ECE 3849 D2021
Real-Time Embedded Systems
Lab 0: Tutorial, Button Handling, Stopwatch

1) Getting Started

Before starting the lab, you should use the documents posed in Canvas to

- Purchase a lab kit
- Install software: Code Composer Studio and TI-RTOS
- Become familiar with the Lab Reference Material posted in Canvas

2) Lab Objectives

- Setup a CCS project.
- Output to the 128×128 LCD display
- Configure general purpose timers
- Configure the interrupt controller
- Read and debounce user buttons

3) Setup the Lab 0 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.

1) 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.
2) Download the **ece3849_lab0_starter** CCS project from the Lab0 Module on Canvas
3) Start Code Composer Studio version 9.3
4) Enter the directory path of your workspace for Lab 0.
5) Extract the zip file, placing the ece3849_lab0_starter folder as a sub-folder of your Workspace folder.
6) Import this project into CCS
   a) Select menu Project → "Import CCS Projects..."
   b) Click Browse, then click OK. Your Workspace folder should already be selected.
   c) Check the ece3849_lab0_starter project and click Finish.
7) Click on the ece3849_lab0_starter project name, then click the Debug button on the toolbar below the main menu.
   a) This compiles the selected project, loads it onto the target board and halts the CPU at main() for debugging.
   b) 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).

8) 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.
9) 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.
10) Click the Terminate button to quit debugging and return to the "CCS Edit" Perspective.
11) 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.
12) 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.

4) Rename the project

Right click on the ece3849_lab0_starter project and select Rename. Enter the new name: "**ece3849d21_lab0_username**", substituting your WPI username. This project naming convention is necessary for electronic source code submission.

5) Review the project settings (no changes required)

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/")
  - 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}/grlib/ccs/Debug/grlib.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.

6) Explore the contents of the starter project:

Files you will need to edit

- 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).
- buttons.c, buttons.h
  - Button and joystick handling module
  - You will need to add functionality to these files as part of Lab0.
- tm4c1294ncpdt_startup_ccs.c
  - Contains the interrupt vector table.
  - You will need to add a Button ISR to this table.

Files that you should NOT edit

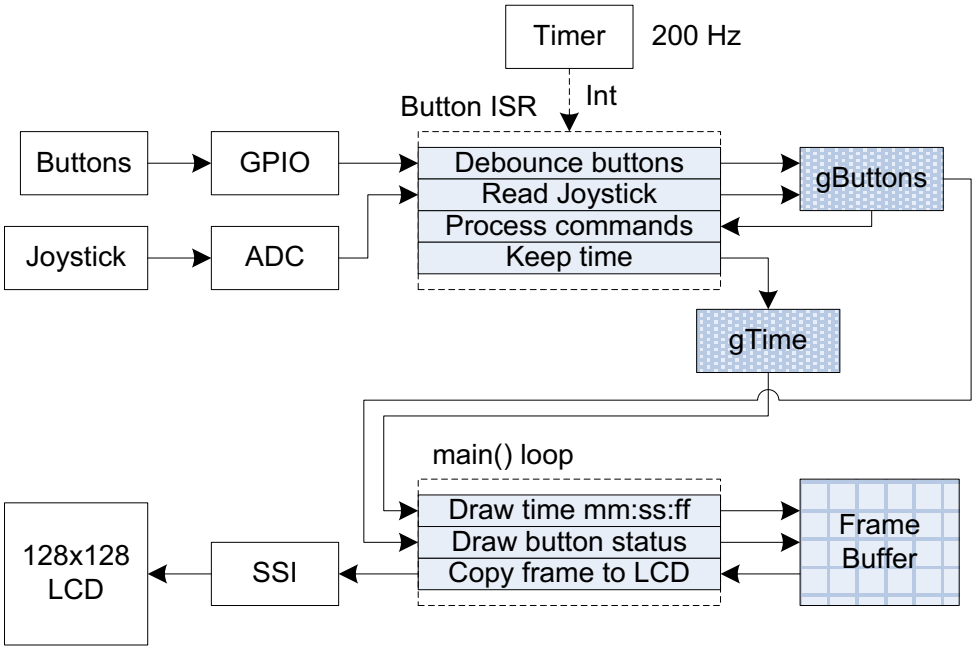
- sysctl_pll.c, sysctl_pll.h
  - PLL clock frequency function from sysctl.c
- Crystalfontz128x128_ST7735.c, Crystalfontz128x128_ST7735.h, HAL_EK_TM4C1294XL_Crystalfontz128x128_ST7735.c, HAL_EK_TM4C1294XL_Crystalfontz128x128_ST7735.h
  - LCD driver customized for the specific combination of LaunchPad and BoosterPack
- tm4c1294ncpdt.cmd
  - Linker command file.
  - Contains the locations and sizes or different memory regions (flash and RAM).
  - You should not edit this file.

7) Lab 0 Application Introduction

In Lab 0 you will be implementing the following functionality

- You will implement a stopwatch that will display elapsed time in the format mm:ss:ff (<minutes>:<seconds>:<fraction of a second>) on the LCD screen.
- You will configure an interrupt to increment the gTime global variable, so that the time increases while the stopwatch is running.
- You will use the user buttons to start and stop the stopwatch, as well as clear the elapsed time.
- You will display the status of the user buttons on the LCD display.

The 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.

8) Update time format display mm:ss:ff on LCD screen

Description Summary

The first task is to properly format the time displayed on the LCD screen such that instead of showing 008345 it displays the time in mm:ss:ff format where mm is minutes, ss is seconds and ff is fractions of a second. “Time = 008345” would be displayed as “Time = 01:23:45”

FPU and Clock Generation Code

Before writing any code, lets walk through what you are already given in the starter project. The following code snippets from **main.c** configure the FPU and the clock generator.

```
1  #include <stdint.h>
   #include <stdbool.h>
   #include "driverlib/fpu.h"
   #include "driverlib/sysctl.h"
   #include "driverlib/interrupt.h"
2  uint32_t gSystemClock; // [Hz] system clock frequency
3  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.

1) 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.
2) The second section contains a global variable holding the clock frequency, gSystemClock.
3) The third section is at the start of main() and from the TivaWare driver library, allowing us to program the FPU (floating point unit), the System Control module and the interrupt controller.
    - The IntMasterDisable() call globally disables interrupts so we can safely initialize the hardware.
    - The FPU commands enable 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).
    - **Do not modify the SysCtlClockFreqSet() call!** Misconfiguring the clock can result in your board being inoperable and unrecoverable (bricked).

LCD Drive Initialization Code

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

```
#include "Crystalfontz128x128_ST7735.h"
Crystalfontz128x128_Init(); // Initialize the LCD display driver
Crystalfontz128x128_SetOrientation(LCD_ORIENTATION_UP); // set screen orientation

tContext sContext;
GrContextInit(&sContext, &g_sCrystalfontz128x128); // 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 is 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.

Displaying Time on the LCD Code

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

```
1  #include <stdio.h>
2  volatile uint32_t gTime = 8345; // time in hundredths of a second
3  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
   }
```

1) The stdio.h file must be included for this functionality.
2) The gTime is a global variable that holds the time value.
3) 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 on Canvas** 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.

Changes needed too reformat the time

Modify the snprintf() call in the infinite while loop 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 – Setup time to increase.

Description Summary

The original code shows a static time value of 01:23:45. To implement the a stop watch this value needs to track the actual time elapsed. For this to happen a timer interrupt must be configured to allow gTime global variable to increment every 10 msec.

Buttons.c and Button.h walk through

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:

1 buttons.h	`#define BUTTON_SCAN_RATE 200    // [Hz] button scanning interrupt rate` `#define BUTTON_INT_PRIORITY 32  // button interrupt priority`

2
buttons.c
top
	`#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"`

3 buttons.c top	`extern uint32_t gSystemClock;   // [Hz] system clock frequency`

4
buttons.c
ButtonInit
	`// 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);`

5
Buttons.c
ButtonISR
	`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:

1) #define statements for the interrupt rate and priority in buttons.h
2) includes in buttons.c
3) global variable for system clock frequency imported into buttons.c from main.c
4) timer initialization code in ButtonInit() in buttons.c
5) 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** posted in Canvas. The general-purpose timer peripheral is described in detail in the **TM4C1294NCPDT Microcontroller Datasheet** posted in Cavnas.

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.

Changes needed to activate the timer interrupts

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

Changes needed to enable partial button reading

You may have noticed that buttons.c contains some block-commented code. If it is not already uncommented, you should uncomment this code now to enable partial button reading and debouncing functionality:

1
button.c
top
	`#include "driverlib/gpio.h"`
	`#include "driverlib/adc.h"`
	`#include "sysctl_pll.h"`

2 button.c top	`volatile uint32_t gButtons = 0; // debounced button state`

3
button.c
ButtonInit
	`// 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)`

4
button.c
ButtonISR
	`// 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`

5
button.c
ButtonISR
	`if (presses & 1) { // EK-TM4C1294XL button 1 pressed`
	`running = !running;`
	`}`

The five sections are:

1) driver library includes for the new peripherals used in this code
2) global holding the debounced button state
3) setup in ButtonInit()
   a) Configure GPIO to read the two buttons on the EK-TM4C1294XL
   b) Configure analog inputs for the joystick
   c) Configure ADC0 to sample the joystick X and Y analog signals
   d) Uncomment the whole block not just the lines shown in the table.
4) button handling in ButtonISR()
   a) Read the raw state of the buttons into a bitmap
   b) Debounce the button state such that potentially dirty transitions (multiple rising and falling edges) are cleaned up to single rising and falling edges
   c) Read the analog joystick state and convert to a button-like interpretation
   d) Detect button press events (transitions from not pressed to pressed)
5) command processing in ButtonISR()
   a) start/stop the stopwatch using a button press event

Button handling functionality walk through

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.

In ButtonInit(), study how this code reads the hardware state of the USR_SW2 and USR_SW1 buttons (PJ0 and PJ1). Both buttons are connected to the same GPIO port J, pins 1 and 0.

Section 3, ButtonISR(), reads the GPIO port using a driver function call. Inverts the raw button state because hardware buttons are active low (0 = pressed), and we want active high (1 = pressed). Finally, it mask out all non-button bits using a bitwise AND operation.

Section 4, ButtonInit(), 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 in only necessary if the hardware button is simply a switch to ground, lacking a pull-up resistor.

Section 5, Button ISR(), 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.

Changes needed for USR_SW2 to clear time

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. Toward the end of ButtonISR(), you only need to check for a press event in `presses`, bit position 1.

Other button functions

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).

Part 9: Adding and displaying button status

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** to quickly find to which GPIO port and pin each button is connected. Add code to ButtonInit() to configure each GPIO port. See **BoosterPack Checker for Pinout file** posted in Lab 0 module canvas for details.

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. Pressing multiple buttons simultaneously should produce multiple 1s in the binary value.

Print the 9 least significant bits of `gButtons` in binary below the stopwatch time on the LCD. Your goal is to permit the grader to evaluate your lab only by running your code and interacting with the on-board controls.

Sign-off Procedure

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

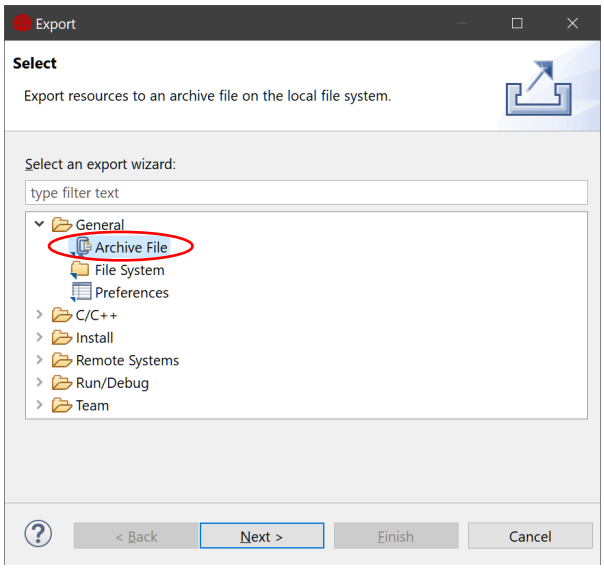
Signoff can be done in-person in AK113 during the officially scheduled lab times on Thursday 10-1:50 or 2-4:50 or via zoom during any lab time or office hour on the Weekly calendar.

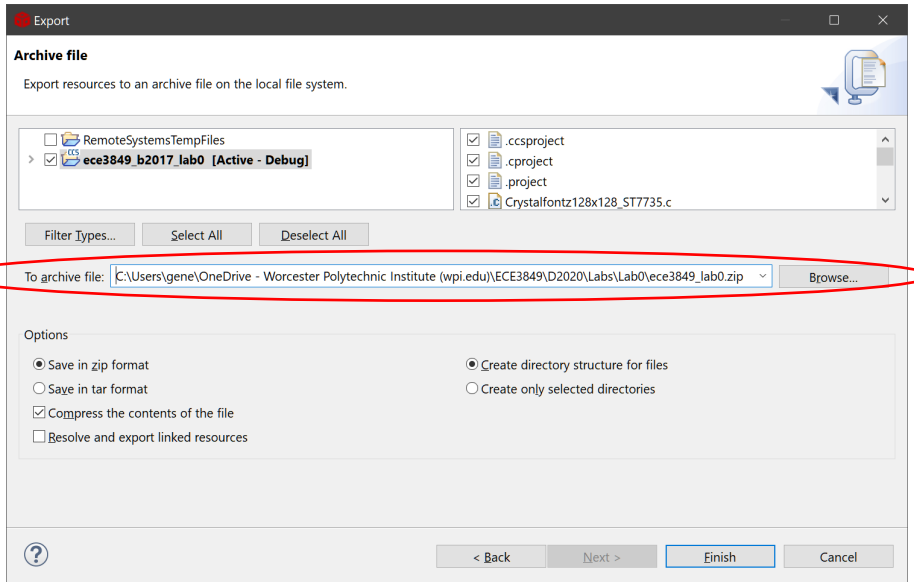
If signing-off remotely verify that the functions of your Lab kit can be clearly displayed with zoom's video capability prior to signing off. This may require a use of an external camera, phone , tripod or other equipment.

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 0 project is named "**ece3849d20_lab0_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.





Sign-Off Check List (80 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. Time is displayed in mm:ss:ff format (10 points)

- Time has proper format.

2. Time is incrementing by default (10 points)

- Before pressing any buttons, time is incrementing.

3. USR_SW1 switch start and stops stopwatch (10 points)

- Press USR_SW1 several times verify that time starts and stops incrementing

4. USR_SW2 clears stopwatch (12 points)

- With stopwatch stopped at a non-zero time, press USR_SW2, verify time is cleared to 00:00:00.

5. gButton's value is displayed on LCD (8 points)

- Verify that there is a 9-bit binary value is displayed below the time.

6. Verify that all 9 buttons work correctly (18 points -2 per)

- Verify that all 9 function correctly and are displayed in the correct bit field of the binary value displayed.

8	7	6	5	4	3	2	1	0
Down	Up	Left	Right	Select	S2	S1	USR_SW2	USR_SW1

Canvas Submissions Completed (2 points)

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

Source Code Rubric (20 points)

1. Code is well written, commented and formatted (5 points)

- Where changes were made there should be comments about functionality.
- Proper indentation and no excessively long lines.
- Functionality being implemented is clear.
- Good use of defines and functions.
- Variables well named making functions obvious.
- Good use of global and local variables, i.e. globals used only when needed.

2. Expected functionality is present (15 points)

- Main.c is modified with time formatting and gButton display.
- Main.c is modified to enable timer functions.
- Button.c is modified with enabled pins, additional press statements.
- tm4c1294ncpdt_startup_ccs.c vector table is updated.
- Archived project must build in CCS without errors and run successfully on hardware.