ECE 3849 B2023
Real-Time Embedded Systems
Lab 2: Improved Oscilloscope

This lab will improve on the 1 Msps digital oscilloscope implemented last lab by adding triggers, scaling, and displaying CPU load.

To complete all labs in this course, you will need to submit your source for that lab, and demonstrate the lab in operation to one of the course staff (a TA or tutor). You will be graded on the completion of the steps in this lab document. After you have received a signoff, follow the instructions at the end of the document to submit your source code.

This lab is due on **Wednesday, November 15**. Late assignments will receive a 20% penalty.

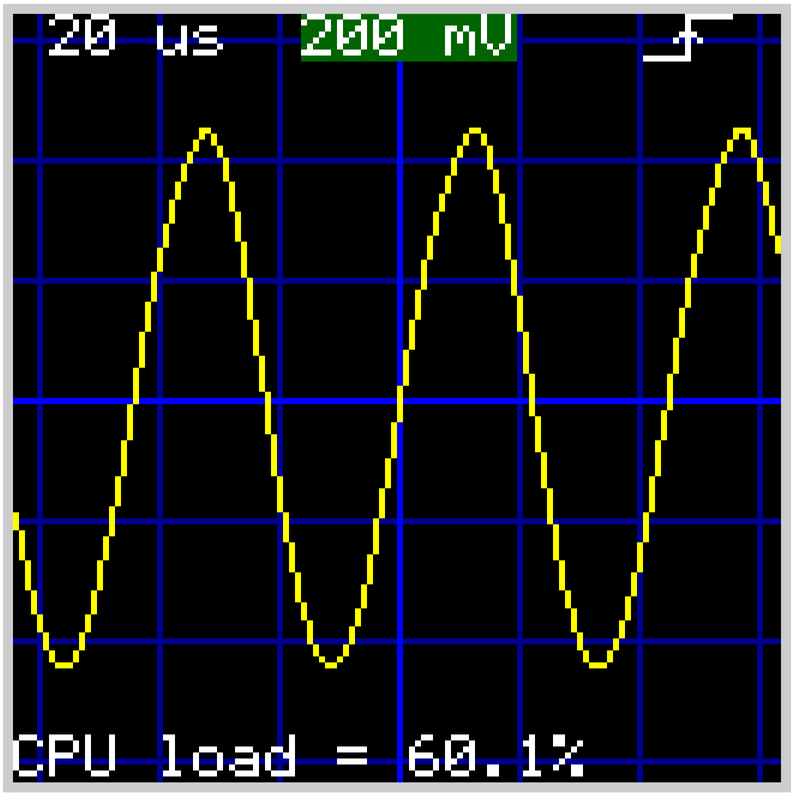
Step	**Max Points**	**Score**
Smooth display of waveform at 20us/div and 1V/div with rising edge trigger, high frame rate.	10	
Selectable trigger slope	15	
Adjustable voltage scale: 2V, 1V, 500mV, 200mV, 100mV	10	
CPU load of ISRs displayed as percentage on screen	25	
Extra Credit: Adjustable time scale	10	
Source Code & Report	25	
Button inputs passed to main through a FIFO without shared data bugs	15	
Total	100 + 10	

Objectives

- Develop a realistic real-time application without an operating system
- Meet tight timing constraints
- Use interrupt prioritization and preemption
- Write performance-sensitive code
- Access shared data without disrupting real-time performance
- Use real-time debugging techniques

Assignment

In this lab, you will improve on the oscilloscope you implemented in the last lab by scaling the signal to the grid, making the scaling level of the signal adjustable by buttons, and implementing a trigger search as a replacement for the snapshot mode. Additionally, you will display the CPU load of ISRs on the screen.



When writing your software, do not use a real-time operating system. You may use TivaWare, TI Graphics Library and other libraries. Document the source of any external libraries that you use in the lab report.

Part 0: Copy CCS Project

Copy your Lab 1 CCS project by selecting it in Project Explorer and pressing Ctrl+c, then Ctrl+v. CCS should prompt you to rename the new project: use the convention for online code submission "ece3849_lab2_username1_username2," substituting your username. When the time comes to submit your source code, follow the instructions at the end of the Lab.

Part 1: Button Command Processing

To control the various features of the oscilloscope, you will rely on user inputs. In this part of the lab, you will modify the button command processing to push button inputs to a FIFO that `main()` can process and respond to.

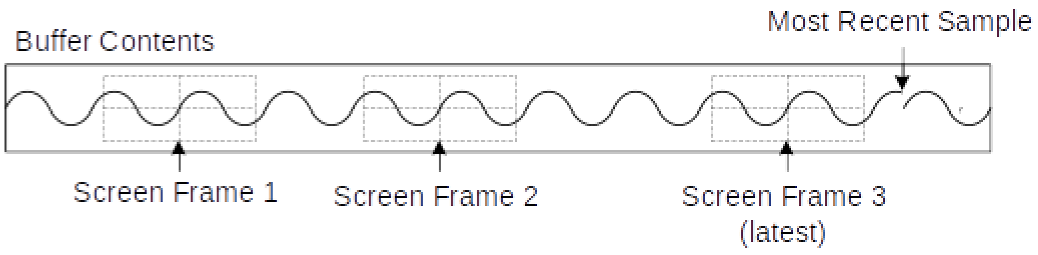
You may copy the FIFO data structure discussed in lecture from the ece3849_shared_data project on the course website, but you will need to fix it. Watch out for shared data bugs in the `fifo_get()` function, as it can be interrupted anywhere by the button ISR. As explained in lecture, it is possible to create a circular FIFO data structure free of shared data bugs without disabling interrupts.

Modify your existing code for reading the buttons in the button ISR to store the button id in a FIFO for `main()` to process. A capacity of 10 button presses is enough. Using a FIFO will ensure that the `main()` loop will not miss any button presses if a loop iteration takes a considerable amount of time.

Part 2: Trigger Search

Here, you will implement a trigger search to allow continuously displaying samples to the screen with a consistent display.

The ADC ISR stores the newly acquired samples in the circular buffer `gADCBuffer`. The software processing the acquired samples must keep track of its own index into this buffer. If the processing software is too slow in processing the samples, the ADC ISR will overwrite them. Luckily, to implement a simple oscilloscope, we do not necessarily need to process all the acquired ADC samples. The behavior illustrated in the following figure is perfectly acceptable of our oscilloscope:



The buffer contents are shown as an analog waveform. The oscilloscope only needs to present the viewer with the general shape of the waveform. To accomplish this, it may extract the newest acquired ADC samples that conform to a **trigger** (waveform crossing the specified voltage level in the specified direction at a certain point in time) and ignore the older samples. There may be large gaps of ignored samples between displayed frames (Frame 1, 2 and 3), and the oscilloscope performance is not affected. In this example, Frame 1 and 2 are older frames, displayed some time ago, while Frame 3 is the currently displayed frame. The dotted lines indicate screen borders and the trigger location (crossing in the middle of the frame).

The following is the suggested algorithm for trigger search in main():

1. Initialize the trigger index (into `gADCBuffer`) to half a screen width behind the most recent sample. (Why half-screen? If the trigger is found immediately, enough samples are available to display on the right half of the screen. Screen size constants are available in Crystalfontz128x128_ST7735.h.)
2. Keep moving this index backwards until the buffered waveform crosses the trigger level in the desired direction (e.g. current sample is at or above trigger level, next older sample is below trigger level). The trigger level in ADC units is the same as `ADC_OFFSET` in the next section "Step 3: ADC sample scaling."
3. If not finding a trigger, give up the search after traversing half of the ADC buffer. (Why half? We are reserving the other half for overwriting by the ISR.) In this case, reset the

trigger index to its initial location (as in step 1) to display the newest samples unsynchronized.

4. Copy samples from half a screen behind to half a screen ahead of the trigger index from `gADCBuffer` into a local buffer. (Remember, 1 pixel width = 1 sample interval.)

```
int RisingTrigger(void) // search for rising edge trigger
{
    // Step 1
    int x = gADCBufferIndex - <...>/* 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 &&
         <...>/* next older sample */ < ADC_OFFSET)
        break;
    }
    // Step 3
    if (x == x_stop) // for loop ran to the end
        x = <...>; // reset x back to how it was initialized
    return x;
}
```

Note that `gADCBuffer` and `gADCBufferIndex` are shared between the ADC ISR and `main()`. We have learned that under a preemptive scheduler, accesses to shared data must be treated with care. Because of the extremely tight timing of the ADC ISR, we cannot disable interrupts when accessing these shared variables in `main()`. Your approach should be to treat the code accessing the circular buffer as real-time: Perform the desired operations on half of the buffer while the ISR overwrites the other half. Reads of `gADCBufferIndex` are atomic, so pose no shared data issues (if the programmer does not expect it to remain unchanged).

To implement the trigger search, use the provided code to find a trigger, then modify your code in main to copy the samples around the trigger into the buffer that is drawn to the screen. Once trigger search is working, modify your code to search for a falling trigger, and add a button to switch between rising and falling triggers. Finally, draw an indicator on the screen to show whether the trigger is rising, falling, or not found.

Part 3: ADC Sample Scaling

Now that trigger search is working, instead of drawing raw ADC samples on the screen, they should be scaled to conform to a volts/division scale.

Assume your oscilloscope input has a DC offset added before it is sampled: $V_{ADC} = V_{in} + 1.65V$. The oscilloscope input voltage range is from −1.65 V to +1.65 V, while the ADC input voltage range is 0 V to +3.3 V. Zero input voltage corresponds to the middle of the ADC range.

Here is a simple conversion method that produces the pixel y-coordinate:

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

Where:

`uint16_t sample` = raw ADC sample

`ADC_OFFSET` = ADC value when Vin = 0 V (middle of ADC range)

`LCD_VERTICAL_MAX` = vertical dimension of the LCD in pixels (predefined)

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

Where:

`VIN_RANGE` = total Vin range in volts = 3.3

`PIXELS_PER_DIV` = LCD pixels per voltage division = 20

`ADC_BITS` = number of bits in the ADC sample (look it up in the datasheet)

`float fVoltsPerDiv` = volts per division setting of the oscilloscope

The scale factor fScale should remain constant during waveform plotting in a frame for performance reasons.

Implement several voltage scales, and use button inputs to switch between them.

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

Finally, draw the current voltage scale on the screen.

Part 4: CPU Load Measurement

In this section, you will measure the CPU load in ISRs, and display it to the screen.

Recall from lecture that the overall CPU utilization by your real-time tasks (only ISRs in this case) is an indicator of schedulability of your lowest priority tasks. To measure the CPU load of the ISRs it is suggested to follow the example project ece3849_int_latency. This example configures Timer3 in one-shot mode. It then starts it 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. Measure the former for every frame displayed, and the latter only once, before starting the main() loop. Note that the timer polling code should be in a separate function that should not be inlined. If the polling code is duplicated, the optimizing compiler may compile each version differently, resulting in erroneous measurements.

You will need to set the timer timeout interval to 10 ms. The timer one-shot mode is explained in Section 13.3.3.1 of the datasheet. Briefly, the “load value” argument of TimerLoadSet() is the timeout in system clock cycles. In periodic mode, the same parameter is the period minus 1 clock cycle.

Extra Credit: Adjustable ADC Sampling Rate

To achieve an adjustable ADC sampling rate lower than 1 Msps, you will need to trigger the ADC conversions from a timer. You may follow the example of Lab 0 when configuring a second timer to specify the ADC sampling rate. There is a separate driver function call to enable the ADC trigger output of a timer. You need to implement many new time scales for full credit: 100 ms/div, 50 ms/div, 20 ms/div, 10 ms/div, etc., down to 50 µs/div, also including 20 µs/div from the main assignment. The 20 µs/div time scale is a special case where the ADC should be in the “always” trigger mode rather than triggered from a timer.

If you implement an adjustable time scale, you may sample the microphone (GPIO: PD5, ADC: AIN6) instead of the PWM signal. If you do, mention this in your lab report. To receive the extra credit, you need to demonstrate the time scale adjustment during the TA signoff.

Part 5: Submitting Source Code

Make sure your Lab 1 project is named “**ece3849_lab1_username**” with your username substituted. **Clean** your project (remove compiled code). Right click on your project and select “Export...” Select General → “Archive File.” Click Next. Click Browse. Find a location for you archive and specify the exact same file name as your project name. Click Finish. Submit the resulting .zip file to Canvas. This concludes Lab 1.