Lab 1: Digital Oscilloscope

Yil Verdeja, ECE Box 349
April 3 2019

Table of Contents

List of Figures

1 Introduction

The purpose of this lab was to create a versatile one mega sample per second (Msps) oscilloscope that could read a sample waveform. By attaching an oscillator circuit to the analog input of the board, the waveform was sampled to a buffer using an Analog to Digital Converter (ADC). Using the Liquid Crystal Display (LCD), a smooth waveform was drawn on the screen showing a time scale of 20 µs per division at a high frame rate. This oscilloscope is versatile since it includes multiple features such as: selectable trigger slopes (falling or rising), adjustable voltage scales (100mV/div to 1V/div), and a measured Central Processing Unit (CPU) load of the system. This project was a good example of developing a real-time application without an operating system. Tight timing constraints were met, and interrupt prioritization and preemption were used.

This project was developed to be a realistic real-time application. Because of the extremely tight timing of the ADC Interrupt Service Routine (ISR), interrupts cannot be disabled when accessing shared variables in main(). By not disabling interrupts, latency isn't introduced into the high priority ISR. Two ISRs were created: an ADC ISR and a Button ISR. Since the ADC continuously take samples from the analog input, it was set to the highest priority since it would be interrupted most often.

2 Methodology, Results and Discussion

This section describes the implementation of each major step in the lab by going through them and describing how each was completed. The steps include the implementation of a circular buffer, a waveform display on the LCD, a First-in First-out (FIFO) data structure for button presses, a selectable trigger slope, an adjustable voltage scale and the CPU load. It also includes the implementation of an extra step which was to create an adjustable time scale.

2.1 Circular Buffer

The output of the oscillating circuit designed was connected directly to an analog input on the board which was sampled via the ADC1. An ADC ISR was created to interrupt the ADC every 1 µs and process the sample acquired on the ADC prior to the next one being obtained. The functionality of the ADC ISR is simple:

1. Acknowledge the ADC interrupt by clearing its interrupt flag
2. Detect if a deadline was missed by checking the overflow flag of the ADC hardware.
3. Read a sample from the ADC and store it in a buffer array

```
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 the implementation of the ADC ISR described in the steps above. After running this code and verifying that the ISR was being called, with the oscillator connected, a sampled waveform was observed in the gADCBuffer. As expected, the ADC samples into a circular buffer at 1 Msps without missing any samples. Figure 2 provides a snippet of the ADC sampling into the ADC buffer.

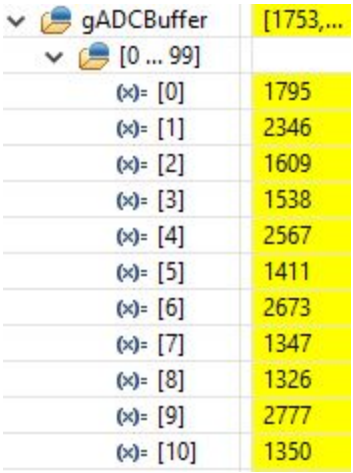


Figure 2. Sampled Waveform on the gADCBuffer

2.2 Initial Waveform Display

Figure 3 shows a working waveform display at 20µs/div and 1V/div with a rising edge trigger at the 0 V level. While the frame rate is high, at certain times the waveform *glitches* and can't be fully considered a smooth waveform. Although hard to catch, when it *glitches* the waveform either breaks or shifts for less than a sample. This is due to the trigger being reset constantly.

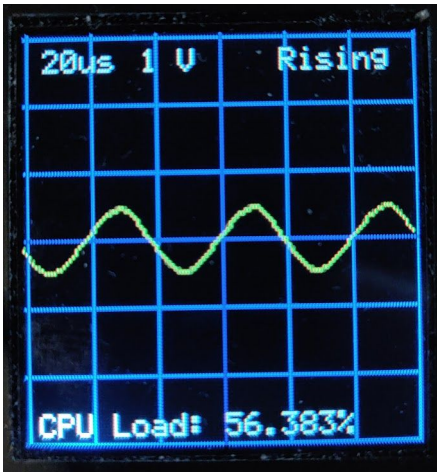


Figure 3. Initial Waveform Display with 20µs/div and 1V/div

2.2.1 Zero-crossing point

To create this oscilloscope display, using the ADC circular buffer, a zero crossing point was determined. The zero-crossing point can be found at the middle of the waveform. This was found to be around an analog value of 2055. However, to stay accurate within each sample taken from the ADC, a function was written to dynamically determine the midpoint of the waveform. Instead of a using a constant ADC offset of 2055, the dynamic function *zero_crossing_point()* was preferred since it did not affect the CPU load.

2.2.2 Trigger Search

As shown in Figure 4, the oscilloscope displays a specific screen frame, where the zero-crossing point is centered on the screen of the LCD. On the LCD each pixel represents a sample from the ADC. Since the LCD is 128 pixels wide, the waveform is captured within 128 samples.

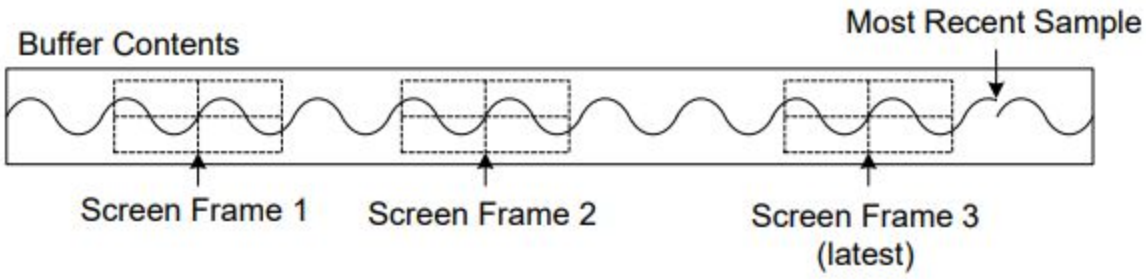


Figure 4. Fitting ADC Samples into Screen Frame

In order to capture the correct 128 pixels, the *triggerSearch()* function looks for the zero-crossing point value (i.e. 2055 ± 5). This is done by iterating backwards through the entire ADC buffer until the midpoint of the waveform is found. Once that conditional statement is met,

the *trigger_index* is set to an index that is 64 samples ahead of the zero-crossing point index. Since the width of the LCD is 128 pixels, the index where the zero-crossing point occurs must be on the 64th pixel, halfway across the screen. The ADC_BUFFER_WRAP macro is used to determine a non-existing index on the array by wrapping around the circular buffer (i.e. with an array of 2048 values, indices exist between 0 and 2047. Using the wrap macro, an index of 2048 would correspond to an index of 0 on the array).

```
254 void triggerSearch(void){
255     int32_t trigger_index = -1;
256     int32_t offset = 5;
257     int32_t i;
258     for (i = ADC_BUFFER_SIZE - 1; i >= 0; i--) {
259         if ((gADCBuffer[i] >= trigger_value - offset) && (gADCBuffer[i] <= trigger_value + offset)) {
260             trigger_index = ADC_BUFFER_WRAP(i + 64);
261             break;
262         }
263     }
264
265     // looks inside gADCBuffer, finds trigger_index, and gets half the values from that point
266     for (i = 0; i < ADC_TRIGGER_SIZE; i++){
267         trigger_samples[i] = gADCBuffer[ADC_BUFFER_WRAP(trigger_index - (ADC_TRIGGER_SIZE - 1) + i)];
268     }
269
270 }
```

Figure 5. triggerSearch() function implementation

As shown in Figure 5, the *trigger_samples* array is a local buffer that holds 128 samples to display on the screen. Using the *trigger_index* found, the *trigger_sample* array collects 128 samples in front of the sample at the *trigger_index*.

2.2.3 ADC Sample Scaling and LCD Drawing

The waveform was drawn by scaling the *trigger_samples* such that they conform to the volts per division scale. The raw ADC samples were converted using the conversion shown below:

```
int y = LCD_VERTICAL_MAX/2 - (int)roundf(fScale * ((int)sample - ADC_OFFSET));
```

Note: The fScale variable is set by a product of the ADC resolution and the ratio of pixels per div to voltage per div.

As shown in the figure below, the waveform was drawn using a series of lines.

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

Figure 6. Waveform Draw Function

2.3 FIFO Button Presses

Previously, button presses were read directly from the global *gButtons* variable. However, reading *gButtons* could pose a problem since the main() loop may not catch that the button was pressed. To fix this issue, a FIFO data structure was implemented.

2.3.1. FIFO Data Structure

```
// 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_tail + 1;
    if (new_tail >= FIFO_SIZE) new_tail = 0; // wrap around
    if (fifo_head != new_tail) {     // if the FIFO is not full
        fifo[fifo_tail] = data;      // store data into the FIFO
        fifo_tail = 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_head != fifo_tail) {    // if the FIFO is not empty
        *data = fifo[fifo_head];     // read data from the FIFO
        if (fifo_head + 1 >= FIFO_SIZE)
            fifo_head = 0;
        else
            fifo_head++;
        return 1;                    // success
    }
    return 0;   // empty
}
```

Figure 7. FIFO Data Structure - Shared data fix

The figure above is the implementation of the circular FIFO data structure. The shared data bug was fixed within the *fifo_get()* function by only allowing one write per *fifo_get()* call.

2.3.2 Accessing Button Presses

To listen to button presses, the main() loop runs a conditional that gets the FIFO data structure. If a button press exists and *gButtons* is triggered with the specific button that is pressed, then it takes action accordingly. If there is nothing inside the FIFO data structure, *fifo_get()* would return 0 and no action is taken.

```
if (fifo_get(bpresses)) {
    // read bpresses and change state based on bpresses buttons and gButtons
    for (i = 0; i < 10; i++){
        if (bpresses[i]==('u') && gButtons == 4){ // increment state
            stateVperDiv = (++stateVperDiv) & 3;
        } else if (bpresses[i]==('t') && gButtons == 2){ //trigger
            risingSlope = !risingSlope;
        }

    }
}
```

Figure 8. Reading Button Presses

Since every *fifo_get()* function clears the FIFO data structure, the value of *gButtons* also has to be checked to further determine if that specific button is being pressed.

2.4 Selectable Trigger Slope

To implement the selectable trigger slope, after some specific button is pressed, a character of 't' is sent to the FIFO. When the button press is retrieved in the main loop, it inverts a global boolean *risingSlope* which determines whether the waveform should be triggered on the falling edge or rising edge. As shown below, two lines are added into the first for loop of the *triggerSearch()* function (see Figure 5).

```
for (i = ADC_BUFFER_SIZE - 1; i >= 0; i--) {
    if ((gADCBuffer[i] >= trigger_value - offset) && (gADCBuffer[i] <= trigger_value + offset)) {
        bool isRising = gADCBuffer[ADC_BUFFER_WRAP(i - 1)] > gADCBuffer[ADC_BUFFER_WRAP(i + 1)]; // checks whether rising slope or falling slope
        if (isRising && risingSlope || !isRising && !risingSlope) { // set trigger index depending on whether its a rising slope or falling slope
            trigger_index = ADC_BUFFER_WRAP(i + 64);
            break;
        }

    }
}
```

Figure 9. Modified Trigger Search for Trigger Slope Detection

First, the *isRising* variable is a boolean that determines whether the zero-point index is at a rising slope. The second line is a conditional that checks whether the value of *isRising* matches the

global *risingSlope* variable. If it doesn't match, it finds the next occuring zero-point index on the ADC Buffer.

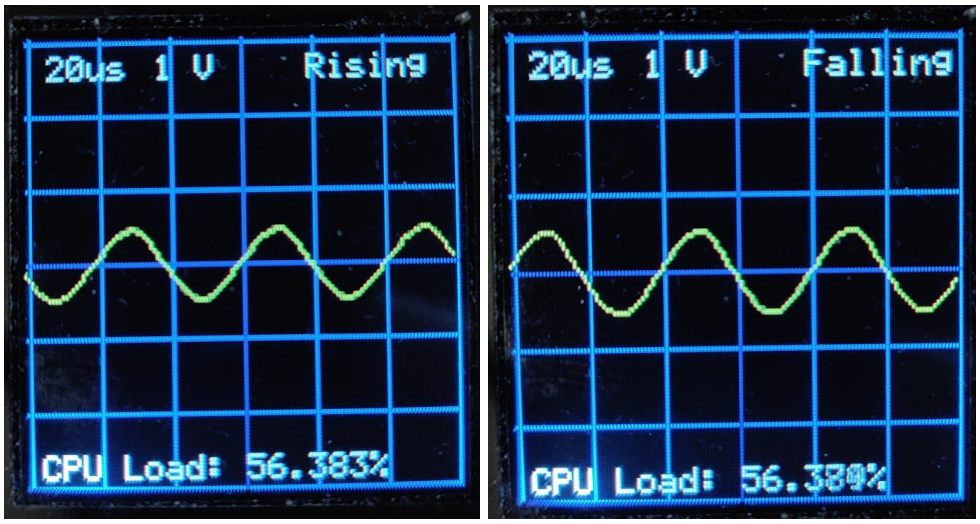


Figure 10. Selectable Trigger Slope - Rising (Left) and Falling (Right)

2.5 Adjustable Voltage Scale

As mentioned previously, the 128 *trigger_samples* is scaled by the variable fScale which is dependent on the voltage per division. In the main() loop, a certain button press ('u') would alter the state of the voltage scale. Using a constant array with values of 1V/div, 0.5V/div, 0.2V/div and 0.1V/div, the fScale variable changes depending on what state it is in. Because the fScale changes at different states, the voltage scale of the drawn waveform is changed thus creating an adjustable voltage scale feature.

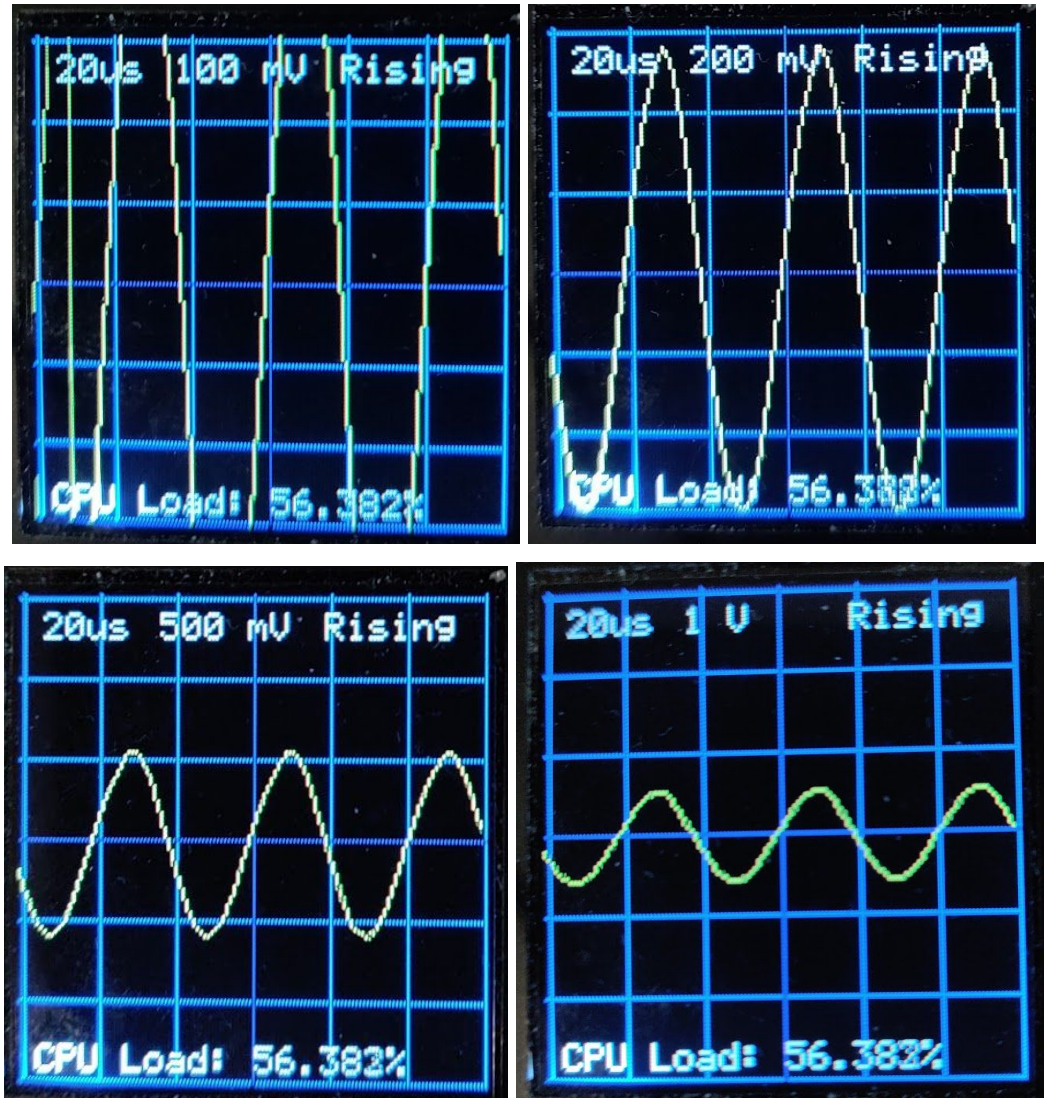


Figure 11. Adjustable Voltage Scale - 100mV/div to 1V/div

The figure above show the screen of the LCD at different voltage scales. The voltage scales are 100mV/div, 200mV/div, 500mV/div and 1V/div. In every case, the 0V level is calibrated to always be in the middle of the screen.

2.6 CPU Load

The *cpu_load_count()* function measures the CPU load of the ISRs by configuring Timer3 in one-shot mode. This is done by counting iterations. The more *cpu_load_count()* is interrupted, the less iterations there are. The CPU load can be estimated by comparing the iteration counts when interrupts are enabled and when they are disabled. Since the ADC ISR has an extremely high timing, interrupts cannot be disabled in the main() loop. Therefore the CPU load count when interrupts are disabled must be 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 on the main() loop. As shown by all the figures above, the estimated CPU load of this system is 56.383%. This value is obtained by averaging the CPU load over 10ms.

2.7 Extra Credit: Adjustable Time Scale

In this portion, an adjustable time scale is implemented. However, unlike the adjustable voltage scale which required changing the *fScale* variable, the ADC conversions need to be triggered from a timer. Timer 1 is enabled and given a load set that matches the time scale selected whilst the ADC1 is changed from always triggering, to triggering from the timer. As shown in figure 12, this is only changed in the setup.

```
// Timer 1 setup with
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

// 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
```

Figure 12. Trigger and ADC Setup Changed for Adjustable Time Scale

When the load set is set to *gSystemClock*, the timer runs on a 1 second interval. By multiplying *gSystemClock* by the time scale, the ADC samples the waveform at the rate provided. 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. These changes can be seen in figure 13 which is a snippet of the code added to button ISR.

```
// Checks for state change
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 13. Changing timer interval depending on the Time Scale

A global and volatile variable *stateTperDiv* holds the 12 states of the potential time scales: 100ms, 50ms, 20ms, 10ms, 5ms, 2ms, 1ms, 500µs, 200µs, 100µs, 50µs and 20µs. After a certain

button press (‘h’), the state of the time scale is changed to the next value. The images can be seen on figure N on the Appendix.

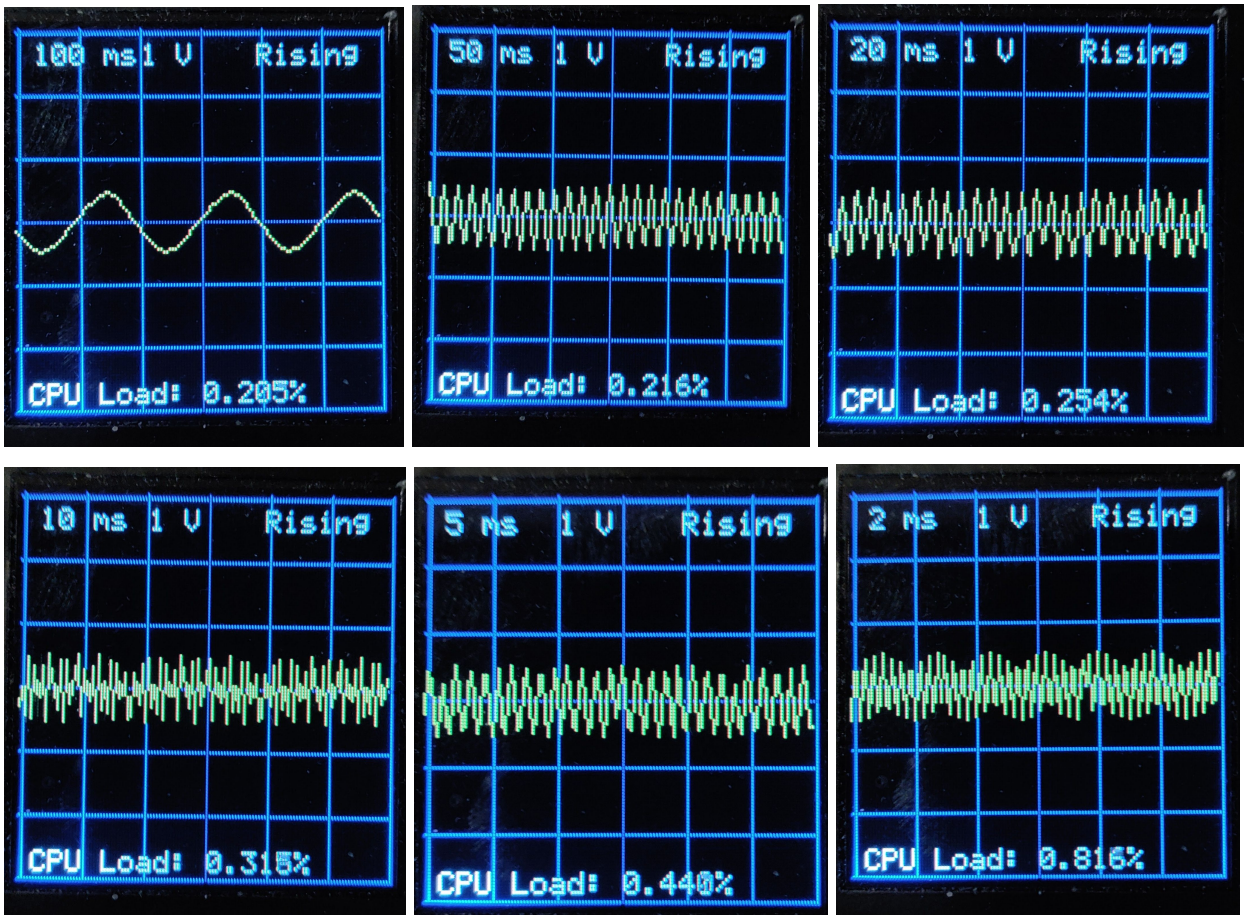
Rather than being triggered from a timer, the 20µs/div scale should be created when the ADC is in the “always” trigger mode. This is because the ADC is at a higher priority and is being interrupted the fastest, which creates the 20µs/div time scale. The implementation of this feature uses a conditional to check whether that state is reached. If it isn’t, it changes the timer load set and configures the ADC sequence. As expected, if the 20µs/div state is reached, it keeps timers disabled and changes the ADC to trigger always at 1 Msps.

3 Conclusion

The most important outcomes of this project in creating a real-time application were by using interrupt prioritization and preemption, accessing shared data without disrupting performance, and using real-time debugging techniques. Debugging in real-time proved to be a great tool because it not only allowed to better understand what was happening in the code, but it also helped in fixing errors when they occured. Prior to this lab, an ISR was already written to handle button presses. By creating a new ISR for the ADC, it enforced how interrupt prioritization and preemption worked. Since the ADC ISR was set to interrupt at a much faster than the button ISR, the ADC ISR must preempt the button ISR. This order of interrupt prioritization allows for no samples of data to be missed when retrieving data from the ADC. Lastly, shared data bugs were reduced by practicing the steps of reentrancy which include: using local storage as much as possible and by properly handling access to global data and resources atomically.

The extra implementation of the adjustable time scale was hard to implement because the timer and ADC settings had to be changed. Without realization, initially the settings were changed inside the ADC ISR which caused inaccurate waveforms. This was fixed by placing the change in the settings inside the button ISR. These settings were placed here because the its affected by a button press and not by sampling the ADC. It is interesting to notice how the CPU load decreases as the sampling rate of the ADC decreases. However, this makes sense because the CPU load is dependent on the amount of interrupts that occur. Therefore the less interrupts there are, the smaller the CPU load.

Appendix: Extra Credit Time Scale



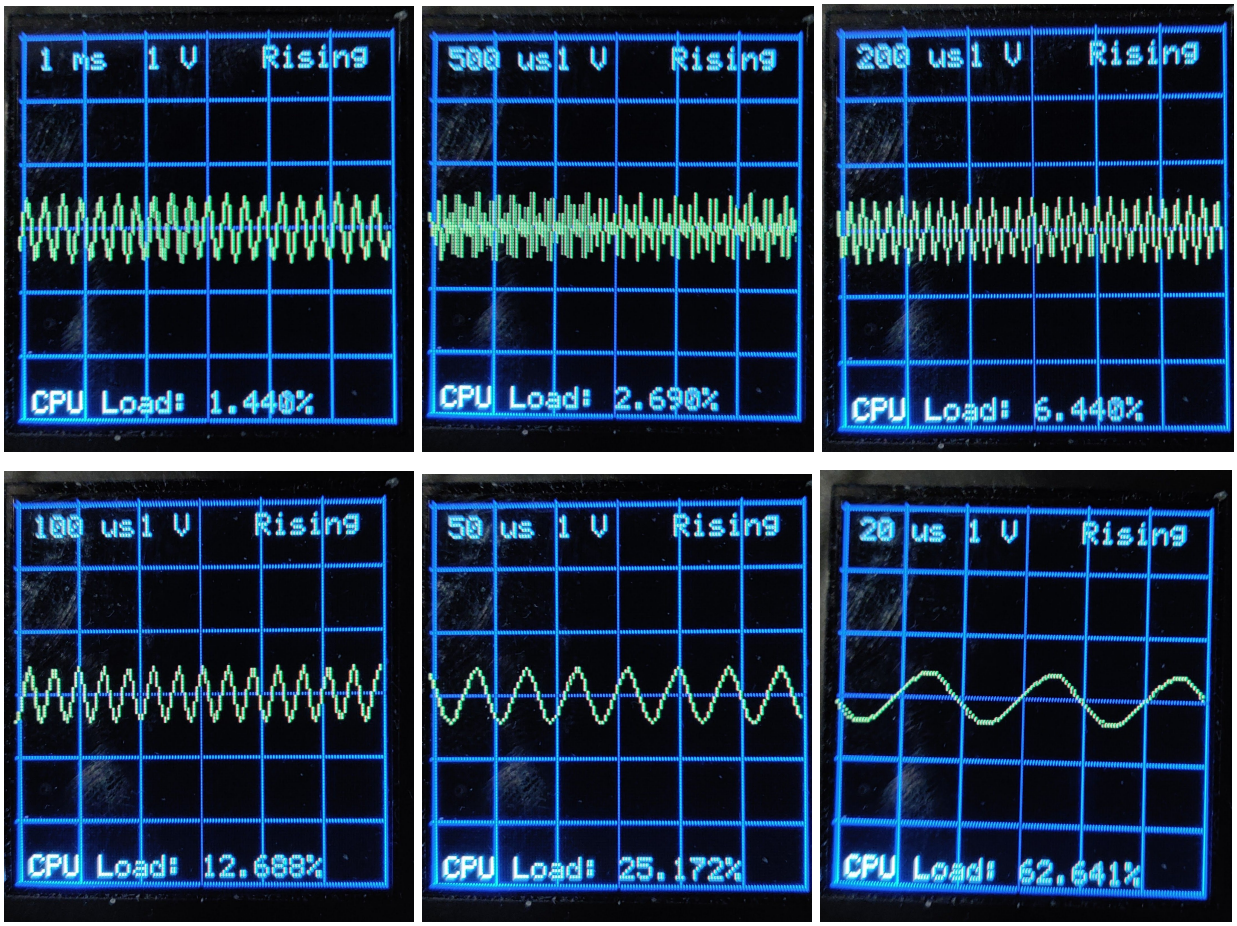


Figure 14. Adjustable time scale from 100ms to 20µs