ECE 3849 B2023
Real-Time Embedded Systems
Lab 1: Digital Oscilloscope

This lab will implement a 1 Msps digital oscilloscope by generating a signal, reading it with an ADC, and displaying it on the LCD screen.

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 before you start Lab 2 on **Tuesday November 7**. Late assignments will receive a 20% penalty.

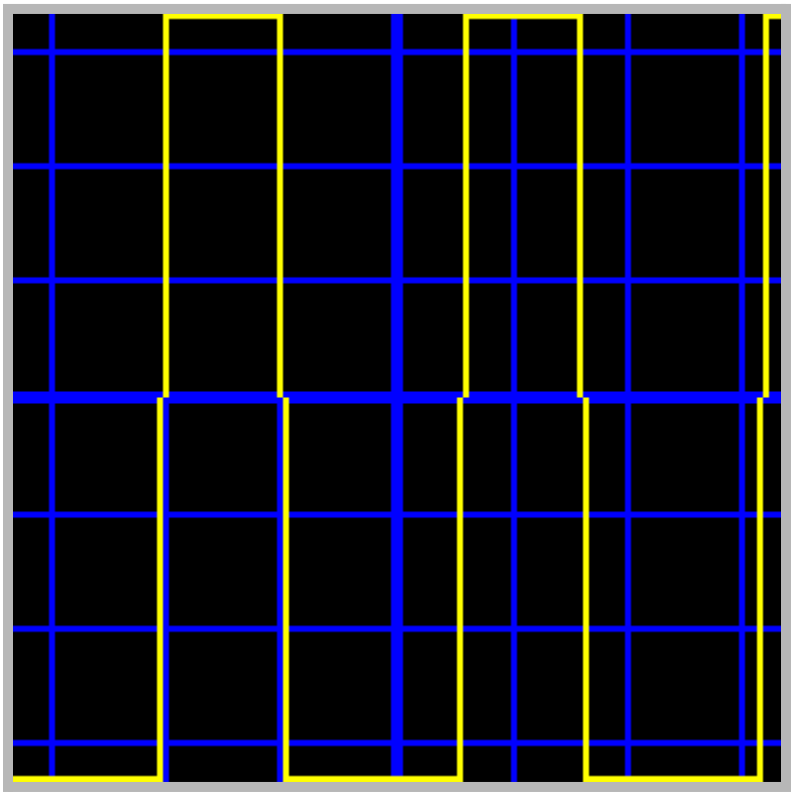
Step	**Max Points**	**Score**
ADC Samples into a circular buffer at 1,000,000 samples/sec without missing samples, and all other features active	25	
Voltage divisions are drawn 20 pixels apart, centered, behind the signal	10	
Signal is drawn on screen with a smooth waveform	10	
Signal is captured by pressing a button and stays the same until button is pressed again	15	
Source Code & Report	25	
Signal is captured without shared data bugs	15	
Total	100	

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 safely
- Use real-time debugging techniques

Assignment

In this lab, you will write software that will turn the lab kit into a simple 1 Msps oscilloscope, resembling the following figure. When drawing, ensure samples are interconnected with lines for a smooth signal display.



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 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 "ece3849_lab1_username1_username2," substituting your username. When the time comes to submit your source code, follow the instructions at the end of the Lab.

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

Part 1: Signal Source

In this part, you will generate a signal for your oscilloscope to measure.

You will generate a very simple signal for the oscilloscope to measure: a PWM (pulse-width modulation) output from the same board. For a more interesting signal source, you may use the microphone on the BoosterPack (GPIO: PD5, ADC: AIN6). However, the microphone signal looks best with a lower sampling rate, which you only implement as extra credit.

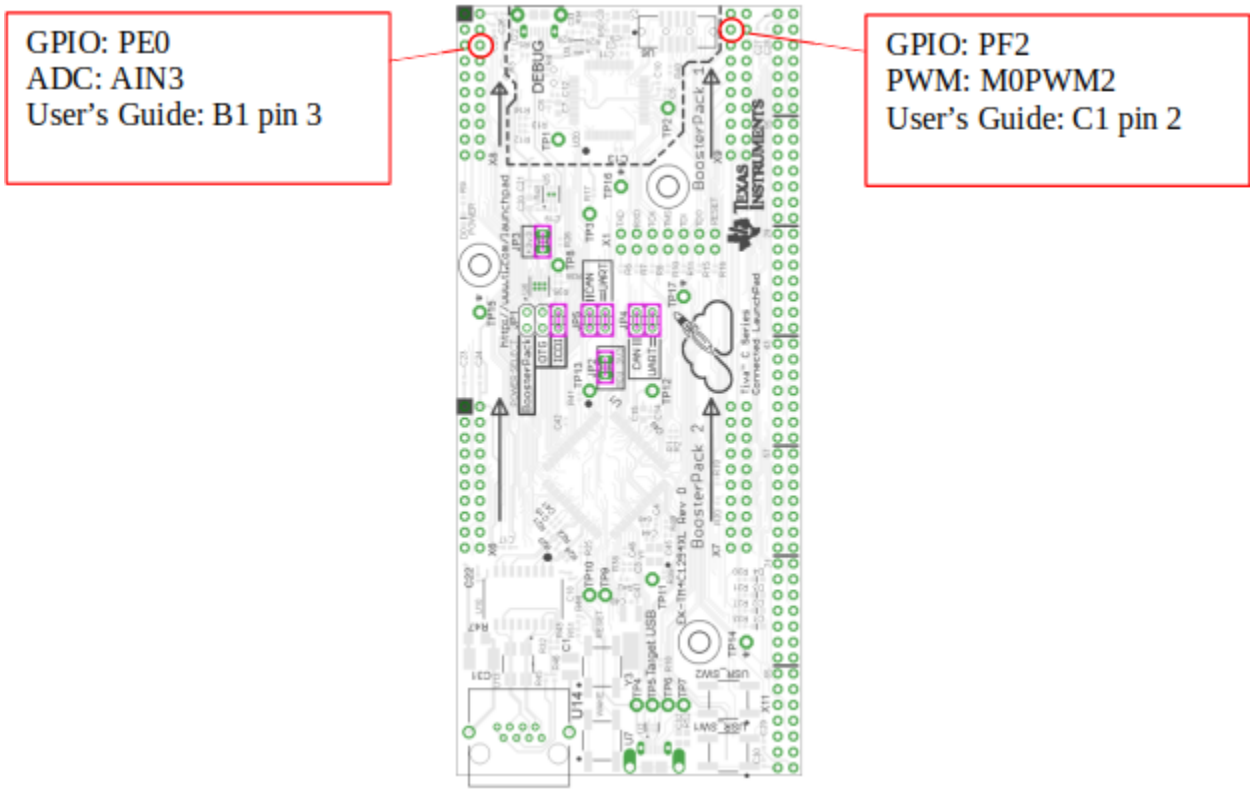
Insert the following code into your main(). The #includes/#define go with the other #includes. The second part goes into the hardware initialization portion of your main(), after the clock has been configured. It is best to place this code in a separate function, such as signal_init(), and call it from 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
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_2);
GPIOPinConfigure(GPIO_PF2_M0PWM2);
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_2,
                 GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);
// configure the PWM0 peripheral, gen 1, outputs 2 and 3
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
 // use system clock without division
```

```
PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_1);
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));
PWMOutputState(PWM0_BASE, PWM_OUT_2_BIT, true);
PWMGenEnable(PWM0_BASE, PWM_GEN_1);
```

Connect PE0 to PF2 on the EK-TM4C1294XL using a jumper wire.



Part 2: ADC Sampling

This section handles capturing 1,000,000 samples per second with the ADC without missing any.

You will configure **ADC1**, using the ADC0 initialization in buttons.c as a reference. The following incomplete code should provide a starting point.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO?);
GPIOPinTypeADC(...); // GPIO setup for analog input AIN3
// initialize ADC peripherals
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
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);
// choose ADC1 sequence 0; disable before configuring
ADCSequenceDisable(...);
ADCSequenceConfigure(...); // specify the "Always" trigger
// in the 0th step, sample channel 3 (AIN3)
// enable interrupt, and make it the end of sequence
ADCSequenceStepConfigure(...);
// enable the sequence. it is now sampling
ADCSequenceEnable(...);
// enable sequence 0 interrupt in the ADC1 peripheral
ADCIntEnable(...);
IntPrioritySet(...); // set ADC1 sequence 0 interrupt priority
 // enable ADC1 sequence 0 interrupt in int. controller
IntEnable(...);
```

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.

The correct arguments to the driver function calls can be found in the ADC chapter of the TivaWare Peripheral Driver Library User's Guide. To call the ADC driver functions, you need to include "`driverlib/adc.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
// index wrapping macro
#define ADC_BUFFER_WRAP(i) ((i) & (ADC_BUFFER_SIZE - 1))
 // latest sample index
volatile int32_t gADCBufferIndex = ADC_BUFFER_SIZE - 1;
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
    <...>;
    // check for ADC FIFO overflow
    if(ADC1_OSTAT_R & ADC_OSTAT_OV0) {
        gADCErrors++; // count errors
        ADC1_OSTAT_R = ADC_OSTAT_OV0; // clear overflow condition
    }
    gADCBufferIndex = ADC_BUFFER_WRAP(gADCBufferIndex + 1)
    // read sample from the ADC1 sequence 0 FIFO
    gADCBuffer[gADCBufferIndex] = <...>;
}
```

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 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). To access registers directly, include "`inc/tm4c1294ncpdt.h`".

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:

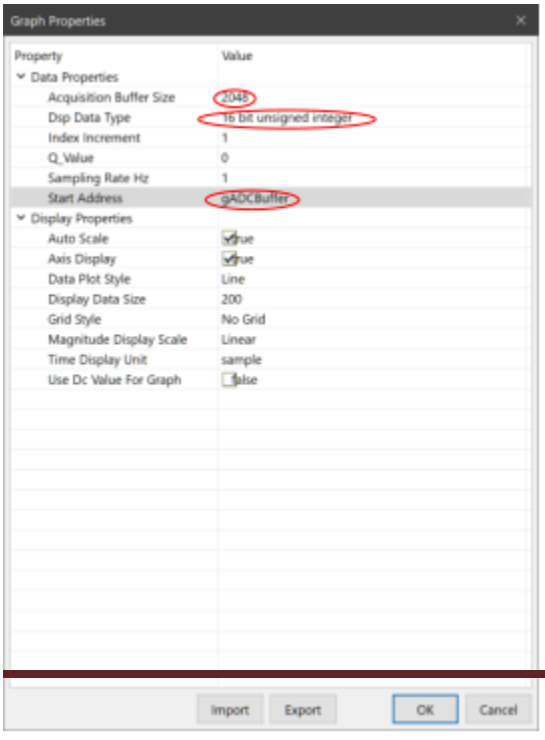
`<C peripheral name>_<register name>_<field name>`

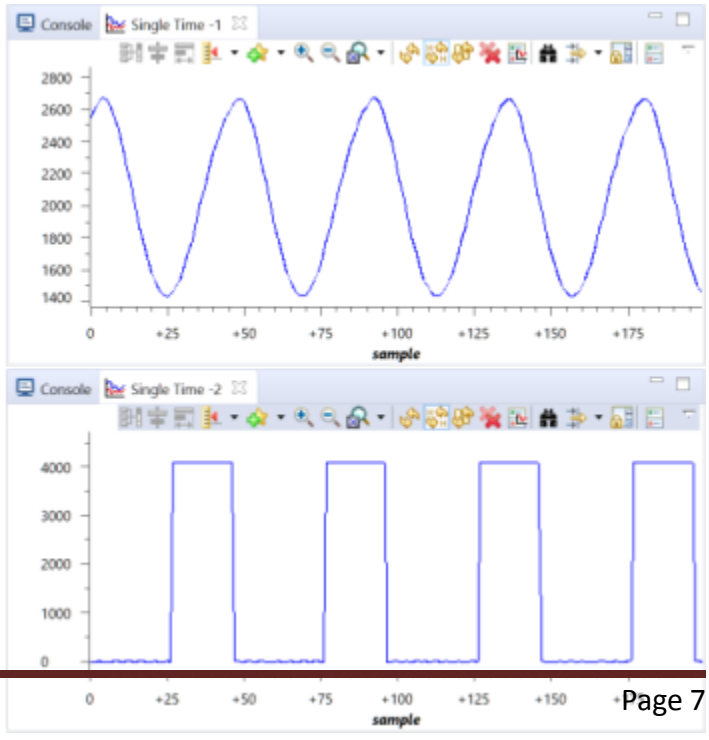
For example, the `TATOCINT` (timer timeout interrupt) bit of the `GPTMICR` register can be accessed using `TIMER_ICR_TATOCINT` in C code.

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.

Create a new module (.c file), e.g. sampling.c, for your ADC initialization code and ISR. To this file, add the function to initialize ADC1. Also include in this file the ADC ISR. Once these have been completed, call the initialization code in main(), and place the ADC ISR in the appropriate **interrupt vector** in tm4c1294ncpdt_startup_ccs.c.

Now, run the code and verify the ISR is being called and `gADCErrors` stays at 0. 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). If your signal source is connected, you should be able to observe the sampled waveform in `gADCBuffer` using the debugger. Pause the program, and verify that `gADCBuffer` is filled with alternating periods of low (< 10) and high (> 4086) values. To verify it visually, go to Tools → Graph → "Single Time." The lower-right graph is the PWM source.



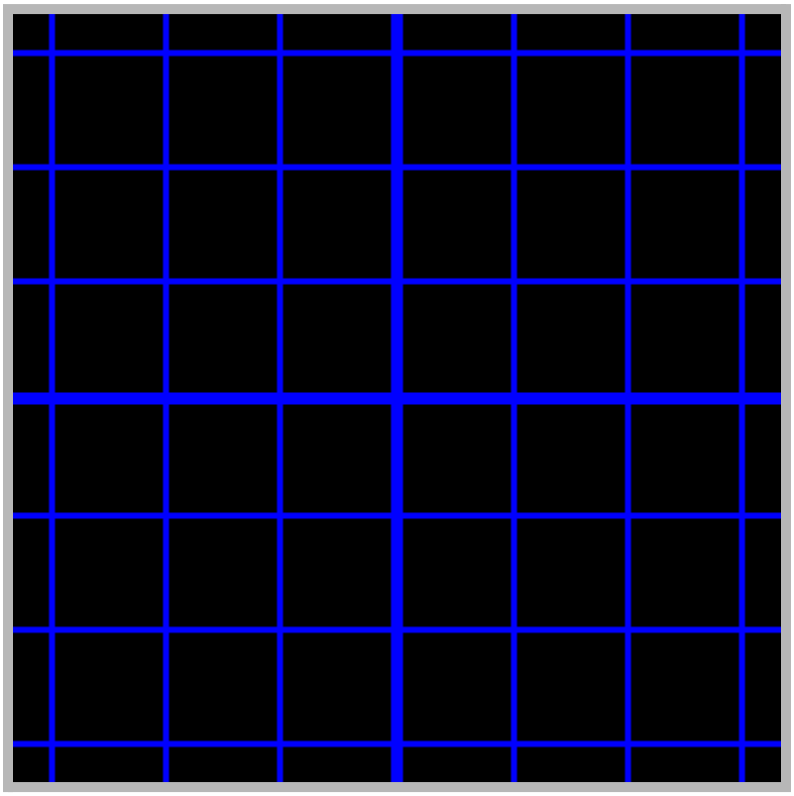


Part 3: Draw Screen Background

Now that we are reading ADC samples, we need something to draw them onto. Draw a grid on the screen with 20 pixels per division to display the signal onto.

See the graphics library documentation for the various screen drawing functions. To change the color that is being drawn, call `GrContextForegroundSet` before drawing to the screen. It is recommended you use a define, such as `PIXELS_PER_DIV` to set the spacing between lines.

The final result should look something like the below figure:



Part 4: Draw the Signal to the Screen

Now that there's a grid for the signal to be drawn on, we should display the signal on the screen.

As the ADC Buffer is constantly being overwritten by the ISR, we must take caution to ensure that we don't encounter any shared data bugs when drawing to the screen. To prevent the data being drawn to the screen from being overwritten, samples should be copied into a buffer, with the buffer then being drawn to the screen.

It is also important to ensure the display on the screen is stable. If we simply drew directly from the buffer to the screen every frame, the display would constantly shift and be unreadable. To solve this problem, we will use a snapshot approach. Here, the user will press a button to take a "snapshot" of the most recent 128 samples, which will then be displayed to the screen. As copying from the samples to the display buffer will take some time, this should be implemented in the `main()` function to avoid adding delay to the button ISR.

The final detail that needs attention is scaling the samples to draw them on the screen. As the max range of the ADC is a product of the screen height, scaling can be implemented simply by dividing the sample values to be in the range of [0, `LCD_VERTICAL_MAX - 1`]. In the next lab, we will implement a system to scale our samples to the divisions on the screen, as well as improving how we select samples to draw to the screen.

Add new code to buttons.c to register a button press and signal `main()` to copy the most recent 128 samples to a buffer. Scale the samples from this buffer and draw them on to the screen.

Part 10: 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.