SE420 Laboratory Assignment 2 (One Week Lab)
Introduction to TMS320F28377S GPIO Programming and Texas Instruments Code Composer Studio

Goals for this Lab Assignment:

1. Use CPU Timer to periodically perform desired procedures/code.
2. Work with port inputs and port outputs.
3. What to do with a compiler error.
4. Debugging your source code with Breakpoints and the Watch Window.

Prelab

Complete: http://coecsl.ece.illinois.edu/se420/SE420Lab2ExtraPrelab.pdf

LED's Default GPIO Assignments:

LED1	GPIO64, Controlled with Registers GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE
LED2	GPIO91, Controlled with Registers GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE
LED3	GPIO92, Controlled with Registers GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE
LED4	GPIO99, Controlled with Registers GPDDAT, GPDSET, GPDCLEAR and GPDTOGGLE

Push Button's Default GPIO Assignments:

PB1	GPIO41, Read bit status with Register GPBDAT
PB2	GPIO66, Read bit status with Register GPCDAT
PB3	GPIO73, Read bit status with Register GPCDAT
PB4	GPIO78, Read bit status with Register GPCDAT

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 by referencing the "Using the SE420 Repository" document and performing the steps in the section "Course File Updates". These steps will pull the latest updates from the class repository you merged in Lab 1. This procedure can be a bit confusing so 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.

Now that you have the updates, import "LABstarter" and call it lab2. Follow the instructions in Lab 1 on how to create a new project in your workspace. 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 to the text LCD. Find this UART_printfLine() function call in the main() while loop. Does it make sense that when you set "UARTPrint" to 1, the while loop will call the UART_printfLine function? Why is UARTPrint set to zero inside the "if" after UART_printfLine 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. Build and Debug your code and check that the LaunchPad's blue LED is still blinking and your text is printing to the text LCD.

4. Write two worker functions "void SetLEDsOnOff(int16_t leds)" and "int16_t ReadSwitches(void)".
   - void SetLEDsOnOff(int16_t leds) takes a 16 bit integer as a parameter. The four least significant bits of this integer determine if the four 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. So for example if 11 (0xB, which is binary 1011) is passed to your function then LED1, LED2 and LED4 should be turned on. Use four if statements inside your function to check, using the bitwise AND, &, operator, if the integer passed to your function has the least significant four 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 definitions and example code on writing to the registers that control the LEDs. I want you using the GP?SET and GP?CLEAR registers to turn on or off the LEDs. To test this function you could 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 15? **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 UART_printfLine 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 2:

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 0.25 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 4 bits of your count variable to the four LEDs. 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 another use of bitwise operators in C. Create a global 16 bit integer and inside your cpu_timer2_isr assign it the value returned from ReadSwitches(). Then use an "if" statement and the bitwise C operator & to check the global 16 integer 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 DSP. When your code is working, **demonstrate your application to your TA**.

Exercise 3: 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 4:

Write a program that spells out a 3 or 4 letter word in Morse code using all 4 LEDS as one light source. See http://en.wikipedia.org/wiki/Morse_code for specifics on Morse code. Here I want you using an int16_t array to indicate to your CPU timer 2 interrupt function whether it should turn all 4 LEDs or turn off all 4 LEDs. Each time into the CPU timer 2 interrupt function, check an index into the Morse code array. If the value at that index is set to 1, then turn on all 4 LEDs. If the value at that index is 0, turn off all LEDs. Then before exiting from CPU timer 2's interrupt function, increment the index by one so that the next time into the interrupt you will be looking at the next index in your array. Your array will have a certain length. Make sure not to increment past the length of your array by setting the index back to zero when index has reached the size of your array. This will cause your Morse code communication to start from the beginning. Below is an example array for "S O S".

```
int16_t morse[34]={1,0,1,0,1,0,0,0,1,1,1,0,1,1,1,0,1,1,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0};
```

Demo this to your TA.

Lab Check Off:

1. Demonstrate your first application that continually checks the status of the four pushbuttons and displays their current state on the four LEDs.
2. Demonstrate your second application that updates a counter every quarter second and outputs the least significant 4 bits of the count to the four LEDs. The count should stop if both pushbuttons 2 and 3 are pressed and resume when one or both of them are released.
3. Demonstrate that you know how to use Breakpoints and the Watch Window to debug your source code.
4. Demonstrate your Morse code program working.
5. Make sure to commit your Lab 2 to your repository so you have this code for future labs.