LAB 4

PUSH BUTTON DIGITAL INPUTS AND CYBOT-PC COMMUNICATIONS

INTRODUCTION

This week you will program push button messages using general-purpose input/output pins (GPIO), bitwise operations, and polling. The GPIO pins are grouped into sets of 8 pins called ports. A port is accessed with a data register, and the data register is accessed using a memory address – this is called memory-mapped input/output. A programmer must initialize a port before using it for input/output. Initialization is done using a set of configuration registers associated with a GPIO port. You will use relevant concepts and skills to develop functionality that allows a user to interact with a running program on the CyBot through push buttons, the LCD screen, and other communications.

Read each step in this manual completely before attempting it so that you do not miss out on important details.

REFERENCE FILES

The following reference files will be used in this lab:

- button.c, contains functions that you will implement for this lab and use in future labs
- button.h, header file for button.c
- lab4_template.c, contains a main function template that you will implement for this lab
- cyBot_uart.h, header file for pre-compiled CyBot-PC UART communication library
- libcybotUART.lib: pre-compiled library for CyBot-PC UART communication (note: must change extension of file from .txt to .lib after copying)
- lcd.c, program file containing various LCD functions
- lcd.h, header file for lcd.c
- timer.c, program file containing various wait commands
- timer.h, header file for timer.c
- TI Tiva TM4C123G Microcontroller Datasheet
- TI TM4C123G Register Definitions C header file: REF_tm4c123gh6pm.h
- Cybot baseboard and LCD schematics: Cybot-Baseboard-LCD-Schematic.pdf
- GPIO register list: GPIO-registers.pdf
- GPIO Reading Guide: reading-guide-GPIO.pdf **← BROWSE THROUGH THIS**

The code files are available to download.

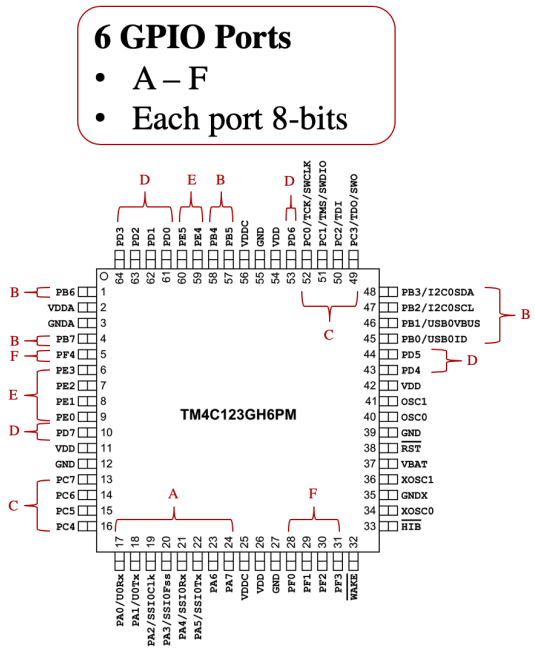
Being familiar with the GPIO reading guide may help you find specific information in various resources. This lab manual also cites specific resources and contents. **Your goal is to be aware of information so that you can find and read it in more detail if and when needed. You are not expected to read or know everything at once.** Take it one step at a time. Be patient but persistent and purposeful. What do you need to know? What do you know that you can build on? What don't you know that might take more effort? Where or how might you get started? (Arthur Ashe: "Start where you are. Use what you have. Do what you can.")

PRELAB

See the prelab assignment in Canvas and submit it prior to the start of lab.

STRUCTURED PAIRING

You are expected to continue to use structured pairing in this lab and in future labs. It was introduced in Lab 2.



PART 1: READING PUSH BUTTONS

Write a short program in main using a polling method (continuously checking the status of a device in software using a loop) that repeatedly calls the function **button_getButton()** to check if a button has been pushed and displays the result on the LCD screen.

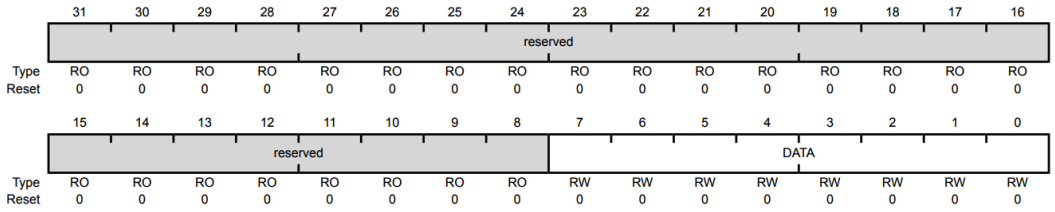
To begin, you will need to complete an API function for reading the push buttons on the LCD board by initializing the GPIO port in the function **button_init()** and finishing the **button_getButton()** function in button.c. Some template code has been provided for you in the button.c file.

Chapter 10 (pg. 649) in the Tiva datasheet provides details on using GPIO ports on the microcontroller. Remember that the microcontroller contains 6 GPIO ports (A-F), which each contain 8 bits. Each port has a set of registers associated with it, and a full list of registers can be seen in table **10-6. GPIO Register Map** on pg. 660.

You will be using GPIO Port E to program the buttons on the LCD board. You will need to use your knowledge of bitwise operations to mask bits in the data register and detect which button has been pressed.

You will find section **10.3 Initialization and Configuration** in the datasheet helpful for initializing and configuring the GPIO port pins. Initialization, or configuration, of the GPIO pins is necessary in order to use the GPIO port and to read the status of a push button, i.e., pressed or not (note: buttons are active low, meaning that pressing a button results in a low or logic 0 signal on the digital input).

Page 662 in the Tiva datasheet corresponds to the **GPIODATA** register. Read through the register description in the datasheet to get familiar with the information provided. Your code will need to read the GPIODATA register in order to determine which button has been pressed. **Pins 0 – 3 of GPIO Port E correspond to buttons 1 – 4 on the LCD board.** The **GPIODATA** register diagram from the datasheet is shown below.



Once you have completed **button_init()**, you should complete **button_getButton()** to return the button pressed. If no button is pressed, 0 should be returned.

****Reminder: Don't forget to initialize your I/O interfaces. Initialize the button interface by calling the button_init() function that you implemented for this lab. Initialize the LCD screen by calling lcd_init() (note: also call timer_init() before lcd_init() since it uses timer functions). Initializations should only be called once and should always be called at the beginning of your main function (i.e. NOT in a loop).**

CHECKPOINT:

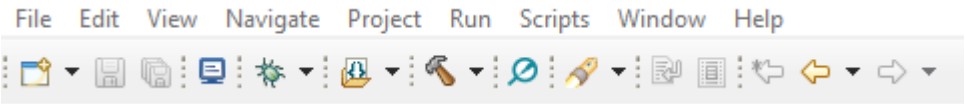
The position of the button pressed should be displayed on the LCD screen.

PART 2: DEBUGGING PUSH BUTTONS

In Lab 1, you explored the variables, registers, and expressions tabs. To solidify what you have observed about the debugger already, we will go over how to: create watches, look at the value of a variable, set breakpoints, and step-over lines of code. Additionally, you'll want to familiarize yourself with other features of Code Composer Studio, such as the outline tab and the disassembly tab.

2.1 DEBUGGING: ADDING BREAK POINTS, STEP INTO, STEP OUT AND STEP OVER

As you have already learned in Lab 1, you can debug your program by selecting **Run → Debug**, clicking the green bug-like button in the toolbar, or using the keyboard shortcut **F11**.



You can step through your program one instruction at a time using the **Step Over** button (the second yellow arrow) or the keyboard shortcut **F6**.

****Tip:** Hovering over icons with the mouse will show names of buttons and their shortcuts.

When you come to a function call, you can enter the function by using the **Step Into** button (the first yellow arrow) or the keyboard shortcut **F5**. When you are in a function, if you want to step out of the function, use **Step Return** (the third yellow arrow) or the keyboard shortcut **F7**. Stepping out of a function returns to the calling function on the line directly following the call to the function. You should experiment with these buttons to increase your understanding of the code that you are writing.

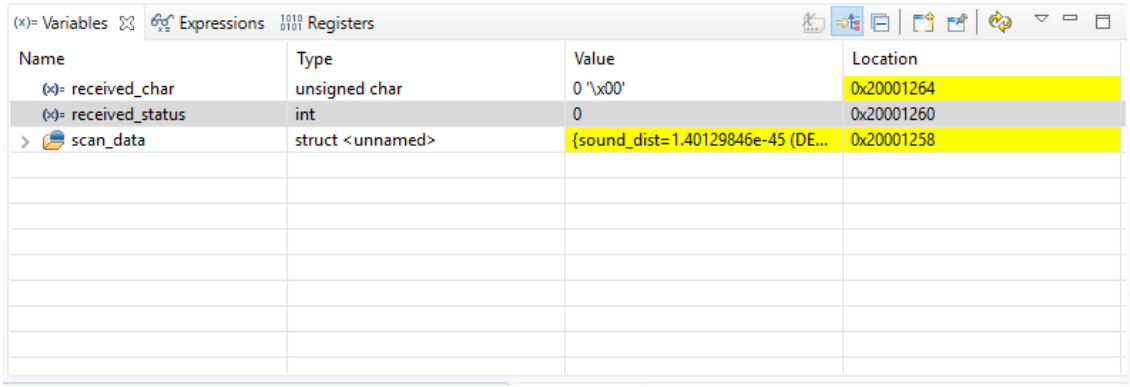
A very helpful feature in debugging is setting breakpoints if you know the specific section of code that needs to be debugged or you need to determine where problems start in your code. To add a break point to your program, double click on the line number that you would like your program to stop at. A circle will appear indicating a break point has been added. You can toggle the break point by double clicking on it again.

```
10    oi_t *sensor_data = oi_alloc();
11    oi_init(sensor_data);
12
13    move_forwardCollisionDetect(sensor_data, 200);
```

Place a break point somewhere in your program, then click the **Continue** button (the green play button) or the keyboard shortcut **F7**. The processor will run your program until it encounters your break point. At this point you can step over additional lines of code line if needed or set a new breakpoint and resume your program again.

2.2 Variables, Expressions, and Registers Tabs

While the program execution is paused, you can inspect the current state of all variables and registers. Variables declared in the current function are displayed in the **Variables Tab,** which shows the variable name, type, values stored, and the location in memory. Code Composer Studio also allows you to create a watch expression by using the **Expressions Tab.** This tab is used to monitor the value of variables in other parts of your code.



To add a watch expression, you can either right click on a variable name in your code and select "**Add Watch Expression**", or you can click the green plus button in the **Expressions Tab** and type a variable name or expression. The variable and its value will appear in the **Expressions Tab** (next to **Variables Tab**). You can even edit the value of a variable in the tabs while debugging. Experiment with adding watches and practice looking at variable values while you step through your code to increase your understanding.

One additional tab that you may find useful is the **Registers Tab.** This tab shows information about the core registers, as well as GPIO registers, timer registers, etc. You may find this feature particularly useful as you are initializing peripheral registers. For example, if you have initialized all registers and find that your code is not working, you may accidentally be disabling something in a register.

2.3 Disassembly Tab

Take time to familiarize yourself with the other debugging features of Code Composer Studio by exploring the following toolbar icons:

- **Disassembly Tab**
  Shows the assembly code of your program. Assembly instructions have a one-to-one correspondence with machine code. The address of the assembly instruction in program memory is listed on the left, followed by a description of the assembly instruction as well as the line of code that it corresponds to. To open this, go to **Window → Show View → Disassembly**. This feature may be particularly helpful towards the end of the semester when you study assembly language.

- **Outline Tab**
  Shows an outline of your program that includes: include statements, global variables, defines, and function prototypes. This may be helpful for quickly navigating your code files or checking that you have all of the right includes and global variables.

Checkpoint:

You should be able to explain and use debugging features such as stepping over, stepping into, and step return and the variables, expressions, and registers tabs.

Part 3: CyBot Communication Using PuTTY

In this part of the lab, you will send messages from the Cybot to your desktop PC when push buttons are pressed on the Cybot.

1. Set up communication between the PC and the Cybot as in Lab 3. The communication uses a UART serial port on the Cybot and a PuTTY terminal on the PC and your code uses the libcybotUART library. Refer as needed to the quick reference sheet for configuring PuTTY on the PC.

2. Send a message to PuTTY when you press a button on the LCD board. You should program one message for each button.

Checkpoint:

You should be able to press a button and send a message specific to that button from the Cybot to the PC.

DEMONSTRATIONS:

1. **Functional demo of a lab milestone –** Specific milestone to demonstrate in Lab 4: Checkpoint for Part 3
2. **Debug demo using debugging tools to explain something about the internal workings of your system –** The TA will announce any specific debugging requirements at the start of lab; otherwise you will create your own debug demo based on your needs and interests in the lab.
3. **Q&A demo showing the ability to formulate and respond to questions –** This can be done in concert with the other demos.