ECE 3849 D2022
Real-Time Embedded Systems
Lab 1: Digital Oscilloscope

Adam Grabowski, Michael Rideout
April 4, 2022

Introduction

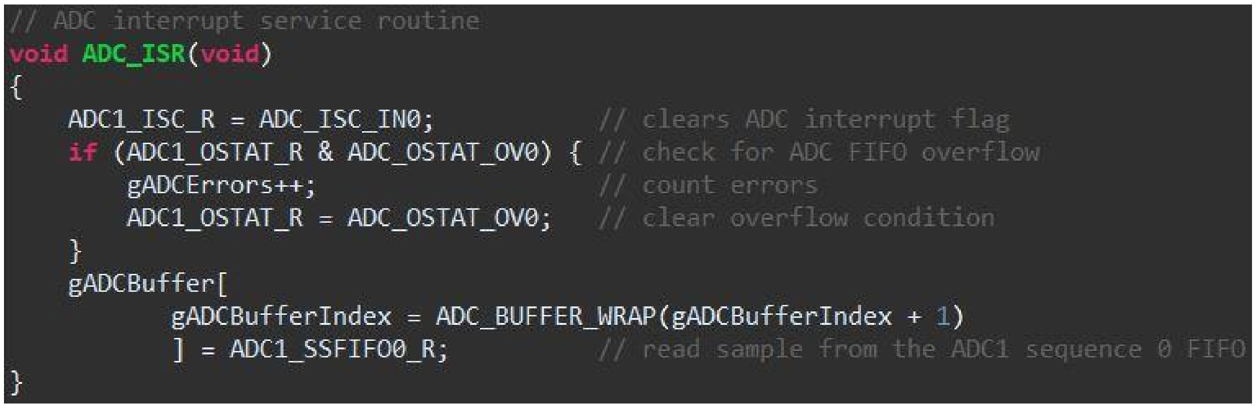
In this lab, we implemented a 1 mega sample per second (Msps) digital oscilloscope using our ECE 3849 lab kits. We attached an built-in oscillator circuit to the analog input on the board so that the waveform could be sampled to a buffer using an analog to digital converter. Using the liquid crystal display on our lab boards, a smooth waveform was drawn on the screen with an adjustable trigger, voltage scale, and time scale. To accomplish this, we implemented the 20 μs/div time scale, the 2 V/div, 1 V/div, 500 mV/div, 200 mV/div and 100 mV/div voltage scales, real-time measurement and triggering. Additionally, we implemented a measured central processing unit (CPU) load of the system to display on the LCD. This project was developed as a realistic real-time application. An ADC ISR was created to read samples from the ADC for storage in a buffer array. Also, a button ISR was created to read samples from a FIFO data structure holding each button press for trigger, voltage scale, and time scale adjustments.

Discussion and Results

This section describes our implementation of each major lab component in the signoff.

ADC Sampling/Circular Buffer

The ADC acquires a sample and interrupts every 1 μs, and must process this sample before the next one is aquired. This means that the ADC ISR has a relative deadline of only 120 CPU cycles. First, the ADC ISR acknowledges the ADC interrupt so it will not interrupt again on return. Then, it detects if a deadline was missed by checking the overflow flag of the ADC hardware. Lastly, it reads from the ADC and stores the result in a buffer array.



```
// ADC interrupt service routine
void ADC_ISR(void)
{
    ADC1_ISC_R = ADC_ISC_IN0;           // clears ADC interrupt flag
    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 1: ADC ISR Implementation

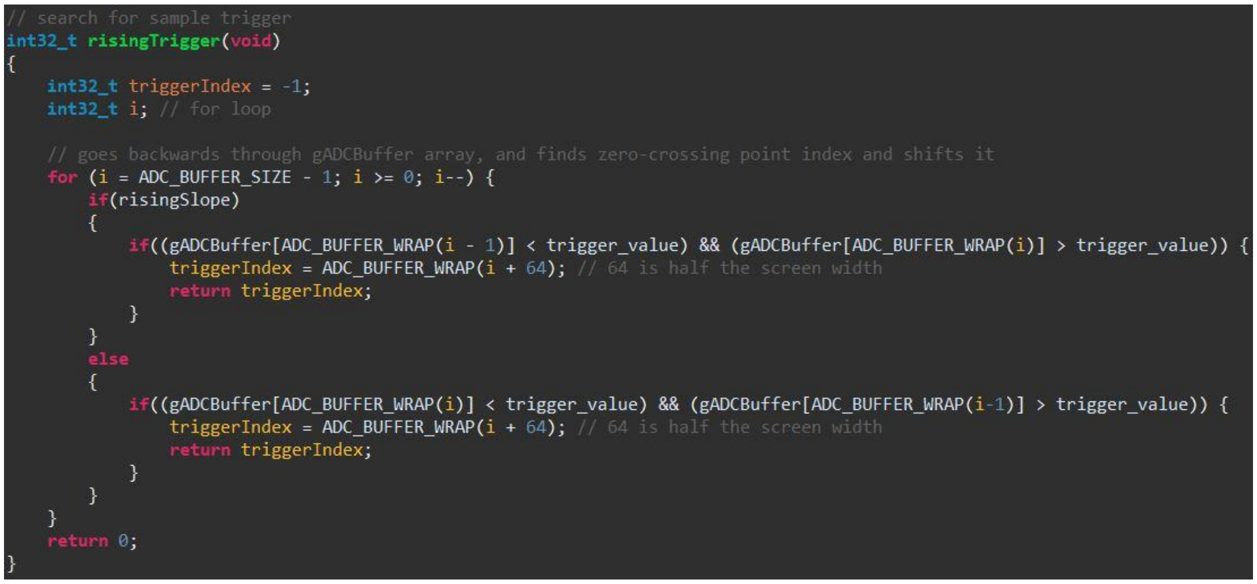
Figure 1 shows our implementation of the ADC ISR described in the steps above. Using the TM4C1294NCPDT Datasheet, we determined how to clear the interrupt flab that originally caused this ISR to be called. This was accomplished by setting `ADC1_ISC_R` equal to `ADC_ISC_IN0` which directly accesses the respective registers. Also, a sample was read from the ADC1 sequence 0 FIFO data structure using `ADC1_SSFIFO0_R` which was also found on the register datasheet.

Waveform Display

A working waveform display was created with adjustable voltage and time scales, and a toggleable trigger slope between rising and falling.

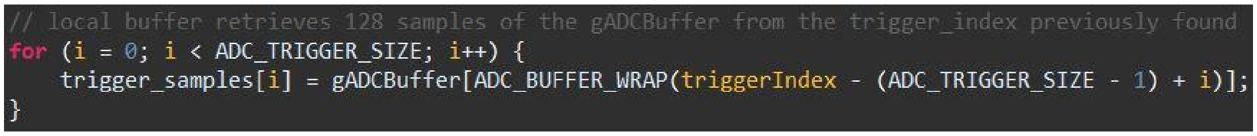
Trigger Search

In order to capture the correct pixels, the `risingTrigger` function looks for the point where the input voltage to the ADC is 0 Volts. This is accomplished by iterating through the ADC buffer until the midpoint of the waveform is found. Then, the `triggerIndex` is set to an index 64 samples ahead of this zero-crossing point because the width of the LCD is 128 pixels. The `ADC_BUFFER_WRAP` macro was used to determine an index on the array by wrapping around the circular buffer, so an index of 2048 corresponds to an index of 0 in the array.



```
// search for sample trigger
int32_t risingTrigger(void)
{
    int32_t triggerIndex = -1;
    int32_t i; // for loop

    // goes backwards through gADCBuffer array, and finds zero-crossing point index and shifts it
    for (i = ADC_BUFFER_SIZE - 1; i >= 0; i--) {
        if(risingSlope)
        {
            if((gADCBuffer[ADC_BUFFER_WRAP(i - 1)] < trigger_value) && (gADCBuffer[ADC_BUFFER_WRAP(i)] > trigger_value)) {
                triggerIndex = ADC_BUFFER_WRAP(i + 64); // 64 is half the screen width
                return triggerIndex;
            }
        }
        else
        {
            if((gADCBuffer[ADC_BUFFER_WRAP(i)] < trigger_value) && (gADCBuffer[ADC_BUFFER_WRAP(i-1)] > trigger_value)) {
                triggerIndex = ADC_BUFFER_WRAP(i + 64); // 64 is half the screen width
                return triggerIndex;
            }
        }
    }
    return 0;
}
```



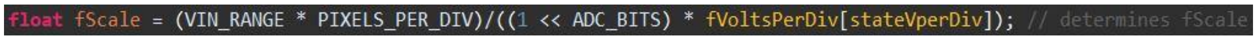
```
// local buffer retrieves 128 samples of the gADCBuffer from the trigger_index previously found
for (i = 0; i < ADC_TRIGGER_SIZE; i++) {
    trigger_samples[i] = gADCBuffer[ADC_BUFFER_WRAP(triggerIndex - (ADC_TRIGGER_SIZE - 1) + i)];
}
```

Figures 2 and 3: Trigger Search Implementation

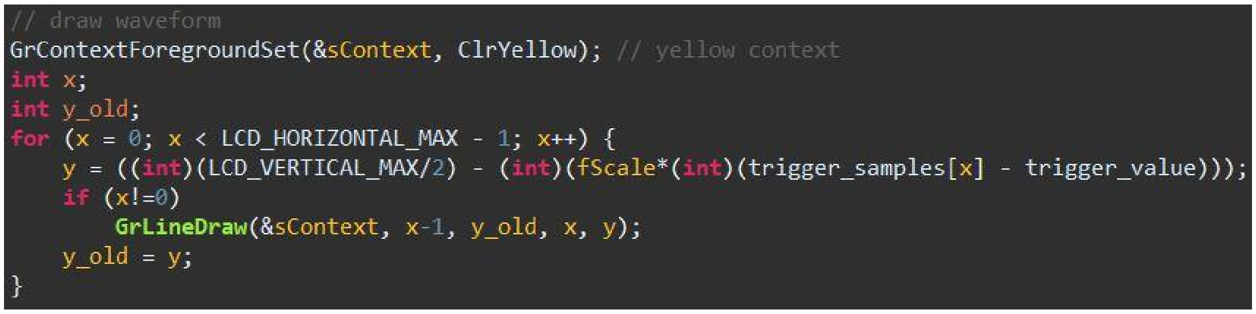
As shown in Figures 2 and 3 above, the `trigger_samples` array is a buffer in main.c that holds 128 samples to display on the screen. Using the `triggerIndex` found by the `risingTrigger` function in sampling.c, the `trigger_sample` array collects samples in front of the sample at `triggerIndex`.

ADC Sample Scaling

The waveform was drawn by scaling samples from trigger_samples such that they align with the volts per division scale. When scaling the ADC values, we understood that the ADC value corresponding to an input voltage of 0 is in the middle of the ADC range. The raw ADC samples were converted using the formulas in Figures 4 and 5 as shown below.



```c
float fScale = (VIN_RANGE * PIXELS_PER_DIV)/((1 << ADC_BITS) * fVoltsPerDiv[stateVperDiv]); // determines fScale
```



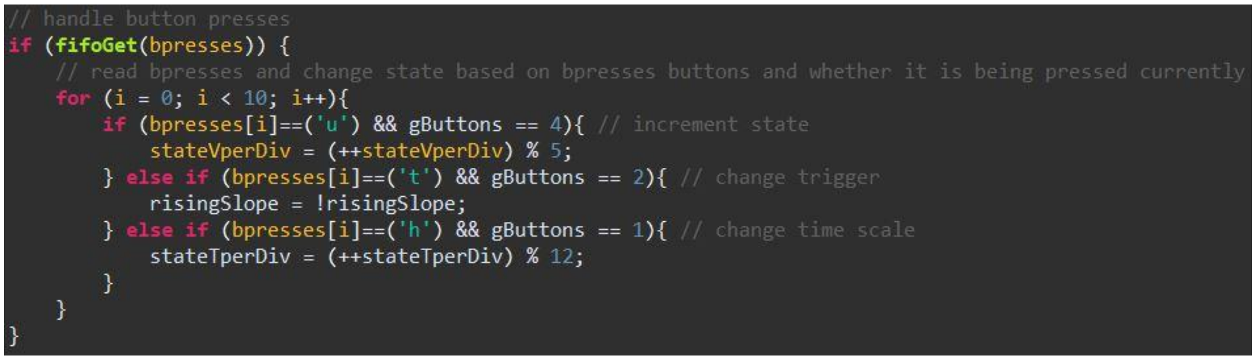
```c
// draw waveform
GrContextForegroundSet(&sContext, ClrYellow); // yellow context
int x;
int y_old;
for (x = 0; x < LCD_HORIZONTAL_MAX - 1; x++) {
    y = ((int)(LCD_VERTICAL_MAX/2) - (int)(fScale*(int)(trigger_samples[x] - trigger_value)));
    if (x!=0)
        GrLineDraw(&sContext, x-1, y_old, x, y);
    y_old = y;
}
```

Figures 4 and 5: Waveform Drawing

As shown in Figure 5 above, the waveform was drawn using a series of lines by iterating through each pixel of the LCD horizontally.

Button Command Processing

For button command processing, the loop in main.c gets the FIFO data structure. If a button press exits inside of the FIFO and `gButtons` is triggered with the button that is pressed, then the respective action would be taken. If there are no events inside of the FIFO, the `fifoGet` function would return 0, so no action would be taken.



```
// handle button presses
if (fifoGet(bpresses)) {
    // read bpresses and change state based on bpresses buttons and whether it is being pressed currently
    for (i = 0; i < 10; i++){
        if (bpresses[i]==('u') && gButtons == 4){ // increment state
            stateVperDiv = (++stateVperDiv) % 5;
        } else if (bpresses[i]==('t') && gButtons == 2){ // change trigger
            risingSlope = !risingSlope;
        } else if (bpresses[i]==('h') && gButtons == 1){ // change time scale
            stateTperDiv = (++stateTperDiv) % 12;
        }
    }
}
```

Figure 6: Handling Button Presses

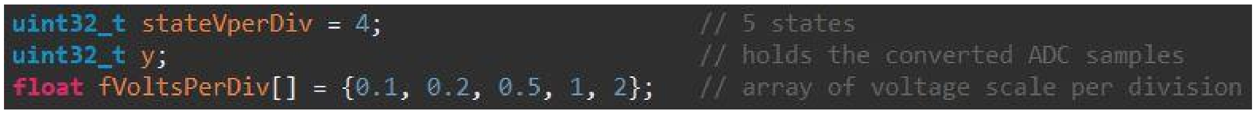
As shown in Figure 6 above, the `fifoGet` function clears the FIFO data structure, and the value of `gButtons` is checked to determine if a specific button is pressed.

Selectable Trigger Slope

In order to implement the selectable trigger slope, after a button is pressed, the character ‘t’ is sent to the FIFO data structure. When the button press is received in the main loop, it toggles the global boolean `risingSlope` which determines whether the waveform is triggered on the falling or rising edge. As shown in Figure 2 above, a check was added to the `risingTrigger` function to handle this. If `risingSlope` is true, the previous sample has to be less than the current sample and if it is false, the previous sample has to be greater than the current sample.

Adjustable Voltage Scale

As shown previously, the samples in the trigger_samples array are scaled by the variable `fScale` which is dependent on the voltage per division. In the main loop, a button press that sends the character ‘u’ alters the state of the voltage scale. Using a constant array with values of 2 V/div, 1 V/div, 0.5 V/div, 0.2 V/div, and 0.1 V/div, the `fScale` variable changes depending on its state. Therefore, when the state changes, the voltage scale of the drawn waveform changes.



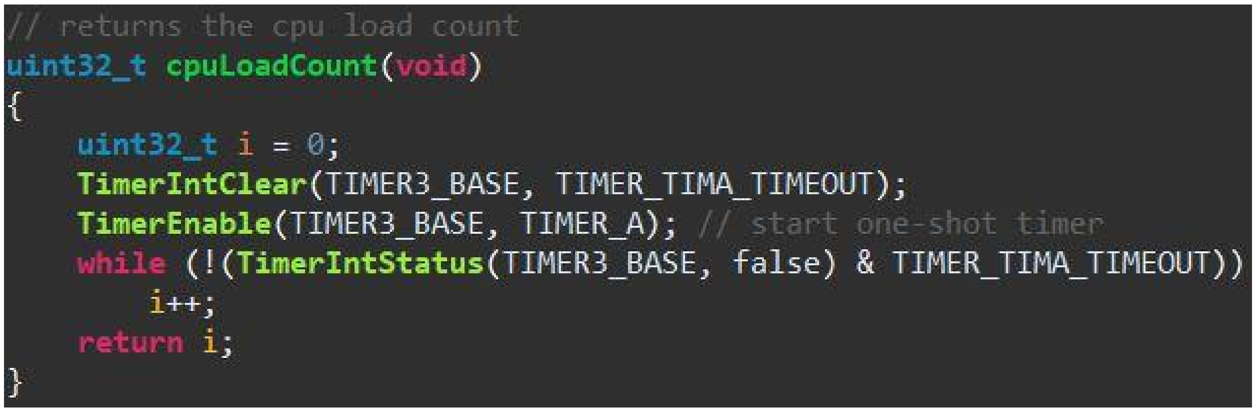
```
uint32_t stateVperDiv = 4;                      // 5 states
uint32_t y;                                     // holds the converted ADC samples
float fVoltsPerDiv[] = {0.1, 0.2, 0.5, 1, 2};   // array of voltage scale per division
```

Figure 7: Voltage Scale Array

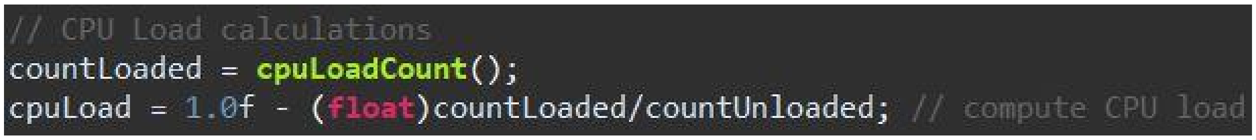
As shown in Figures 4, 6, and 7 above, the `stateVperDiv` variable changes when a ‘u’ character is sent to the FIFO data structure. Also, this `stateVperDiv` is used to index the `fVoltsPerDiv` array which holds all of the respective voltage scales.

CPU Load

The `cpuLoadCount` function measures the CPU load of the ISRs by configuring Timer 3 in one-shot mode. This is accomplished by counting iterations while `cpuLoadCount` is uninterrupted, so the more `cpuLoadCount` is interrupted, the less iterations there are. We estimated the CPU load by comparing the iteration counts when interrupts are enabled and when they are disabled. The CPU load count when interrupts are disabled was determined prior to the main loop, before interrupts are enabled. This unloaded value is then compared to the loaded CPU load count after every iteration of the main loop as shown in Figures 8 and 9 below.



```
// returns the cpu load count
uint32_t cpuLoadCount(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;
}
```

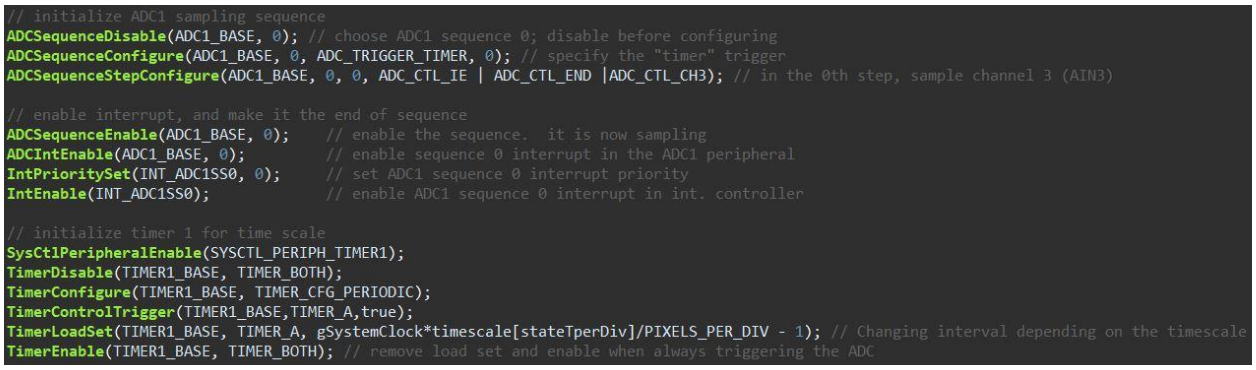


```
// CPU Load calculations
countLoaded = cpuLoadCount();
cpuLoad = 1.0f - (float)countLoaded/countUnloaded; // compute CPU load
```

Figures 8 and 9: CPU Load

Adjustable Time Scale

The adjustable time scale was implemented by enabling Timer 1 to trigger the ADC conversions. As shown in Figure 10 below, Timer 1 was enabled and given a load set that matches the selected time scale while ADC1 is changed from always triggering to triggering from the timer.



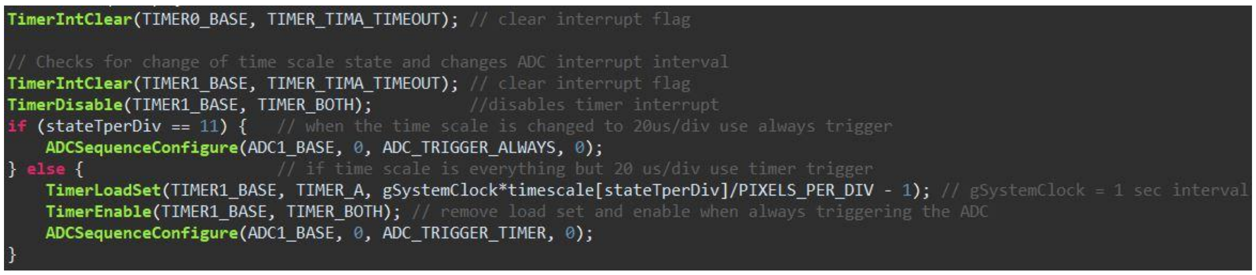
```
// initialize ADC1 sampling sequence
ADCSequenceDisable(ADC1_BASE, 0); // choose ADC1 sequence 0; disable before configuring
ADCSequenceConfigure(ADC1_BASE, 0, ADC_TRIGGER_TIMER, 0); // specify the "timer" trigger
ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_IE | ADC_CTL_END |ADC_CTL_CH3); // 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

// initialize timer 1 for time scale
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
TimerDisable(TIMER1_BASE, TIMER_BOTH);
TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
TimerControlTrigger(TIMER1_BASE,TIMER_A,true);
TimerLoadSet(TIMER1_BASE, TIMER_A, gSystemClock*timescale[stateTperDiv]/PIXELS_PER_DIV - 1); // Changing interval depending on the timescale
TimerEnable(TIMER1_BASE, TIMER_BOTH); // remove load set and enable when always triggering the ADC
```

Figure 10: Time Scale ADC Initialization

Also, In order to change the load set in real-time, when the button ISR interrupts, the timer is changed depending on the state of the adjustable time scale as shown in Figure 11 below.



```
TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // clear interrupt flag

// Checks for change of time scale state and changes ADC interrupt interval
TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT); // clear interrupt flag
TimerDisable(TIMER1_BASE, TIMER_BOTH);          //disables timer interrupt
if (stateTperDiv == 11) {   // when the time scale is changed to 20us/div use always trigger
    ADCSequenceConfigure(ADC1_BASE, 0, ADC_TRIGGER_ALWAYS, 0);
} else {                    // if time scale is everything but 20 us/div use timer trigger
    TimerLoadSet(TIMER1_BASE, TIMER_A, gSystemClock*timescale[stateTperDiv]/PIXELS_PER_DIV - 1); // gSystemClock = 1 sec interval
    TimerEnable(TIMER1_BASE, TIMER_BOTH); // remove load set and enable when always triggering the ADC
    ADCSequenceConfigure(ADC1_BASE, 0, ADC_TRIGGER_TIMER, 0);
}
```

Figure 11: Time Scale Button ISR

A global variable `stateTperDiv` holds the 12 states of the potential time scales which are 100 ms, 50 ms, 20 ms, 10 ms, 5 ms, 2 ms, 1 ms, 500 µs, 200 µs, 100 µs, 50 µs, and 20 µs. After the character ‘h’ is sent to the button FIFO, the state of the time scale is changed to the next value as shown in Figure 6. The ADC should be in the always trigger mode when the 20 µs scale is created because the ADC is being interrupted the fastest since it is at a higher priority. Therefore, if the 20µs/div state is reached, the program keeps timers disabled and changes the ADC to trigger always.

Difficulties

Initially, the triggering functionality of the code didn't work correctly. Switching the board between rising and falling edge triggers did nothing, and the image of the wave was displayed based on the timer, leaving it off center. This was fixed by reorganizing the code, and adjusting the conditions of a few if statements containing the relevant code.

Our square wave did not work at first, as the right addresses had to be found and plugged into the functions in `ADC_Init` in sampling.c, and the peripherals had to be correctly configured to fix this.

Conclusions

In this project, we implemented a real-time application by accessing shared data safely, using interrupt prioritization and preemption, and using real-time debugging techniques. By implementing a second ISR for the ADC, interrupt prioritization and preemption was emphasized, since the button ISR was previously implemented. The ADC ISR was set to interrupt at a faster rate compared to the button ISR, so the ADC ISR had to preempt the button ISR for no samples to be missed. Also, shared data bugs were reduced by practicing steps for reentrancy which included using local storage as much as possible and properly handling access to global resources. For example, the FIFO data structure for button events meant that data can be safely placed into and retrieved for button presses. Also, we learned how the CPU load is dependent on the amount of interrupts that occur, so the less interrupts there are, the smaller the CPU load is.