Part 1. Push Button System

System Sketch

Similar to the system sketch you did in Lab 1, sketch a diagram that shows how the 4 push buttons connect to the microcontroller. Your diagram should show the port and the bit of the port that each button is connected to and how they connect to the microcontroller pins (give the pin number and detailed port names used on the microcontroller). Use the following resources: (1) Datasheet, (2) CyBot Baseboard diagram, (3) CyBot LCD board diagram. As part of your figure, indicate the port and bit of the port that each button is connected.

Questions: The port connected to the push buttons is associated with a specific GPIO data register.

a) What is the specific name of the data register as given in the Tiva TM4C Datasheet?

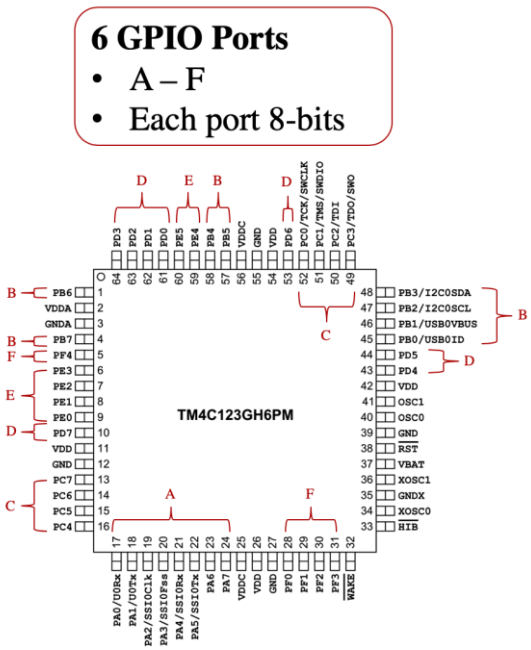
b) What is the specific name of the data register as given by the #define macro in the header file, tm4c123gh6pm.h (TM4C123GH6PM Register Definitions)?

c) What is the 32-bit memory address of the data register (in hex)?

Part 2: Push Buttons Using Polling

Write a short program in main using a polling method that repeatedly calls the function **button_getButton()** to check if a button has been pushed and displays the result on the screen.

To begin, you will need to complete an API function for the push buttons on the LCD display board by initializing the GPIO ports 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.

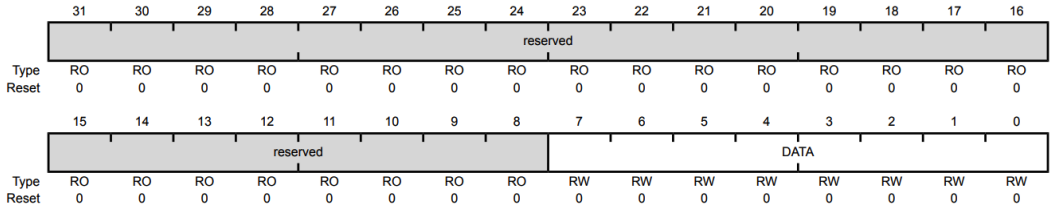


Chapter 10 (pg. 649) in the Tiva Datasheet provides details on using GPIO ports on the microcontroller. Remember that the microcontroller contains 6 GPIO Ports (A-F) which each contain 8 bits. Each port has a set of registers associated with it and a full list of registers can be seen in table **10-6.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

You will be using GPIO Port E to program the buttons on the LCD module. You will need to use your knowledge of bitwise operators to masks bits in the data register and detect which button has been pressed (Note: Buttons are Active Low).

You will find section **10.3 Initialization and Configuration** in the data sheet helpful for initializing and configuring the GPIO port E pins. Initialization and configuration of the GPIO pins is necessary in order to use the GPIO ports and to read in data signaling that a button has been pressed.

Page 662 in the Tiva datasheet corresponds to the **GPIODATA** register. It is recommended that you read the register description as you will need to read the GPIODATA register in order to tell which button has been pressed. Pins 0 – 3 on the GPIO port correspond to buttons 1 – 4 on the LCD module.



Once you have completed **button_init()**, you should complete **button_getButton()** to return the button pressed. When multiple buttons are pressed, return the rightmost button. For example, if both buttons 1 and 3 are being pressed at the same time, the function should only return button 3, as it is the rightmost button being pressed, and ignore the other buttons. If no button is pushed, 0 should be returned.

****Reminder: Don't forget to initialize buttons by calling the button_init() function that you completed and the lcd screen by calling lcd_init(). Initializations should only be called once and should always be called at the beginning of your main function (i.e. NOT in a loop).**

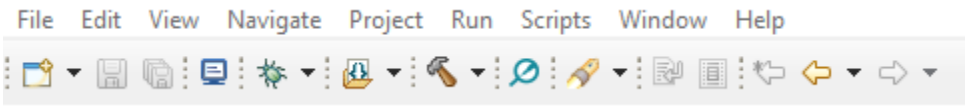
Checkpoint: Demonstrate push buttons to your mentor. The position of the rightmost button pressed should be displayed to the LCD screen.

Part 3: Debugging Push Buttons

In Lab 1 you analyzed 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.

3.1 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**.

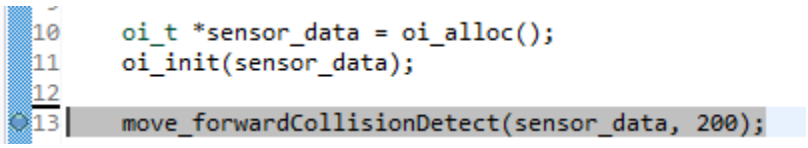


You can step through your program one instruction at a time using the **Step Over** button (the second yellow arrow) or the keyboard shortcut **F6**.

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

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 like 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.

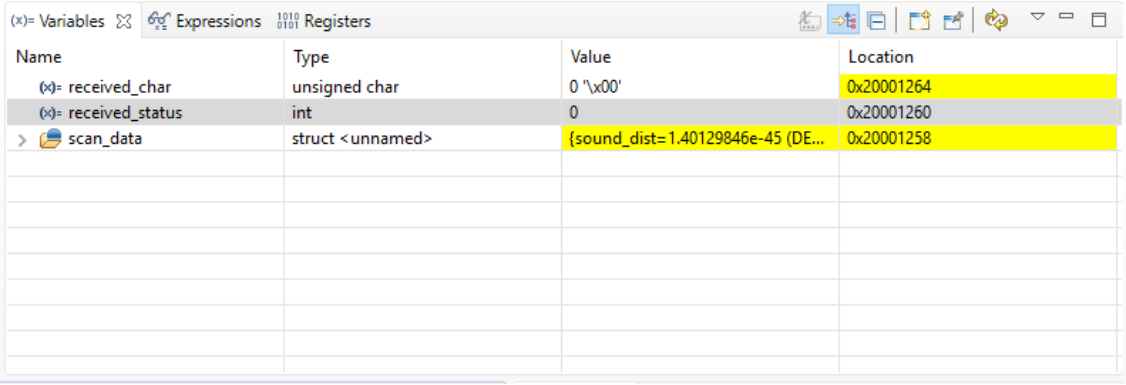
A very helpful feature in debugging is setting breakpoints if you know the specific section of code that needs to be debugged or you need to determine where problems start in your code. To add a break point to your program, double click on the line number that you would like your program to stop at.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



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.

3.2 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 the variables 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 will be able to look into this more in the next part of this lab. You may find this feature particularly useful as you are initializing registers. For example, if you have initialized all registers and find that your code is not working you may accidentally be clearing a register.

3.3 Registers Tab and Disassembly Window

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: Demonstrate debugging to your mentor. You will need to demonstrate stepping over, stepping into, and step return and explain the variables, expressions, and registers tabs.

Part 4. Communication from CyBOT push button to Putty

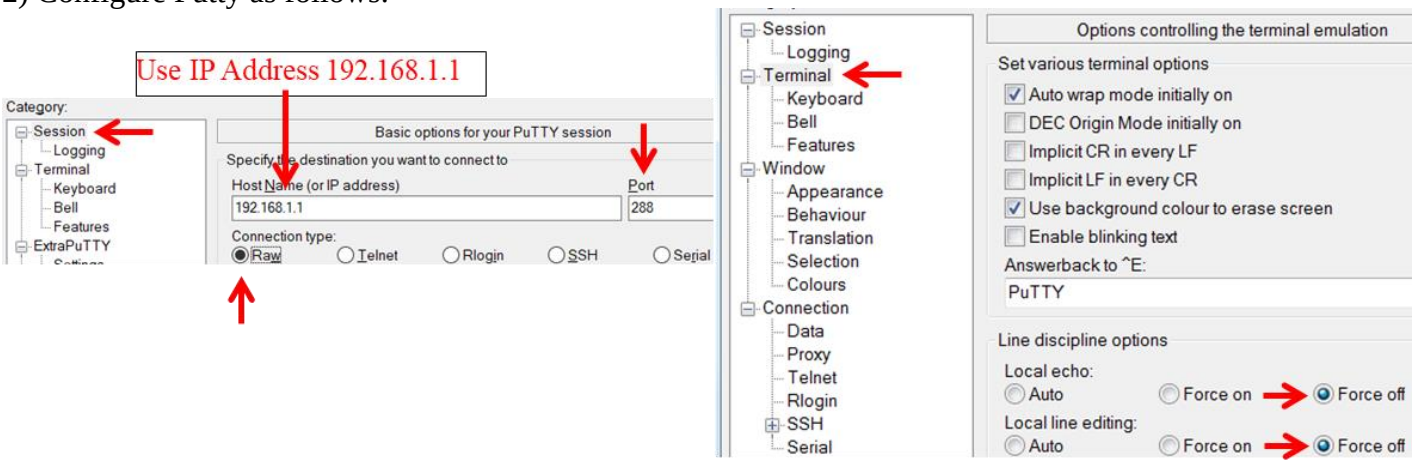
Send a message to PuTTy when you press a button on the LCD module. A message for each button.

Here is a reminder for how to set up PuTTy:

Using Putty to interact with your CyBot over WiFi:

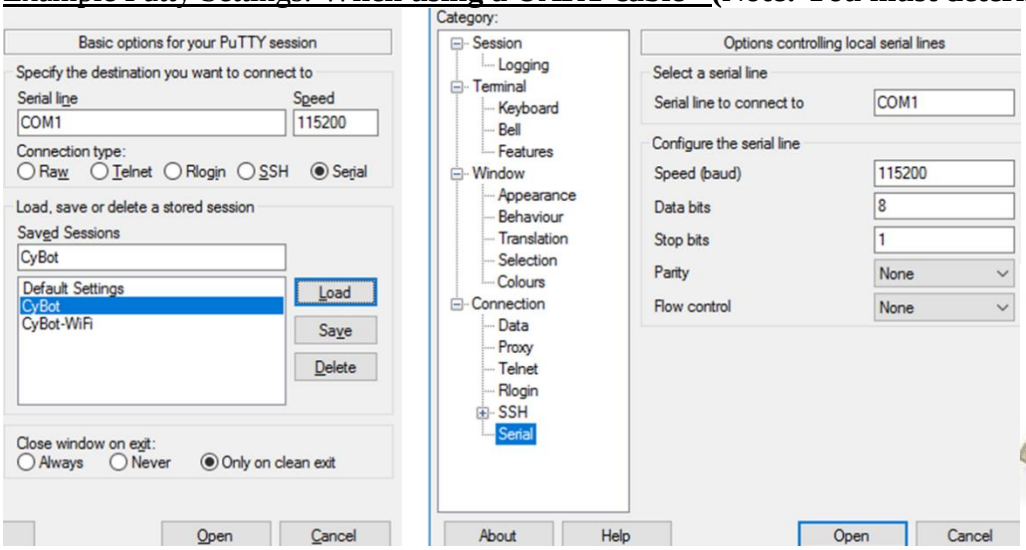
1) Make sure your PC is connected to the WiFi network for your CyBot. The network will be called: **cyBOT #** (where # is your CyBot's number), and the WiFi password is: **cpre288psk**

2) Configure Putty as follows:



****Note:** When using the Virtual CyBot: 1) Skip step 1 of the WiFi setup, and 2) Configure Putty using: a) the IP address given by 288SimGUI, and port 50000

Example Putty Settings: **When using a UART cable - (Note: You must determine which COM port to use)**



UART cable

Checkpoint: Demonstrate communicating between your Desktop and Cybot platform to your mentor when a button on the LCD module is pressed.

Part 5. Putty / UART system

Similar to part 1, make use of the following documents to draw a figure that shows how your Desktop electrically connects to the Microcontroller's UART.

1) Data Sheet
2) Cybot Baseboad
3) Cybot LCD board

A) As part of your figure, specify the **Port** and **wires** being used by the Microcontroller's UART RX (receive) and TX (transmit) signals to electrically connect to the UART cable.

B) Within the Microcontroller, what values do the i) Alternative Function Register (AFSEL), and ii) Port Control Register (PCTL) need to be set to enable the UART device to receive and transmit data to your Desktop computer?