

ECE 3849 Lab 0

ECE 3849 D2019 Real-Time Embedded Systems Lab 0: Tutorial, Button Handling, Stopwatch

This lab is intended as a tutorial to help you familiarize yourself with the ECE 3849 embedded system and its software development environment, as well as review some ECE 2049 material. This lab does not require a lab report, but you must submit your source code. **Source code submission is required for completion of every lab**, which in turn is required to receive a grade in the course. Complete this lab early to allow yourself more time to work on the more challenging Lab 1.

Have one of the course staff fill in your scores in the following signoff sheet on or before the **signoff due date** listed on the course website to receive full credit. Since there is no report, hand in your signoff sheet to the grader. Fill in one ECE mailbox number for return. Follow the instructions in the “**Lab report guidelines**” document, or at the end of this document to submit your source code.

Step	Max pts	Score
Import the ece3849_lab0_starter project. Reformat the time on the LCD to mm:ss:ff.	3	
Activate the timer interrupt. The time should be incrementing on the LCD.	3	
Uncomment button handling code in buttons.c. EK-TM4C1294XL button 1 should start/stop the stopwatch. Add the ability to reset the time to 00:00:00 using button 2.	3	
Add the two BoosterPack buttons and the joystick select switch to the button bitmap. Draw the gButton state in binary on the LCD. Demonstrate all 9 buttons functioning correctly.	8	
Question about this lab answered	5	Student1: Student2:
Source code submitted on Canvas	3	
Total points	25	Student1: Student2:

Student 1: _____ Student 2: _____

Grader: _____ Date: _____ Mailbox: _____


Learning objectives

- Set up a Code Composer Studio project
- Explore the development board EK-TM4C1294XL w/ BOOSTXL-EDUMKII
- Consult appropriate datasheets, manuals and help pages
- Output to the 128×128 LCD display
- Configure general purpose timers
- Configure the interrupt controller
- Read and debounce user buttons

Part 1: Software development tools

This term we will be developing our embedded software using Texas Instruments Code Composer Studio (CCS). Several commercial and open source IDEs (integrated development environment) and compilers for ARM (the CPU architecture) are available on the market. The most important commercial ones are IAR and Keil. They are well-polished, but 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.











You may download the CCS installer for your own PC directly from TI (http://processors.wiki.ti.com/index.php/Download_CCS). Install CCS **version 8.1.0** so you can easily transfer projects between your own PC and AK113 machines. It is recommended to download the offline installer. When installing, under Processor Support make sure the box “**TM4C12x ARM Cortex-M4F core-based MCUs**” is checked.

If performing your own installation, you will also need to install **TI-RTOS for TivaC**. Run Code Composer Studio 8.1. Open the Resource Explorer from the View menu, select “TI-RTOS for TivaC” under Software, and click “Download and install” .

Part 2: Development board

The course staff will be signing out lab kits containing the TI EK-TM4C1294XL **LaunchPad** with the BOOSTXL-EDUMKII Educational **BoosterPack** MKII. You may purchase your own directly from TI or from a distributor, such as Digi-Key. If you borrow a lab kit, you must return it to receive a grade in this course.

Besides the two boards, the lab kit contains the following breadboard components in an anti-static bag. They must also be returned with the kit at the end of the term. When using the breadboard components, make sure to trim their leads to about 0.5”.

Quantity	Component	Photo/label
10	Jumper Wire 6" Female/Female	
20	Header .100" (number of pins)	
2	Inductor 1 mH	
2	Capacitor 0.1 μ F	 μ1J100
1	Capacitor 0.047 μ F	 47nJ63
1	Capacitor 22 μ F	 220
1	Transistor PNP 2N3906	
3	Resistor 10 k Ω	
1	Resistor 200 Ω	
1	Resistor 60.4 Ω	

If the two boards are not connected, plug the BOOSTXL-EDUMKII (board with LCD and joystick) into the **BoosterPack 2** (middle of the board) connector of the EK-TM4C1294XL. The direction is such that the joystick is on the left when the Ethernet connector is on the bottom.

Locate the primary documentation for this hardware in the Pages → Datasheets section of the course website:

- **TivaWare Peripheral Driver Library User's Guide**
 - Peripheral programming using easy-to-read driver function calls
- **TivaWare Graphics Library User's Guide**
 - LCD graphics primitives
- **TM4C1294NCPDT Microcontroller Datasheet**
 - Low-level MCU hardware details
 - Peripheral programming using direct register access
- **EK-TM4C1294XL LaunchPad User's Guide**
 - Routing of the MCU pins to the on-board buttons and Launchpad connectors
- **BOOSTXL-EDUMKII BoosterPack User's Guide**
 - Routing of the BoosterPack peripheral signals to the Launchpad connectors

When looking up the LaunchPad and BoosterPack pinout, TI provides a handy **BoosterPack Checker** website (also linked from Pages → Datasheets):

- <https://dev.ti.com/bpchecker>
 - Select the LaunchPad and BoosterPack (they are in separate tabs on the left)
 - Click **Toggle Connector** such that the BoosterPack is on **Connector #2**.
 - Hover the mouse over the Connector #1 and #2 pins to receive information on the pin function, GPIO port and pin number, and peripheral signal name.


Part 3: Code Composer Studio


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”. The Workspace is a directory structure used internally by CCS to store your projects. Most of this directory structure is reflected in the OS file system. It is relatively easy to copy CCS projects using the file system, with the only exception that the copied project must be imported into the Workspace using the Project menu. You can switch to a different Workspace folder at any time using the File menu.

Create a folder for your Workspace on your network drive (or USB drive), not on the local machine in AK113. Alternatively, you may run CCS and store the workspace on your own laptop. A known issue is that CCS becomes sluggish when the workspace with many projects is stored on a network drive. To minimize lag, close all unused projects: right-click on a project or multiple selected projects in the Project Explorer and select “Close Project” in the context menu. These projects then become dormant, and need to be opened again (similar procedure) to be accessed. Alternatively, you may store your workspace on your laptop or a USB drive, but make sure to keep backups. Compiler performance is much better with local project storage.


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 (monitor its progress by double-clicking the driver installer tray icon).

Download the **ece3849_lab0_starter** CCS project from the Lab0 assignment area of the course website. Extract the zip file, placing the ece3849_lab0_starter folder as a sub-folder of your Workspace folder. Import this project into CCS by going to the menu Project → “Import CCS Projects...” Click Browse, then click OK. Your Workspace folder should already be selected. Check the ece3849_lab0_starter project and click Finish.

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” for editing code and “CCS Debug” for debugging (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. The GUI layout here is also very intuitive. In the Project Explorer you can double-click on a project/folder to expand it, and on a file to edit it. Useful shortcuts when editing C code are **Ctrl+i** to **auto-indent** a selection of code, **Ctrl+/** to **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), navigate the menu Project → Clean..., select the project you want recompiled (instead of all projects), select “Start a build immediately” and “Build only the selected projects,” then click OK.

Part 4: Starter project

Right click on the ece3849_lab0_starter project and select Rename. Enter the new name: “ece3849d19_lab0_username1_username2,” where username1 and username2 are the usernames of you and your partner. This project naming convention is necessary for electronic source code submission.

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_00_08\products\TivaWare_C_Series-2.1.1.71b”
 - This is the location of TivaWare, the MCU device driver library
 - 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}/glibc/ccs/Debug/glibc.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

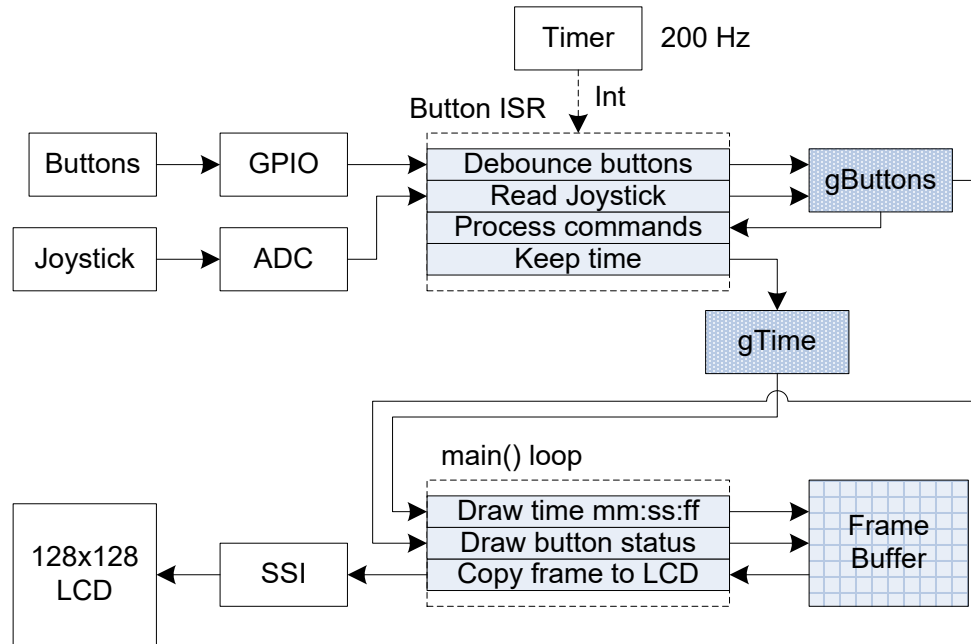
- Debug
 - Flash Settings
 - “**Reset target during program load to Flash memory**” needs to be checked
 - This makes sure all peripherals are in their default (after reset) state before you start programming them.

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

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. You will also display the status of all the user buttons on the LCD. 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 responds to all the buttons and the joystick.

Part 6: Outputting the time to the LCD display

Before writing any code, let's walk through what you are already given in the starter 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
```

ECE 3849 Lab 0

```
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 are required C libraries defining fixed size integers (such as `uint32_t`) and Booleans. 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 and 1 (it will become redundant in subsequent labs).

The next step is to initialize the LCD driver, also given to you:

```
#include "Crystallfontz128x128_ST7735.h"
Crystallfontz128x128_Init(); // Initialize the LCD display driver
Crystallfontz128x128_SetOrientation(LCD_ORIENTATION_UP); // set screen orientation

tContext sContext;
GrContextInit(&sContext, &g_sCrystallfontz128x128); // Initialize grlib context
GrContextFontSet(&sContext, &g_sFontFixed6x8); // select font
```

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.

The next step is to display the time on the LCD, also given to you:

```
#include <stdio.h>
volatile uint32_t gTime = 8345; // time in hundredths of a second
```



```
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);
    GrRectFill(&sContext, &rectFullScreen); // fill screen with black
    time = gTime; // read shared global only once
    snprintf(str, sizeof(str), "Time = %06u", time); // convert time to string
    GrContextForegroundSet(&sContext, ClrYellow); // yellow text
    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

The main functionality of this lab is partially given to you in buttons.c and buttons.h. We will now activate the ISR (called ButtonISR()) that is called 200 times per second by a timer.

The code to get the ISR going is already given to you in buttons.c and buttons.h:

```
#define BUTTON_SCAN_RATE 200 // [Hz] button scanning interrupt rate
#define BUTTON_INT_PRIORITY 32 // button interrupt priority
#include <stdint.h>
#include <stdbool.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, (float)gSystemClock / BUTTON_SCAN_RATE - 0.5f);
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) {
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // clear interrupt flag

    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 the timer interrupts, you need to perform the following steps:

- Add the following right after the other includes in `main.c`:

```
#include "buttons.h"
```

- This brings in function prototypes from `buttons.c` so we can call them from `main.c`
- It also gives access to global variables from `buttons.c`

- Add the following after the LCD/graphics initialization but before the main loop in `main.c`:

```
ButtonInit();  
IntMasterEnable();
```

- This calls the timer initialization code in `buttons.c`
- It then globally enables interrupts

- 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.

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

Part 8: Reading and debouncing buttons

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"  
  
// 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
ButtonDebounce(gpio_buttons);       // Run the button debouncer. Output = gButtons.
ButtonReadJoystick();               // Convert joystick state to button presses
uint32_t presses = ~old_buttons & gButtons; // detect button presses
presses |= ButtonAutoRepeat();       // autorepeat presses if a button is held

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

The four sections are:

- driver library includes for the new peripherals used in this code
- 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. 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_SW2	USR_SW1

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 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. We will study button debouncing details later in lecture.

Study how this code reads the hardware state of the USR_SW2 and USR_SW1 buttons. Both of these 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.

Your task for this part of the lab is to handle another command: reset the time to 00:00:00. Program the USR_SW2 press event to reset the time. Note that this button is already read and debounced by existing code. You only need to check for a press event in **presses**, bit position 1.

Also take a 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).

Part 9: Completing the button handling

Follow the hardware button reading example from the previous section to read the three remaining buttons (S1, S2 and Select) and bitwise OR them into the **gpio_buttons** bitmap. You will need to shift the button bits into the locations specified in the table in Part 8. Be aware that all 3 buttons you are responsible for are in different GPIO ports. Use the TI BoosterPack Checker website (see Part 2 of this lab) to quickly find to which GPIO port and pin each button is connected. Add code to **ButtonInit()** to configure each GPIO port.

To verify that all 9 buttons are now functioning, you may use the debugger. Add **gButtons** to the Expressions window, change its Number Format to binary, run your code and turn on Continuous Refresh 🔄. Now press and hold one button at a time. You should see a single 1 bit in **gButtons** in that button's position.

To practice outputting to the LCD, print the 9 least significant bits of **gButtons** in binary below the stopwatch time on the LCD. Demonstrate to the grader that all 9 buttons work, and when pressed, produce a 1 in their bit position on the LCD. Pressing multiple buttons simultaneously should produce multiple 1s in your binary display.

Part 10: Submitting source code

Make sure your Lab 0 project is named “ece3849d19_lab0_username1_username2” with your usernames 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 on your network drive and specify the exact same file name as your project name. Click Finish. Submit the resulting .zip file to Canvas under Lab 0. This concludes Lab 0.