UNIVERSITY OF IDAHO

Lab 1-Digital I/O Pins

Report

Joe Vandal

1/31/2015

Introduction:

The goal of lab one was to become familiar with the general purpose input and output of the PIC32 microcontroller. To facilitate learning the methods of reading inputs and writing outputs, the lab required the implementation of software to read buttons and control LEDs.

One major subsystems of the goal is reading inputs. For this lab reading inputs refers to determining whether or not button events have occurred. In order to determine this two things need to be accomplished: the buttons need to be initialized, and the buttons need to be read. More specifically the pins that the buttons are attached to needs to be set as digital inputs utilizing the TRIS register. Once the pins are initialized the PORT register needs to be read to get the values of the buttons. The initialization of the buttons can be done many ways, but according to the lab handout the best way to initialize the buttons is to utilize the function "PORTSetPinsDigitalIn()" in the peripheral library. The CerebotMX7ck.h file defines the buttons according to the pins they are attached to.

The other major subsystem of the goal is to write outputs, in the form of toggling LEDs. The setup and usage of the LEDs is very similar to that of the inputs. The LEDs are attached to pins that need to be initialized, but instead of setting the pins as digital inputs they need to be set to digital outputs. Once set as digital outputs a one can be written to the LAT register to turn on the LEDs. Again the lab handout recommends using the functions in the plib.h file to initialize the ports. The CErebotMX7ck.h file defines the LEDs according to the pins they are attached to.

Implementation:

The software implementation went as planned. There are five new sections added to the code from the previous lab. The first of these sections is the initialization. This addition is shown in listing 1 below:

Listing 1.

```
PORTSetPinsDigitalIn(IOPORT_G, BTN1 | BTN2);

PORTSetPinsDigitalOut(IOPORT_G, LED1 | LED2 | LED3 | LED4);
PORTClearBits(IOPORT_G, LED1 | LED2 | LED3 | LED4);

unsigned int button_status = 0;
unsigned int led_ctrl = 0;
```

This section is between the system initialization from lab 0 and the while loop. The purpose of this section is to initialize the hardware for the buttons and LEDs, and to declare and initialize the variables used to transfer data between the functions.

To initialize the buttons the peripheral library function PORTSetPinsDigitalIn() is used. In listing 1 the function is used to set the sixth and seventh bits, defined in CerebotMX7cK.h as BTN1 and BTN2 respectively, of port G as inputs. This has the same effect as writing a one to the TRIS register of port G.

The initialization of the LEDs is similar to the buttons. The PORTSetPinsDigitalOut() function of the peripheral library sets the last four bits of port G as outputs. These bits are defined as LED1 through LED4 in CerebotMX7cK.h. The function used to set the pins as outputs has the same effect as writing a zero to the TRIS register of port G. The line of code following the initialization of the LEDs clears the indicated ports. This is like using the CLR register associated with LATG.

The last portion of listing 1 is the declaration and initialization of the variables used to handle the data. The variables are declared as unsigned integers because that is the type of variable expected by the functions used later in the lab.

The second section of the lab is the while loop. In the infinite loop the functions are called repeatedly. This allows the program to run continuously. This section is shown in listing 2.

Listing 2.

```
while(1)
{
	/* Student supplied code inserted here */

	button_status = read_buttons();
	led_ctrl = decode_buttons(button_status);
	control_leds(led_ctrl);
}
```

The next portion of code is the functions. The lab called for three function; the first of which is shown in listing 3. This function is used to read the value of the buttons and return the raw value to the main loop. The value of the buttons is read by the PORTReadBits() function provided in the peripheral library. The function only reads the indicated bits of the specified port. A temporary variable is used to hold the value within the function.

Listing 3.

```
int read_buttons(void)
{
	/* Student supplied code inserted here */

	int btn_val = 0;   // temporary variable to hold the value of BTN1 and BTN2
```

```
        btn_val = PORTReadBits(IOPORT_G, BTN1 | BTN2); // read the value of the BTNs

        return btn_val;   // return the value of the BTNs

}
```

The second function takes the value of the buttons and selects the appropriate mask to return to the main loop. The function utilizes a switch statement to select the mask. This function is shown below in listing 4.

Listing 4.

```
int decode_buttons(int buttons)
{
   /* Student supplied code inserted here */

   /* Only use the values of BTN1 and BTN2 on Port G for the switch statement.
    * This is only needed if the software is reading all of Port G.
    * buttons = buttons & (BTN1|BTN2);
    */

   switch(buttons)
   {
      case 192:            // if buttons indicates both buttons are pressed
         return 0x8000;    // generate code to turn on LED4
         break;
      case 128:            // if buttons indicates BTN2 is pressed
         return 0x4000;    // generate code to turn on LED3
         break;
      case 64:             // if buttons indicates BTN1 is pressed
         return 0x2000;    // generate code to turn on LED2
         break;
      case 0:              // if buttons indicates no buttons are pressed
         return 0x1000;    // generate code to turn on LED1
         break;
      default:             // if buttons indicates a value not specified
         return 0xF000;    // generate code to turn on all LEDs
         break;
   }
}
```

Originally the commented out line was assumed to be necessary, because when a PORT register is read all of the bits are read. Considering that the PORTReadBits() function only reads the specified bits, this line was considered unnecessary.

The final function of the software is the LED controller. This function makes use of the PORTWriteBits() function in the peripheral library to turn on the led indicated by the mask generated by the function in listing 4. Using the PORTClearBits() function, same as in the initialization section, and the PORTWriteBits() function, eliminates the need for a read-modify-write sequence. This is possible because the two functions only modify the indicated bits. Listing 5 shows the LED control function.

Listing 5.

```
void control_leds(int leds)
{
  /* Student supplied code inserted here */

  PORTClearBits(IOPORT_G, LED1 | LED2 | LED3 | LED4); // turn off all LEDs
  PORTSetBits(IOPORT_G, leds);  // turn on the LEDs indicated by leds

  //LATG = leds;              // Write to port G to set LEDs

}
```

Testing and Verification:

Testing the code was simple. Reading the buttons was tested by setting a breakpoint in the while loop at the second function. Once the code stopped at the breakpoint the button_status variable could be looked at to determine the value of the buttons. The second function was tested by setting a breakpoint at the beginning of the last function in the while loop. The value of the led_cntl could be compared to the expected value to determine whether or not the switch statement was setup correctly. Finally, testing the led control function was just letting the function run and visually inspect the LEDs to determine if the appropriate LED was turned on. Each of the tests indicated that the program was working properly.

Questions:

One of the potentially difficult tasks in this lab was reading from and writing to specific IO pins without affecting other pins. This can be accomplished a couple of ways. By using the functions provided by the peripheral library, specific bits can be read or modified without affecting other pins on the port. Another method of reading and toggling specific bits is to use a read-modify-write sequence. To implement a read-modify-write sequence there are several steps. The LAT register needs to be read and stored. Then a mask is applied to clear the bits, in the value of the LAT register, that need modified. Then the modified value can be combined with the new state of the bits. Finally, the value can be written back to LAT register.

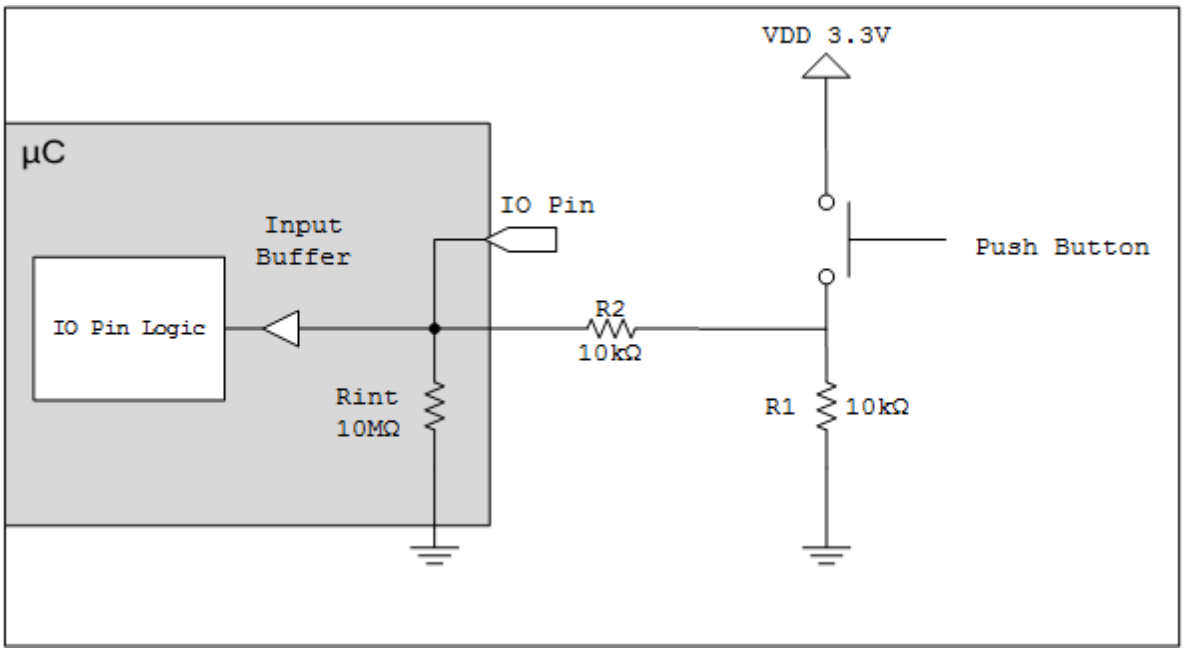

Figure 1: Cerebot push-button schematic

Figure 1 shows the push button schematic for the Cerebot board. The current on the IO Pin can be described as i=VDD/(R1+R2), and the voltage on the IO Pin can be described as v=3.3-iR2. Using the values presented in the schematic the current and voltage on the IO Pin, when the button is pressed, is 0.165mA and 1.65V.

Pull-up and pull-down resistors are used to regulate the state of a pin. A pull-down resistor, like R1 in figure 1, is used to keep the pin from floating by pulling the voltage to a low logic level. A pull-up resistor also keeps a pin from floating, but it pulls the voltage to a high logic level. If R1 were removed then the circuit would have high impedance when the button was not pressed. This could cause the software to determine that the pin is high or low and act in an unintended manner.

The I/O pin sources 5.53mA to the LEDs. The PIC32 is capable of sourcing or sinking 18mA to any I/O pin. The maximum current that the microcontroller can source or sink on all I/O pins simultaneously is 200mA.

Conclusion:

The lab was a success but there are some limitations to this program. One limitation is the lack of control over the timing of the software. This limitation is not very significant because the program executes so rapidly that the program will not miss any button push events. Another limitation is that the software cannot detect issues with the hardware. During demonstration the buttons on a board were not making adequate connection, which lead to intermittent issues. Once the code was programed on another board everything worked as intended.