ECE 3849 – Laboratory Report

Real - Time Embedded Systems

Worcester Polytechnic Institute

D - Term 2021

Lab Section: D01

Laboratory 1

Digital Oscilloscope

Submitted by:

Jonathan R. Lopez

ECE Mailbox #178

4/12/2021

Professor: Jennifer Stander

Table of Contents ***Page #***

Introduction:

In this lab, we will be making a 1Msps oscilloscope using the material stated above. In addition, we will be implementing the features such as CPU load percentage, rising/falling edge triggers and implementing different voltage scales.

Materials:

Texas Instruments EK-TM4C1294XL
Texas Instruments Educational BoosterPack MKII (BOOSTXL-EDUMKII)
Jumper wire (x1)
Micro-USB to USB-A cable
Computer with Code Composer Studio ver. 9.3
Computer with TI-RTOS installed

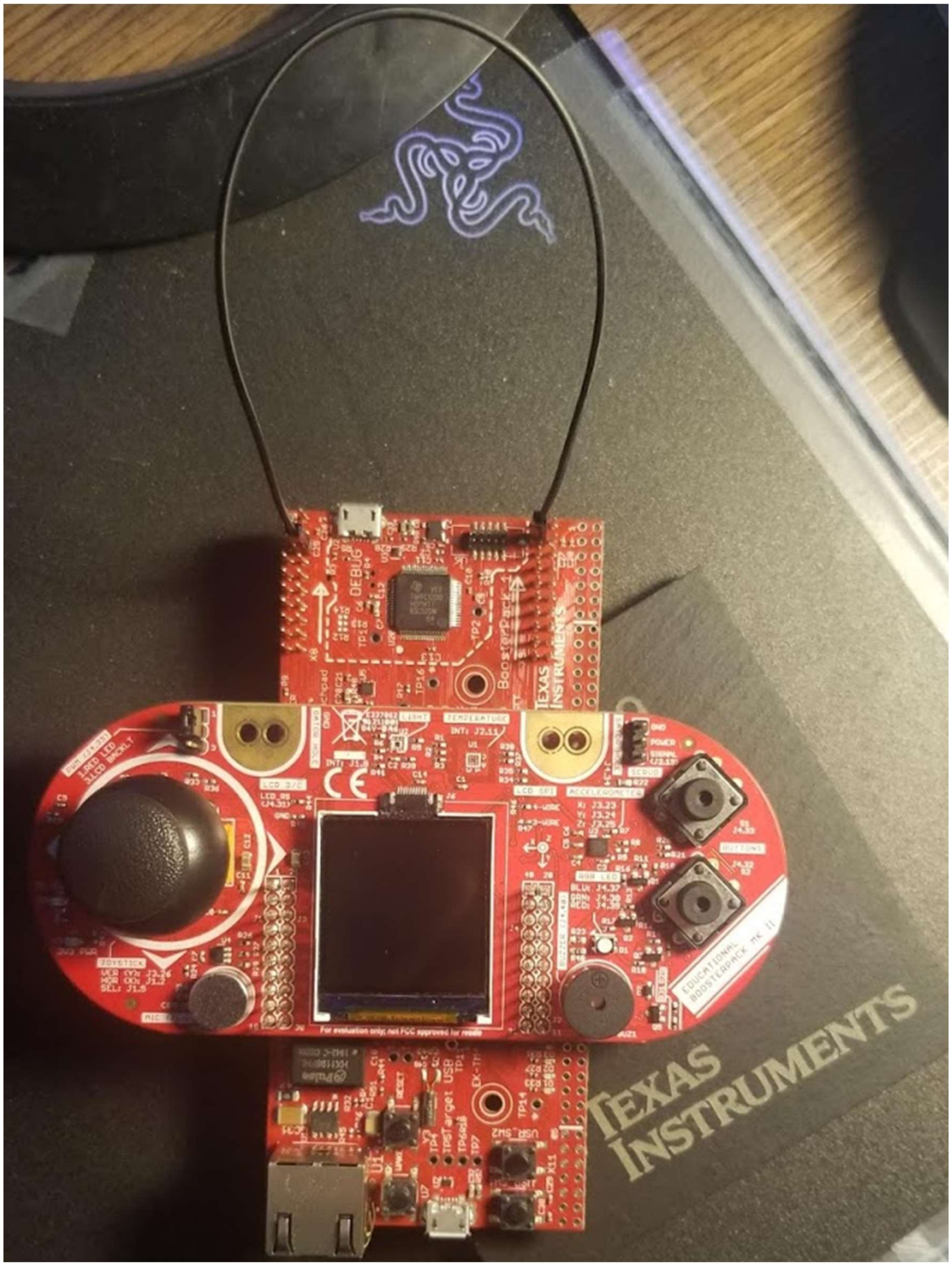


Figure 1: Board setup

Discussion and Results:

Lab 1: ECE 3849: Real-Time Embedded Systems

Digital Oscilloscope

After copying the Lab 0 files we are ready to start Lab 1

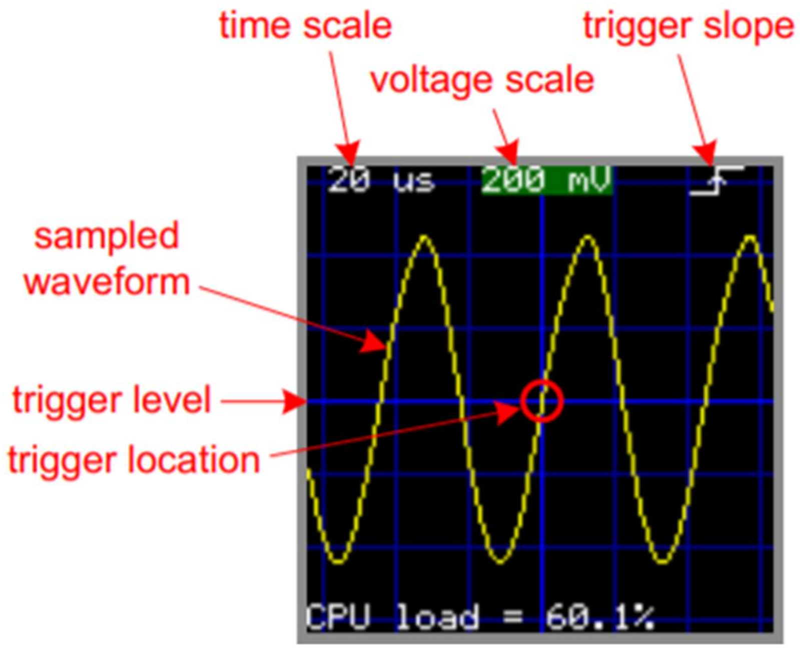


Figure 2: Example of the screen layout

Figure 2 shows an example of what the screen of the Educational BoosterPack MKII should look like. In our lab, we will be using a PWM signal and it will be a square wave and not a sine wave.

For the base time scale, we will make a 20μs/div scale.
For the base voltage scale, we will make a 1V/div scale.

Figure 3 shows the software block diagram for this lab

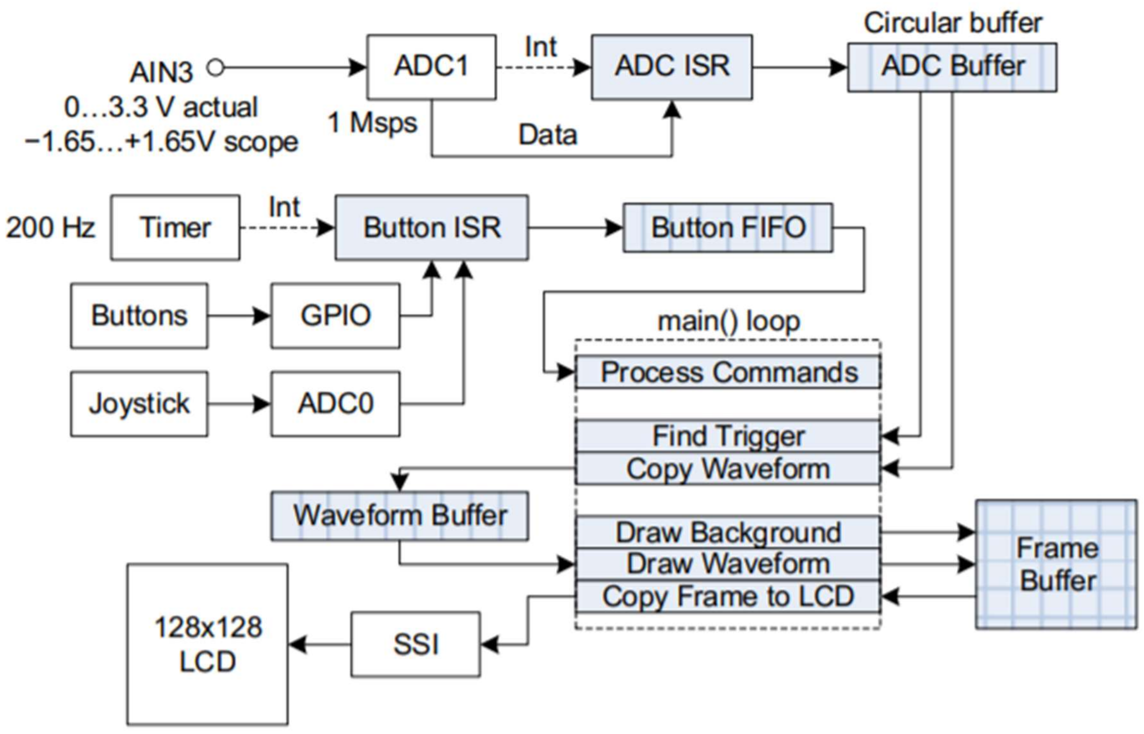


Figure 3: Software Block Diagram

<u>Step 1: Source Voltage</u>

As for the source of the waveform, we will be using one of the onboard sources from the TM4C1294XL Launchpad board. This will go through an Analog to digital Converter (ADC) to sketch the waveform. The range of this will be from 0-3.3V. It will be centered around the center of the screen for a full range of +1.65V and -1.65V. Figure 4 shows what pins we need to connect. Figure 1 shows the pins connected with a jumper wire. Figure 5 shows the code to set up the input signal.

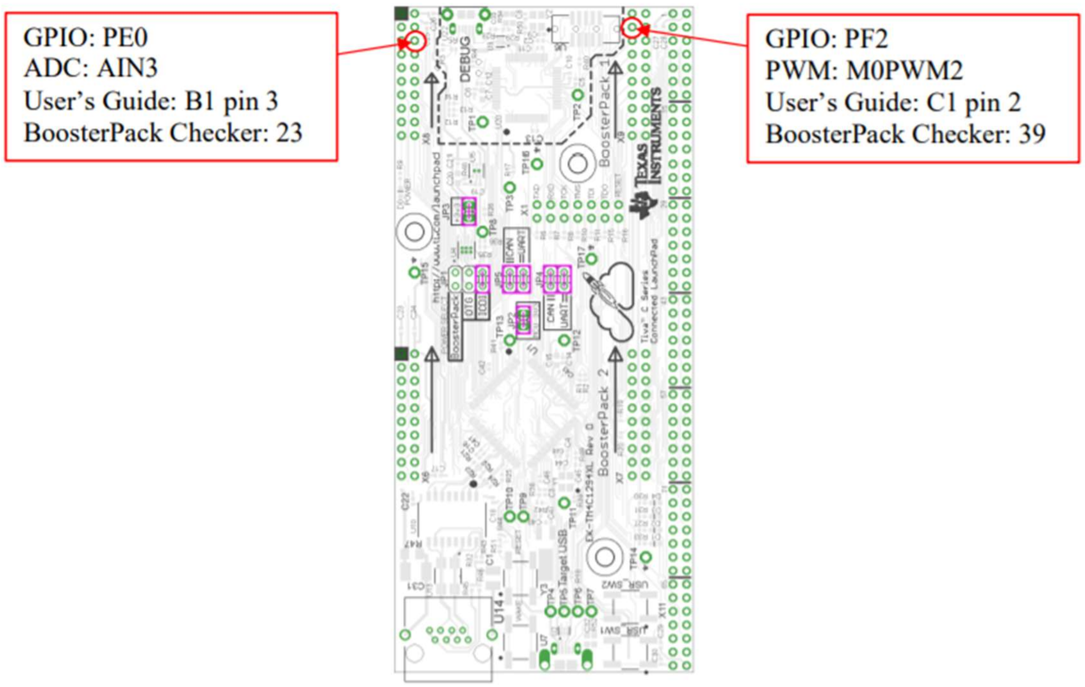
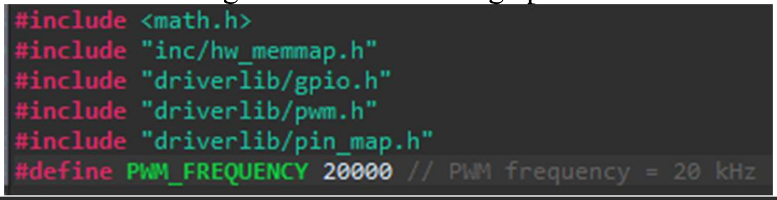
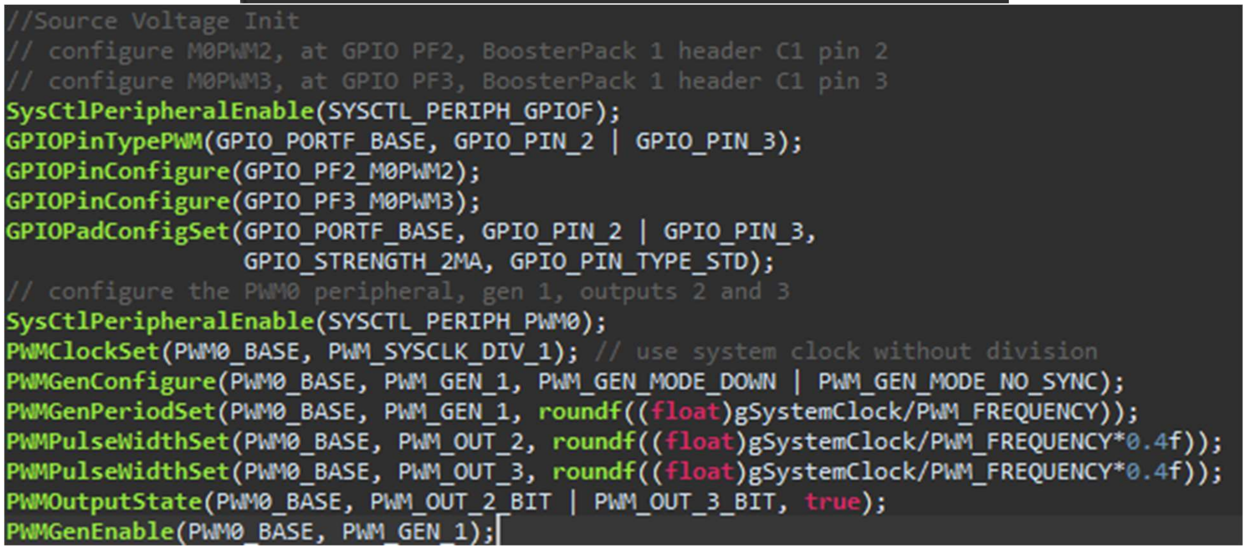


Figure 4: Source Voltage pins



```
#include <math.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pwm.h"
#include "driverlib/pin_map.h"
#define PWM_FREQUENCY 20000 // PWM frequency = 20 kHz
```



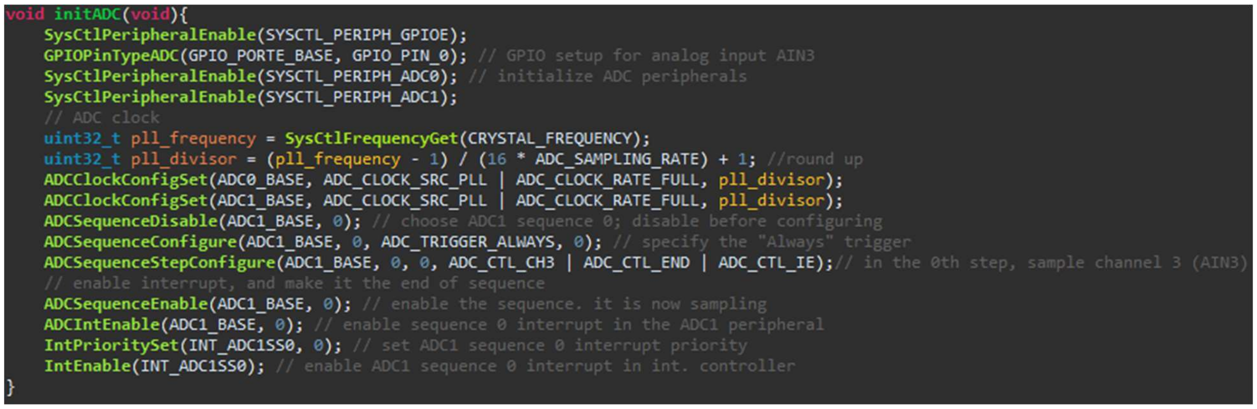
```
//Source Voltage Init
// configure M0PWM2, at GPIO PF2, BoosterPack 1 header C1 pin 2
// configure M0PWM3, at GPIO PF3, BoosterPack 1 header C1 pin 3
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);
GPIOPinConfigure(GPIO_PF2_M0PWM2);
GPIOPinConfigure(GPIO_PF3_M0PWM3);
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 5: Source Setup

Step 2: Sampling.c and Sampling .h

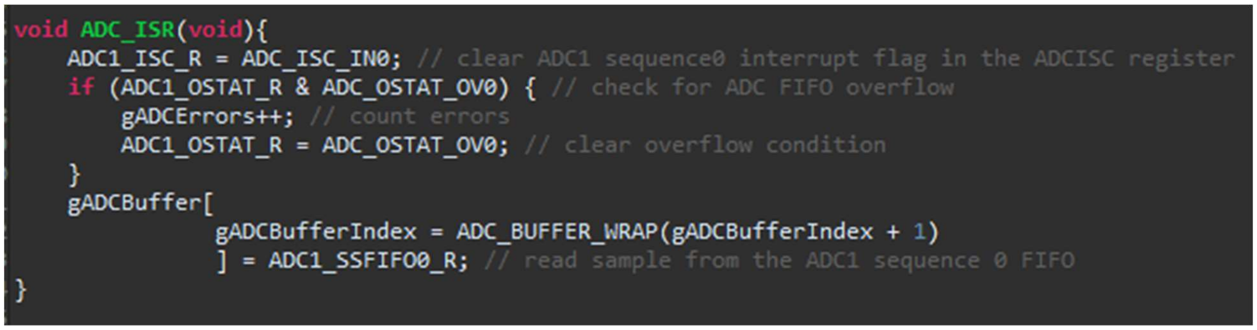
A new .c and .h files are needed to take in the samplings of the waveform to display. In order to keep the code neat we will put this in its own compartment. (Its own .c and .h files). The defines and function prototypes will be in the .h file and the function definitions will be in the .c file. We will be taking in 1,000,000 sampling per second.

Using the various data sheets on Canvas as well as the lab write up, I was able to set up ADC1 as shown in Figure 6 and Figure 7 shows the ADC_ISR function all filled out. Lastly Figure 8 shows the waveform in CCS.



```
void initADC(void){
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0); // GPIO setup for analog input AIN3
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0); // initialize ADC peripherals
    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);
    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_END | ADC_CTL_IE);// in the 0th step, sample channel 3 (AIN3)
    // enable interrupt, and make it the end of sequence
    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 6: initADC(void);



```
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 7: ADC_ISR (void);

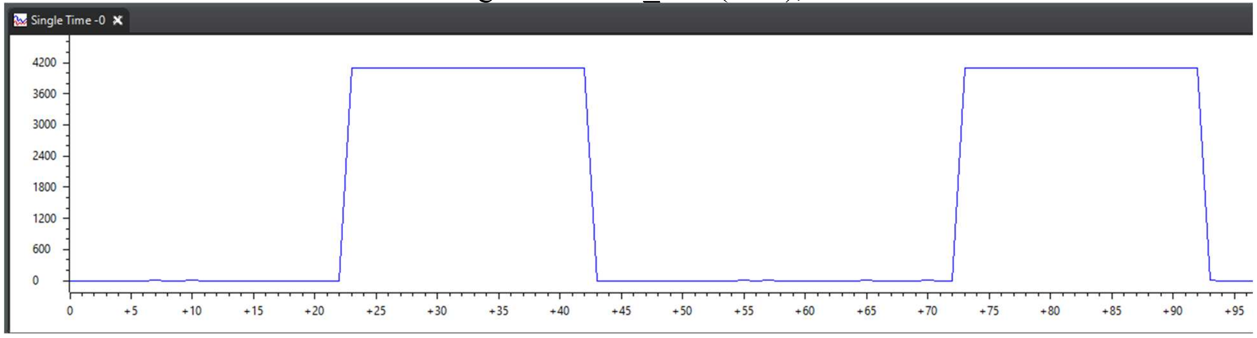


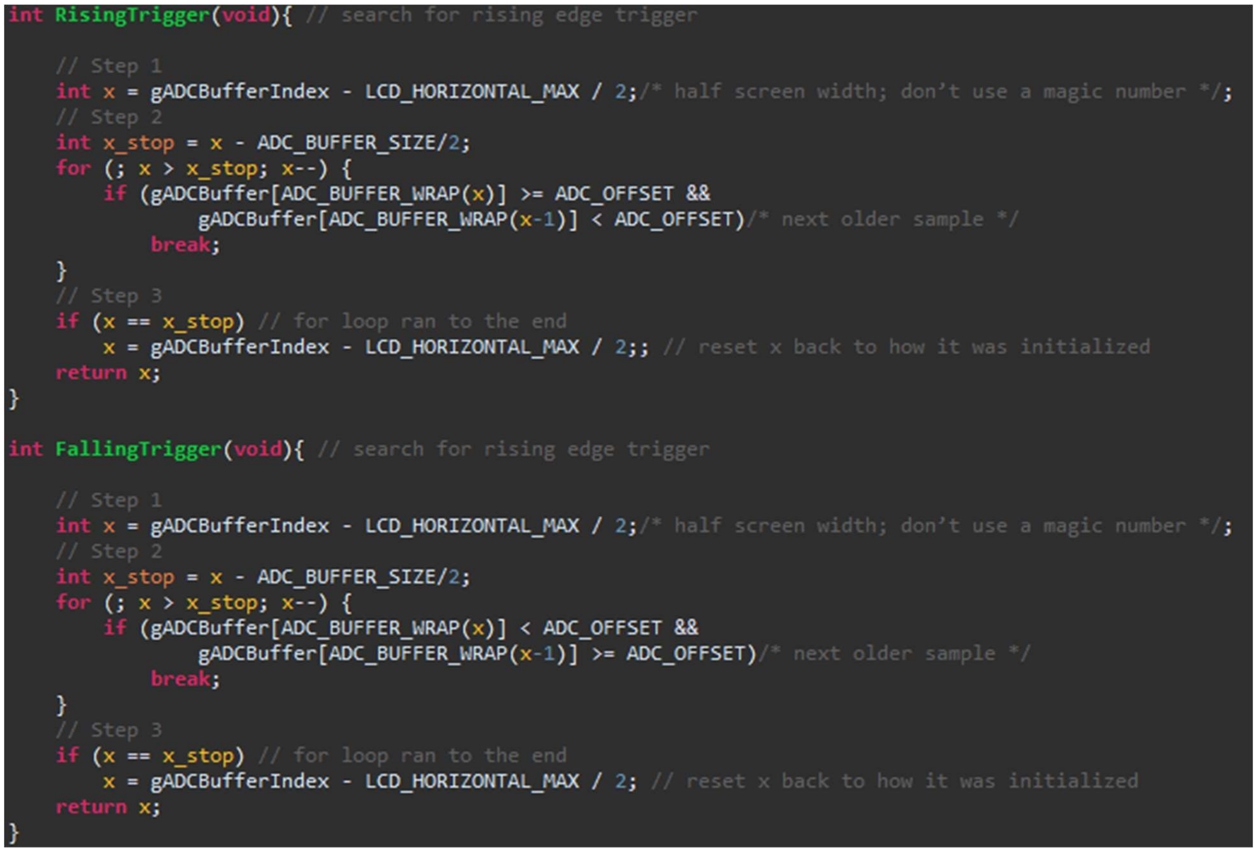
Figure 8: Square Wave

Step 3: Trigger Search

Triggering. The trigger level and location will be in the center of the screen and with the press of a button the trigger level will change from rising to falling. This means when the square wave is rising the rising edge will be at the center of the screen. In falling trigger mode, the falling edge of the square wave will be at the center of the screen. In this lab, we will make the trigger 0V. When 0V (falling or rising) is where it meets the trigger level and the trigger location. When the button is pressed it is passed from an ISR to main() via a FIFO (First In, Last Out).

Displayed on the screen will be a trigger slope icon as shown in Figure 1. The up arrow will indicate rising edge. A down arrow will indicate a falling edge.

The samplings are stored in the gADCBuffer. We will take the samples and conform them to a trigger as explained above. We will ignore the rest of the sampling that we don't need.
Trigger searching has led me to make 2 functions. RisingTrigger(); and FallingTrigger();
These are shown in Figure 9.


```
int RisingTrigger(void){ // search for rising edge trigger

    // 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
    return x;
}

int FallingTrigger(void){ // search for rising edge trigger

    // 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
    return x;
}
```
Figure 9: Rising and Falling Trigger Functions

Step 1 is to use half the screen width

Step 2 is to go back through the buffer to find where the buffer waveform crosses the trigger level

Step 3 is if it can't find the trigger in half the buffer, then stop and give up. We only search half because the other is reserved to be over written by the ISR. We will set the trigger index to its initial location and display the newest samples unsynchronized.

Step 4 is to copy the samples from half a screen behind to half a screen ahead of the trigger index from the gADCBuffer to a local buffer.

We will manipulate the circular buffer index by wrapping macro ADC_BUFFER_WRAP();
It accepts a positive or negative index and returns an index properly wrapped into its valid range of gADCBuffer. This will be in sizes of powers of 2. This speeds up and simplifies the wrapping operation to a very fast bitwise AND.

Trigger search is a low priority event. The only time constraint it has is that it should find the trigger before the ADC_ISR circles back around and overwrites the sample in the range.

This goes into shared data. If we try to draw the waveform from gADCBuffer the ISR will likely overwrite the samples its drawing. Drawing is a very slow operation. This will likely cause discontinuous waveforms, waveforms not crossing the trigger at the desired location, and unsteady waveforms. Copying the part, we want into a local buffer is fast operation and therefore we don't need to worry about the ISR interrupting our drawing from the global buffer.

Shared data must be treated with care. Atomic operations only. gADCBuffer and gADCBufferIndex are shared between the ADC_ISR and main(). In this case of the timing with the ADC_ISR, we can't disable interrupts. We will miss the samplings that we can't miss. Therefore, we will read from gADCBuffer. This is an atomic operation and pose no threat to the ISR. We will also read from half the buffer while thew ISR writes to the other half.

<u>Step 4: ADC Sampling Scale</u>

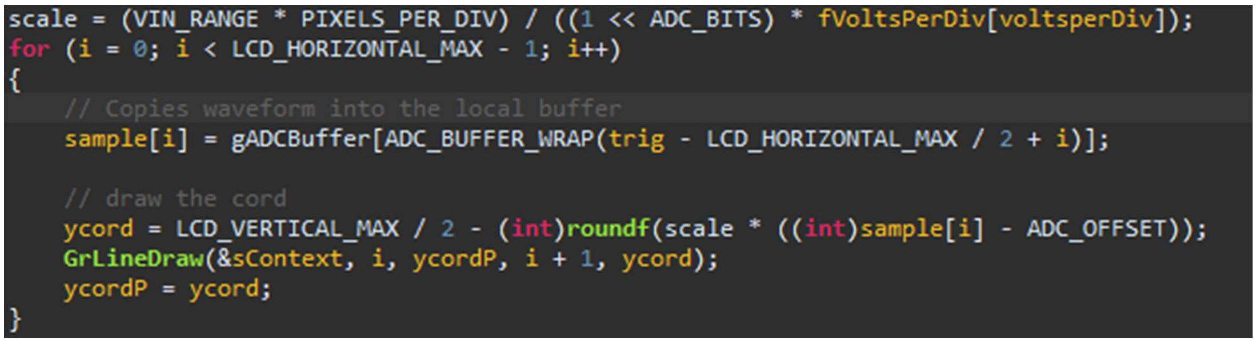
We have to conform the ADC samples such that they conform to the Volts/division scale.
I put all the variable that stayed the same in #define statements so they won't change. Figure 10 shows how I conformed the samples to a voltage scale.

The scale id the Vin range *pixels per division divided by the ADC resolution bit shifter over 2 * the volts per division array with it being set to 2V/div right now.

The for loops goes into drawing the waveform.

For the horizontal length, it reads from the local buffer and draws the coordinate. 1 pixel is 1 sample interval.

Finally setting the previous y coordinate.



```
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 cord
    ycord = LCD_VERTICAL_MAX / 2 - (int)roundf(scale * ((int)sample[i] - ADC_OFFSET));
    GrLineDraw(&sContext, i, ycordP, i + 1, ycord);
    ycordP = ycord;
}
```

Figure 10: ADC Sampling Conformation

Step 5: Waveform Display Formatting

The rest of the displaying is shown in Figure 11:

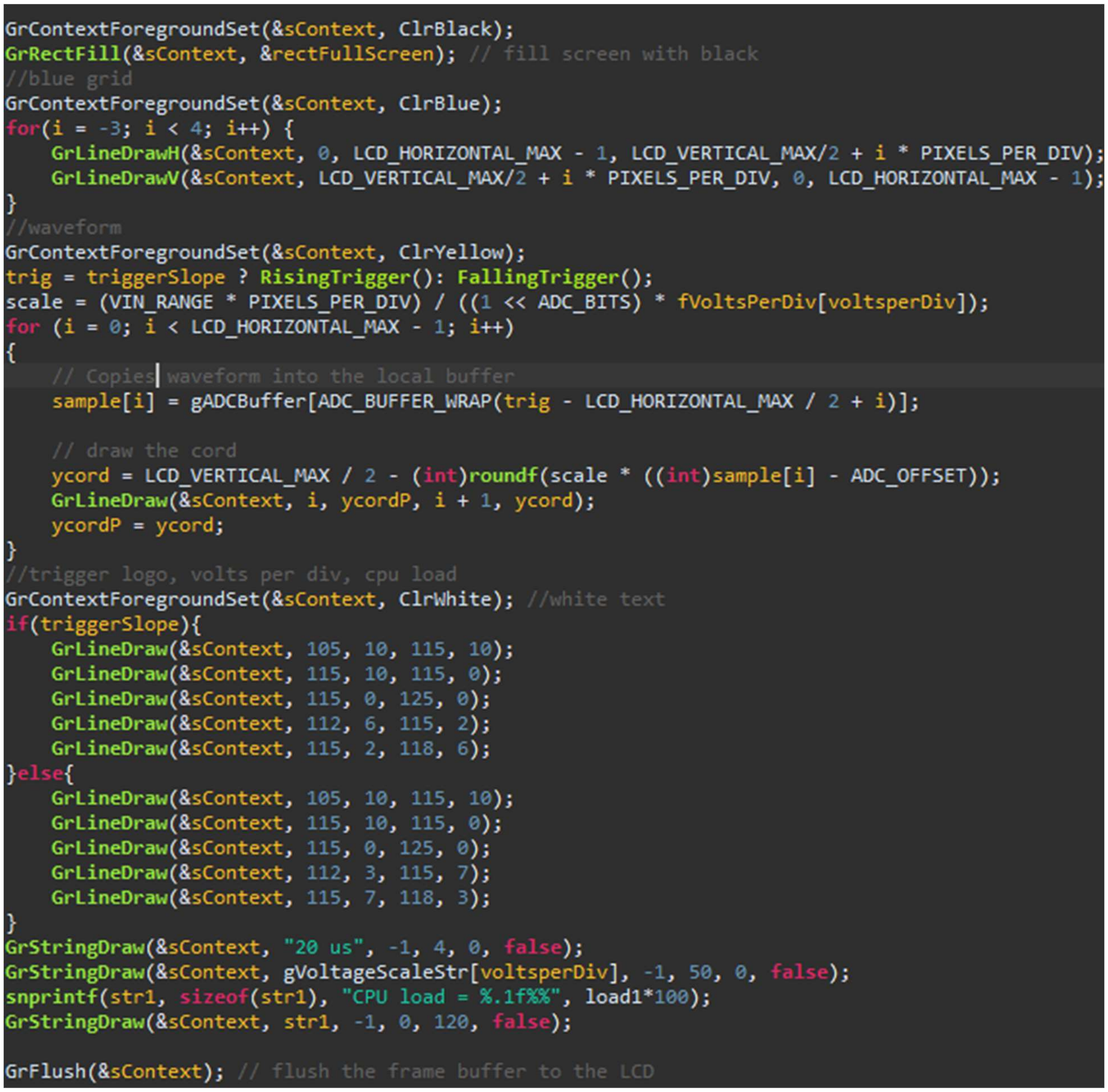
Starting from the top we first fill the screen with black like in lab 0

Then we introduce the blue grid lines

The waveform is then displayed as shown in Section 4: ADC Sampling code.

Next, draw the trigger logo as shown in Figure 2, the time scale, voltage scale and the CPU load percentage. This will be discussed in step 6.

Lastly, we flush all this to the screen.



```
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 cord
    ycord = LCD_VERTICAL_MAX / 2 - (int)roundf(scale * ((int)sample[i] - ADC_OFFSET));
    GrLineDraw(&sContext, i, ycordP, i + 1, ycord);
    ycordP = ycord;
}
//trigger logo, 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);
}
GrStringDraw(&sContext, "20 us", -1, 4, 0, false);
GrStringDraw(&sContext, gVoltageScaleStr[voltsperDiv], -1, 50, 0, false);
snprintf(str1, sizeof(str1), "CPU load = %.1f%%", load1*100);
GrStringDraw(&sContext, str1, -1, 0, 120, false);

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

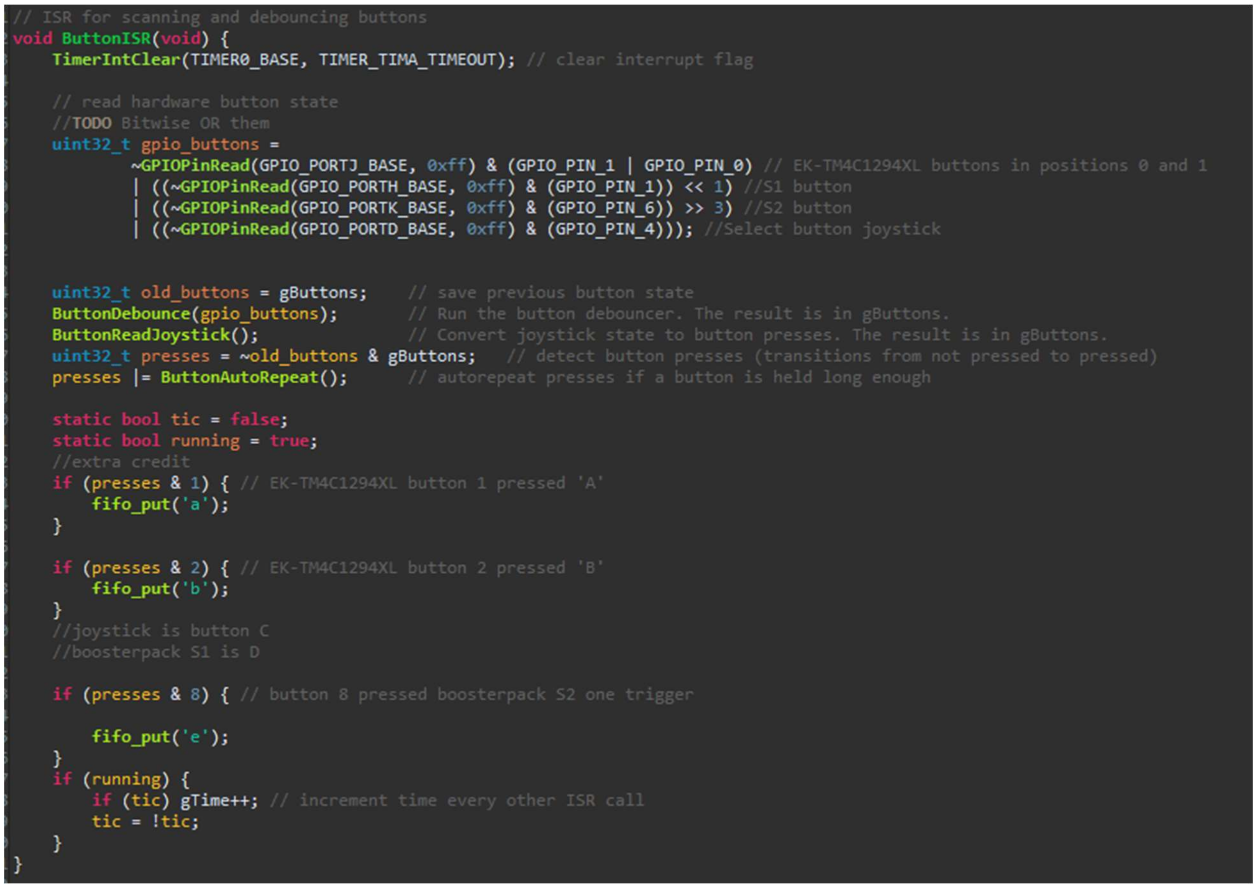
Figure 11: Displaying Information

Step 5a: Button Command Processing

For the button handling I decided to label the buttons A, B, C, D, E for simplicity.

A and B are the 2 buttons on the TM4C1294XL board SW 2 and SW1 respectively (Up to Down)
C is the Joystick button
D and E are the 2 buttons on the BoosterPack S1 and S2 respectively (Up to Down)
We also need a char to pass through to the fifo_put function.

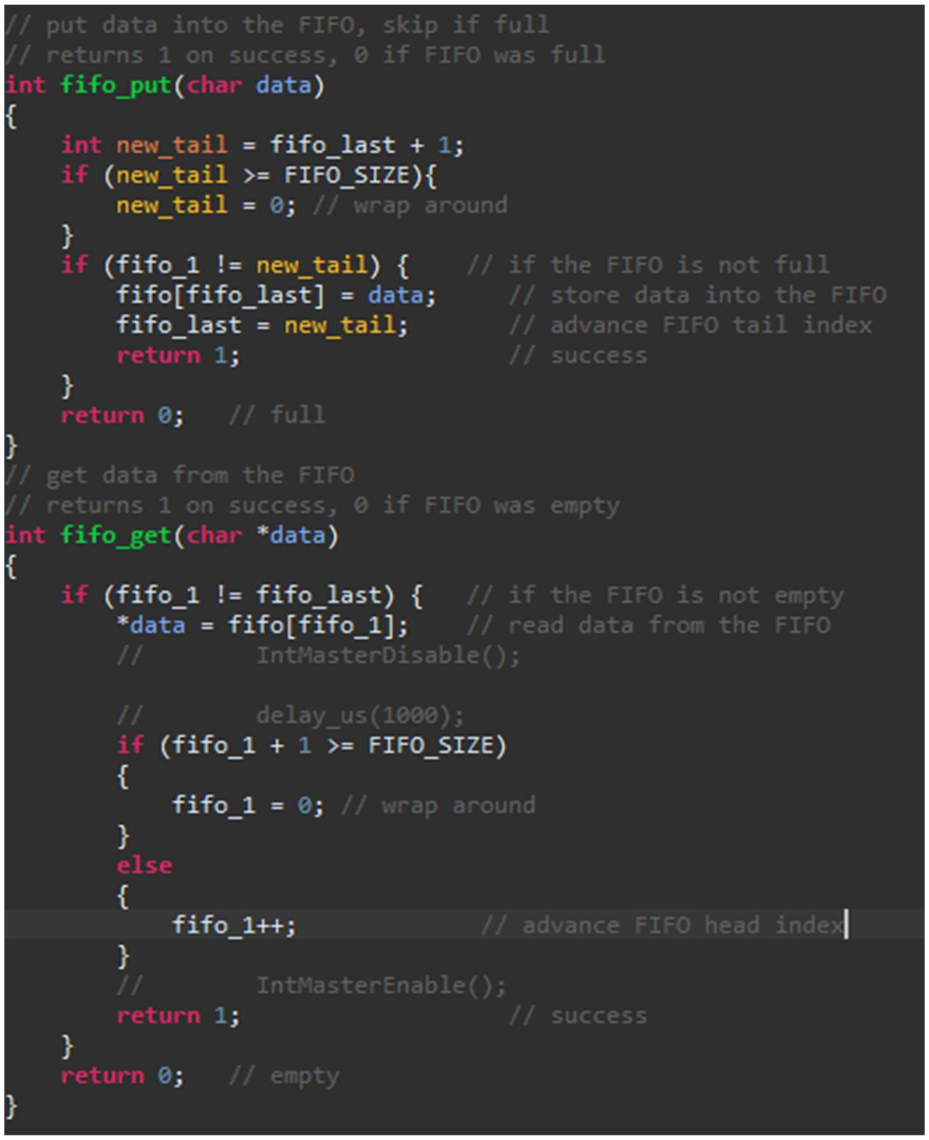
Figure 12 shows the button ISR
Figure 13 shows the fifo_put and fifo_get functions



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

    // read hardware button state
    //TODO Bitwise OR them
    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) //S1 button
            | ((~GPIOPinRead(GPIO_PORTK_BASE, 0xff) & (GPIO_PIN_6)) >> 3) //S2 button
            | ((~GPIOPinRead(GPIO_PORTD_BASE, 0xff) & (GPIO_PIN_4))); //Select button joystick


    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;
    //extra credit
    if (presses & 1) { // EK-TM4C1294XL button 1 pressed 'A'
        fifo_put('a');
    }

    if (presses & 2) { // EK-TM4C1294XL button 2 pressed 'B'
        fifo_put('b');
    }
    //joystick is button C
    //boosterpack S1 is D

    if (presses & 8) { // button 8 pressed boosterpack S2 one trigger

        fifo_put('e');
    }
    if (running) {
        if (tic) gTime++; // increment time every other ISR call
        tic = !tic;
    }
}
```

Figure 12: ButtonISR(void)

I copied them from the shared-data project that we went over in class. And updated the data type that we needed. I also updated so that it wraps around and avoid some shared data bugs that were associated with that process. I also commented out the disabled interrupts and reenabling them.

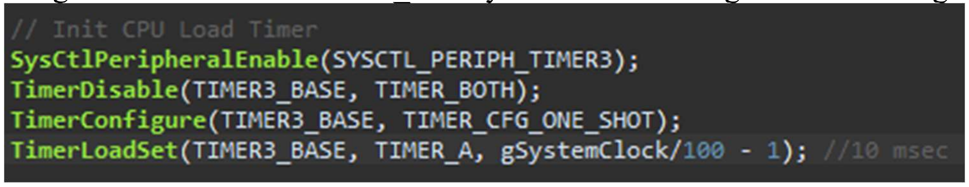


```
// put data into the FIFO, skip if full
// returns 1 on success, 0 if FIFO was full
int fifo_put(char data)
{
    int new_tail = fifo_last + 1;
    if (new_tail >= FIFO_SIZE){
        new_tail = 0; // wrap around
    }
    if (fifo_1 != new_tail) {    // if the FIFO is not full
        fifo[fifo_last] = 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
        //        IntMasterDisable();

        //        delay_us(1000);
        if (fifo_1 + 1 >= FIFO_SIZE)
        {
            fifo_1 = 0; // wrap around
        }
        else
        {
            fifo_1++;                // advance FIFO head index
        }
        //        IntMasterEnable();
        return 1;                   // success
    }
    return 0;   // empty
}
```

Figure 13: fifo_put and fifo_get.

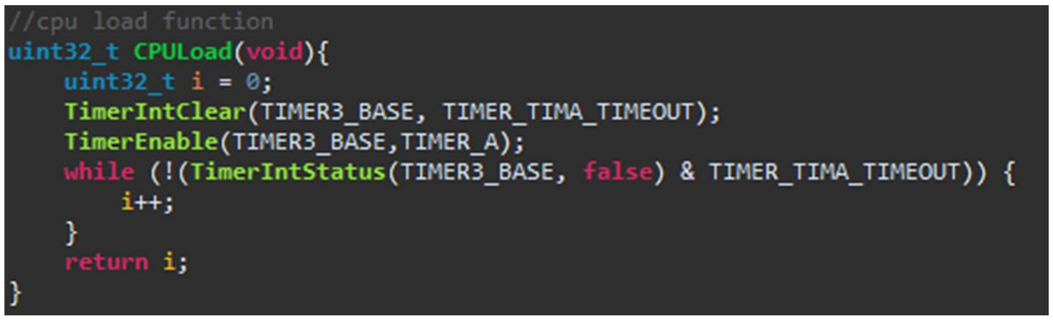
Step 6: CPU Load Measurement

After consulting the data sheet and the int_latency the load timer is give below in Figure 14



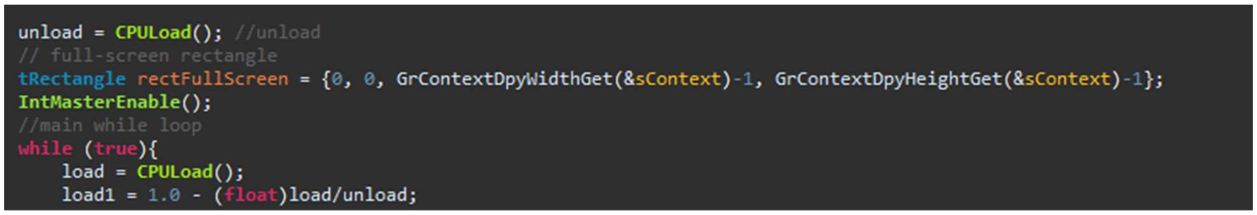
```
// Init CPU Load Timer
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
TimerDisable(TIMER3_BASE, TIMER_BOTH);
TimerConfigure(TIMER3_BASE, TIMER_CFG_ONE_SHOT);
TimerLoadSet(TIMER3_BASE, TIMER_A, gSystemClock/100 - 1); //10 msec
```

Figure 14: Load Timer



```
//cpu load function
uint32_t CPULoad(void){
    uint32_t i = 0;
    TimerIntClear(TIMER3_BASE, TIMER_TIMA_TIMEOUT);
    TimerEnable(TIMER3_BASE,TIMER_A);
    while (!(TimerIntStatus(TIMER3_BASE, false) & TIMER_TIMA_TIMEOUT)) {
        i++;
    }
    return i;
}
```

Figure 15: CPU Load Function



```
unload = CPULoad(); //unload
// full-screen rectangle
tRectangle rectFullScreen = {0, 0, GrContextDpyWidthGet(&sContext)-1, GrContextDpyHeightGet(&sContext)-1};
IntMasterEnable();
//main while loop
while (true){
    load = CPULoad();
    load1 = 1.0 - (float)load/unload;
```

Figure 16: taking measurements.

We take the unload measurement in the main but before the while(1) loop and before interrupts are enabled again.

The load is taken in the while loop and when interrupts are enabled. The load percentage is one minus the load value over the unload value. Then multiplied by 100 to get it into percentage.

Figure 15 shows the function to get CPU load.
Figure 16 shows where in main() we take the unload and load measurements
Figure 11 shows how the CPU load is displayed. (towards the bottom)

The CPU load percentage will be displayed on the Oscilloscope screen as well as grid lines and the volts/div scales and the time scales.

My CPU utilization is 56.4%

Extra Credit:

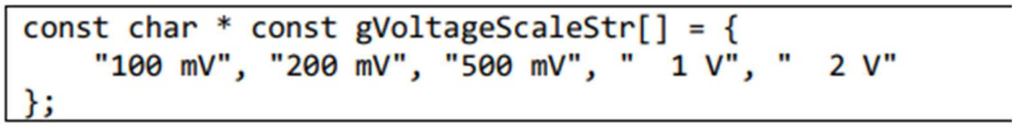
For extra credit we will implement different voltage scales. (2 V/div, 1 V/div, 500 mV/div, 200 mV/div and 100 mV/div)

This will be triggered by a different button. Like the triggering. The ISR will be passed through main via FIFO. The new scale number will be displayed on the screen.

The CPU load percentage will be displayed on the Oscilloscope screen as well as grid lines and the volts/div scales and the time scales.

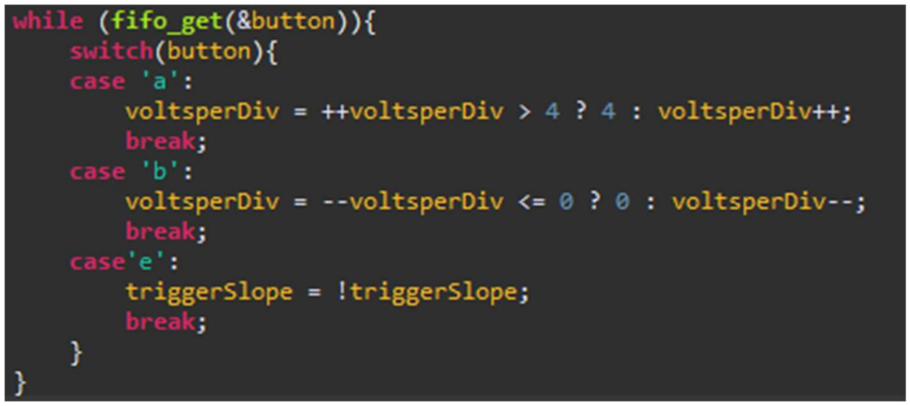
I will use the A and B buttons I mentioned earlier in Step 5. SW2 and SW1 respectively on the TM4C1294XL Launchpad Board.

By default, it is set to a 2V/div scale. SW2 will decrease the scale to 1V, 500mV, 200mV, and lastly 100mV. SW1 will increase the scale from the lower scales. Maximum is 2V and minimum is 100mV.



```
const char * const gVoltageScaleStr[] = {
    "100 mV", "200 mV", "500 mV", "  1 V", "  2 V"
};
```

Figure 17: Voltage char for different scales.



```
while (fifo_get(&button)){
    switch(button){
    case 'a':
        voltsperDiv = ++voltsperDiv > 4 ? 4 : voltsperDiv++;
        break;
    case 'b':
        voltsperDiv = --voltsperDiv <= 0 ? 0 : voltsperDiv--;
        break;
    case'e':
        triggerSlope = !triggerSlope;
        break;
    }
}
```

Figure 18: Button Case Statement

When SW2 is hit it checks to see if if's reach 4 which is the maximum. If not, then increase the index of the char array by 1 (voltsperDiv) and if it reaches 4 keep it 4. Can't go any bigger.

Similar with the button B. I check to see if it reaches 0 which is the minimum. If it's no 0 it decreases the index of the char array by 1 (voltsperDiv) and if it reaches 0 keep it 0. Can't go any smaller.

This then multiplies the samplings in the local buffer of the wave to make it either bigger or smaller on the screen when drawing it. This doesn't touch the global variable buffer at all. Therefore, not creating any shared data bugs.

This is shown in Figure 11 with drawing the y coordinate and displaying the cord at the end.

Screenshots:

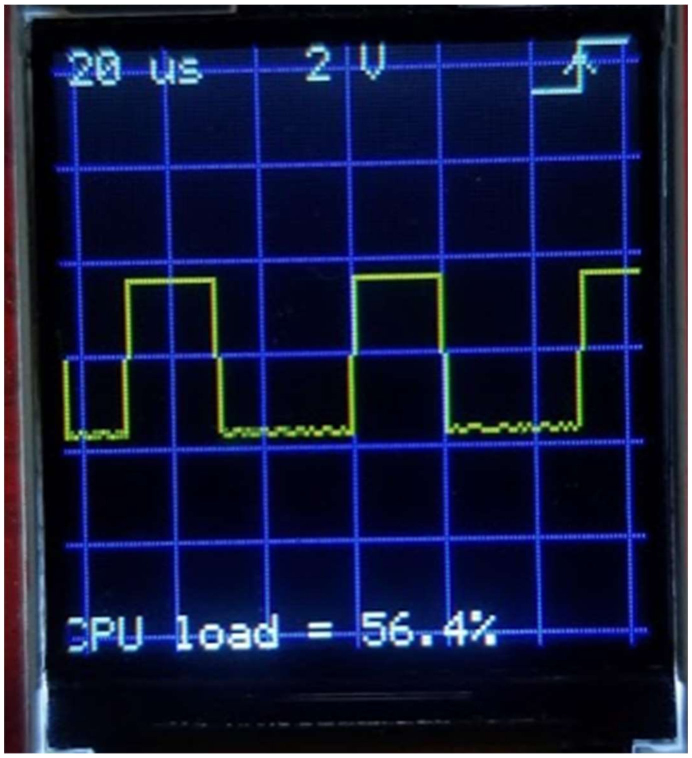


Figure 19: Rising Trigger 2V/div

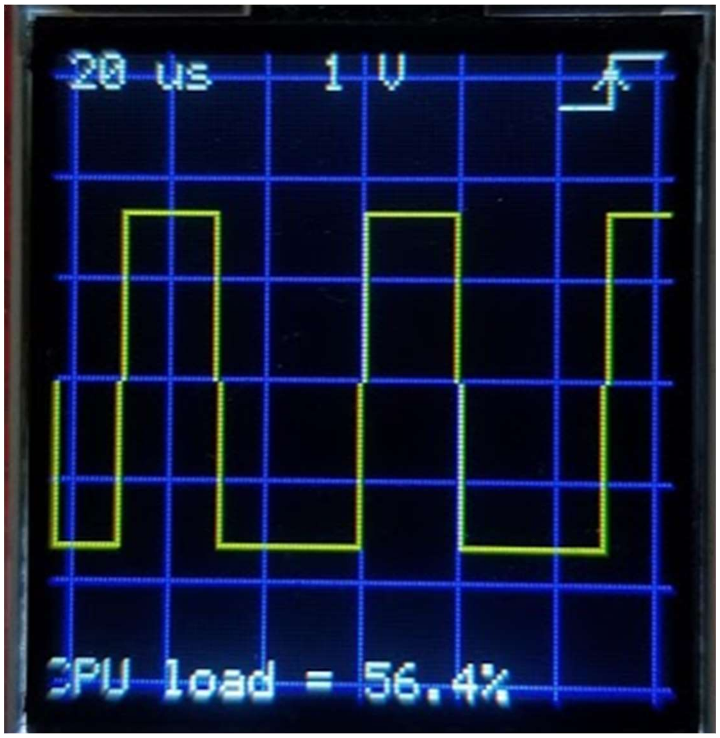


Figure 20: Rising Trigger 1V/div

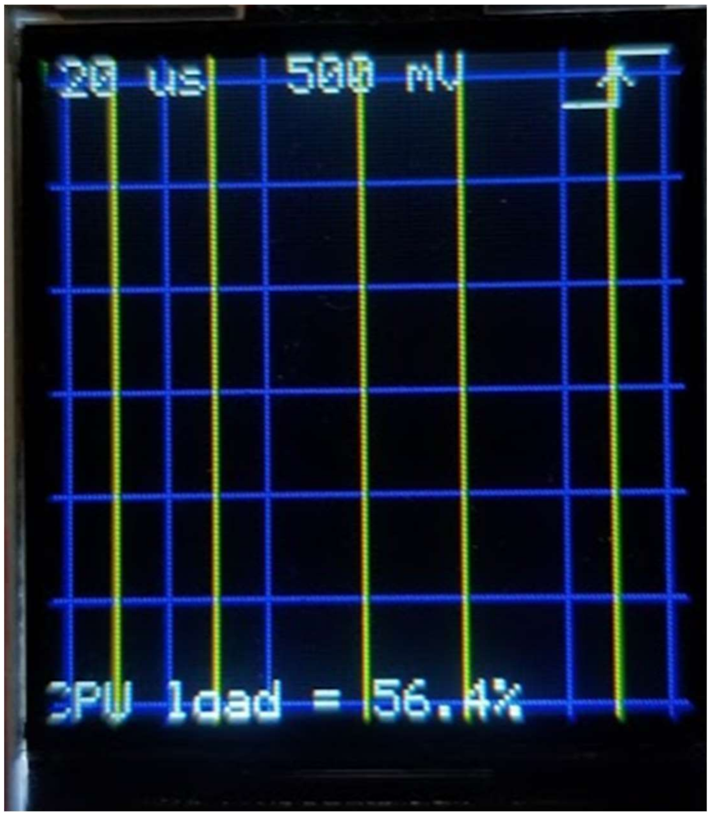


Figure 21: Rising Trigger 500mV/div

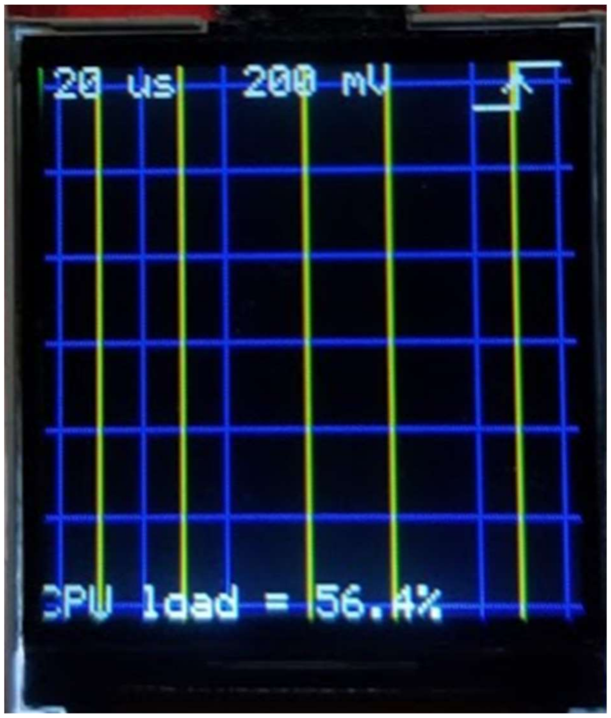


Figure 22: Rising Trigger 200mV/div

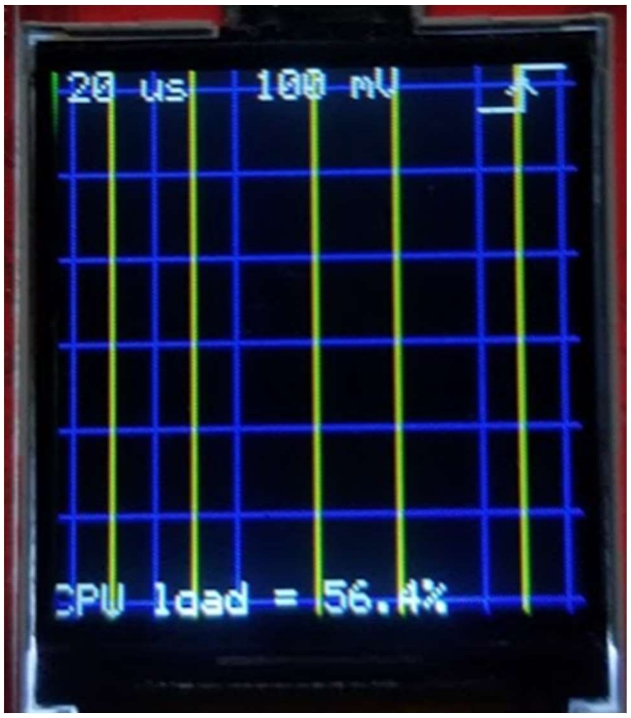


Figure 23: Rising Trigger 100mV/div

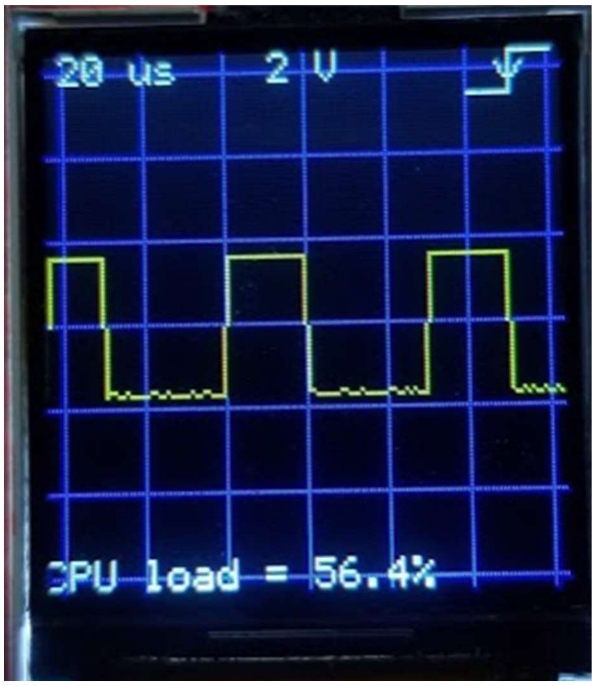


Figure 24: Falling Trigger 2V/div

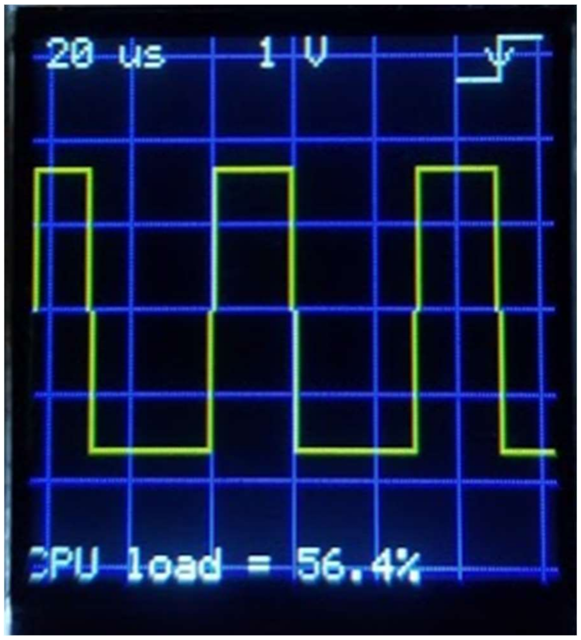


Figure 25: Falling Trigger 1V/div

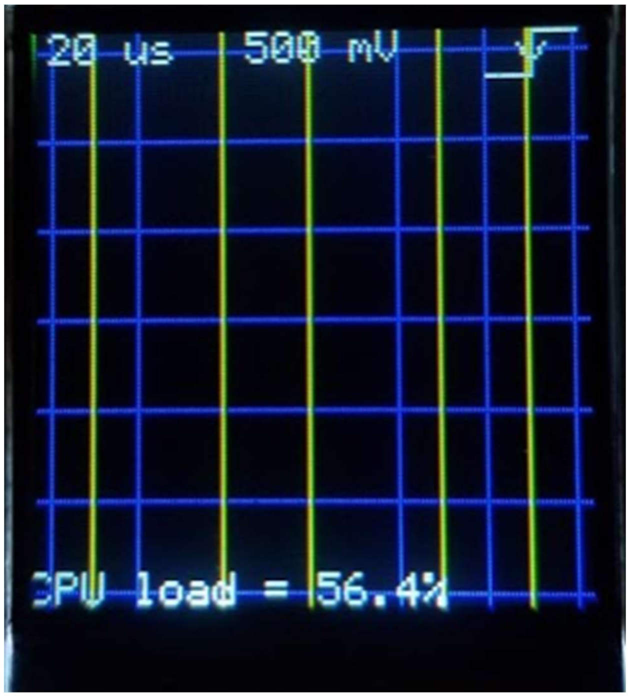


Figure 26: Falling Trigger 500mV/div

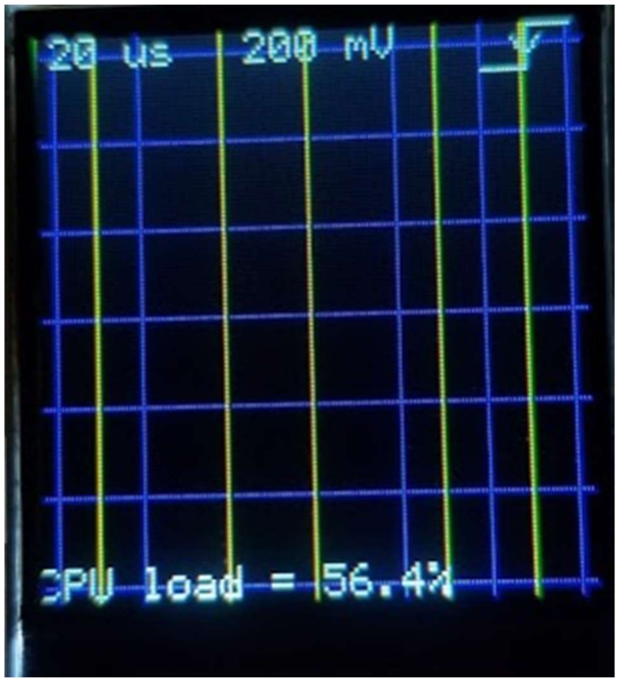


Figure 27: Falling Trigger 200mV/div

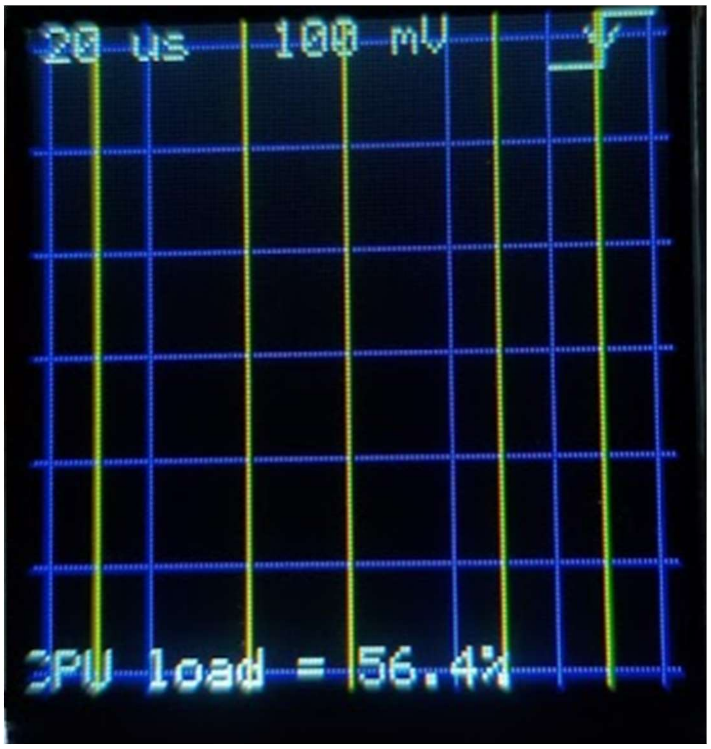


Figure 28: Falling Trigger 100mV/div

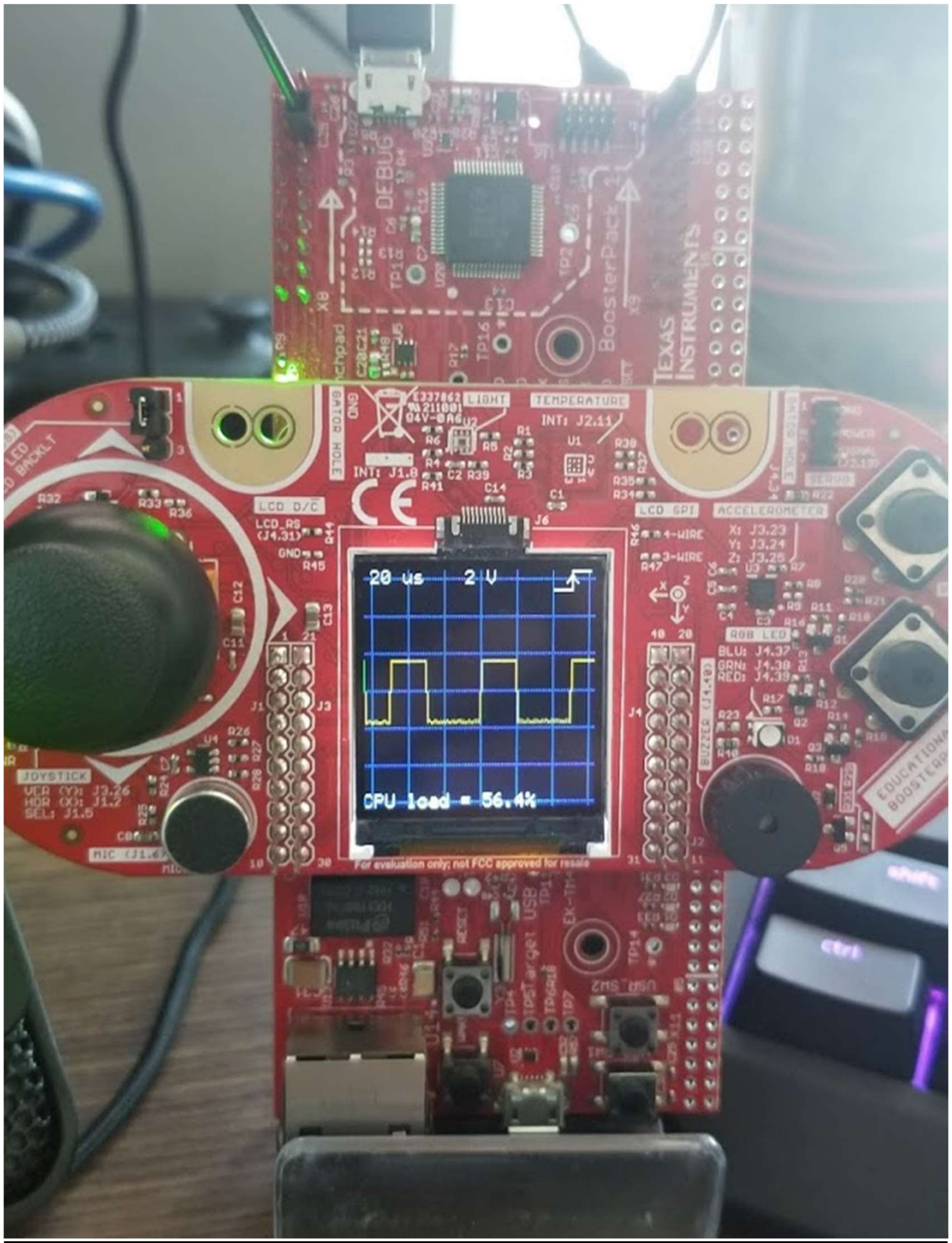


Figure 29: Whole board

Conclusion:

In conclusion. This lab was a lot, but it applied everything we learned in class. Shared data, latency and scheduling as well as disabling interrupts and fixing shared data bugs. Atomic operations and increasing my C programming skills. This concludes this laboratory project with making a basic digital oscilloscope.

Archive project file is submitted on Canvas
Signed off: Wednesday, 4/14/21 (During Simon's help session)

Appendix:

TivaWare Graphics Library User Guide:
https://canvas.wpi.edu/courses/24014/files/3630516?module_item_id=565672

TivaWare Peripheral Driver Library:
https://canvas.wpi.edu/courses/24014/files/3630518?module_item_id=565670

TM4C1294NCPDT Datasheet:
https://canvas.wpi.edu/courses/24014/files/3630515?module_item_id=565673

ECE 3849 ARM Assembly PDF:
https://canvas.wpi.edu/courses/24014/files/3630509?module_item_id=565667

ARM Cortex-M4 Technical Reference Manuel:
https://canvas.wpi.edu/courses/24014/files/3630511?module_item_id=565669

BIOS User Guide:
https://canvas.wpi.edu/courses/24014/files/3630510?module_item_id=565668

BOOSTXL-EDUMKII User Guide
https://canvas.wpi.edu/courses/24014/files/3630512?module_item_id=565666

EK-TM4C1294XL Launchpad User Guide:
https://canvas.wpi.edu/courses/24014/files/3630513?module_item_id=565671

Lab 1 Write Up:
https://canvas.wpi.edu/courses/24014/files/3685199?module_item_id=580282

int_latency program:
https://canvas.wpi.edu/courses/24014/files/3666274?module_item_id=580160

shared_data program:
https://canvas.wpi.edu/courses/24014/files/3685202?module_item_id=580156

lab1 source fragments
https://canvas.wpi.edu/courses/24014/files/3685201?module_item_id=580155

Lab 0 starter file
https://canvas.wpi.edu/courses/24014/files/3653407?module_item_id=573570

Lab Materials:
https://canvas.wpi.edu/courses/24014/files/3535924?module_item_id=565422

Pinout pdf:
https://canvas.wpi.edu/courses/24014/files/3630727?module_item_id=565431

TI Pinout website:
https://dev.ti.com/bpchecker/#/