SE 423 Mechatronics
Laboratory Assignment #2

Introduction to Raspberry PI4 and Introduction to TMS320F28379D GPIO Programming and Texas Instruments Code Composer Studio.

One-Week Lab

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

1. If there were such a 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?
   (a) `1101110000011011`
   (b) `0001111100110101`
   (c) `1000000010110011`
3. In question 2a, is bit `10` high `1` or low `0`? Remember, we start numbering with bit `0` as the 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) {
    // Other code
}
```

Goal for this Lab Assignment

1. Build a simple LabVIEW program to communicate over Ethernet (TCP/IP) to the robot's Raspberry Pi4 board.
2. Use CPU Timer to perform desired procedures/code periodically.
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.

Reference Material

LED's Default GPIO Assignments: (Listed here for reference)

Table 1: LED GPIO Mapping and Control Registers

LED	GPIO	Bank	DAT	SET	CLEAR	TOGGLE
LED1	22	GPA	GPADAT	GPASET	GPACLEAR	GPATOGGLE
LED2	94	GPC	GPCDAT	GPCSET	GPCCLEAR	GPCTOGGLE
LED3	95	GPC	GPCDAT	GPCSET	GPCCLEAR	GPCTOGGLE
LED4	97	GPD	GPDDAT	GPDSET	GPDCLEAR	GPDTOGGLE
LED5	111	GPD	GPDDAT	GPDSET	GPDCLEAR	GPDTOGGLE

Push Button's Default GPIO Assignments:

Table 2: Push Button's GPIO Mapping and Control Registers

LED	GPIO	Bank	DAT
PB1	157	GPE	GPADAT
PB2	158	GPE	GPADAT
PB3	159	GPE	GPADAT
PB4	160	GPF	GPADAT

GPIO Register Use when GPIO pin set as Output

The GPIO Registers are 32-bit registers, but we use unions and bit fields in C/C++ to control a single bit of each register at a time. The "`.all`" part of the C/C++ union is the entire 32-bit register. The "`.bit.GPIO11`" 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,
                                     // as we are setting the 11th bit.
```

Table 3: GPIO Register Functional Definitions and Usage Constraints

Register	Usage	Example

GP?DAT	`GP?DAT.bit.GPIO? = 1;`
	Sets Pin High, 3.3V
	`GP?DAT.bit.GPIO? = 0;`
	Sets Pin Low, 0V/GND
		FOR TECHNICAL REASONS, DO NOT USE DAT FOR TURNING ON AND OFF LEDS. JUST USE IT TO READ THE STATE OF THE PUSHBUTTONS BELOW.

GP?SET	`GP?SET.bit.GPIO? = 1;`
	Sets 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 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;`
		State toggles (3.3V ↔ 0V)
		`GpioDataRegs.GPDTOGGLE.bit.GPIO98 = 0;`
		Does Nothing

GPIO Register Use When GPIO Pin Set as Input:

Each GPIO pin, when configured as an input, has an internal pull-up resistor that can be enabled or disabled. With the passive push button on our breakout board, we will need to enable the pull-up resistor.

Table 4: GPIO Register Functional Definitions and Usage Constraints

Register	Usage	Example

GP?DAT	If `GP?DAT.bit.GPIO? = 1`, the pin is logic high (3.3 V).
	If `GP?DAT.bit.GPIO? = 0`, the pin is logic low (0 V/GND).
		`if (GpioDataRegs.GPADAT.bit.GPIO19 == 1) {`
		`  // Code executed when input pin`
		`  // GPIO19 is High (3.3V)`
		`} else {`
		`  // Code executed when input pin`
		`  // GPIO19 is Low (0V/GND)`
		`}`

Laboratory Exercises

Exercise 1

First, to give you practice performing these steps if needed in the future, check if the class repo has any changes or additional files to merge into your repo already on the lab PC. Under Lab 1, find the Git help file titled "Using the SE423 Repository" and read and perform the steps of the 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. *There probably will not be any updates, but in the future, if your instructor makes some additions, you will need to perform these steps again.*

In Exercise 1, you will create a simple LabVIEW VI from scratch to communicate with your robot. Use the steps below, 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. Then in File Explorer, find your repo's folder and explore into "`workspace\LabVIEW`". Create a folder "Lab2" here. *In "`workspace\LabVIEW`" is where I recommend you save all your LabVIEW files. This way,*

when you add your CCS projects to GitHub by running "`git add workspace`*" you will also be adding your LabVIEW files.*

3. Save this new project and VI file to your "`workspace\LabVIEW\Lab2`" folder.

4. Use the block diagram picture below and the remaining instructions to build your LabVIEW program.

5. 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.

6. In the second frame add a `while` Loop.

7. 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.

8. 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 ensure it is in the state you want when you reach your main code. To control the Stop button's state, use a local variable to toggle it. 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 ensure the Stop button is false at the start 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.

9. In the steps below, we are going to set up a TCPIP connection between the LabVIEW program and a Linux program running on the robot. A slight delay is necessary at the beginning and end during the TCPIP server connection and termination. So, in the first and third sequence frames, add a "Wait (ms)" of 1 second (1000 ms) to make sure the first and third frames take at least 1 second.

10. 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.

11. In the fourth sequence frame, add a "TCP Close Connection."

12. 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, then press the send button to send the text. Then the robot car will receive that data, 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.

13. 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.

14. Inside the `while` loop add an Event Structure. Add three events to this structure. One (labeled event "0" in the block diagram) is the default Timeout event. Second (labeled "1"), a Send button Mouse Down event. Third (labeled "2") is a Stop button Value Change event.

15. 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's timeout to a constant setting of 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 to the output port.

16. 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`".

17. That completes sending 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.

18. 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."

19. The final addition for the "TCP Read" block is to create a constant to connect to the "mode" input. Once again, the easiest way to do this is to right-click the "mode" input node and select "Create→Constant". The data transfer method between the robot and LabVIEW is slightly different from that between LabVIEW and the robot. For the LabVIEW-to-robot transfer, we placed start and stop characters at the beginning and end of the string. The "TCP Read" block can be configured to use a different transfer, waiting until the carriage return and line feed characters are received 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."

20. So that the event structure works correctly with the Send button created in step 12, move the Send button's icon inside the Send Button Mouse Down Event.

21. 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 hourglass icon. Right-click the input of this hourglass icon, 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, place the Stop button icon here in the Stop button event. Double-check the picture below, and your LabVIEW program should be finished now.

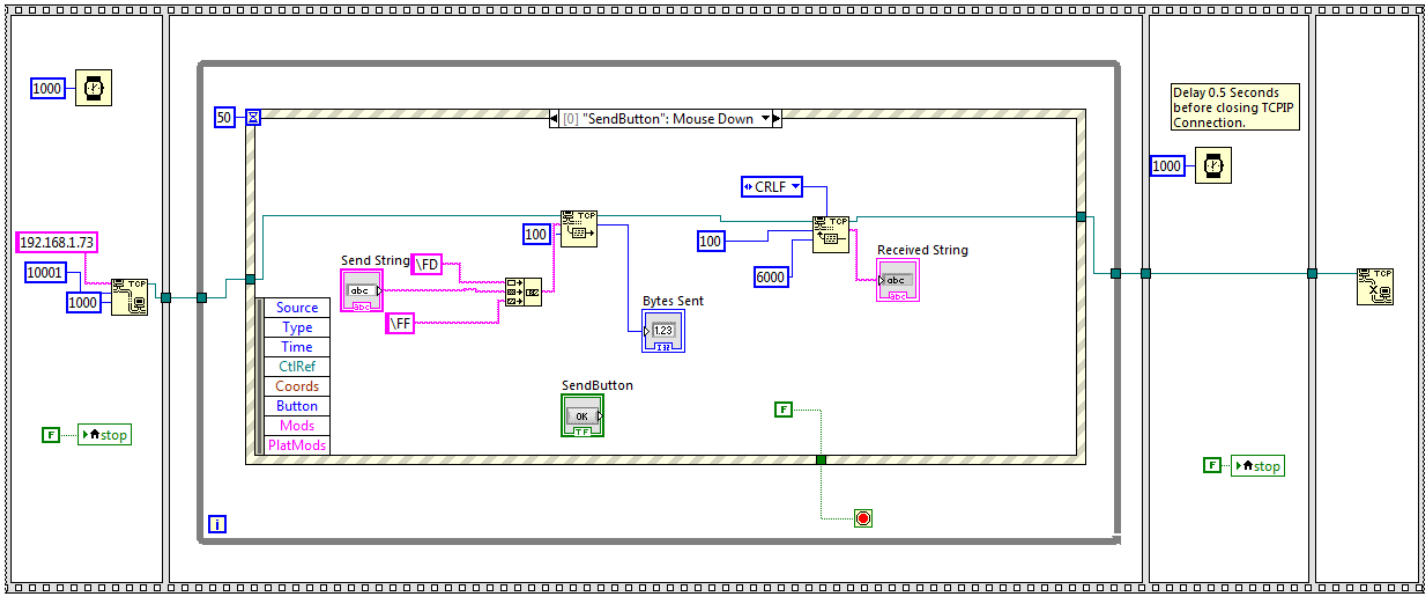
22. Next step is to get the correct program running on your Raspberry PI4 board on your robot. We need a TCP/IP server application running that waits for text from your LabVIEW program, receives it, and sends it back (echoes it). You will use a program on your lab PCs called "PuTTY." To run PuTTY, open a Windows command prompt. At the command prompt, type "`putty pi@192.168.1.7?`" filling in the ? with your robot's IP. (pi is the username @ its ip address) A terminal window should display a prompt for you to enter the password. Type the password, "`f33dback5`", and then you have a wireless terminal of your RPI4's Linux operating system. Type ls (list) and see all the folders of the user pi's home directory. The program we want to run is called LVecho, and it is located in the folder lvecho. Change directory into the `lvecho` folder by typing "`cd lvecho`" (cd is change directory). 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.

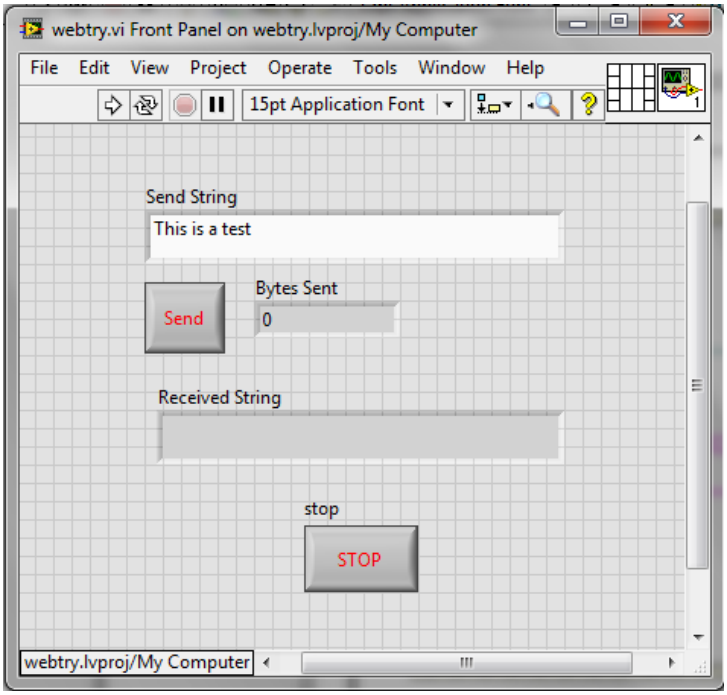
23. Run your LabVIEW application, and type a message into your Send String control. Click on the **Send** button, and your message should print at the Linux console and echo back to the Received String indicator.

24. 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.

25. **Wait here for your TA to tell you to power off your RPI4. We may have other things to show you on the RPI4 before you power off. Before you go on to exercise 2, let us power down the Raspberry Pi.** At your Pi terminal type "`sudo shutdown now`". This will shut down the Pi, and it takes about 5 seconds.

26. Then have your TA show you how to disconnect power from the Pi, so we do not have to worry about shutting it down again for this lab.





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 steps below.

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. How can I comment out 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, the blue LED on the Launchpad is blinking on and off. Build and debug this code to ensure it 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 us 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 the 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 increments by 1 every 1/200000000 seconds (200 MHz). 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 reset to 0 to restart timing. 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. In this exercise, you will gain more experience creating global variables and understanding best practices for printing text to the robot's text LCD screen. Create a global `int32_t` variable and name it something like "`numTimer2calls`." This variable will perform the same job as the variable `CpuTimer2.InterruptCount`, which is already being used in the function. Please create another variable to practice creating a global variable. 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 the robot's text LCD screen. Notice that `UARTPrint` is already set to one inside an if statement in `cpu_timer2_usr`. You can leave it there or comment it out. It sets `UARTPrint` to the function every 10th time, but your code sets it every time. Find the `UART_printfLine` function calls in the `main()` while loop. Does it make sense that when you set "`UARTPrint`" to 1, the while loop will call the two `UART_printfLine` functions? Why is `UARTPrint` set to zero inside the "if" after the `UART_printfLine` calls? **Explain to your TA.** Change the `UART_printfLine` call for line 1 that now prints your "`numTimer2calls`" global variable instead of `CpuTimer2.InterruptCount` variable. Since "`numTimer2calls`" is a 32-bit integer, you will need to use the `%ld` formatter, which is the way it is already set. Debug and Run your code to see that indeed you are still printing the number of times `cpu_timer2_isr` is called. As one more exercise, add to your "line 1" `UART_printfLine` call printing both `numTimer2calls` and `3*numTimer2calls` with a comma between the two numbers. Look at the line 2 call of `UART_printfLine` to see how to use two formatters to print two items. Debug and run your code and check that the LaunchPad's blue LED is still blinking and your text is printing to the LCD. **Show your TA.**

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 to use 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, where the least significant four bits indicate the state of the four push buttons. (Note that when each of the push buttons is 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 pull-up resistor.) This function should have four if statements and use the bitwise OR, "|" operator to appropriately set bits of a local variable that this function will return. So start the return variable at zero. Then, if switch 1 is pressed OR 0x1 is set in a local variable. If switch 2 is pressed OR 0x2 with a 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 connected to the push buttons, which are set up as inputs with pull-up resistors enabled in the default code.

5. Now that you have these worker functions, make your program 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 assigning 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 text (or shortening the text so it fits on the LCD) to one of the `UART_printfLine` function calls 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 the two worker functions you created from your previous project. **Do not modify these worker functions. Instead, use them "as is" in the steps below.**

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 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 the variable's value appear. Run your code, halt it again, and again put your cursor over the variable to confirm that it changes.

2. A more straightforward method than repeatedly using the cursor is to add the variable to the Expressions window. When the DSP is halted, the Expressions window displays the current value of each variable. 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 its current value. 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 enable easy breakpoint use and code stepping. At the top of your C-file, but below the `#include`s, 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 halts 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 Exercises 2 and 3, make a few modifications. For many of our lab assignments, we will want at least one of our timers running at a fast periodic rate—most of the time, that will be somewhere between 1ms and 5ms. I would not be surprised, though, if some of your projects will require you to run code at an even faster rate, and the F28379D can definitely handle a 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 so that CPU Timer 2's interrupt function is called once every millisecond.

The F28379D can execute a large number of instructions every 1ms, but there are some things you do not want the processor to perform every 1ms. For example, if we printed to the robot's text LCD every 1ms, our eyes would not be able to see all the text spilling onto 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 print only every

100th time it 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 when both pushbuttons 2 and 3 are pressed and resume when either or both 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 Gradescope. 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 clear in your submission which code corresponds to each exercise. I do not want short, hard-to-understand comments. Instead, I would like short paragraphs explaining the code you wrote.