SE423 Laboratory Assignment 2 (One Week Lab)
Introduction to Raspberry PI4 and
Introduction to TMS320F28379D GPIO Programming and Texas Instruments Code Composer Studio

A few questions to get your brain thinking in binary number representation:

1. If there was such thing as a 24 bit signed integer, what would be the largest positive number it could represent and what is the smallest negative number it could represent?
2. Below are three (signed) int16_t integers represented in binary format. What are these numbers in decimal format?
    i. 1101110000011011
    ii. 0001111100110101
    iii. 1000000010110011
3. In question 2i, is bit 10 high 1 or low 0? Remember we start numbering with bit 0 as right most bit.
4. **Explain to your TA** what this if statement is checking for (think in binary) (& is bitwise AND)

```
if ((myregister & 0x4) == 0x4) {
}
```

Goals for this Lab Assignment:

1. Build a simple LabVIEW program to communicate over Ethernet (TCPIP) to the robot's Raspberry Pi4 board.
2. Use CPU Timer to periodically perform desired procedures/code.
3. Work with port inputs and port outputs.
4. What to do with a compiler error.
5. Debugging your source code with Breakpoints and the Watch Window.

LED's Default GPIO Assignments:

LED1	GPIO22, Controlled with Registers GPADAT, GPASET, GPACLEAR and GPATOGGLE
LED2	GPIO94, Controlled with Registers GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE
LED3	GPIO95, Controlled with Registers GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE
LED4	GPIO97, Controlled with Registers GPDDAT, GPDSET, GPDCLEAR and GPDTOGGLE
LED5	GPIO111, Controlled with Registers GPDDAT, GPDSET, GPDCLEAR and GPDTOGGLE

Push Button's Default GPIO Assignments:

PB1	GPIO157, Read bit status with Register GPEDAT
PB2	GPIO158, Read bit status with Register GPEDAT
PB3	GPIO159, Read bit status with Register GPEDAT
PB4	GPIO160, Read bit status with Register GPFDAT

GPIO Register Use when GPIO pin set as Output: The GPIO Registers are 32 bit registers but we use unions and bitfields in the C/C++ programming language to control just one bit of the 32 bit register at a time. The ".all" part of the

C/C++ union is the entire 32bit register. The ".bit.GPIO19" is just one bit in the 32 bit register. So these two lines of C code perform the same operation:

GpioDataRegs.GPASET.all = 0x00000800; //You have to think a little with this code to know that bit 11 is being set.

GpioDataRegs.GPASET.bit.GPIO11 = 1; //This line of code is somewhat easier to understand that we are setting the 11th bit.

Register	Usage	Example
GP?DAT	GP?DAT.bit.GPIO? = 1, Sets that Pin High, 3.3V GP?DAT.bit.GPIO? = 0, Sets that Pin Low, 0V/GND	**GpioDataRegs.GPADAT.bit.GPIO19 = 1; Sets GPIO19 High/3.3V** **GpioDataRegs.GPADAT.bit.GPIO19 = 0; Sets GPIO19 Low/0V**
GP?SET	GP?SET.bit.GPIO? = 1, Sets that Pin High, 3.3V GP?SET.bit.GPIO? = 0, Does Nothing	**GpioDataRegs.GPBSET.bit.GPIO37 = 1; Sets GPIO37 High/3.3V** **GpioDataRegs.GPBSET.bit.GPIO37 = 0; Does Nothing**
GP?CLEAR	GP?CLEAR.bit.GPIO? = 1, Sets that Pin Low, 0V/GND GP?CLEAR.bit.GPIO? = 0, Does Nothing	**GpioDataRegs.GPCCLEAR.bit.GPIO70 = 1; Sets GPIO70 Low/0V** **GpioDataRegs.GPCCLEAR.bit.GPIO70 = 0; Does Nothing**
GP?TOGGLE	GP?TOGGLE.bit.GPIO? = 1, Sets Pin opposite of its current state. GP?TOGGLE.bit.GPIO? = 0, Does Nothing	**GpioDataRegs.GPDTOGGLE.bit.GPIO98 = 1; was 3.3V then 0V or was 0V then 3.3V** **GpioDataRegs.GPDTOGGLE.bit.GPIO98 = 0; Does Nothing**

GPIO Register Use When GPIO Pin Set as Input: Each GPIO pin, when setup as an input, has an internal pull-up resistor that can either enabled/connected or disabled/disconnected to that GPIO pin. With the passive push button on our breakout board, we will need to enable the pull-up resistor.

Register	Usage	Example
GP?DAT	If GP?DAT.bit.GPIO? is equal to 1 then the Pin is High, 3.3V If GP?DAT.bit.GPIO? is equal to 0 then the Pin is Low, 0V/GND	**if (GpioDataRegs.GPADAT.bit.GPIO19 == 1) {** **//code that needs to run when input pin GPIO19 is High/3.3V** **} else {** **// code that needs to run when input ping GPIO19 is Low/0V** **}**

Laboratory Exercises

Exercise 1:

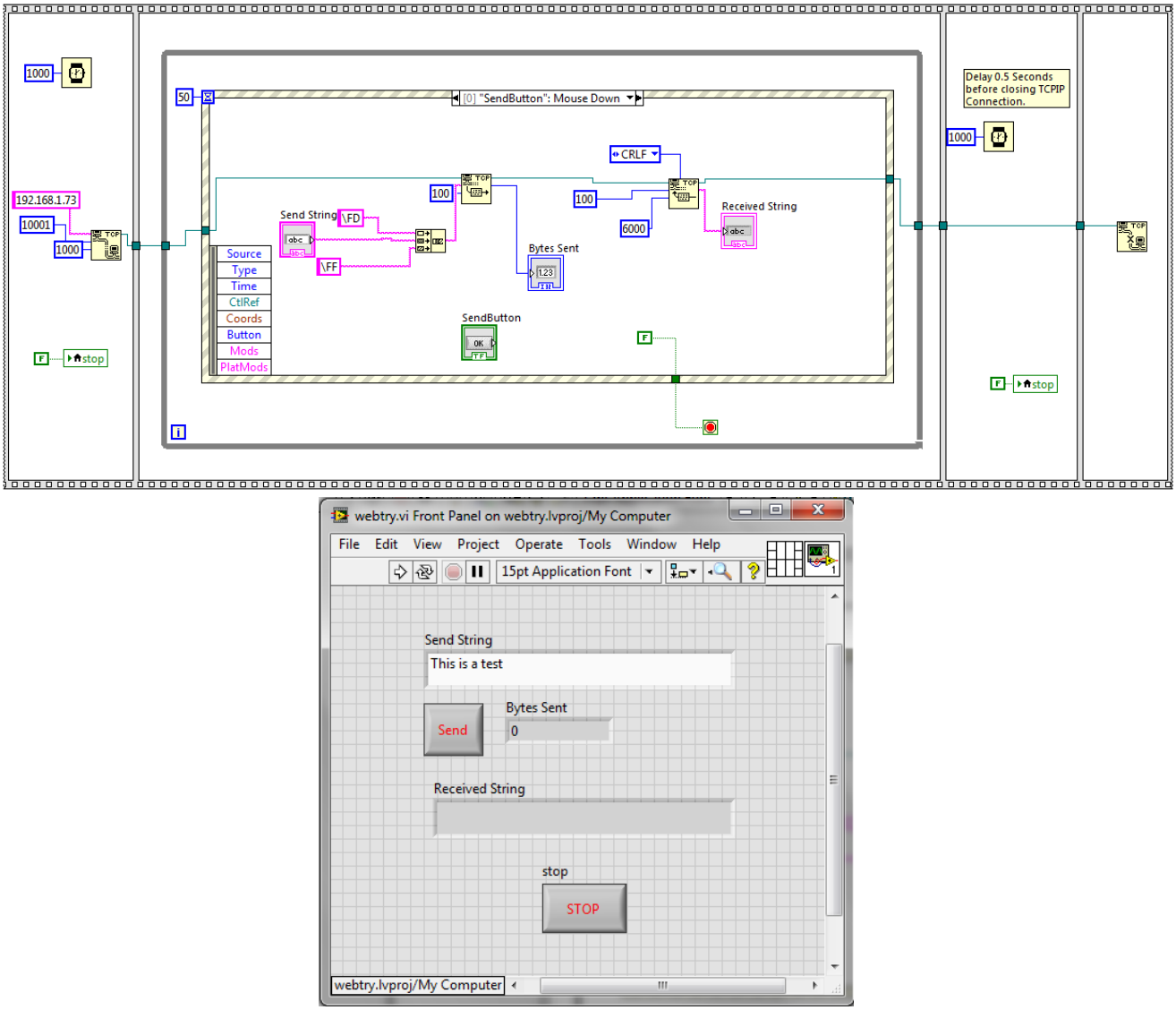
First, make sure your repository is up to date. Under Lab 1, find the Git help file titled "Using the SE423 Repository" and read and perform the steps of the last section of the document titled "Course File Updates." These steps will pull the latest updates from the class repository. Ask your TA for help if needed. **You should perform these steps each time you come to a new lab session** to make sure you have the latest starter code.

In this exercise 1 you will create a simple LABVIEW VI from scratch to communicate with your robot. Use the below steps along with the picture of the block diagram below to create this LabVIEW application.

1. Open a new LABVIEW project and then its VI file. Remember Ctrl-E switches you back and forth between the front panel view and the block diagram view.
2. Use the below block diagram picture and the remaining instructions to build your LABVIEW program.
3. Add a 4 frame flat sequence structure to your block diagram. The flat sequence will allow for initializations, the main body of the program and termination.
4. In the second frame add a While Loop.
5. Add a Stop button in the front panel view but do not wire the Stop button in the block diagram yet. Change the Mechanical action of the Stop button to "Switch When Released." For now in the block diagram view, place the Stop button outside the While Loop.

6. I like using the Mechanical Action "Switch When Released" (or "Switch When Pressed") for the Stop button but it requires initialization and termination code to make sure it is in the state you wish when you get to your main code. To control the state of the Stop button you can use a local variable to change its state. In the first sequence frame add a local variable located under "Structures" items. Click on the "?" of the local variable and select "Stop" (or whatever you named your stop button.) Connect a "False" Boolean constant to the input of the local variable. This will force the state of the Stop button to be false at the beginning of the program. The while loop stops when the Stop button returns true. For completeness add a "stop" local variable in the third sequence frame and also set it to false there.
7. In below steps we are going to setup a TCPIP connection between the LABVIEW program and a Linux program running on the robot. A small delay is necessary at the beginning and end during the TCPIP server connection and termination. So in the first and third sequence frame add a "Wait (ms)" of 1 second (1000 ms) to make sure the first and third frames take at least 1 second.
8. We will finish up the initialization and termination. Add a "TCP Open Connection" found under "Data Communication->Protocols->TCP to the first sequence frame. Right click at TCP Open Connection's "address". "remote port" and "timeout (ms)" ports and select create constant. Set the timeout to 1000 ms, remote port to 10001 and address to "192.168.1.7?" where ? is dependent on the robot you are using.
9. In the fourth sequence frame add a "TCP Close Connection."
10. Now we will add the main code to our VI. The goal of the VI is to open a TCPIP connection to a Linux program running on the robot car. Our front panel will have a string control, a Send button and a string indicator. The user will type a string to send to the robot car in the string control and then press the send button to send the text to the robot car. Then the robot car will receive that data and echo it back to the LABVIEW program and it will be displayed in the string indicator. This will be repeated until the Stop button is pressed.
11. Add an OK button to the front panel and make its Mechanical action "Switch Until Released." Also change the OK text to SEND, and change the button's name to something like "sendbutton." For now place this button outside of the while loop.
12. Inside the While loop add an Event Structure. Add three events to this structure. One (labeled event "0" in the block diagram) being the default Timeout event. Second (labeled "1") a Send button Mouse Down event. Third (labeled "2") a Stop button Value Change event.
13. In the Send button event first place a TCP Write block and a Concatenate String block. Wire the "Connection ID" output of the "TCP Open Connection" block to the "Connection ID" input of the TCP Write block. Wire the TCP Write block timeout to a constant setting it to 100 ms. Wire the output of the Concatenate String block to the "data in" port of the TCP Write block. Finally create an Indicator to display the number of bytes written output port.
14. Drag down on the Concatenate String block so that three items will be concatenated together. Create a constant string for the first element, a control string for the second element (call it "Send String") and a constant for the third element. The Linux program is waiting for the start character 0xFD to begin the string and the stop character 0xFF to end the string. To send these hexadecimal numbers as a character select the constant string and right click on it. Down towards the bottom of the displayed menu you should find and select "'\' Codes Display". Do this for the two constants and type in the first "\FD" and the second "\FF".

15. That completes the sending of a string to the robot, now wire the code to receive the string sent back to LABVIEW. Place a "TCP Read" block after the TCP Write block. First wire the "Connection ID" output of the TCP Write block to the "Connection ID" input of the "TCP Read" block. Also while we are here, wire the "Connection ID" output of the "TCP Read" block to the "Connection ID" input of the "TCP Close Connection" block which is located in the fourth sequence panel.
16. Create constants for the "TCP Read" block setting the Timeout to 3000 ms and the bytes to read to 100. Also create an indicator in the front panel view and connect its input to the "data out" of the "TCP Read" block in the block diagram. Name this indicator "Received String."
17. The final addition for the "TCP Read" block is to create a constant to connect to the "mode" input. Once again, this is easiest to do by right clicking on the "mode" input node and selecting "Create->Constant". The data transfer method between the robot and LABVIEW is slightly different then the LABVIEW to robot transfer. For the LABVIEW to robot transfer we placed a start and stop character at the beginning and end of the string. The "TCP Read" block can be set to implement a different transfer waiting until the carriage return and line feed characters are seen in sequence. This sequence indicates the end of the string. To put the "TCP Read" block in this mode, click the down arrow of the "mode" constant and select "CRLF."
18. So that the event structure works correctly with the Send button that was created in step 12, move the Send button's icon inside the Send Button Mouse Down Event.
19. The Timeout and Stop events are simple. In both events pass the "Connection ID" output of the "TCP Open Connection" block through to the output on its way to the "TCP Close Connection" block. For the Timeout event connect a False Boolean constant to the Stop button port. Also in the top left of the event structure there is a small hour glass icon. Right click on the input of this hour glass icon and create a constant and set the Timeout to 50ms. In the Stop button event connect a True Boolean constant to the Stop button port. Also here in the Stop button event place the Stop button icon.
20. At your robot's Linux console (Tera Term connected to COM1) first login with "pi" as the user name and "f33dback5" for the password. Change directory into the lvecho directory with "cd lvecho". Run the Linux application **LVecho** by typing "./LVecho". The "./" tells Linux to look in the current directory for the application to execute. This application will be used to verify your LABVIEW GUI is working. It simply receives the messages you send from LABVIEW, prints them to the Linux console and echoes them back to your LABVIEW application.
21. Run your LABVIEW application, and type a message into Send String control. Click on the **Send** button and your message should print at the Linux console and echo back to the Received String indicator.
22. When you are done playing with this end, click the STOP button and close the LVecho application at that Linux console by typing Ctrl-C.
23. Before you go on to exercise 2, let's power down the Raspberry Pi. At your Pi terminal type "sudo shutdown now". This will shutdown the Pi and it takes about 5 seconds or so to shutdown.
24. Then have your TA show you how to power off the Pi.



Exercise 2:

In this exercise we are back to working with Code Composer Studio and the F28379D Laundpad board (red board). Import "LABstarter", instructions in Lab 1 or your HowTo document, to create a new project in your workspace and call it lab2<yourinitials>. Once you have your new lab2 project perform the below steps.

1. For this lab, you will only be using CPU Timer 2's interrupt service routine "cpu_timer2_isr(void)". We will leave the timer0 and timer1 functions in our source code but we will not enable timer0 or timer1. So in main() find the two lines of code that set the TIE (Timer Interrupt Enable) bit to enable timer 0 and timer 1. Comment these two lines so they are not included in your program. i.e.

   //CpuTimer0Regs.TCR.all = 0x4000;

   //CpuTimer1Regs.TCR.all = 0x4000;

2. In main() find the "ConfigCpuTimer" function call for CPU Timer 2 and set its period to 0.25 seconds. Also find CPU Timer 2's interrupt function "cpu_timer2_isr." Note that in this function, it is blinking on and off the blue LED on the Launchpad. Build and Debug this code to make sure that the code compiles and runs. You should see

the blue LED blinking on and off every half second. Once that is working, terminate your debug session so you are back in Edit mode. Before going to the next step, let's take a few minutes to think about the period value that you passed to the "ConfgiCpuTimer" function. 0.25 seconds is a large number of microseconds so you may have thought about what the largest acceptable number is to pass to this period parameter. To find this largest period setting we need to look at the TIM (timer register) and PRD (period register) registers of the CPU Timers. Both the PRD and TIM registers are 32 bits long and they each store a 32 bit unsigned integer. The TIM register starts at 0 and counts up by 1 every 1/200000000 seconds (200Mhz). Whenever the TIM register reaches the value stored in the PRD register an interrupt event is issued calling the CpuTimer2 interrupt service routine. At this moment, the TIM register is also set back to 0 to start timing again. Knowing that a 32 bit unsigned integer has a maximum value, what is the largest period in seconds that the CPU Timers can be set to? **Explain your answer to your TA.**

3. Create a global int32_t variable and name it something like "numTimer2calls." Inside the cpu_timer2_isr function increment that variable by one each time that function is entered. In addition, every time the function is entered, set the already defined global variable "UARTPrint" to 1. By doing this you are telling the main() while loop to print text through a UART serial port to your PC. Find this serial_printf() function call in the main() while loop. Does it make sense that when you set "UARTPrint" to 1, the while loop will call the serial_printf function? Why is UARTPrint set to zero inside the "if" after serial_printf is called? **Explain to your TA.** Change the text so that it prints your "numTimer2calls" global variable. Since "numTimer2calls" is a 32 bit integer you will need to use the **%ld** formatter. Also have the serial_printf() function print the value of the numRXA variable just as it does in the default serial_printf() statement.

   To see this printed text you need to install a serial terminal on you PC. TeraTerm is already installed on the Windows machines in lab. We need to figure out what serial port COM number your USB serial port is using. The easiest way to find this is to run "Device Manager" in Windows and find the "Ports" item. Under ports find the COM number for the device titled "USB Serial Port". Run Tera Term and select the "Serial" item and find the USB Serial COM port in the list of COM ports. Final thing to do is change the Baud (or Bit) speed of the COM port. Still in Tera Term select the menu item "Setup" and then "Serial Port…". Change the "Speed" to 115200 if it is not already. Build and Debug your code and check that the LaunchPad's blue LED is still blinking and your text is printing to Tera Term. As in Lab 1, type text in Tera Term to increment the "numRXA" variable that you are printing. We will not use numRXA in this lab, but just wanted to show that the UART is receiving characters along with transmitting characters.

4. Write two worker functions "void SetLEDsOnOff(int16_t LEDvalue)" and "int16_t ReadSwitches(void)".
   - void SetLEDsOnOff(int16_t LEDvalue) takes a 16 bit integer as a parameter. The five least significant bits of this integer determine if the five LEDs are on or off. Bit 0 determines LED1's state. Bit 1 determines LED2's state. Bit 2 determines LED3's state. Bit 3 determines LED4's state. Bit 4 determines LED5's state. So for example if 18 (0x12, which is binary 10010) is passed to your function then LED5 and LED2 should be ON. Use five if statements inside your function to check, using the bitwise AND, &, operator, if the integer passed to your function has the least significant five bits either individually set or cleared. If set, turn ON the corresponding LED. If cleared, turn OFF the corresponding LED. See the above tables for LED GPIO assignments. I want you using the GP?SET and GP?CLEAR registers to turn on or off the LEDs. To test this

function increment a global int16_t variable by 1 in your CPU timer 2 interrupt routine and pass this value to your SetLEDsOnOff function. What happens if the number passed to SetLEDsOnOff() is greater than 31? **Explain to your TA.**

- int16_t ReadSwitches(void) returns a 16 bit integer that the least significant four bits indicates the state of the four push buttons. (Note that when each of the push buttons are not pressed the GPIO pin reads a 1 or high voltage. When pressed the GPIO pin reads a 0 or ground. This is because the IO pin is using an internal pullup resistor.) This function should have four if statements and use the bitwise OR, "|" operator to appropriately set bits of a local variable that will be returned by this function. So start the return variable at zero. Then if switch 1 is pressed OR 0x1 with local variable. If switch 2 is pressed OR 0x2 with variable. If switch 3 is pressed OR ??? with variable. If switch 4 is pressed OR ??? with variable. Finally return the local variable with the "return" instruction. See the above table for the GPIO pins that are connected to the push buttons and that are setup as inputs with pull-up resistor enabled in the default code.

5. Now that you have these worker functions, make your program a bit more interesting. Add code in your CPU timer 2 interrupt function so that you display to the LEDs the value returned from your ReadSwitches() function. Do this by creating a global int16_t variable and assign it the value returned from ReadSwitches(). Pass this global variable to your SetLEDsOnOff(value) function to see its binary value displayed on the LEDs. Also print this global variable by adding it to the serial_printf function in main()'s while loop. Make sure to use the %d formatter because this is an int16_t variable.

   Show this working to your TA.

Exercise 3:

1. To get some more practice with starting a new project, create **another** new project by importing the LABstarter example and renaming it and its main source file. Again, disable CPU timer0 and timer1's interrupt by commenting out:

   ```
   //CpuTimer0Regs.TCR.all = 0x4000;
   //CpuTimer1Regs.TCR.all = 0x4000;
   ```

   Change the period of CPU timer 2 to 0.10 seconds. Also copy from your previous project the two worker functions you created. **Do not modify these worker functions. Instead use them "as is" in the below steps.**

2. Change the code in cpu_timer2_isr to increment a global 32 bit integer (you create) by 1 every time timer 2's interrupt function is called. Pass this count variable to the SetLEDsOnOff() function to display the least significant 5 bits of your count variable to the five LEDs. This is similar to what you coded to test your SetLEDsOnOff() function in exercise 1. Compile, download to the DSP and verify that indeed the LEDs are counting in binary. Add one more item to this code as an exercise to see the use of bitwise operators in C. Calling the ReadSwitches() function, use an "if" statement and the bitwise C operator & to check if push buttons 2 and 3 are pressed. If both of these push buttons are pressed, stop incrementing the global count integer. If one or both are released, continue counting. Again compile and download to the microcontroller. When your code is working, **demonstrate your application to your TA**.

Exercise 4: Breakpoints and Watch Windows

Starting with the code you just finished, we want to experiment with adding breakpoints to your code and using the "Expressions window" to edit the values of your variables.

1. In your previous code (with the DSP halted), put your cursor over the integer variable that you are incrementing. You should see that the value of the variable appears. Run your code, halt it again, and again put your cursor over the variable to confirm that it changes.
2. An easier method than using the cursor repeatedly is to add the variable to the Expressions window. When the DSP is halted, the Expressions window displays the current value of each variable in the Expressions window. To add your counting integer variable to the Expressions window, highlight the variable and then right-click, then select **Add Watch Expression…**. The variable will appear in the Expressions window with the current value of the variable. The Expressions window dialog is also found under the View menu.
3. Next play a bit with adding breakpoints and single stepping through a section of code. The code you have written to this point is very small. Add the following nonsense code to allow for easy use of breakpoints and code stepping. At the top of your C-file, but below the #includes, add the following global variables:

   ```
   float x1= 6.0;
   float x2= 2.3;
   float x3= 7.3;
   float x4= 7.1;
   ```

   Then inside your CPU timer 2 interrupt function add this nonsense code:

   ```
   x4 = x3 + 2.0;
    x3 = x4 + 1.3;
    x1 = 9*x2;
    x2 = 34*x3;
   ```

   Build and load your code. Add a breakpoint to your code by double clicking on the left gray margin of your source file. A breakpoint is a location where the program will literally halt during execution. This allows you to check the values of your variables during operation. After a breakpoint, you can single step through your code (F5) and watch the variables update as different calculations are performed. You remove breakpoints by again clicking in the left gray margin.
4. If you happened not to receive any compiler errors during any of the above exercises, you should intentionally add some errors to your code so that you will see how CCS will alert you during the build process. Try double clicking on the error message. The editor will then take you to the line of code that has the error.

Exercise 5:

Still using the code from Exercise 2 and 3 make a few modifications. For many of our lab assignments we will want to have at least one of our timers running at a fast periodic rate. Most of the time that will be somewhere between a period of 1ms to 5ms. I would not be surprised though, if some of your projects will require you to run code at an even faster rates and the F28379D can definitely handle periodic rate of 0.1ms to 0.02ms. For this exercise, use a period of 1ms. which can also be stated as a sample frequency of 1Khz. Change CPU Timer 2's period to 1ms in order that CPU Timer 2's interrupt function is call once every millisecond.

The F28379D can do a huge amount of instructions every 1ms, but there are some things you do not want the processor performing every 1ms. For example the printing to Tera Term, if we printed every 1ms our eyes would not be able to see all the text spilling to the screen. Also calling the SetLEDsOnOff() function every 1ms would cause a blur if LED changes. So add code to your CPU Timer 2 interrupt function to only print every 100th time the function is called. The % (mod) operator is perfect for this. Mod returns the remainder of an integer divided by another integer. i.e. (56 % 5) = 1. So using the int32_t integer that you are incrementing every time in the timer interrupt, write an if statement with a % (mod) condition that causes the if statement to be true every 100th time in the timer interrupt. Inside this if statement, perform all code that makes sense to run at the slower rate.

Demo this to your TA.

Lab Check Off:

1. Demonstrate your LabVIEW programming talking back and forth to your robot's Raspberry Pi board.
2. Demonstrate your first application that continually checks the status of the four pushbuttons and displays their current state on the five LEDs. One LED should always be off since there are only four push buttons.
3. Demonstrate your second application that updates a counter every quarter second and outputs the least significant 5 bits of the count to the five LEDs. The count should stop if both pushbuttons 2 and 3 are pressed and resume when one or both of them are released.
4. Demonstrate that you know how to use Breakpoints and the Watch Window to debug your source code.
5. Demonstrate your 1ms timer period code working.
6. For your lab submission submit your working **commented** code to your Box folder in a subfolder named "Lab2". Take time to add comments explaining what you understand is happening in the code you wrote and the functions in which your code is running. Please make it obvious in your submission which code is for each exercise. I do not want short, hard to understand, comments. Instead, I would like short paragraphs explaining the code you wrote.
7. HowTo Document items:
    a. Anything you learned and need to remember about writing you own functions and calling your own functions. (Or anything else you would like to is always good.)
    b. Study the code given, and that you wrote, for using GPIO157, 158, 159, & 160 as inputs for sensing if the four pushbuttons are pressed or not pressed. Note the initializations of these four GPIOs in the main() function. For the HowTo Document create a section titled "Setting up and using a GPIO as an input". GPIO52 happens to be a GPIO we currently do not use in the lab assignments. In this section, describe the code that would be needed to setup GPIO52 as an input to read a passive pushbutton type sensor. Then what line of code would you write to read if the pushbutton is pressed or not pressed? Finally if the signal connected to GPIO52 was actively driven high or low by a device, (like an IC, integrated circuit) what would you change in the setup lines of code in main()?
    c. Take some time to understand bitfields in C better. In your own words, explain what bitfields are doing for us in C. For Example here is the bitfield for the GPADAT register:

```
struct GPADAT_BITS {                    // bits description
    Uint16 GPIO0:1;                     // 0 Data Register for this pin
    Uint16 GPIO1:1;                     // 1 Data Register for this pin
    Uint16 GPIO2:1;                     // 2 Data Register for this pin
    Uint16 GPIO3:1;                     // 3 Data Register for this pin
```

```
    Uint16 GPIO4:1;                 // 4 Data Register for this pin
    Uint16 GPIO5:1;                 // 5 Data Register for this pin
    Uint16 GPIO6:1;                 // 6 Data Register for this pin
    Uint16 GPIO7:1;                 // 7 Data Register for this pin
    Uint16 GPIO8:1;                 // 8 Data Register for this pin
    Uint16 GPIO9:1;                 // 9 Data Register for this pin
    Uint16 GPIO10:1;                // 10 Data Register for this pin
    Uint16 GPIO11:1;                // 11 Data Register for this pin
    Uint16 GPIO12:1;                // 12 Data Register for this pin
    Uint16 GPIO13:1;                // 13 Data Register for this pin
    Uint16 GPIO14:1;                // 14 Data Register for this pin
    Uint16 GPIO15:1;                // 15 Data Register for this pin
    Uint16 GPIO16:1;                // 16 Data Register for this pin
    Uint16 GPIO17:1;                // 17 Data Register for this pin
    Uint16 GPIO18:1;                // 18 Data Register for this pin
    Uint16 GPIO19:1;                // 19 Data Register for this pin
    Uint16 GPIO20:1;                // 20 Data Register for this pin
    Uint16 GPIO21:1;                // 21 Data Register for this pin
    Uint16 GPIO22:1;                // 22 Data Register for this pin
    Uint16 GPIO23:1;                // 23 Data Register for this pin
    Uint16 GPIO24:1;                // 24 Data Register for this pin
    Uint16 GPIO25:1;                // 25 Data Register for this pin
    Uint16 GPIO26:1;                // 26 Data Register for this pin
    Uint16 GPIO27:1;                // 27 Data Register for this pin
    Uint16 GPIO28:1;                // 28 Data Register for this pin
    Uint16 GPIO29:1;                // 29 Data Register for this pin
    Uint16 GPIO30:1;                // 30 Data Register for this pin
    Uint16 GPIO31:1;                // 31 Data Register for this pin
            };
```

Each of the values in the Bitfield, GPIO0, … , GPIO24, etc can only be set to 0 or 1 because each GPIO is given only 1 bit (:1). (0 sets the GPIO to low or GND. 1 sets the GPIO to high or 3.3V.) If GPIO2 for example is set to 5, the code would compile, but it would be the same as setting GPIO2 to 1 because only the bottom bit (bit 0) is looked at in the assignment statement. So answer this question where the ".all" statement is the entire 32 bit register and the Bitfield elements only access one bit of the register. After answering this below question, would you say the Bitfield method is a bit easier to understand?

Using bitfields, we could check if two bits where set in the GPADAT register with the following if statement:

```
if ((GpioDataRegs.GPADAT.bit.GPIO12 == 1) && (GpioDataRegs.GPADAT.bit.GPIO13==1)) {
        // do something
}
```

How could you perform this same check using the GPADAT.all, entire 32bit GPADAT register? Note the other bits in the GPADAT register could be 1 so your if condition will have to read GPADAT.all and clear (or mask) all the other bits besides 12 and 13.

d. One more Bitfield question to answer so you can better remember how to use them in coming labs. Here is another Bitfield we may use this semester and it is for the ADCSOC1CTL register. The F28379D processor has some 16 bit registers and some 32 bit registers. Counting the used bits in this Bitfield, is ADCSOC1CTL 16 bits or 32 bits? Notice that all elements are unsigned. What is the largest value that can be assigned to ACQPS, CHSEL and TRIGSEL.

```
struct ADCSOC1CTL_BITS {            // bits description
    Uint16 ACQPS:9;                 // 8:0 SOC1 Acquisition Prescale
    Uint16 rsvd1:6;                 // 14:9 Reserved
    Uint32 CHSEL:4;                 // 18:15 SOC1 Channel Select
    Uint16 rsvd2:1;                 // 19 Reserved
    Uint16 TRIGSEL:5;               // 24:20 SOC1 Trigger Source Select
    Uint16 rsvd3:7;    };             // 31:25 Reserved
```