ECE 3849 D2021
Real-Time Embedded Systems
Lab 1: Digital Oscilloscope

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

Deliverables

There are three deliverables for this lab:

- Lab signoff completed: 65 points
- Archived project submitted to canvas: 20 points
- Lab report submitted to canvas: 15 points

Each deliverable has a rubric at the end of this report.

To receive credit for the code and report submissions you must sign off.

Any deliverable submitted after the deadline in canvas, will receive a 20% late penalty and can be submitted up to a week late. No submissions after one week late will receive credit.

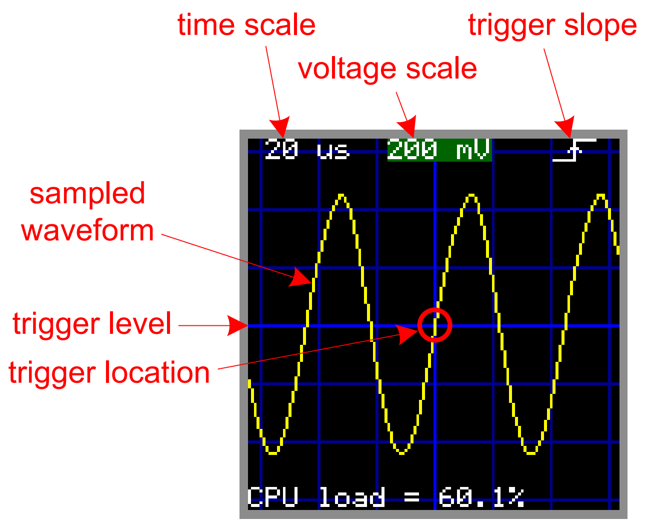
Restrictions

- When writing your software, do not use a real-time operating system.
- You may use TivaWare, TI Graphics Library and other libraries. External / non-TI provided libraries should not be needed.
- Code should be written in C.

Lab Overview

In this first full lab assignment, you will implement a 1 Msps digital oscilloscope using your ECE 3849 lab kit. In addition, you will implement real-time CPU load measurement, which measure the percentage of time spent in ISRs.

Write software that turns your lab kit into a 1 Msps single-channel digital oscilloscope. The LCD display should resemble the following figure. (Your waveform will be a square wave).



Time Scale

- Implement the 20 µs/div time scale.
- The time scale displayed is per division, which is one grid square on the screen (20 pixels).
    - This time scale is 1 ADC sample per pixel at 1 Msps.

Voltage Scale

- The ADC input voltage, $V_{ADC,}$ has a range of 0 to 3.3V. This waveform displayed on the oscilloscope removes the DC offset for a range of -1.65 to +1.65V. $V_{oscope} = V_{ADC} - 1.65V$. $V_{oscope} = 0V$, corresponds to the middle of the ADC range and is displayed in the middle of the screen.
- Implement the 1V/div voltage scale.
- The voltage scale displayed is per division, which is one grid square on the screen (20 pixels).

Triggering

- Implement triggering: when redrawing the waveform, it should cross the trigger level at the same location on the screen.
    - The trigger level should be at $V_{oscope} = 0$ 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 button(s). Button press events should pass from an ISR to main() through a FIFO.
  - If the waveform does not cross the trigger level, display the newest samples unsynchronized. Do not wait forever for a trigger!

Oscilloscope display features (see above example)

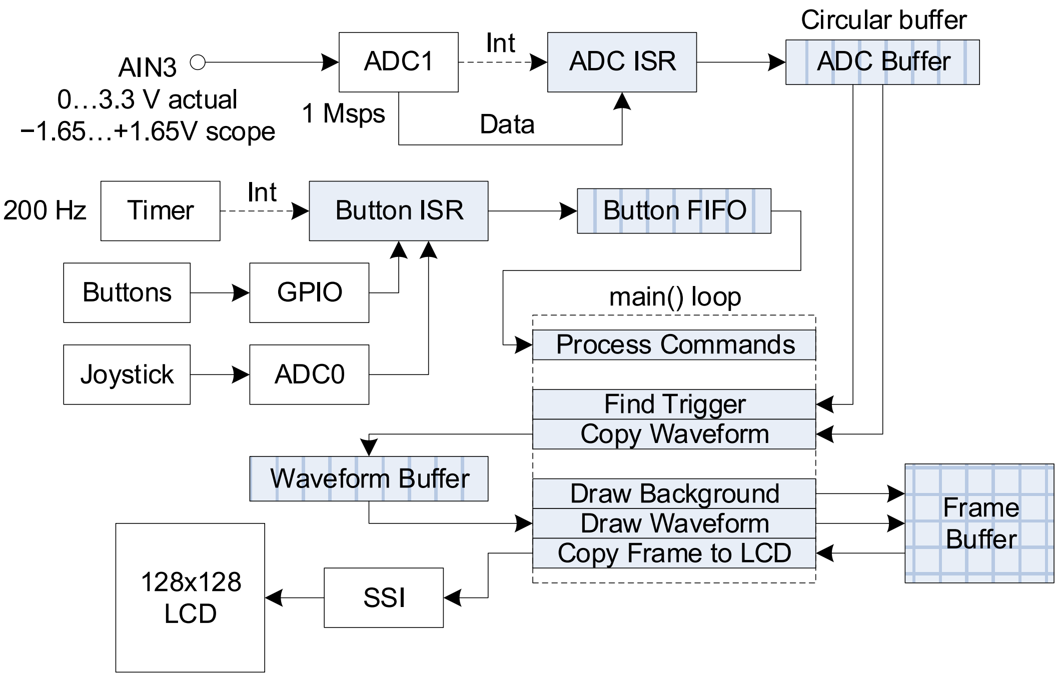
- Waveform should be drawn using line functions calls to give it a smooth waveform look.
- CPU load must be displayed at bottom screen.
- Horizontal and Vertical grid lines must be drawn.
- Text at top of screen includes time per division, voltage per division and either icon or text showing rising or falling edge trigger setting.

Extra credit

- For **extra credit** implement the 2 V/div, 1 V/div, 500 mV/div, 200 mV/div and 100 mV/div voltage scales.
  - Voltage scale must be **adjustable** using button(s). Button press events should pass from an ISR to main() through a FIFO.
  - Voltage scale text on screen should be updated to show scale.
- For **extra credit**, implement additional, switchable time scales: 100 ms/div, 50 ms/div, , 10 ms/div, 5 ms/div, 1 ms/div, 500us/div, 100usec/div 50 µs/div (and retain 20 µs/div from the main assignment).
  - 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.

Software block Diagram

The following block diagram shows the recommended Lab 1 structure.



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 “ece3849d20_lab1_username,” substituting your username. When the time comes to submit your source code, follow the instructions at the end of Lab 0.

You will be reusing the button and joystick processing part of Lab 0. Keep your older completed labs intact.

Step 1: Setting-up the input signal source to the oscilloscope

In this lab we will use a very simple signal source for your oscilloscope to sample: a PWM (pulse-width modulation) output from the same board.

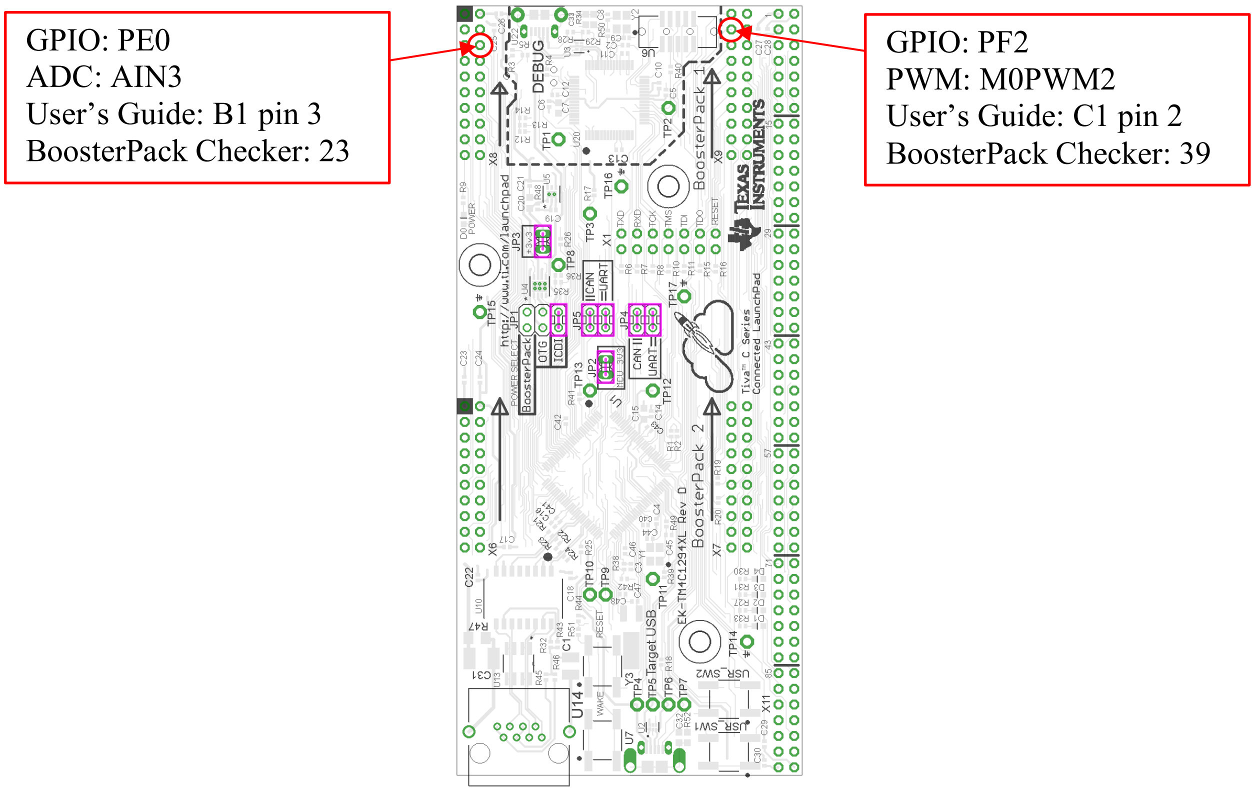
Insert the following code into your main(). The #includes/#define go with the other #includes. The second part goes into the hardware initialization portion of your main(), after the clock has been configured. It is best to place this code in a separate function, such as signal_init(), and call it from main(). This code produces a 20 kHz PWM square wave with a 40% duty cycle on the PF2 (M0PWM2) and PF3 (M0PWM3) outputs.

```
#include <math.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pwm.h"
#include "driverlib/pin_map.h"
#define PWM_FREQUENCY 20000 // PWM frequency = 20 kHz
```

```
// configure M0PWM2, at GPIO PF2, BoosterPack 1 header C1 pin 2
// configure M0PWM3, at GPIO PF3, BoosterPack 1 header C1 pin 3
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);
GPIOPinConfigure(GPIO_PF2_M0PWM2);
GPIOPinConfigure(GPIO_PF3_M0PWM3);
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3,
                 GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);

// configure the PWM0 peripheral, gen 1, outputs 2 and 3
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_1); // use system clock without division
PWMGenConfigure(PWM0_BASE, PWM_GEN_1, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, roundf((float)gSystemClock/PWM_FREQUENCY));
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, roundf((float)gSystemClock/PWM_FREQUENCY*0.4f));
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_3, roundf((float)gSystemClock/PWM_FREQUENCY*0.4f));
PWMOutputState(PWM0_BASE, PWM_OUT_2_BIT | PWM_OUT_3_BIT, true);
PWMGenEnable(PWM0_BASE, PWM_GEN_1);
```

Connect PE0 to PF2 on the EK-TM4C1294XL using a jumper wire. Learn to interpret both the EK-TM4C1294XL LaunchPad User's Guide and the TI BoosterPack Checker website. The following figure shows the locations of PE0 and PF2 on the board layout.



Step 2: ADC sampling

Your first challenge is to acquire ADC samples at 1,000,000 samples/sec without missing any. **Do not place your ADC code in buttons.c**. Start a new module, e.g. **sampling.c.** Move the ADC initialization code from button.c to sampling.c. Configure **ADC1** using the ADC0 initialization in as a reference. Use the following incomplete setup code as a starting point for sample.c:

```
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 currently defined in buttons.h, they should be moved to sampling.h. In sampling.c you will need to include "sysctl_pll.h" to be able to call `SysCtlFrequencyGet()`. "sysctl_pll.h" should be found automatically as it is already be in your TI include paths. Please note that both ADCs must be initialized to the same clock configuration, as they share the clock divider. As such it is recommended to configure ADC0 and ADC1 in the same function in the sampling.c file as shown above.

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. `ADC1_BASE`.

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 a tight timing requirement, but the Cortex-M4F is 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 = 0;                       // 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 takes samples at a fast sampling rate. We need a place to store the data so it can be read and displayed on the oscilloscope functions without having shared data problems. ADC_BUFFER_SIZE defines how many samples are stored. In this case we are storing 2048 samples.

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 out how to clear the interrupt flag that originally caused this ISR to be called, and to read in the ADC sample. You must 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). The TM4C1294NCPDT datasheet Chapter 15 contains information about the ADC operations. Once you have decoded the direct register name, you can use the below instructions to find it in the tm4c1294ncpdt.h file. This file also contains bit field definitions for each bit in the register.

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>`

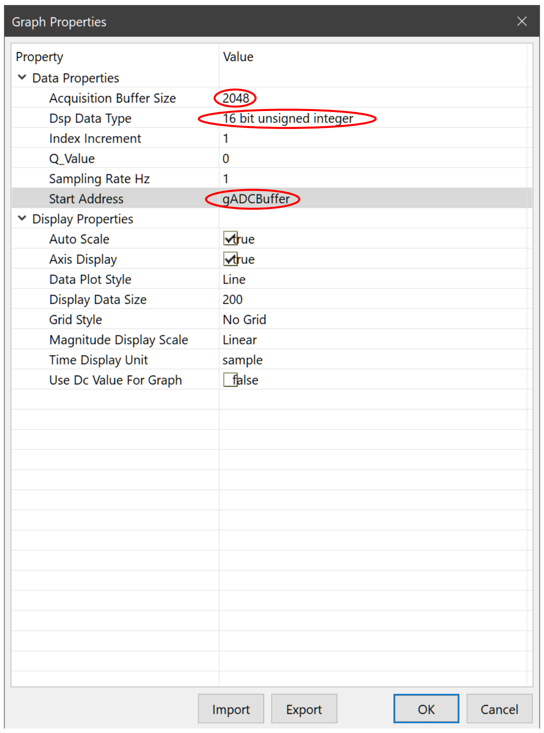
For example, the `TATOCINT` (timer timeout interrupt) bit of the `GPTM`**`ICR`** register can be accessed using `TIMER_ICR_TATOCINT` in C code.

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.

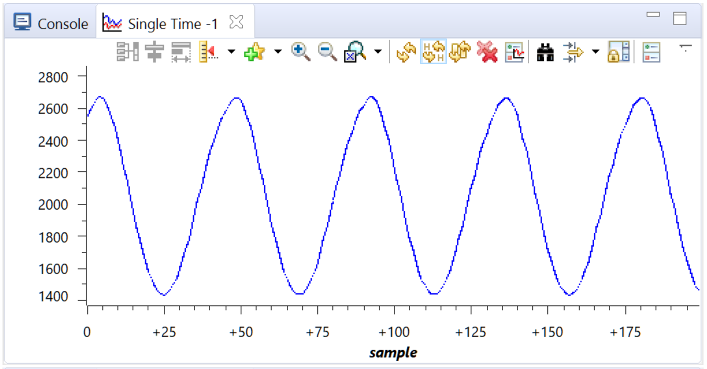
Make sure to refer to this ISR in the appropriate **interrupt vector** in tm4c1294ncpdt_startup_ccs.c (see Lab 0 for an example). At this point, you can run your code and verify that the ISR is being called and the ADC error counter `gADCErrors` stays at zero (it should increment if you pause your program 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 interrupt priorities). As part of sign-off staff will check that gADCErrors stays at zero while running and increments when paused.

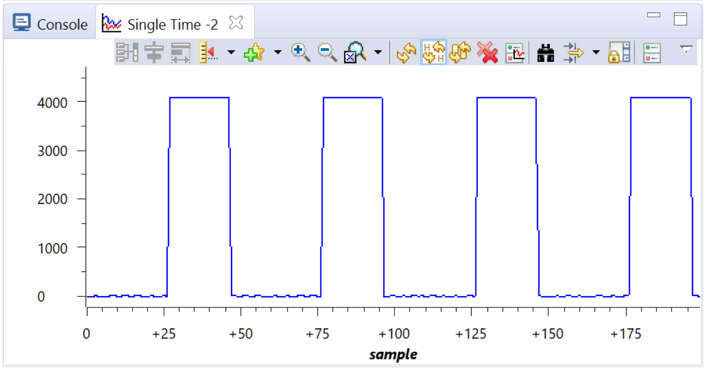
If your signal source is connected, you should be able to observe the sampled waveform in `gADCBuffer` using the debugger. Pause your program in the debugger and navigate the menu to Tools → Graph → "Single Time." You only need to fill in the fields circled below (left screenshot) to get a useful graph (right screenshots). The lower-right graph corresponds to the PWM source.

A care inspection of the graph shows that the raw ADC samples can take on values from 0 to 4095, this is because the ADC has 12-bits of resolution, 2^12 = 4096. The maximum reading will be 2^12-1. The ADC_OFFSET, the value in the middle of the range is half of 4096 = 2048. This is important later when finding the trigger point and plotting the data.



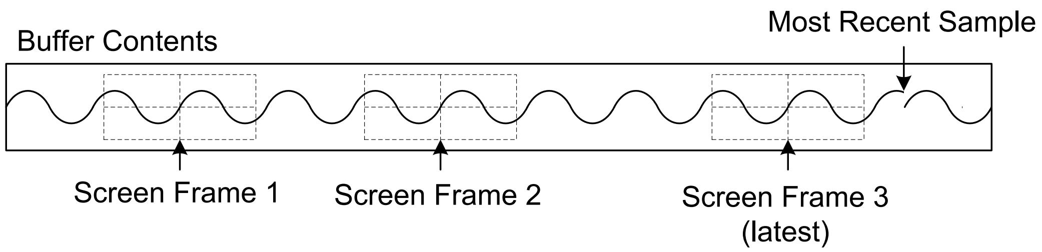
Property	Value
Data Properties	
Acquisition Buffer Size	2048
Dsp Data Type	16 bit unsigned integer
Index Increment	1
Q_Value	0
Sampling Rate Hz	1
Start Address	gADCBuffer
Display Properties	
Auto Scale	✓ true
Axis Display	✓ true
Data Plot Style	Line
Display Data Size	200
Grid Style	No Grid
Magnitude Display Scale	Linear
Time Display Unit	sample
Use Dc Value For Graph	false





Step 3: Trigger search

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. In this lab the trigger voltage is $V_{oscope} = 0$ V, which is in middle of the ADC range. 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 4: 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.)

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).

The first 3 steps for a rising-edge trigger are partially given to you in this function:

```
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;
}
```

Trigger search is a lower priority 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. Copy the part of the waveform you are displaying into a local buffer is a quick operation. Then, draw from the local waveform buffer without any interference from the ISR.

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 in real-time: Perform the desired read 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).

Step 4: ADC sample scaling

To draw the waveform, you will need to scale the raw ADC samples such that they conform to a volts/division scale. When scaling the ADC values, keep in mind that the ADC value corresponding to V_{oscope} = 0 V is in 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 V_{oscope} = 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 V_{ADC} range in volts = 3.3V
`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. Make sure to interconnect your samples with lines for a smooth waveform look, see next step for details.

Step 5: Waveform display formatting

To draw the ADC sample waveform use the Graphics Library functions such as the GrLineDraw functions, these prototypes are in grlib.h. They are also documented in the TivaWare Graphics Library User Guide.

After you have the waveform display working, draw the rest of the oscilloscope screen elements, such as the grid (20×20 pixel spacing, behind the waveform) and the settings. See the Lab overview section for details and example display. GrStringDraw() can be used to display text values similar to Lab 0.

Step 5: Button command processing

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 main() to process. FIFO capacity of 10 button presses is enough. 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.

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. Watch out for shared data bugs in the fifo_get() function, as it can be interrupted anywhere by the button ISR. Interrupts should not be disabled. As explained in lecture, it is possible to create a circular FIFO data structure free of shared data bugs without disabling interrupts.

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. 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. Before interrupts are enabled, the cpu_load_count() function should be run once to establish the count when interrupts are disabled. Once in the main loop the cpu_load_count() function should be run for each frame displayed. The timer polling code should be in a separate function that should not be in-lined with the frame processing. If the polling code is duplicated, the optimizing compiler may compile each version differently, resulting in erroneous measurements. The percent utilization should be displayed on the LCD screen as shown in the Lab Overview section.

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: Implement different voltage scales

Implement different voltage scales. 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", "  2 V"
};
```

Extra Credit: Implement different time scales

Changing the time scale will require an 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 calls to enable the ADC trigger output off a timer. You need to implement many new time scales for full credit: 100 ms/div, 50 ms/div, 10 ms/div, 5 ms/sec, 1 ms/sec, 500us/div, 100us/div, 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 adjustable time scale, you may sample the microphone (GPIO: PD5, ADC: AIN6) instead of the PWM signal.

Sign-off Procedure

Every lab MUST be signed-off to receive credit for the lab.

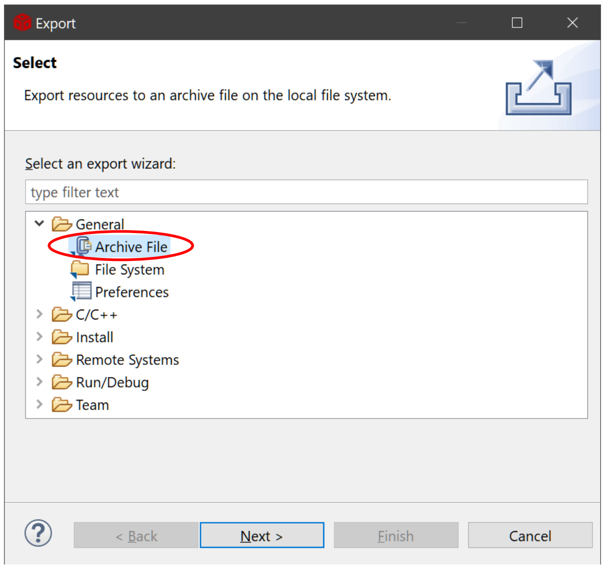
Signoff can be done in-person in AK113 during the officially scheduled lab times on Thursday 10-1:50 or 2-4:50 or via zoom during any lab time or office hour on the Weekly calendar.

If signing-off remotely verify that the functions of your Lab kit can be clearly displayed with zoom's video capability prior to signing off. This may require a use of an external camera, phone , tripod or other equipment.

1) Have your CCS project open and ready to show the TA.
2) The TA will ask you to give a brief walk through the code that you wrote or modified and briefly talk about what was done where in your code.
3) With the TA present Build and Run your CCS project. TA will verify there are no preventable warnings in the code after building.
4) The TA will go through the sign-off check-list and ask you to demonstrate each function. The TA will record results and assign points for all functionality that is correct.
   a) You can demo as many times as needed.
   b) You will always be rewarded points for the functions that are work.
5) Once you are happy with your demonstration results.
   a) You will archive the project that you just demoed and submit the generated zip immediately to the Code Archive assignment in canvas.
   b) The TA will verify the archived project is submitted and complete the sign-off process.

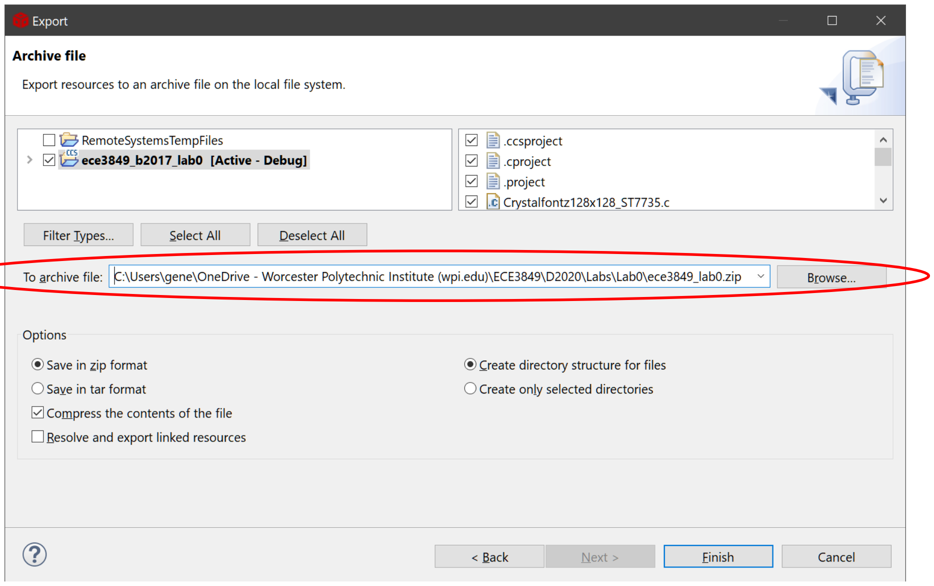
Instructions for archiving

i) Make sure your Lab 1 project is named "**ece3849d21_lab1_username**" with your username substituted.
ii) **Clean** your project (remove compiled code).
iii) Right click on your project and select "Export..." Select General → "Archive File." Click Next.
iv) Click Browse. Find a location for you archive and specify the exact same file name as your project name.
v) Click Finish.



Export resources to an archive file on the local file system.

Select an export wizard:



Export resources to an archive file on the local file system.

To archive file: C:\Users\gene\OneDrive - Worcester Polytechnic Institute (wpi.edu)\ECE3849\D2020\Labs\Lab0\ece3849_lab0.zip

● Save in zip format
○ Save in tar format
☑ Compress the contents of the file
☐ Resolve and export linked resources
● Create directory structure for files
○ Create only selected directories

Sign-Off Check List (65 points)

Project Status (10 points)

1. Student Walk through (5 points)
    - Student was able to give a brief walk through of their work. Showing where and how functionality are implemented. If things are unclear, student should be able to answer questions about their code for clarification.
2. No preventable warnings (3 points)
    - Check the problems window, verify that all preventable warnings are resolved.
3. Project Builds and Runs (2 points)

Functionality Check List

1. Verify gADCErrors (10 points)
    - Run code with gADCErrors displayed continuously updating should stay constant when running and increment when paused.
2. Display has proper formatting (6 points)
    - Gridlines in correct locations.
    - Text at top of screen shows time/division, volts/division and rising / falling trigger text or icon.
3. Waveform is properly displayed (15 points)
    - Waveform is properly centered in screen.
    - Disconnect input and verify flat line.
    - Has expected signal swing and frequency at 1V/division and 20 usec/division.
    - Drawn using lines with smooth look.
4. Selectable trigger slope (rising & falling (12 points)
    - When button is pushed trigger slope changes appropriately and trigger text / icon updated.
5. CPU load (10 points)
    - CPU load is displayed in % at button of screen and updating value.
6. Extra Credit: Adjustable voltage scale (5 points)
    - 2V/div, 1V/div, 500 mV/div, 200 mV/div and 100 mV/div implemented and set via button pushes. Text display updates accordingly.
7. Extra Credit: Adjustable time scale (5 points)
    - 100 ms/div, 50 ms/div, 10 ms/div, 5 ms/sec, 1 ms/sec, 500us/div, 100us/div, and 50 μs/div implemented and set via button pushes. Text display updates accordingly.

Canvas Submissions Completed (2 points)

1. Archived project submitted in canvas at time of sign-off

Source Code Rubric (20 points)

- [8 pts] Button presses pass from an ISR to main through a FIFO. FIFO is free of shared data bugs.
- [3 pts] Performance where needed
  - Do not perform extraneous computations in performance-sensitive code, such as ISRs, critical sections, etc.
  - Use direct register access in the shortest-period ISRs.
  - Avoid excessive optimization where performance is not critical, and readability is more important.
- [3 pts] Structure
  - Control flow should be concise and clear:
    - No excessive nested loops.
    - No long switch statements that could be replaced by a simple computation.
  - Use global variables only if you intend to share them between functions (or for debugging purposes).
  - Separate software into modules for each subsystem, e.g. buttons and sampling. A module is a separate .c file, usually with an accompanying .h file.
- [3 pts] Readability
  - Use driver function calls for hardware initialization.
  - Initialize each subsystem (e.g. buttons and sampling) in a separate function.
  - Do not use “magic numbers.” Define named constants or use existing constants.
  - Use clear variable and function names that reflect their functionality.
- [3 pts] Comments
  - Code should be well-commented: explain the high-level functionality.
  - Each function should have an explanation of its arguments, return value and functionality.

Report Rubric (15 points)

- [2 points] Introduction
  - State objectives of the lab and what you actually accomplished. This section is brief and informs the reader of what they should expect to find in the rest of the document
- [10 points] Discussion and Results
  - Explain the system requirements and real-time goals. Comment on specific things your implementation did to meet these goals.
    - How were tasks prioritized? What was high , middle, low priority?
  - For each major lab component or step explain your implementation to demonstrate your understanding.
    - What functions did you create? How did they work together to implement the required functionality?
    - How did various parts of the code pass information or communicate?
    - If code was provided or given as an example, how did was it incorporated in the design? What modifications were needed to make it work?
- [3 points] Conclusion
  - List significant problems that occurred and how / if they were solved.
  - What you learned from the lab.
  - What you found valuable / interesting about the lab and how it might be improved in the future.