

ECE 3849 B2023
Real-Time Embedded Systems
Lab 0: Tutorial, Button Handling, Stopwatch

This lab is meant to serve as a tutorial, introducing you to the embedded system you will utilize in this course and the software development environment for it. It will also serve as a review of some material from ECE 2049.

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 1 on **Tuesday October 31**. Late assignments will receive a 20% penalty.



Step	Max Points	Score
Import the starter project and reformat the time to mm:ss:ff	10	
Activate the timer interrupt. Time is incrementing on LCD display	10	
Uncomment button handling code in buttons.c USR_SW1 starts/stops stopwatch Add ability for USR_SW2 to reset the time to 0:00:00	20	
Add BoosterPack buttons and joystick select to bitmap Demonstrate functionality of the buttons in the debugger or by printing them on the LCD screen	40	
Source Code	20	
Total	100	

Objectives

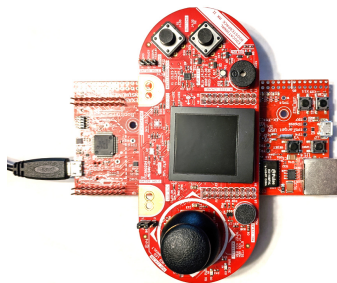
- Install software: Code Composer Studio and TI RTOS
- Set up a Code Composer Studio project
- Output to the 128×128 LCD display
- Configure general purpose timers
- Configure the interrupt controller
- Read and debounce user buttons

Part 1: Lab Kit

There are two main components of the lab kit:

Component	Photo
TI EK TM4C1294XL https://www.ti.com/tool/EK-TM4C1294XL	
TI BOOSTXL-EDUMKII (Educational BoosterPack MKII) https://www.ti.com/tool/BOOSTXL-EDUMKII	

The BOOSTXL-EDUMKII is plugged into the **BoosterPack 2** (middle of the board) connector of the EK-TM4C1294XL. The joystick is on the left when the ethernet connector is on the bottom. The USB connector on the opposite end of the ethernet connector (marked DEBUG) is used to connect the board to a PC.



The primary documentation for the hardware is available from TI's website, and can also be found on the course website.

- Software
 - Available from <https://www.ti.com/tool/SW-TM4C>
 - Peripheral Driver Library: peripheral programming using easy-to-read driver function calls
 - <https://www.ti.com/lit/ug/spmu298e/spmu298e.pdf>
 - Graphics Library: LCD Graphics primitives
 - <https://www.ti.com/lit/ug/spmu300e/spmu300e.pdf>
- Hardware
 - TM4C1294NCPDT Microcontroller Datasheet
 - Low-level MCU hardware details
 - Peripheral programming using direct register access
 - Available from <https://www.ti.com/product/TM4C1294NCPDT>
 - <https://www.ti.com/lit/ds/symlink/tm4c1294ncpdt.pdf>
 - EK-TM4C1249XL LaunchPad User's Guide
 - Routing of MCU pins to the on-board buttons and Launchpad connectors
 - Available from <https://www.ti.com/tool/EK-TM4C1294XL>
 - <https://www.ti.com/lit/ug/spmu365c/spmu365c.pdf>
 - BOOSTXL-EDUMKII BoosterPack User's Guide
 - Routing of the BoosterPack peripheral signals to the Launchpad connectors
 - Available from <https://www.ti.com/tool/BOOSTXL-EDUMKII>
 - <https://www.ti.com/lit/ug/slau599b/slau599b.pdf>

Part 2: Development Tools

This section will deal with installing the development tools that will be used in these labs.

While there are robust, commercial embedded development tools like IAR and Keil, they are expensive or come with code size restrictions. TI's CCS, on the other hand, is free for use with TI microcontrollers, reasonably user friendly and integrates well with TI's real time operating system, TI RTOS. CCS is based on the open source tools Eclipse and GCC.

Before you install the TI software, make sure you have at least 6 GB of free space. A word of caution: Some TI software, specifically parts of TI RTOS, cannot handle spaces anywhere in a file

path. The issue manifests itself as a compilation failure, with an obscure error message. Make sure CCS and your CCS workspace are in locations without spaces in folder names. Default installation folders are recommended.

Download **Code Composer Studio v12.5.0** from <https://www.ti.com/tool/download/CCSTUDIO>
When installing, check the box for TM4C12x processor support, and keep only the default debug probes.

Download **TI-RTOS for TivaC v2.16.01.14** from
https://downloads.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/tirtos/index.html

TI-RTOS should be installed with CCS closed. When you run CCS, go to Window, then Preferences. Under preferences, open “Code Composer Studio”, go to Products, and click the refresh button. You should see XDCtools and TI-RTOS as installable products. Click install.


Part 3: Code Composer Studio

In this part of the lab, you will familiarize yourself with Code Composer Studio and run the starter project.


CCS is based on Eclipse and inherits its basic interface. The very first time you launch CCS, it will ask you to select a folder for the “Workspace” where all your projects will be stored. You can switch to a different Workspace folder at any time using the File menu. Try to select a Workspace folder that sees regular backups, e.g. on OneDrive. Using network drives is not recommended due to slow compilation speed.

Connect your EK TM4C1294XL board to the PC through the USB connector labeled DEBUG (there is also a second USB connector that you should not use). If this is the first time connecting this board, give Windows time to install some drivers.


To begin, download the **ece3849_lab0_starter** CCS project from Canvas. In CCS, go to the Project menu, and select “Import CCS Projects”. In this menu, choose “Select Archive File”, and browse to the downloaded file. This will ensure that the project is properly copied into your workspace.

Once the project is imported, click on the ece3849_lab0_starter project name, then click the Debug  button on the toolbar below the main menu. This compiles the selected project,

loads it onto the target board and halts the CPU at `main()` for debugging. This should also switch CCS to the “CCS Debug” Perspective. The two main Perspectives (window and menu layouts) are “CCS Edit” and “CCS Debug” (indicated by icons in the upper right corner).

Once stopped at `main()`, click the Resume  button on the Debug toolbar. The starter project should run, and you should see “Time = 008345” on the LCD screen. If you cannot get to this point, ask the course staff for help.

Try out the debugger controls. You should be able to halt the executing program, set breakpoints and single step through the code. Tooltips should be enough to identify the functionality of the debugger buttons. There are windows for observing variables, registers (CPU and on chip peripherals), memory and disassembly.

Click the Terminate  button to quit debugging and return to the “CCS Edit” Perspective. Useful shortcuts when editing C code are **Ctrl+i** to **auto-indent** a selection of code, **Ctrl+/
toggle comment** (//) on a selection of code and **Ctrl+click** (or F3) to look up the **definition** of a function, variable, constant, etc. Any compilation errors and warnings are listed in the Problems window and highlighted in the code itself. To completely recompile a project (sometimes necessary because of bugs in CCS), right click on the project name and select “Clean Project,” then build your project again.

Part 4: Starter Project

In this section, you will rename the project and examine its settings and code.

First, right click on the `ece3849_lab0_starter` project and select Rename. Enter the new name: “**ece3849_lab0_username**”, using your WPI username.

Although the starter project is pre-built for you, you should be aware of its most important settings. If you ever need to create a project from scratch, you would have to modify all these settings. Right click the project name and select Properties. Verify the following:

- General
 - Device must be Tiva TM4C1294NCPDT
 - Connection must be Stellaris In-Circuit Debug Interface
- Build
 - Variables tab

- A variable called SW_ROOT needs to exist with the value "C:\ti\tirtos_tivac_2_16_01_14\products\TivaWare_C_Series-2.1.1.71b"
- This is the location of TivaWare, the MCU device driver library.
- This is a Windows path. On a different OS, please browse for the correct path. (e.g. on MacOS, the path typically starts with "/Applications/ti/", and on Linux, "/home/\$USERNAME/ti")
- ARM Compiler
 - Optimization
 - Optimization level
 - Select "off" for easiest debugging (default)
 - Select "1 - Local Optimization" for better execution time at the expense of somewhat harder debugging
 - You may use a higher setting when debugging capability is less important
 - Include Options
 - Add dir to #include search path
 - "\${SW_ROOT}" needs to exist
 - This permit access to the TivaWave includes (.h files)
 - Predefined Symbols
 - PART_TM4C1294NCPDT needs to be defined
- ARM Linker
 - Basic Options
 - Set C system stack size = 2048
 - The default 512-byte stack is too small. It results in stack overflow when running the snprintf() function.
 - File Search Path
 - Include library file or command file as input
 - "\${SW_ROOT}/glib/ccs/Debug/glib.lib" needs to exist
 - This is the pre-compiled TI graphics library
 - "\${SW_ROOT}/driverlib/ccs/Debug/driverlib.lib" needs to exist
 - This is the pre-compiled TivaWare driver library
- Debug
 - Flash Settings
 - **"Reset target during program load to Flash memory"** needs to be checked
 - This makes sure all peripherals are in their default (after reset) state before you start programming them.

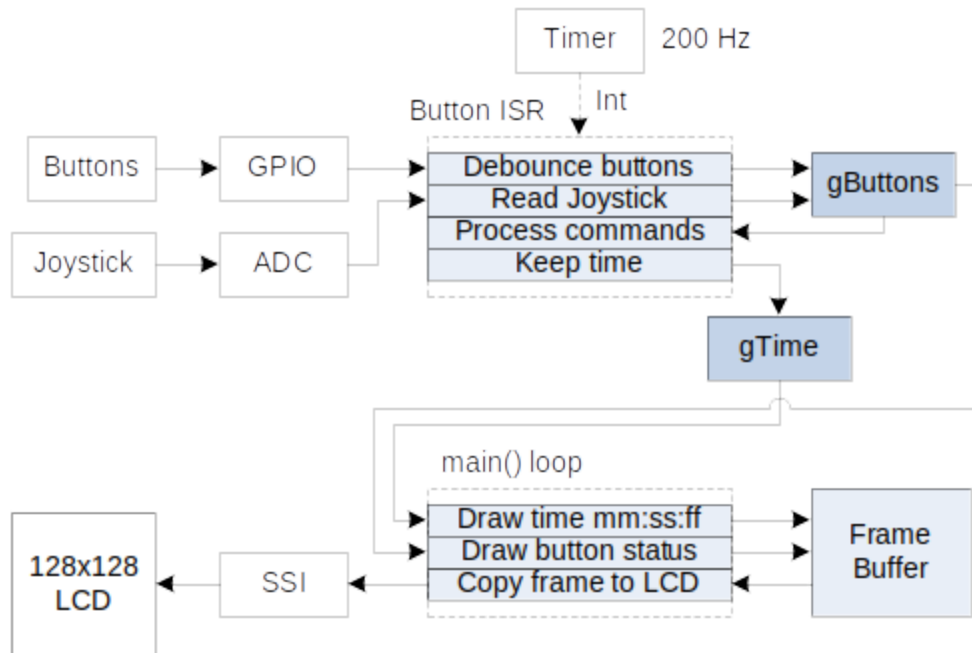
Also, explore the contents of the starter project

- buttons.c, buttons.h
 - Button and joystick handling module
 - You will need to add functionality to these files as part of Lab0.
- Crystalfontz128x128_ST7735.c Crystalfontz128x128_ST7735.h, HAL_EK_TM4C1294XL_Crystalfontz128x128_ST7735.c, HAL_EK_TM4C1294XL_Crystalfontz128x128_ST7735.h
 - LCD driver customized for the specific combination of LaunchPad and BoosterPack
 - You should not modify these files.
- sysctl_pll.c, sysctl_pll.h
 - PLL clock frequency function from sysctl.c – do not modify
- main.c
 - The location of main(), where your application starts up
 - You may implement the entire lab in main.c or break it up into modules (separate .c files with .h files specifying the shared globals and functions).
 - Tm4c1294ncpdt_startup_ccs.c
 - The location of the interrupt vector table and various exception service routines (exceptions are handled just like interrupts).
 - You will need to add ISR references to the interrupt vector table.
 - tm4c1294ncpdt.cmd
 - Linker command file.
 - Contains the locations and sizes of different memory regions (flash and RAM).
 - You should not edit this file.

Part 5: The Lab 0 Application

This part will handle how the lab is structured, and how to approach completing it.

The main objective of this lab is to complete the button handling code in buttons.c. To verify the timing of your code as well as the button handling features, you will add a simple stopwatch as the top-level application in this lab. Your stopwatch will display elapsed time in the format mm:ss:ff (<minutes>:<seconds>:<fraction of a second>) on the LCD screen. User buttons will start and stop the stopwatch, as well as clear the elapsed time. The recommended structure of lab 0 is illustrated in the following figure.



Peripherals (Timer, Buttons, GPIO, etc.), threads (ISR and main() loop) and important global variables (gTime, gButtons, Frame Buffer) are illustrated. Arrows indicate data and command flow. Hardware is in clear boxes, while variables use patterned shading. Commands use dotted lines.

The best way to approach building a complex system is one small step at a time, getting each piece of the system working before continuing. The suggested construction order for this lab is:

1. The `main()` loop that outputs the time in mm:ss:ff format on the LCD.
2. Timer ISR that increments the time.
3. Basic button reading and debouncing functionality in the ISR.
4. Working stopwatch, able to start/stop and reset time.
5. Button ISR that responds to all the buttons and the joystick.

Part 6: Output the time to the LCD display

Here, you will update the code that prints the time to the LCD display to output in the mm:ss:ff format discussed above.

Before writing any code, let's walk through what you are given in the start project. The following code configures the FPU and the clock generator:

```
#include <stdint.h>
```



```
#include <stdbool.h>
#include "driverlib/fpu.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"

uint32_t gSystemClock; // [Hz] system clock frequency

IntMasterDisable();

// Enable the Floating Point Unit, and permit ISRs to use it
FPUEnable();
FPULazyStackingEnable();

// Initialize the system clock to 120 MHz
gSystemClock = SysCtlClockFreqSet(SYSCTL_XTAL_25MHZ |
                                   SYSCTL_OSC_MAIN |
                                   SYSCTL_USE_PLL |
                                   SYSCTL_CFG_VCO_480,
                                   120000000);
```

This code is broken into three sections. The `#include` statements belong at the start of `main.c`. The first two headers, `stdint.h` and `stdbool.h` are required C libraries defining fixed size integers (such as `uint32_t`) and the `bool` type.

The last three are from the TivaWare driver library, allowing us to program the FPU (floating point unit), the System Control module and the interrupt controller.

The second section contains a global variable holding the clock frequency.

The third section is at the start of `main()`. It first globally disables interrupts so we can safely initialize the hardware. It then enables the FPU, so we can use native floating-point math, and configures the CPU clock generator. The clock frequency is saved in the global variable `gSystemClock` for when we need to program timers and similar functionality. Leave this code unmodified at the **beginning of `main()`** for Labs 0, 1, and 2 (it will become redundant in subsequent labs).

Do not modify the `SysCtlClockFreqSet()` call! Misconfiguring the clock can result in your board being inoperable and unrecoverable (bricked).

Next, the LCD driver is initialized:

```
#include "Crystalfontz128x128_ST7735.h"
```

```
// Initialize the LCD display driver
Crystalfontz128x128_Init();
// set screen orientation
Crystalfontz128x128_SetOrientation(LCD_ORIENTATION_UP);

tContext sContext;
// Initialize grlib context
GrContextInit(&sContext, &g_sCrystalfontz128x128);
// select font
GrContextFontSet(&sContext, &g_sFontFixed6x8);
```

There is an `#include` section (LCD driver header that also brings in the TI graphics library header) and a section that goes right after the clock generator code in `main()`. This initializes the SSI/SPI (serial peripheral interface), the LCD controller chip, and the TI graphics library.

After that, time is displayed on the LCD:

```
#include <stdio.h>

// time in hundredths of a second
volatile uint32_t gTime = 8345;

uint32_t time; // local copy of gTime
char str[50]; // string buffer
// full-screen rectangle
tRectangle rectFullScreen = {0, 0,
                             GrContextDpyWidthGet(&sContext)-1,
                             GrContextDpyHeightGet(&sContext)-1};

while (true) {
    GrContextForegroundSet(&sContext, ClrBlack);
    // fill screen with black
    GrRectFill(&sContext, &rectFullScreen);
    time = gTime; // read shared global only once
    // convert time to string
    snprintf(str, sizeof(str), "Time = %06u", time);
    // yellow text
    GrContextForegroundSet(&sContext, ClrYellow);
    GrStringDraw(&sContext, str, /*length*/ -1, /*x*/ 0,
                /*y*/ 0, /*opaque*/ false);
    GrFlush(&sContext); // flush the frame buffer to the LCD
}
```

The sections are `#include`, global variable, and main loop. The infinite `while()` loop is the last thing in the `main()` function.

The graphics driver is buffered for greater performance. Functions like `GrRectFill()` draw to a RAM frame buffer. Then, at the end of the `main()` loop, the function `GrFlush()` copies the RAM frame buffer to the LCD memory through SPI (this is a somewhat time-consuming operation that should be done only when the frame has been completely drawn). Browse the TivaWare Graphics Library User's Guide Section 3.3 for graphics function definitions, particularly how to draw lines.

With the expectation that the global `gTime` will be incremented in an ISR, we read it into a local variable before doing any conversions on it for display (as it could be modified in the middle of our conversion code). If you are not familiar with the `volatile` keyword, look it up or wait until we cover it in lecture.

Modify the `snprintf()` call to reformat the time into the desired `mm:ss:ff` (should display "Time = 01:23:45" instead of "Time = 008345" that this code produces). If you need documentation on the C library function `snprintf()`, look it up on the web.

Part 7: Timer Interrupts

In this section of the lab, you will activate the interrupt for the timer, enabling the time display to start incrementing.

The main functionality of this lab is in `buttons.c` and `buttons.h`. The code for the ISR, and enable it in the peripheral is already provided:

```
#define BUTTON_SCAN_RATE 20    // [Hz] button scanning interrupt
rate
#define BUTTON_INT_PRIORITY 32 // button interrupt priority

#include <stdint.h>
#include <stdbool.h>
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_ints.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/interrupt.h"
#include "buttons.h"

extern uint32_t gSystemClock; // [Hz] system clock frequency

// initialize a general purpose timer for periodic interrupts
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerDisable(TIMER0_BASE, TIMER_BOTH);
```

```
TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
TimerLoadSet(TIMER0_BASE, TIMER_A,
              roundf((float)gSystemClock / BUTTON_SCAN_RATE) - 1);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
TimerEnable(TIMER0_BASE, TIMER_BOTH);

// initialize interrupt controller to respond to timer interrupts
IntPrioritySet(INT_TIMER0A, BUTTON_INT_PRIORITY);
IntEnable(INT_TIMER0A);
void ButtonISR(void) {
    // clear interrupt flag
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    static bool tic = false;
    static bool running = true;
    if (running) {
        if (tic) gTime++; // increment time every other ISR call
        tic = !tic;
    }
}
```

The five sections are:

- #define statements for the interrupt rate and priority in buttons.h
- Includes in buttons.c
- Global variable imported into buttons.c from main.c
- Timer initialization code in ButtonInit() in buttons.c
- The ISR in buttons.c

The timer (Timer0) is configured in 32-bit mode, which consumes both halves of the timer, A and B. In periodic mode, the Load value is the period in system clock cycles minus 1. The documentation for the driver functions used is in the TivaWare Peripheral Driver Library User's Guide. The general-purpose timer peripheral is described in detail in the TM4C1294NCPDT Microcontroller Datasheet.

The timer timeout interrupt is enabled both in the timer peripheral and subsequently in the interrupt controller (NVIC), which is built into the ARM Cortex-M4F core. This interrupt controller is simple to configure and very powerful for real-time work. It supports up to 256 vectored interrupts/exceptions and up to 256 priorities. The Cortex-M4F only implements 8 priorities, specified in the upper 3 bits of the 8-bit priority number. Therefore, the **active priorities** are in steps of 32: **0 = highest, 32 = second highest, 64 = third highest**, etc. This interrupt controller permits never globally disabling interrupts at all. Higher priority interrupts

preempt lower priority ones. This affords extremely low latency for the highest priority interrupt. We will discuss this in detail in lecture.

Note an important feature of the ISR: the clearing of the interrupt flag as the very first step. This is the timeout interrupt flag in the timer peripheral. It is not cleared automatically by hardware because there are multiple interrupt flags in the timer peripheral. Multiple interrupt sources could be handled by the same ISR. If you do not clear the right interrupt flag, the CPU will be stuck executing your ISR over and over.

The only application-related action of this ISR is to increment the global `gTime` every other ISR call (so every 10 ms). Note the use of the `static` keyword. Look it up if you do not know what it means.

In order to activate timer interrupts, the following steps are required.

1. Include `buttons.h` in `main.c`, after the other includes
 - This brings in the function prototypes from `buttons.c` so we can call them from `main.c`
 - It also gives access to the global variables from `button.c` in `main.c`
2. After the LCD graphics are initialized, but before the main loop, call `ButtonInit()` and `IntMasterEnable()`
 - This will call the timer initialization code in `buttons.c`
 - Then, it will globally enable interrupts
3. Finally, add the following to `tm4c1294ncpdt_startup_ccs.c`:

```
void ButtonISR(void);
```

```
ButtonISR, // Timer 0 subtimer A
```

- The function prototype goes in the beginning of the file (look for the comment “External declarations for the interrupt handlers used by the application”).
- The second line **replaces** `IntDefaultHandler` for `Timer0A` in the interrupt vector table. This tells the interrupt controller where to find your ISR.

Then, run the modified project. If all went well, you should see the time incrementing on the LCD.

Part 8: Reading and Debouncing Buttons

Your task for this section is to enable the code to read and debounce buttons, and reset the time when one of the buttons is pressed.

You may have noticed that `buttons.c` contains some block commented code. You should uncomment this code now to enable partial button reading and debouncing functionality:

```
#include "driverlib/gpio.h"
#include "driverlib/adc.h"
#include "sysctl_pll.h"

volatile uint32_t gButtons = 0; // debounced button state

// GPIO PJ0 and PJ1 = EK-TM4C1294XL buttons 1 and 2
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0 | GPIO_PIN_1);
GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0 | GPIO_PIN_1,
                  GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
// configure analog inputs (code not shown)
// configure ADC0 (code not shown)

// read hardware button state
uint32_t gpio_buttons =
~GPIOPinRead(GPIO_PORTJ_BASE, 0xff) & (GPIO_PIN_1 | GPIO_PIN_0);
uint32_t old_buttons = gButtons; // save previous button state
// Run the button debouncer. Output = gButtons.
ButtonDebounce(gpio_buttons);
ButtonReadJoystick(); // Convert joystick state to button presses
uint32_t presses = ~old_buttons & gButtons; // detect button
presses
// autorepeat presses if a button is held
presses |= ButtonAutoRepeat();

if (presses & 1) { // EK-TM4C1294XL button 1 pressed
    running = !running;
}
```

The five sections are:

- driver library includes for the new peripherals used in this code
- global holding the debounced button state
- setup in `ButtonInit()`
 - Configure GPIO to read the two buttons on the EK TM4C1294XL
 - Configure analog inputs for the joystick
 - Configure ADC0 to sample the joystick X and Y analog signals
- button handling in `ButtonISR()`
 - Read the raw state of the buttons into a bitmap

- Debounce the button state such that potentially dirty transitions (multiple rising and falling edges) are cleaned up to single rising and falling edges
- Read the analog joystick state and convert to a button like interpretation
- Detect button press events (transitions from not pressed to pressed)
- command processing in `ButtonISR()`
 - start/stop the stopwatch using a button press event

The most important part of this lab is to understand and complete the button handling functionality, as this will be used in all subsequent labs. The state of all the buttons and button-like inputs (joystick) is stored in a single bitmap in the global variable `gButtons`. The format of this 32 bit bitmap is as follows:

Bits #31..9	8	7	6	5	4	3	2	1	0
unused	Down	Up	Left	Right	Select	S2	S1	USR_SW_2	USR_SW_1

Each bit indicates whether the corresponding button is pressed: 1 = pressed, 0 = not pressed. The least significant 5 bits correspond to actual hardware buttons that must be debounced. `USR_SW1` and `USR_SW2` are the EK TM4C1294XL user buttons. `S1` and `S2` are the BoosterPack buttons on the right. “Select” is activated by pressing down on the joystick. The remaining 4 bits correspond to the joystick directions. Only 9 of the 32 bits in this bitmap are used for buttons. The rest should read zero.

The button debouncing function `ButtonDebounce()` in `buttons.c` accepts as an argument the raw state of all hardware buttons in a 32 bit bitmap `gpio_buttons`, formatted the same as above, but with only the least significant 5 bits. The analog joystick is handled by a separate function. The output of `ButtonDebounce()` is in `gButtons` (global variable). The debouncing algorithm requires a button to read “pressed” for at least 2 samples before changing its debounced state to “pressed.” In the other direction it requires a button to read “not pressed” for at least 5 samples before changing its debounced state to “not pressed.” This suppresses button contact bounces and noise glitches in the button signal, while preserving fast response to button press events.

Study how this code reads the hardware state of the `USR_SW2` and `USR_SW1` buttons. Both buttons are connected to the same GPIO port J, pins 1 and 0. Read the GPIO port using a driver function call. Invert the raw button state because hardware buttons are active low (0 = pressed), and we want active high (1 = pressed). Finally mask out all non button bits using a bitwise AND operation.

Now study how the GPIO peripheral is configured to permit reading these buttons. First, enable the `GPIOJ` peripheral in the System Control module, or you will get an exception trying to access this peripheral. Then program pins 0 and 1 as GPIO inputs. Finally, enable the weak pull ups on these pins (`GPIO_PIN_TYPE_STD_WPU` argument). The last step is only necessary if the hardware button is simply a switch to ground, lacking a pull up resistor.

Examine the last new part of the code that processes commands. It uses the variable `presses` as the input. This is also a bitmap, formatted the exact same way as `gButtons`. However, its bits have a slightly different meaning: 1 = button just transitioned from not pressed to pressed, 0 = all other cases (even if a button is held down). Button presses are checked by using a bitwise AND operation. Pressing `USR_SW1` should start/stop the stopwatch.


Also look at how the ADC is initialized and used to read the joystick. The analog joystick readings are converted to 4 equivalent buttons, with debouncing handled by hysteresis. You do not need to fully understand this code yet. However, you will need to program the ADC yourself for Lab 1. There is also a button press auto repeater thrown in for fun (hold a button for half a second and it starts spitting out new button presses 20 times per second).

Here, you just need to handle another command: reset the time to 00:00:00. Program the `USR_SW2` press event to reset the time. As this button is already read and debounce, you only need to check for a press event in `presses`.

Part 9: Completing Button Handling

In this part, you will complete the button handling code to read inputs from SW1, SW2, and the select button.

The easiest way to find the port and pin each button is connected to is to use the BoosterPack's user guide to find where the inputs you want to connect are, then check the silkscreen on the back of the EK-TM4C1294XL to find what port that matches. All three buttons are in different ports.

To verify that all the buttons are functioning, you can use the debugger. Add `gButtons` to the Expressions window, run your code, and turn on Continuous Refresh . When you press and hold buttons, the value associated with `gButtons` will change. If you change the number format to binary, you can see the bits associated with each button.

ECE 3849 Lab 0

To complete this step, add the code to enable reading from the buttons to `ButtonInit()`. Once you have done this, add code in `ButtonISR()` to read each button and shift it to the correct location in the bitmap.

Part 10: Submitting Source Code

Make sure your Lab 0 project is named “**ece3849_lab0_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 0.

