SE423 Laboratory Assignment 2 (One Week Lab)
Introduction to TMS320F28379D 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.

LED’s Default GPIO Assignments:

<table>
<thead>
<tr>
<th>LED</th>
<th>GPIO, Controlled with Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED1</td>
<td>GPIO22, GPADAT, GPASET, GPACLEAR and GPATOGGLE</td>
</tr>
<tr>
<td>LED2</td>
<td>GPIO94, GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE</td>
</tr>
<tr>
<td>LED3</td>
<td>GPIO95, GPCDAT, GPCSET, GPCCLEAR and GPCTOGGLE</td>
</tr>
<tr>
<td>LED4</td>
<td>GPIO97, GPDDAT, GPDSET, GPDCLEAR and GPDTOGGLE</td>
</tr>
<tr>
<td>LED5</td>
<td>GPIO111, GPDDAT, GPDSET, GPDCLEAR and GPDTOGGLE</td>
</tr>
<tr>
<td>LED6</td>
<td>GPIO130, GPDEAT, GPESET, GPECLEAR and GPETOGGLE</td>
</tr>
<tr>
<td>LED7</td>
<td>GPIO131, GPDEAT, GPESET, GPECLEAR and GPETOGGLE</td>
</tr>
<tr>
<td>LED8</td>
<td>GPIO25, GPADAT, GPASET, GPACLEAR and GPATOGGLE</td>
</tr>
<tr>
<td>LED9</td>
<td>GPIO26, GPADAT, GPASET, GPACLEAR and GPATOGGLE</td>
</tr>
<tr>
<td>LED10</td>
<td>GPIO27, GPADAT, GPASET, GPACLEAR and GPATOGGLE</td>
</tr>
<tr>
<td>LED11</td>
<td>GPIO60, GPBDAT, GPBSET, GPBCLEAR and GPBTOGGLE</td>
</tr>
<tr>
<td>LED12</td>
<td>GPIO61, GPBDAT, GPBSET, GPBCLEAR and GPBTOGGLE</td>
</tr>
<tr>
<td>LED13</td>
<td>GPIO157, GPDEAT, GPESET, GPECLEAR and GPETOGGLE</td>
</tr>
<tr>
<td>LED14</td>
<td>GPIO158, GPDEAT, GPESET, GPECLEAR and GPETOGGLE</td>
</tr>
<tr>
<td>LED15</td>
<td>GPIO159, GPDEAT, GPESET, GPECLEAR and GPETOGGLE</td>
</tr>
<tr>
<td>LED16</td>
<td>GPIO160, GPFDAT, GPFSET, GPFCLEAR and GPFTOGGLE</td>
</tr>
</tbody>
</table>

Push Button’s Default GPIO Assignments:

<table>
<thead>
<tr>
<th>PB</th>
<th>GPIO, Read bit status with Register GPADAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB1</td>
<td>GPIO4, GPADAT</td>
</tr>
<tr>
<td>PB2</td>
<td>GPIO5, GPADAT</td>
</tr>
<tr>
<td>PB3</td>
<td>GPIO6, GPADAT</td>
</tr>
<tr>
<td>PB4</td>
<td>GPIO7, GPADAT</td>
</tr>
<tr>
<td>JoyStick PB</td>
<td>GPIO8, GPADAT</td>
</tr>
</tbody>
</table>

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 bit with this code to know that bit 11 is being set.
GpioDataRegs.GPASET.bit.GPIO11 = 1; //This line of code is easier to understand that we are setting the 11th bit.
### 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.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP?DAT</td>
<td>if GP?DAT.bit.GPIO? is equal to 1 then the Pin is High, 3.3V</td>
<td>GpioDataRegs.GPADAT.bit.GPIO19 = 1; Sets GPIO19 High/3.3V</td>
</tr>
<tr>
<td></td>
<td>If GP?DAT.bit.GPIO? is equal to 0 then the Pin is Low, 0V/GND</td>
<td>GpioDataRegs.GPADAT.bit.GPIO19 = 0; Sets GPIO19 Low/0V</td>
</tr>
<tr>
<td>GP?SET</td>
<td>if (GpioDataRegs.GPBSSET.bit.GPIO37 == 1) {</td>
<td>GpioDataRegs.GPBSET.bit.GPIO37 = 1; Sets GPIO37 High/3.3V</td>
</tr>
<tr>
<td></td>
<td>// code that needs to run when input pin GPIO37 is High/3.3V</td>
<td></td>
</tr>
<tr>
<td>GP?CLEAR</td>
<td>if (GpioDataRegs.GPCCLEAR.bit.GPIO70 == 1) {</td>
<td>GpioDataRegs.GPCCLEAR.bit.GPIO70 = 1; Sets GPIO70 High/3.3V</td>
</tr>
<tr>
<td></td>
<td>// code that needs to run when input pin GPIO70 is High/3.3V</td>
<td></td>
</tr>
<tr>
<td>GP?TOGGLE</td>
<td>if (GpioDataRegs.GPDTOGGLE.bit.GPIO98 == 1) {</td>
<td>GpioDataRegs.GPDTOGGLE.bit.GPIO98 = 1; was 3.3V then 0V or was 0V then 3.3V</td>
</tr>
<tr>
<td></td>
<td>// code that needs to run when input pin GPIO98 is High/3.3V</td>
<td></td>
</tr>
</tbody>
</table>

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

### 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 you forked 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” again to create a new project in your workspace and call it lab2. If you forget the steps of importing the labstarter project and renaming the project and files with lab2 names, see the lab 1 document. Once you have your new lab 2 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.
   ```c
   //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 and go to the next step.
3. In the cpu_timer2_isr function 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, then the while loop calls the serial_printf function? Why is UARTPrint set to zero inside the if after serial_printf is called? 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. Tera Term is installed on the Windows machines in lab. On Mac do a web search for the “screen” application. 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 “XDS100 Class USB Serial Port”. Run Tera Term and select the “Serial” item and find the XDS100 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. Show your TA this working. 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 SetLEDRowsOnOff(int16_t columns)” and “int16_t ReadSwitches(void)
• void SetLEDRowsOnOff(int16_t columns) takes a 16 bit integer as a parameter. The five least significant bits of this integer determine if the five LED rows are on or off. Bit 0 determines the bottom most row’s state. Bit 1 determines the next up row’s state. Bit 2 determines the middle row’s state. Bit 3 determines the second from the top row’s state. Bit 4 is the top row’s state. So for example if 18 (0x12, which is binary 10010) is passed to your function then the top row of LEDs should be ON and the second to the bottom row of LEDs 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 column. If cleared, turn OFF the corresponding row. See the above tables for definitions and example code in the comments of the LEDPatterns.c file 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 SetLEDRowsOnOff function. What happens if the number increment passed the value of 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() function call. 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 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 LED rows 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 SetLEDRowsOnOff(value) function to see its binary value displayed on the LED rows. Also print this global variable by adding 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 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 creat) by 1 every time timer 2’s interrupt function is called. Pass this count variable to the SetLEDRowsOnOff() function to display the least significant 5 bits of your count variable to the five LED rows. This is similar to what you coded to test your SetLEDRowsOnOff() function in exercise 1. Compile, download to the DSP and verify that indeed the LED rows 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 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:

```c
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:

```c
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:**

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 SetLEDRowsOnOff() 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 first application that continually checks the status of the four pushbuttons and displays their current state on the five LED rows. One row should always be off since there are only four push buttons.
2. Demonstrate your second application that updates a counter every quarter second and outputs the least significant 5 bits of the count to the five LED rows. 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 1ms timer period code working.
5. For your lab submission submit your working commented code to your Box folder in a subfolder named “Lab2”. Submit in that same subfolder the answers to the below questions. Also submit your “HowTo” document.

Additional Questions to answer in your report:

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 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?
4. In the lab starter code, you have modified the period value passed to the function “ConfigCpuTimer”. For example the default code sets CPU Timer 0’s period to 10000 microseconds with the following line of code “ConfigCpuTimer(&CpuTimer0, 200, 10000);” The main functionality of this function is to set the PRD (Period) register to the correct value such that the timer times out every 10000 microseconds. The PRD stores a 32 bit unsigned integer. The TIM (timer register) is also a 32 bit unsigned number. 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 CpuTimer0 interrupt service routine. At this moment the TIM register is also set back to 0 to start timing again. Knowing this, what is the largest period in seconds that the CPU Timers can be set to?
5. Using bitfields, we could check if two bits where set in the GPADAT register with the following if statement:

```c
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.
6. Pushbuttons 1, 2, 3, and 4 are connected to GPIO 4, 5, 6, 7. GPADAT is the register that can be read to see the state (high or low) of these pins. Your ReadSwitches() function probably used the GpioDataRegs bitfield to look at 4, 5, 6, 7 individually and that was correct. GPADAT is for GPIO pins 0 through 31. So 4, 5, 6, 7 are bits 4, 5, 6, 7 in the 32 bit GPADAT register. Using a right shift (>>) operation and then a bitwise and (&) operation (because bits 8 through 31 could be set) and finally a not (~) operation (because when the push button is not pressed the corresponding bit is 1), write the ReadSwitches() function in one line of code that operates on the GpioDataRegs.GPADAT.all register. Remember ReadSwitches returns a number between 0 and 15. i.e.

```c
int16_t ReadSwitches(void) {
    return( Your one line of code here );
}
```

Items you need to perform and submit for this lab, due before your next lab session.

1. Your commented code is your report that will be graded. 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. Submit your report items in your class Box folder making sure to create a subfolder named “Lab2”. Finally submit the current state of your HowTo.docx (or whatever type of file you want to use). This HowTo.docx file should include items that will help you remember how to perform tasks with your F28379D Launchpad board during this semester and after you are done with this course. You will submit this HowTo.docx file at each lab submission.