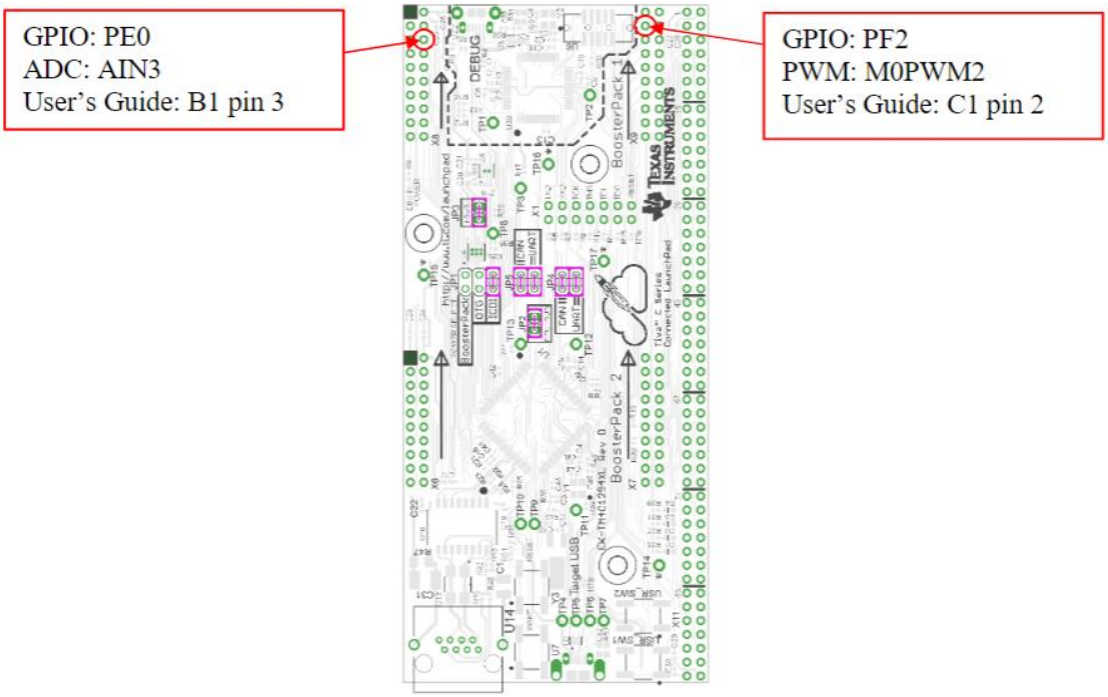

Figure 3: Source Voltage pins

```
// configure M0PWM2, at GPIO PF2, BoosterPack 1 header C1 pin 2
    // configure M0PWM3, at GPIO PF3, BoosterPack 1 header C1 pin 3
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);
    GPIOPinConfigure(GPIO_PF2_M0PWM2);
    GPIOPinConfigure(GPIO_PF3_M0PWM3);
    GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3,
                     GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);

    // configure the PWM0 peripheral, gen 1, outputs 2 and 3
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
    PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_1); // use system clock without division
    PWMGenConfigure(PWM0_BASE, PWM_GEN_1, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, roundf((float)gSystemClock/PWM_FREQUENCY));
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, roundf((float)gSystemClock/PWM_FREQUENCY*0.4f));
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_3, roundf((float)gSystemClock/PWM_FREQUENCY*0.4f));
    PWMOutputState(PWM0_BASE, PWM_OUT_2_BIT | PWM_OUT_3_BIT, true);
    PWMGenEnable(PWM0_BASE, PWM_GEN_1);
```

Figure 4: Source setup

Step 2: ADC Sampling:

In this step I was tasked with configuring the ADC to sample at 1,000,000 sample/sec without missing any deadlines. To do this I created two new modules, sampling.c and sampling.h, to configure ADC1 and the ISR associated with it. Most of the code was given but I needed to find the proper registers values to fill in the code with so that the ADC had its desired function.

```
void adcInit(void)
{

	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
	GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0); // GPIO setup for analog input AIN3

	// Initialize ADCs
	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC1);

	//ADC clock
	uint32_t pll_frequency = SysCtlFrequencyGet(CRYSTAL_FREQUENCY);
	uint32_t pll_divisor = (pll_frequency - 1) / (16 * ADC_SAMPLING_RATE) + 1; //round up
	ADCClockConfigSet(ADC0_BASE, ADC_CLOCK_SRC_PLL | ADC_CLOCK_RATE_FULL, pll_divisor);
	ADCClockConfigSet(ADC1_BASE, ADC_CLOCK_SRC_PLL | ADC_CLOCK_RATE_FULL, pll_divisor);

	// Step configuration for ADC1
	ADCSequenceDisable(ADC1_BASE, 0); // choose ADC1 sequence 0; disable before configuring
	ADCSequenceConfigure(ADC1_BASE, 0, ADC_TRIGGER_ALWAYS, 0); // specify the "Always" trigger
	ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_CH3 | ADC_CTL_IE | ADC_CTL_END);// in the 0th step, sample channel 3 (AIN3)
	 // enable interrupt, and make it the end of sequence

	// Fire the interrupt and sequence now
	ADCSequenceEnable(ADC1_BASE, 0); // enable the sequence. it is now sampling
	ADCIntEnable(ADC1_BASE, 0); // enable sequence 0 interrupt in the ADC1 peripheral
	IntPrioritySet(INT_ADC1SS0, 0); // set ADC1 sequence 0 interrupt priority
	IntEnable(INT_ADC1SS0); // enable ADC1 sequence 0 interrupt in int. controller

}
```

Figure 5: ADC Setup

```
void ADC_ISR(void)
{
    ADC1_ISC_R = ADC_ISC_IN0; // clear ADC1 sequence0 interrupt flag in the ADCISC register
    if (ADC1_OSTAT_R & ADC_OSTAT_OV0) { // check for ADC FIFO overflow
        gADCErrors++;                   // count errors
        ADC1_OSTAT_R = ADC_OSTAT_OV0;   // clear overflow condition
    }
    gADCBuffer[
               gADCBufferIndex = ADC_BUFFER_WRAP(gADCBufferIndex + 1)
               ] = ADC1_SSFIFO0_R;              // read sample from the ADC1 sequence 0 FIFO
}
```

Figure 6: ADC ISR

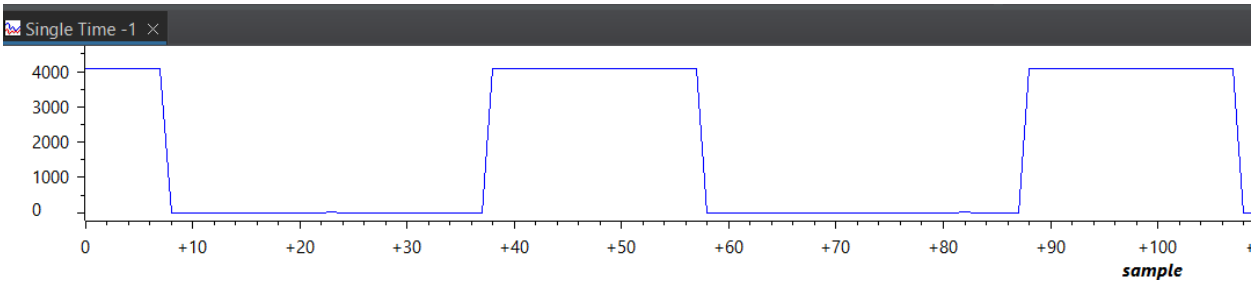

Figure 7: Square Wave

Step 3: Trigger Search:

The next step is to set up a trigger that can read the data associated with the square wave. The trigger location and level will be in the center of the screen and a button press will allow the trigger level to change from rising to falling. In terms of the square wave, the rising edge will be at the center of the screen for the rising trigger, and the falling edge of the wave will be at the center for the falling trigger. This will be indicated on the screen using a trigger slope icon. The trigger will be at 0V on the screen where the trigger level meets the trigger location. When the button is pressed to change between rising and falling trigger, the press will be passed from its ISR to main() through a FIFO displayed in figure 13.

The samples from the ADC are stored in gADCBuffer. In order to access this data to conform to the rising and falling edge triggers I used part of the giving RisingTrigger() function and I also created a FallingTrigger() function displayed below in figure 8.

```
int RisingTrigger(void)   // search for rising edge trigger
{
        checkTrigger = false;
        // Step 1
        int x = gADCBufferIndex - LCD_HORIZONTAL_MAX / 2;/* half screen width; don't use a magic number */;
        // Step 2
        int x_stop = x - ADC_BUFFER_SIZE/2;
        for (; x > x_stop; x--) {
            if (gADCBuffer[ADC_BUFFER_WRAP(x)] >= ADC_OFFSET &&
                    gADCBuffer[ADC_BUFFER_WRAP(x-1)] < ADC_OFFSET)/* next older sample */
                break;
        }
        // Step 3
        if (x == x_stop) {// for loop ran to the end
            x = gADCBufferIndex - LCD_HORIZONTAL_MAX / 2;; // reset x back to how it was initialized
            checkTrigger = true;
        }
        return x;
}

int FallingTrigger(void) // search for falling edge trigger
{
    checkTrigger = false;
    // Step 1
    int x = gADCBufferIndex - LCD_HORIZONTAL_MAX / 2;/* half screen width; don't use a magic number */;
    // Step 2
    int x_stop = x - ADC_BUFFER_SIZE/2;
    for (; x > x_stop; x--) {
        if (gADCBuffer[ADC_BUFFER_WRAP(x)] < ADC_OFFSET &&
                gADCBuffer[ADC_BUFFER_WRAP(x-1)] >= ADC_OFFSET)/* next older sample */
            break;
    }
    // Step 3
    if (x == x_stop) {// for loop ran to the end
        x = gADCBufferIndex - LCD_HORIZONTAL_MAX / 2; // reset x back to how it was initialized
        checkTrigger = true;
    }
    return x;
```

Figure 8: Rising and Falling Trigger

Both functions share the same step 1: use half of the LCD screen width.

Step 2 is where the two functions differ slightly. In this step, a for loop goes through the buffer searching for where the buffer waveform crosses the trigger level. In the rising trigger it checks for when the buffer is greater than the offset and less than the next older sample. In the falling trigger, it checks for when the buffer is less than the offset and greater than the next older sample.

Step three occurs if the trigger is not found in the first half of the buffer. This tells the function to stop and an indicator checkTrigger is implemented to display a warning message that the trigger was missed. This step is only seen when the jumper cable is disconnected causing the square wave to flatline.

Figure 9: Missing Trigger

In order to manipulate the circular buffer index, the wrapping macro ADC_BUFFER_WRAP() is used. It accepts either a positive or negative index and returns an index properly wrapped into the valid range of gADCBuffer. Sizing the buffer in powers of 2 simplifies the wrapping operation to a very fast bitwise AND.

Trigger search is a low priority event. The only time constraint is that it needs to find the trigger before the ADC ISR circles around and overwrites the samples in the search range. We cannot draw the waveform directly from gADCBuffer though. Drawing is a slow operation and doing this from the buffer will likely allow the ISR to overwrite the data in the buffer possibly leading to discontinuous, unsteady waveform or a waveform not crossing the trigger level at the expected

location. This is a shared data bug that needs to be treated with care. The solution will be to copy the buffer into a local buffer which is a fast atomic operation. Then we can draw the waveform from this local buffer and not have to worry about the ISR overwriting our samples. We will Perform the desired read operations on half of the buffer while the ISR overwrites the other half.

Step 4: ADC Sample Scaling

In order to draw the waveform on the display, the raw ADC samples need to be scaled such that they conform to a volts/division scale. Much of this code is given particularly the code that converts to the pixels on the screen. Any parts that would remain constant were placed as define statements at the top of my main() file. The scale function was given code, there are two identical for loops and the reason for this is so that when gADCBuffer is copied to the local sample array there are no other actions that could delay this. The second loop reads data from the local buffer and converts it to the pixel location on the display.

```
scale = (VIN_RANGE * PIXELS_PER_DIV) / ((1 << ADC_BITS) * fVoltsPerDiv[voltsperDiv]);
for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++)
{
    // Copies waveform into the local buffer
    sample[i] = gADCBuffer[ADC_BUFFER_WRAP(trig - LCD_HORIZONTAL_MAX / 2 + i)];
}

// draw the coordinate
 for(i = 0; i < LCD_HORIZONTAL_MAX - 1; i++) {
    y = LCD_VERTICAL_MAX / 2 - (int)roundf(scale * ((int)sample[i] - ADC_OFFSET));
    GrLineDraw(&sContext, i, yP, i + 1, y);
    yP = y;
}
```

Figure 10: Drawing the Waveform

Step 5: Waveform Display Formatting and Button Command Processing:

I followed the formatting of the sample display shown in figure 1. The code is commented to show where certain features of the display are implemented. This also includes the code that displays the missingTrigger indicator that was mentioned in the step 3.

```
GrContextForegroundSet(&sContext, ClrBlack);
GrRectFill(&sContext, &rectFullScreen); // fill screen with black

//blue grid
GrContextForegroundSet(&sContext, ClrBlue);
for(i = -3; i < 4; i++) {
    GrLineDrawH(&sContext, 0, LCD_HORIZONTAL_MAX - 1, LCD_VERTICAL_MAX/2 + i * PIXELS_PER_DIV);
    GrLineDrawV(&sContext, LCD_VERTICAL_MAX/2 + i * PIXELS_PER_DIV, 0, LCD_HORIZONTAL_MAX - 1);
}

//waveform
GrContextForegroundSet(&sContext, ClrYellow);
trig = triggerSlope ? RisingTrigger(): FallingTrigger();
scale = (VIN_RANGE * PIXELS_PER_DIV) / ((1 << ADC_BITS) * fVoltsPerDiv[voltsperDiv]);
for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++)
{
    // Copies waveform into the local buffer
    sample[i] = gADCBuffer[ADC_BUFFER_WRAP(trig - LCD_HORIZONTAL_MAX / 2 + i)];
}

// draw the coordinate
 for(i = 0; i < LCD_HORIZONTAL_MAX - 1; i++) {
    y = LCD_VERTICAL_MAX / 2 - (int)roundf(scale * ((int)sample[i] - ADC_OFFSET));
    GrLineDraw(&sContext, i, yP, i + 1, y);
    yP = y;
}

//trigger direction, volts per div, cpu load
GrContextForegroundSet(&sContext, ClrWhite); //white text
if(triggerSlope){
    GrLineDraw(&sContext, 105, 10, 115, 10);
    GrLineDraw(&sContext, 115, 10, 115, 0);
    GrLineDraw(&sContext, 115, 0, 125, 0);
    GrLineDraw(&sContext, 112, 6, 115, 2);
    GrLineDraw(&sContext, 115, 2, 118, 6);
}else{
    GrLineDraw(&sContext, 105, 10, 115, 10);
    GrLineDraw(&sContext, 115, 10, 115, 0);
    GrLineDraw(&sContext, 115, 0, 125, 0);
    GrLineDraw(&sContext, 112, 3, 115, 7);
    GrLineDraw(&sContext, 115, 7, 118, 3);
}
```

```
GrContextForegroundSet(&sContext, ClrWhite); //white text
GrStringDraw(&sContext, "20 us", -1, 4, 0, false);
GrStringDraw(&sContext, gVoltageScaleStr[voltsperDiv], -1, 50, 0, false);
snprintf(str1, sizeof(str1), "CPU load = %.1f%%", cpu_load*100);
GrStringDraw(&sContext, str1, -1, 0, 120, false);

//Draw Missing Trigger indicator
GrContextForegroundSet(&sContext, ClrRed); //Red text
                while (checkTrigger == true){
                    snprintf(str1, sizeof(str1), "Missing Trigger");
                    GrStringDraw(&sContext, str1, -1, 20, 64, false);
                    break;
                }

GrFlush(&sContext); // flush the frame buffer to the LCD
```

Figure 11: Display formatting

As mentioned earlier, to process the button presses I implemented a FIFO that passes the presses from the buttonISR to main(). I used three buttons in this lab, S2 on the booster pack for switching the rising and falling trigger, and SW1 and SW2 on the board for changing the voltage scale as part of the extra credit.

```
// ISR for scanning and debouncing buttons
void ButtonISR(void) {
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // clear interrupt flag

    // read hardware button state
    uint32_t gpio_buttons =
            (~GPIOPinRead(GPIO_PORTJ_BASE, 0xff) & (GPIO_PIN_1 | GPIO_PIN_0)) | //EK-TM4C1294XL buttons in positions 0 and 1
            (~GPIOPinRead(GPIO_PORTH_BASE, 0xff) & (GPIO_PIN_1)) << 1 |         //BoosterPack button 1
            (~GPIOPinRead(GPIO_PORTK_BASE, 0xff) & (GPIO_PIN_6)) >> 3 |         //BoosterPack button 2
            ~GPIOPinRead(GPIO_PORTD_BASE, 0xff) & (GPIO_PIN_4);                 //BoosterPack buttons Joystick select


    uint32_t old_buttons = gButtons;    // save previous button state
    ButtonDebounce(gpio_buttons);       // Run the button debouncer. The result is in gButtons.
    ButtonReadJoystick();               // Convert joystick state to button presses. The result is in gButtons.
    uint32_t presses = ~old_buttons & gButtons;   // detect button presses (transitions from not pressed to pressed)
    presses |= ButtonAutoRepeat();      // autorepeat presses if a button is held long enough

    static bool tic = false;
    static bool running = true;

    if (presses & 1) { // EK-TM4C1294XL button 1 pressed
        fifo_put('a');
    }
    if (presses & 2) { // EK-TM4C1294XL button 2 pressed
        fifo_put('b');
        }
    if (presses & 8) { // button 8 pressed boosterpack S2 one trigger
        fifo_put('c');
    }
    if (running) {
        if (tic) gTime++; // increment time every other ISR call
        tic = !tic;
    }
}
```

Figure 12: Button ISR

The FIFO code was shown as an example during lecture, so I copied most of it from there. We talked about some shared data problems associated with it so I edited it so that these problems would not occur

```
int fifo_put(char data)
{
    int new_tail = fifo_last + 1;
    if (new_tail >= FIFO_SIZE){
        new_tail = 0; // wrap around
    }
    if (fifo_1 != new_tail) {    // if the FIFO is not full
        fifo[fifo_last] = data;     // store data into the FIFO
        fifo_last = new_tail;       // advance FIFO tail index
        return 1;                   // success
    }
    return 0;   // full
}
// get data from the FIFO
// returns 1 on success, 0 if FIFO was empty
int fifo_get(char *data)
{
    if (fifo_1 != fifo_last) {   // if the FIFO is not empty
        *data = fifo[fifo_1];    // read data from the FIFO
        if (fifo_1 + 1 >= FIFO_SIZE)
        {
            fifo_1 = 0; // wrap around
        }
        else
        {
            fifo_1++;             // advance FIFO head index
        }
        return 1;                   // success
    }
    return 0;   // empty
}
```

Figure 13: FIFO

Step 6: CPU Load Management:

In order to measure the CPU load I used the example code that was shown in lecture. A timer is configured in one shot mode and polls it to timeout, while counting iterations. If this code is being interrupted, it will count fewer iterations than if interrupts are disabled. The CPU load can be estimated from the iteration counts with interrupts enabled and disabled. Figure 14 shows how the timer was configured with a timeout interval of 10ms. Figure 15 shows the function that gets

the CPU load. And Figure 16 shows where the unload and loaded measurements are taken. My CPU utilization was 56.4%.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
TimerDisable(TIMER3_BASE, TIMER_BOTH);
TimerConfigure(TIMER3_BASE, TIMER_CFG_ONE_SHOT);
TimerLoadSet(TIMER3_BASE, TIMER_A, gSystemClock/100 - 1); //10 msec
```

Figure 14: Timer Configuration

```
uint32_t cpu_load_count(void){
      uint32_t i = 0;
      TimerIntClear(TIMER3_BASE, TIMER_TIMA_TIMEOUT);
      TimerEnable(TIMER3_BASE,TIMER_A); //start one-shot timer
      while (!(TimerIntStatus(TIMER3_BASE, false) & TIMER_TIMA_TIMEOUT)) {
           i++;
       }
        return i;
```

Figure 15: CPU load function

```
count_unloaded = cpu_load_count(); //count_unloaded
```

```
count_loaded = cpu_load_count();
cpu_load = 1.0f - (float)count_loaded/count_unloaded; //Calculate CPU utilization
```

Figure 16: Taking CPU load measurements

Extra Credit: Implement different voltage scales:

For extra credit I implemented a variable voltage scale to be drawn on the graph. (2V/div, 1V/div, 500mV/div, 200mV/div, 100mV/div). This takes the button presses from SW1 and SW2 to increase and decrease the scale.

```
while (fifo_get(&button)){ //Loop for switching button states for voltage scale and trigger slope
    switch(button){
    case 'a':
        voltsperDiv = ++voltsperDiv > 4 ? 4 : voltsperDiv++;
        break;
    case 'b':
        voltsperDiv = --voltsperDiv <= 0 ? 0 : voltsperDiv--;
        break;
    case 'c':
        triggerSlope = !triggerSlope;
        break;
```

Figure 17: Button case states

This case statement shows how the FIFO passes the button presses into main() and shows how the buttons increase and decrease the voltage scale and also change the falling and rising trigger. The voltsperDiv is a constant array that contains the respective multipliers for the voltage scales. This scale number is then multiplied by the samples stored in the local buffer giving the changed scale displayed on the screen. Since this only accesses the local buffer there are no shared data problems.

Conclusion:

I ran into a handful of problems while completing this lab. I struggled initially being able to properly configure the ADC to read the square wave signal. Most of the problems came because it was configured in the wrong places. I initially moved all of the ADC configurations to sampling.c but this created issues so I had to remake the configuration function adcInit() in sampling.c which eventually fixed the problem. Another problem I had was figuring out how to properly display the waveform on the display. I took me some time to figure out how to translate the local buffer sample data to the square wave on the screen. I was able to do this with a for loop and the given display code. I also struggled to get the missing trigger signal to display on

the screen but this was because of a syntax error where a } was missing. In this lab I learned many things but perhaps the most important being the significance of shared data bugs. In one of my first attempts at drawing the waveform I did not have a local copy of the buffer data and was met with a strange waveform. I realized that the local copy was necessary to prevent this issue. Overall, this lab was very valuable to me as I was able to see many class concepts (ISRs, FIFOs, Shared Data bugs, scheduling, etc.) and how they were practically used to create a system. I believe this lab does a solid job with these topics albeit a bit difficult for the first lab.