Introduction

The main purpose of this lab1 is to implement a 1 Msps digital oscilloscope using TI EK-TM4C1294XL LaunchPad and BOOSTXL-EDUMKII Educational BoosterPack MKII. We first built a small voltage sine wave signal generator on the breadboard using lab kit, then we used launchpad LCD screen to display the signal. The main features of this digital oscilloscope are shown below.

1. ADC samples into a circular buffer at 1,000,000 samples/sec without missing samples.
2. The waveform can be displayed at different voltage scale from 0.2 V/div to 1V/div.
3. The user can select rising or falling edge trigger.
4. Measure the CPU load of ISRs and display it in percent on the LCD. (Average the CPU load over 10 ms.)

Discussion and Results

- System requirements and real-time goals. Assign thread types and priorities based on the top-level design.

The system requirements consist of the correctness of measurement and ADC, the correctness of offset when interpreting raw ADC values, the continuity of the graph from multiple ADC values (e.g. the given generated example), the correctness of multiple user input to change volt scale and trigger, and the smooth of real-time graph change. Also, The real-time goals are to be able to do three different tasks without error and share data bug. In this lab, tasks can be obviously separated to 3 tasks: taking ADC values, taking user input, and updating screen with correct values. Taking ADC and user inputs should be in interrupt routine services (ISRs), while updating screen should be regular task. Hence, the priorities of the former should be higher than the latter. Because ADC has shorter deadline than user input, taking ADC should have higher priority than taking user input.

- Implementation of each major lab component.

The first step we did was getting and storing ADC samples into a circular buffer at 1,000,000 samples/sec without missing samples. We wrote ADC1 code using the ADC0 initialization in buttons.c as a reference. The only difference was that we had to change ADCSequenceConfigure third element to “Always” so that it’s always triggered. We also wrote an ADC ISR function, which could store the newly acquired samples in the circular buffer gADCBuffer. The code for ADC sample part is shown below.

```
// initialize ADC sampling sequence
gADCBufferIndex = ADC_BUFFER_SIZE - 1;
ADCSequenceDisable(ADC0_BASE, 0);
ADCSequenceDisable(ADC1_BASE, 0);
ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 1);
ADCSequenceConfigure(ADC1_BASE, 0, ADC_TRIGGER_ALWAYS, 0);
ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH13);                              // Joystick HOR(X)
ADCSequenceStepConfigure(ADC0_BASE, 0, 1, ADC_CTL_CH17 | ADC_CTL_IE | ADC_CTL_END);  // Joystick VER(Y)
ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_CH3 | ADC_CTL_IE | ADC_CTL_END);
ADCSequenceEnable(ADC0_BASE, 0);
ADCSequenceEnable(ADC1_BASE, 0);
ADCIntEnable(ADC1_BASE, 0);
IntPrioritySet(INT_ADC1SS0, ADC1_INT_PRIORITY);
IntEnable(INT_ADC1SS0);
```

Figure1. ADC Samples Code

```
// ISR for ADC1
void ADC_ISR(void){
    ADC1_ISC_R = ADC_ISC_IN0;
    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;
}
```

Figure2. ADC ISR Code

The second step we did was searching trigger. In order to implement a simple oscilloscope, we don't have to display all the acquired ADC samples. Instead, We only need to present the general shape of the waveform. Therefore, finding trigger is very important. Our team implemented this trigger search by using suggested algorithm in the lab 1 assignment.

To find a rising edge trigger, we first need to define a trigger offset, which is on the middle of the LCD vertical screen. If there exists an index that gADCBuffer(index-1) is smaller than offset value, and gADCBuffer(index+1) is bigger than offset value, then that index is the rising edge trigger point that we want to find. We wrote a while loop that can determine if the current index is the trigger by comparing gADCBuffer(index-1) and (index+1). If it's not trigger, we will move this index backwards, which is index-1, until we found the trigger index. In addition, we also added a search count in this while loop, the search count increased by 1 if the index is not the trigger index. If the search count is bigger than half of the ADC buffer size, we will give up the search. The code of searching trigger is shown below.

```
// variable initialization
int32_t trig_idx;   // trigger index
volatile int32_t curr_idx;   // current index of ADC when loaded
int32_t search_count;
int16_t buffer[HALF_SCREEN * 2];
```

```
// Create buffer to create oscilloscope
curr_idx = gADCBufferIndex;
trig_idx = ADC_BUFFER_WRAP(curr_idx - HALF_SCREEN);
search_count = 0;
if(gADCBuffer[trig_idx] != ADC_OFFSET){

    if(risingTrig)
        while(!((gADCBuffer[ADC_BUFFER_WRAP(trig_idx + 1)] >= ADC_OFFSET) && (gADCBuffer[trig_idx] < ADC_OFFSET)) && (++search_count < SEARCH_REGION))
            trig_idx = ADC_BUFFER_WRAP(trig_idx - 1);
    else
        while(!((gADCBuffer[ADC_BUFFER_WRAP(trig_idx + 1)] <= ADC_OFFSET) && (gADCBuffer[trig_idx] > ADC_OFFSET)) && (++search_count < SEARCH_REGION))
            trig_idx = ADC_BUFFER_WRAP(trig_idx - 1);

    if(search_count >= SEARCH_REGION)
        trig_idx = ADC_BUFFER_WRAP(curr_idx - HALF_SCREEN);
}
```

Figure3. Searching Trigger Code

After finding the trigger successfully, the next step was displaying the waveform at 20 μs/div on the LCD screen. We first stored 64 sample points before trigger index and 64 samples points after trigger index into a buffer, so that the trigger point is displayed on the middle of horizontal screen.

Then, We need to scale the raw ADC samples based on certain voltage scale, so that we can fit the waveform on the LCD screen. We calculated variable fScale using the formula provided. Then, we stored the scaled ADC sample value into buffer again. Finally, we printed the buffer and displayed the waveform.

```
int32_t i;
for(i = 0; i < HALF_SCREEN * 2; i++)
    buffer[i] = gADCBuffer[ADC_BUFFER_WRAP(trig_idx + i - HALF_SCREEN + 1)];

// change buffer to position on LCD
int temp;
for(i = 0; i < LCD_HORIZONTAL_MAX; i++){
    temp = (int)roundf(fScale * ((int)buffer[i] - ADC_OFFSET));
    if(temp > LCD_VERTICAL_MAX / 2)
        buffer[i] = -1;
    else if(temp < - LCD_VERTICAL_MAX / 2)
        buffer[i] = LCD_VERTICAL_MAX;
    else
        buffer[i] = LCD_VERTICAL_MAX / 2 - temp;

    if(i > 0)
        GrLineDraw(&sContext, i - 1, buffer[i - 1], i, buffer[i]);
}
```

Figure4. Store ADC Samples into Buffer and Display the Waveform Code

After displaying the waveform successfully on the screen, the next task was controlling voltage scale and trigger edge by buttons. We have to store the pressed button as a character into a FIFO, otherwise we may miss the button presses. We used example code FIFO code fifo_put() and fifo_get() functions from lecture, and we also fixed the shared data bug in that code. We put fifo_put() function in buttonISR() function, so every time buttonISR() function is called, the updated pressed button character is put

into FIFO. The fifo_get() function is put in main() function inside the while loop, so we can get the pressed button character from FIFO head index. The code for this part is shown below.

```
// put data into the FIFO, skip if full
// returns 1 on success, 0 if FIFO was full
int fifo_put(DataType 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(DataType *data)
{
    if (fifo_head != fifo_tail) {   // if the FIFO is not empty
        *data = fifo[fifo_head];    // read data from the FIFO
        if (fifo_head == FIFO_SIZE - 1)
            fifo_head = 0; // wrap around
        else
            fifo_head++;
        return 1;                   // success
    }
    return 0;   // empty
}
```

Figure5. FIFO Code

Once we knew which button was pressed using FIFO, we need to change the displayed waveform. If the trigger slope is rising edge, we find the trigger point by using gADCBuffer(index-1) is smaller than offset value, and gADCBuffer(index+1) is bigger than offset value. If trigger slope is failing edge, we find the trigger index by using gADCBuffer(index-1) is bigger than offset value, and gADCBuffer(index+1) is smaller than offset value.

The different voltage scale feature was implemented by fScale variable. The formula to calculate fScale is shown below.

*float fScale = (VIN_RANGE * PIXELS_PER_DIV)/((1 << ADC_BITS) * fVoltsPerDiv)*

We changed variable fVoltsPerDiv to change the displayed waveform voltage scale.

The last step was measuring the CPU load of the ISRs and displaying the result. We also used the example code from lecture. But we modified Timer3 to one-shot mode and timeout interval to 10 ms by changing TimerLoadSet() third element from gSystemClock to gSystemClock /100. The code below shows how to initialize Timer3.

```
// initialize timer 3 in one-shot mode for polled timing
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
TimerDisable(TIMER3_BASE, TIMER_BOTH);
TimerConfigure(TIMER3_BASE, TIMER_CFG_ONE_SHOT);
TimerLoadSet(TIMER3_BASE, TIMER_A, gSystemClock / 100 - 1); // 1 sec interval -> 10 ms
```

Figure5. Initialize Timer3 Code

The CPU load can be estimated from the counts with interrupts enabled and disabled. So we called cpu_load_count() function when interrupts are displayed, and called this function again when interrupts are enabled. Finally, we could calculate CPU load by formula:

CPU = 1 - count_loaded/count_unloaded

- Picture of lab results.

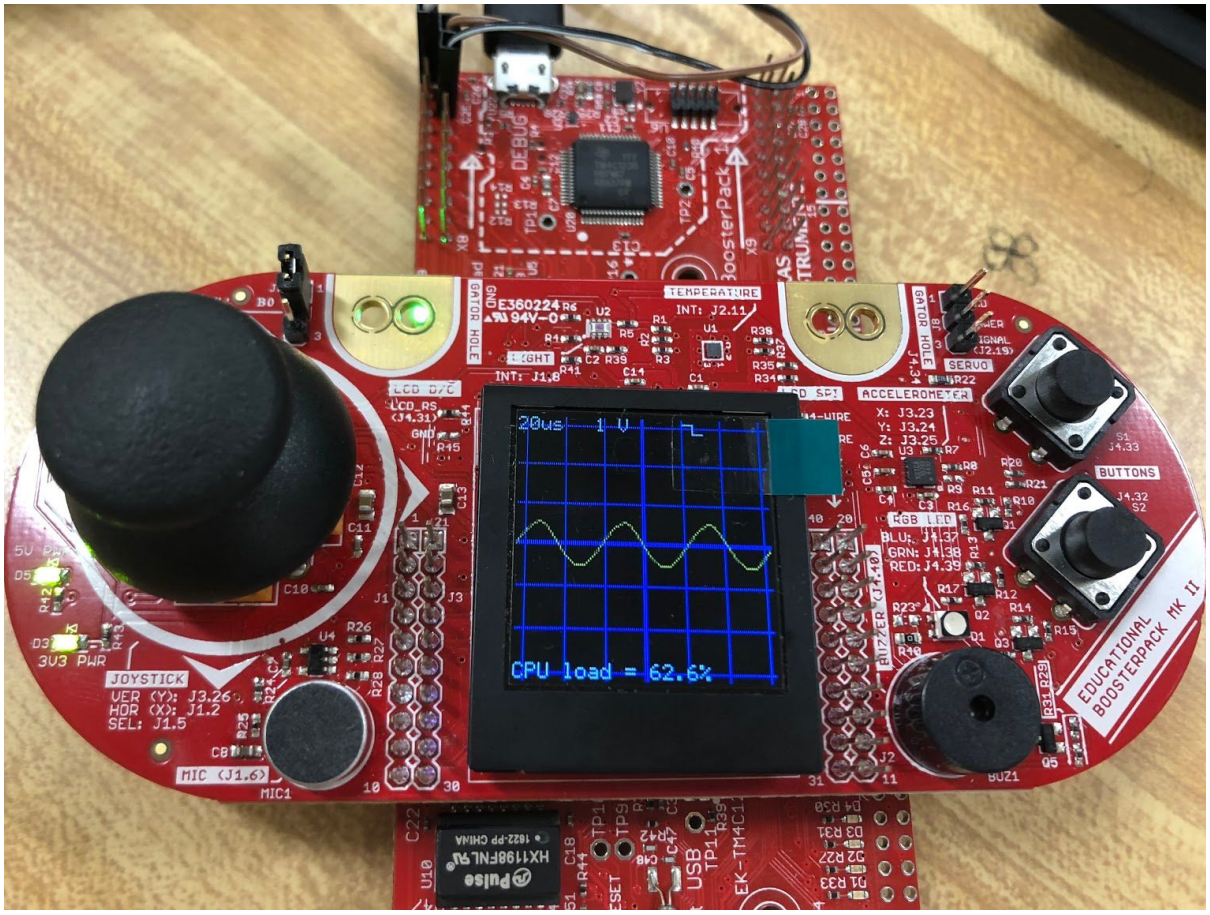

Figure6. Picture of Lab Result

- Serious difficulties we encountered.

The first difficulty we encountered was that ADC ISR was never called. We tried configuring ADC1 using the ADC0 initialization in buttons.c as a reference, then we set a breakpoint at ADC ISR, but the project didn't stop there. We finally fixed the problem by specifying the "Always" trigger in ADCSequenceConfigure(...).

Another we encountered was CPU load measurement. We didn't know how to set Timer3 in one-shot mode, especially how to modify the timeout interval to 10 ms. We finally found that the code should be TimerConfigure(TIMER3_BASE, TIMER_CFG_ONE_SHOT); and TimerLoadSet(TIMER3_BASE, TIMER_A, gSystemClock / 100 - 1). gSystemClock means 1us interval, so gSystemClock/100 stands for 10ms interval.

Conclusions

In summary, our team successfully developed a 1 Msps digital oscilloscope with CPU measurement feature in this lab. First, we learned how to acquire ADC samples at 1,000,000 samples/sec without missing any. Then, we implemented a trigger search by suggested algorithm. In addition, we learned how to store the button ID (a character) into a FIFO while accessing shared data without disrupting real-time performance. We also learned how to use interrupt prioritization and preemption in a real application. Last, we also successfully measured the overall CPU utilization by modifying the timeout interval to 10 ms.