ECE 3849 D2018
Real-Time Embedded Systems
Lab 1: Digital Oscilloscope

In this first full lab assignment you will implement a 1 Msps digital oscilloscope using your ECE 3849 lab kit. Have the course staff fill in your scores in the following signoff sheet before the signoff deadline. This lab requires a report (one per group) that has a separate due date. See the course website for due dates and a lab report guide.

Step	Max pts	Score
ADC samples into a circular buffer at 1,000,000 samples/sec without missing samples, with all other scope features active.	10	
Waveform display works at 20 μs/div and 1 V/div with rising edge trigger at 0 V level, high frame rate, smooth waveform.	15	
Button presses are written to a FIFO by the button ISR. Reads from the FIFO should be free of shared data bugs.	10	
Selectable trigger slope (rising or falling).	3	
Adjustable voltage scale down to 100 mV/div. The 0 V level should be calibrated to always be in the middle of the screen.	7	
Measure the CPU load of your ISRs and display it in percent on the LCD. Average the CPU load over 10 ms.	15	
Adjustable time scale using a timer (extra credit; see assignment) Do not modify the ADC clock.	5	
Question about this lab answered	5	Student1: Student2:
Report and source code (separate due date)	35	
Total points	100 + 5	Student1: Student2:

Student 1: _____ Student 2: _____

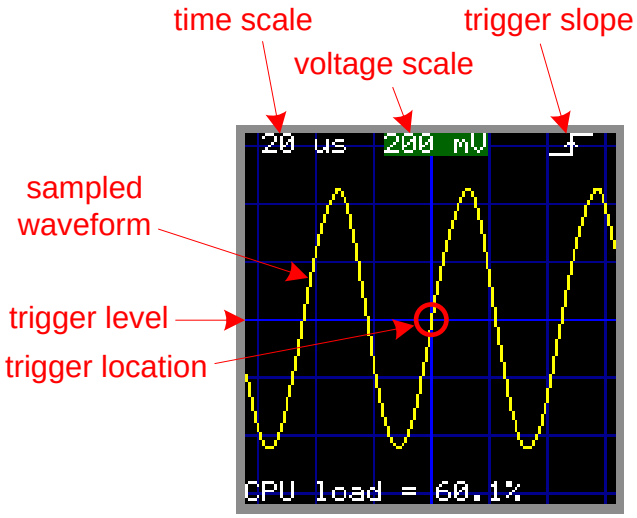
Grader: _____ Date: _____ Mailbox: _____

Learning 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

Write software that would turn your lab kit into a 1 Msps single-channel digital oscilloscope. The LCD display should resemble the following figure.



The units of time and voltage scale are per division, which is one grid square on the screen (20 pixels). Implement the 20 µs/div time scale, one ADC sample per pixel at 1 Msps. For the voltage scale, implement 1 V/div, 500 mV/div, 200 mV/div and 100 mV/div. Voltage scale must be adjustable using buttons or joystick. The trigger level should be 0 V, in the middle of the screen vertically. On the time grid, the trigger location should be in the middle of the screen horizontally, as shown. Trigger slope (rising or falling) should be adjustable using buttons. When drawing, interconnect samples with lines for a smooth waveform look. If the waveform does not cross the trigger level, display the newest samples unsynchronized. Do not wait forever for a trigger!

Your oscilloscope should have an AC coupled input with a 3.3 V voltage range (from −1.65 V to +1.65 V). You will also need a test signal to display on your oscilloscope. Your ECE 3849 lab kit contains breadboard components to implement these features (see Recommended Implementation). If you do not have a breadboard, borrow or buy one from the ECE shop. Wire (22 AWG) will be available in AK113.

When writing your software, do not use a real-time operating system. You may use TivaWare, TI Graphics Library and other libraries.

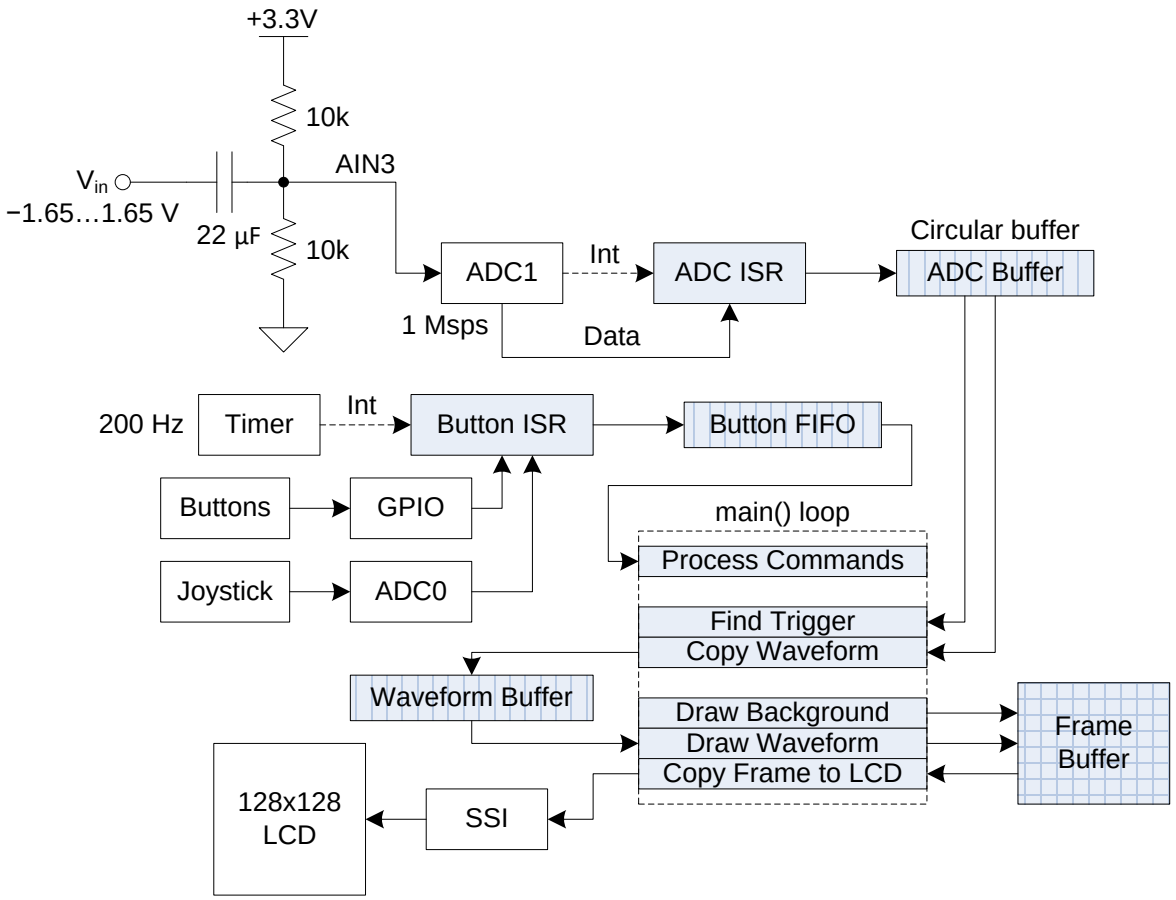
An additional challenge in this lab is to determine the CPU load of the ISRs. Measure the CPU load over a time interval of 10 ms for each displayed frame. Output the ISR CPU load as text on the LCD, e.g. “CPU load = 60.1%” as seen in the figure above.

For extra credit, implement additional time scales: 100 ms/div, 50 ms/div, 20 ms/div, 10 ms/div, etc. down to 50 μs/div. Changing the time scale should change the ADC sampling rate (do not worry about aliasing). Do not modify the ADC clock to change the sampling rate, because the ADC clock is already at its lowest allowed value. Instead, trigger the ADC from a timer when sampling slower than 1 Msps. Extra credit assignments are your responsibility to figure out. Do not expect much help on these from the course staff.

Write a detailed lab report documenting your lab implementation. A lab report guide is available in the Assignment/Labs section of the course website.

Recommended Implementation

The following block diagram shows the recommended Lab 1 structure.



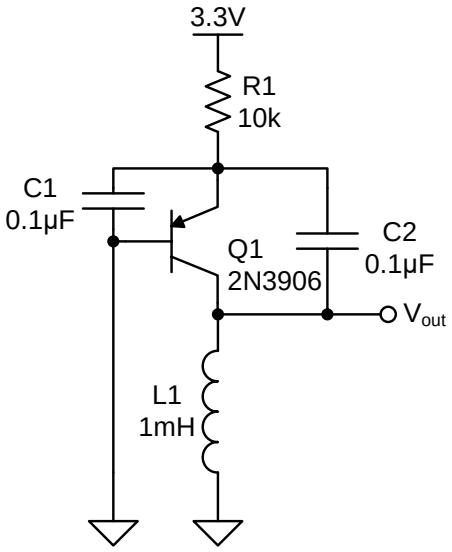
Step 0: Copying the CCS project

Copy your Lab 0 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 "ece3849d18_lab1_username1_username2," substituting your usernames. You will be reusing the button and joystick processing part of Lab 0. Keep your older completed labs intact.

Step 1: Supporting hardware

It is suggested that you build the supporting hardware for this lab first. The input circuit uses a capacitor for AC coupling and a voltage divider to set the center voltage. Build it on a breadboard and connect it to the EK-TM4C1294XL using single-row headers and jumper wires. Look up the location of the analog input AIN3 in the EK-TM4C1294XL LaunchPad User's Guide or TI's BoosterPack Checker website. Use the EK-TM4C1294XL on-board power pins +3.3V and GND to power your breadboard circuit. Do not use the bench power supplies.

Next, construct a very simple Colpitts oscillator as shown in the following schematic. It should produce a near sine wave at ~22.5 kHz with an amplitude of ~0.5 V (verify using a bench scope). V_{out} will be the test signal for your oscilloscope to display. Connect V_{out} to V_{in} on the previous diagram. This oscillator will also be used as a sensor in a subsequent lab. Again, power it directly from the EK-TM4C1294XL board. Hints: The inductor is the large, cylindrical component (there are two of them in your kit). This circuit needs very good contact between the components and the breadboard. To reduce ADC noise, twist the AIN3 and GND jumper wires together. If you cannot get this oscillator to work, ask for help.



Step 2: ADC sampling

Your first challenge is to acquire ADC samples at 1,000,000 samples/sec without missing any. Configure **ADC1** using the ADC0 initialization in buttons.c as a reference. Note that these are different ADC peripherals. Use the following incomplete setup code as a starting point:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO?);
GPIOPinTypeADC(...);          // 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(...);      // choose ADC1 sequence 0; disable before configuring
ADCSequenceConfigure(...);    // specify the "Always" trigger
ADCSequenceStepConfigure(...);// in the 0th step, sample channel 3 (AIN3)
                              // enable interrupt, and make it the end of sequence
ADCSequenceEnable(...);       // enable the sequence.  it is now sampling
ADCIntEnable(...);            // enable sequence 0 interrupt in the ADC1 peripheral
IntPrioritySet(...);          // set ADC1 sequence 0 interrupt priority
IntEnable(...);               // enable ADC1 sequence 0 interrupt in int. controller
```

The CRYSTAL_FREQUENCY and ADC_SAMPLING_RATE constants are defined in buttons.h. Include "sysctl_pll.h" to be able to call SysCtlFrequencyGet(). Please note that both ADCs must be initialized to the same clock configuration, as they share the clock divider.

Consult the ADC chapter of the TivaWare Peripheral Driver Library User's Guide to fill in the arguments of the driver function calls. To call the ADC driver functions, you need to include "driverlib/adc.h". To access registers directly, include "inc/tm4c1294ncpdt.h". It is helpful to browse the driver header files to find the correct constant to use. To jump straight to where the right constants are defined, Ctrl+click on another closely related constant (e.g. ADC0_BASE) in the ADC setup code of buttons.c.

The ADC acquires a sample and interrupts every 1 μs. The ADC ISR must process this sample before the next one is acquired. This gives the ADC ISR a relative deadline of only 120 CPU cycles. This is an extremely tight timing requirement, but the Cortex-M4F turns out to be up to the challenge. We will need to perform some optimization to meet this goal. In a subsequent lab we will learn how to relax this timing constraint. The following is the recommended structure of the ADC ISR:

```
#define ADC_BUFFER_SIZE 2048                                  // size must be a power of 2
#define ADC_BUFFER_WRAP(i) ((i) & (ADC_BUFFER_SIZE - 1)) // index wrapping macro
volatile int32_t gADCBufferIndex = ADC_BUFFER_SIZE - 1;  // latest sample index
volatile uint16_t gADCBuffer[ADC_BUFFER_SIZE];           // circular buffer
volatile uint32_t gADCErrors;                          // number of missed ADC deadlines

void ADC_ISR(void)
{
    <...>; // 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)
               ] = <...>;               // read sample from the ADC1 sequence 0 FIFO
}
```

The ADC ISR is very simple:

1. Acknowledge the ADC interrupt (so it would not interrupt again on return).
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.

Your tasks are to find how to clear the interrupt flag that originally caused this ISR to be called, and to read in the ADC sample. You have to do this using **direct register access**, not driver function calls. In this case the overhead associated with driver function calls is enough to start missing deadlines (you can verify this).

To convert register names from the TM4C1294NCPDT Datasheet to C code:

<datasheet peripheral name>**<register name>** →
<C peripheral name><number>_**<register name>**_R

For example, the datasheet register `GPTM**ICR**` in Timer0 is converted to `TIMER0_ICR_R` in C code. Similarly the register fields are defined as:

```
<C peripheral name>_<register name>_<field name>
```

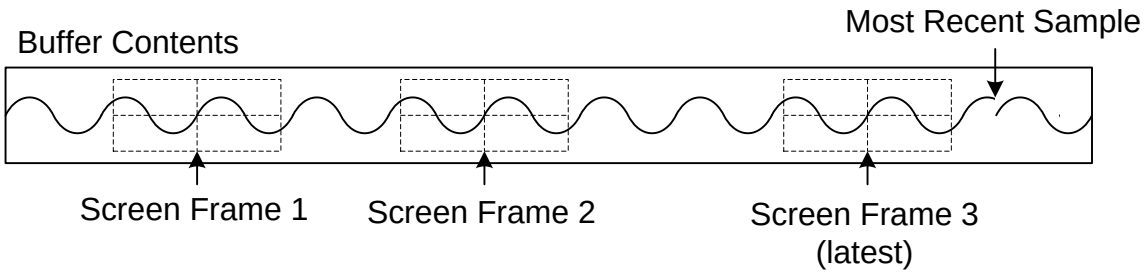
For example, the `TATOCINT` (timer timeout interrupt) bit of the `GPTMICR` register can be accessed using `TIMER_ICR_TATOCINT` in C code.

As before, it is convenient to browse the appropriate header files to locate the correct register definition. Ctrl+click on `ADC1_OSTAT_R` to bring up a list of valid ADC1 registers available for direct access. You would still need to browse the TM4C1294NCPDT Datasheet to help locate the right registers to use, as the header file is not commented. Also see TivaWare Peripheral Driver Library User's Guide Chapter 2.2 for more information on direct register access.

Make sure to refer to this ISR in the appropriate interrupt vector in tm4c1294ncpdt_startup_ccs.c. At this point, you can run your code and verify that the ISR is being called and the ADC error counter stays at zero (it should increment if you pause your program for debugging). If the counter is steadily incrementing, try to find the cause. One possibility is incorrect interrupt priority assignment (the ADC ISR must preempt the button ISR; Lab0 has an explanation of the interrupt priorities). If your oscillator is connected, you should be able to observe the sampled waveform in `gADCBuffer` using the debugger.

Step 3: Trigger search

The ADC ISR stores the newly acquired samples in the circular buffer `gADCBuffer`. This is not a strict FIFO data structure. 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, and the oscilloscope performance is not affected.

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

1. Initialize the trigger index (into the ADC buffer) 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.)
2. Keep moving this index **backwards** until the waveform crosses the trigger level (in ADC units) in the desired direction.
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 (step 1).
4. Copy samples from half a screen behind to half a screen ahead of the trigger index into a local buffer. (Remember, 1 pixel width = 1 sample interval.)

Manipulating the circular buffer index is made much easier by the wrapping macro `ADC_BUFFER_WRAP()`. It accepts either a positive or negative index and returns an index properly wrapped into the valid range of `gADCBuffer`. Sizing the buffer in powers of 2 simplifies the wrapping operation to a very fast bitwise AND (see macro definition in Step 2).

Trigger search is a **real-time** operation performed in your main() loop. The timing constraint is that the trigger must be found before the ADC ISR circles around and overwrites the samples in the search range. If you attempt to draw the waveform (a slow operation) directly from `gADCBuffer`, you will most likely allow the ISR to overwrite the samples you are drawing. Possible symptoms of this are a discontinuous, unsteady waveform or a waveform not crossing the trigger level at the expected location. To get around this issue, copy one screen worth of waveform into a local buffer (a quick operation) before drawing.

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 (as long as the programmer does not expect it to remain unchanged).

Step 4: ADC sample scaling

To draw the waveform you will need to scale the raw ADC samples such that they conform to a certain volts/division scale. When scaling the ADC values, keep in mind that the ADC value corresponding to V_{in} = 0 V will be close to the **middle** of the ADC range, though not exactly. Measure it by grounding V_{in}.

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 V_{in} = 0 V **(do not assume a value; measure it!)**
`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 V_{in} 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` only needs to change when changing the voltage scale. It can remain constant during waveform plotting. Make sure to interconnect your samples with lines for a smooth waveform look.

Step 5: Button command processing

After you have the waveform display working, implement button command processing and draw the rest of the oscilloscope screen elements, such as the grid (20×20 pixel spacing, behind the waveform) and the settings. The button ISR should detect button press events, just like it did in Lab 0, and then store the button ID (a character) into a **FIFO** for the background to process. FIFO capacity of 10 button presses is sufficient. The FIFO is there so that the main() loop does not miss any button presses even if one loop iteration takes a considerable amount of time, say 1 sec.

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. You may copy the FIFO data structure discussed in lecture from the **ece3849_shared_data** project on the course website (section Modules), but you will need to fix it.

Finally, implement different voltage scales and the optional time scale adjustment. It is convenient to store the scaling constants associated with each voltage scale in an array. String indicators, such as "200 mV" for the 200 mV/div scale, can be stored in a separate array, as in the following example, and used directly with the GrStringDraw() function.

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

Step 6: CPU load measurement

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** in the Modules section of the course website. This example configures Timer3 in one-shot mode (you will need to modify the timeout interval to 10 ms). It then starts it and polls it to timeout, while incrementing a counter. If this code is being interrupted, it will count fewer iterations than if interrupts are disabled. The CPU load can be estimated from the 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.

Extra Credit: Time scale adjustment

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. 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.