Lab 1:
Introduction to Working with Microcontrollers

Overview:
In this lab, you will be programming a microcontroller to flash LEDs in response to user input. You will also see the effects of utilizing polling-based approaches to microcontroller development.

Before:
Before this lab, you should be able to do the following:
- Be comfortable with programming in C
  - Read Programming Fundamentals for help
- Understand the difference between microcontroller inputs and outputs

After:
After this lab, you should understand:
- the basic purpose of a microcontroller
- the purpose of pullup and pulldown resistors
- the purpose of data-direction registers and memory registers
- the purpose of the embedded system loop

Required Materials:
- PIC32MX250 Development Board
- 2x LED
- 2x 100 Ohm Resistor
- 2x Push Button
- 2x 10k Resistor (or higher)

Pre-Lab:
- Install SUBLIBinal and the XC32 Compiler
- Install SInterface
- Configure SInterface

Questions:
- What are pullup and pulldown resistors?
- What is a register?
- List five peripherals available on the PIC32MX250F128B and what they can do.
  - Refer to the PIC32MX250 family user guide page 1 and Section 1.0 Device Overview on page 19.
- How can a pin be configured as an input or an output? Can it be configured for both?
- What registers must be used for configuring a pin to be a digital input? What about a digital output?

General Information:

Binary Number Systems

Binary is a way of representing numbers using only ones and zeros. An example of a binary number would be 010. Typically, binary numbers are preceded by a qualifier that denotes them as binary. For example, it is common to see binary numbers displayed as 0b0110, where the 0b designating that the number is binary. An important thing to remember is that all forms of data stored in a computer can ultimately be converted down to a binary representation. Binary numbers are stored in computers in little blocks of memory called bytes. For example, a hard drive may contain 500 gigabytes of memory. This means that it can store up to 500,000,000,000 bytes! Each byte is composed of 8 bits, each of which is capable of storing either a one or a zero. Computers store memory in long strings of organized ones and zeros that are reinterpreted into meaningful information.

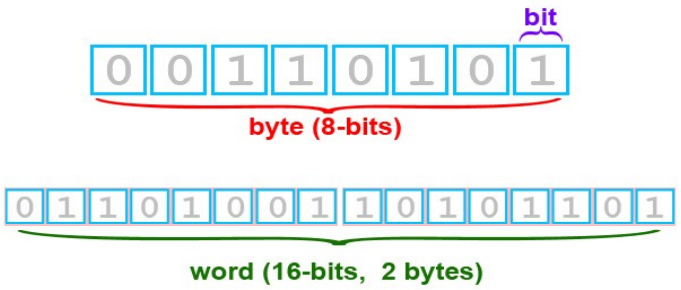

Illustration 1: Representation of a byte in binary form.

Microcontrollers

A microcontroller is a small black chip with a number of metal tabs coming out of the sides called pins. The pins are used for getting output and providing input to the microcontroller. Microcontrollers are also small computers. Just think that instead of a keyboard and mouse for inputs, they use small metal pins.

Programming:

Working with a microcontroller is a bit different than working with your laptop because you don't have a keyboard and mouse to interact with it. Microcontrollers instead work with inputs in the form of voltage. To tell the microcontroller how to react to different inputs, it will need to be programmed. Microcontroller programming can be done in C, C++, or assembly programming languages. Ultimately, all programs are converted down into machine code (binary information) that is placed into the microcontroller's memory.

Pins:

Pins on the microcontroller act as the only way for it to communicate with the outside world. Pins on the PIC32 are divided into a port and a number. The PIC32MX250F128B has port A and port B available, each of which contain up to 32 pins. That means that a pin referenced as A0 belongs to port A and is pin number 0.

A microcontroller pin can either be input or output, but it can't be both. Whenever a pin is used, it will need to be configured to the desired functionality. Configuration is done by setting certain bits in specific registers.

Registers:

In microcontrollers, much of the important memory that you will need to work with is stored in registers. Registers control the entire operation and configuration of the microcontroller. Registers are a collection of a few bytes located at a specific location in memory. Each bit of a register has a specific meaning that is documented within the device datasheet. Typically, there are already useful variables defined by the toolchain (tools used to program the microcontroller) that can be modified to work with a register. For example, modifying the TRISA variable will modify the register associated with the tristate for port A. This means that you can configure inputs and outputs on port A by writing directly to the TRISA variable. The names and functions of registers can be read about in the datasheet for the device.

Peripherals:

Microcontrollers contain a number of peripherals that help them accomplish a wider variety of tasks beyond simple math. An example of a peripheral is a timer on the microcontroller. It helps to track precise time measurements for scheduling tasks. It is important to know what peripherals are available when working with a microcontroller and to know the limitations of the peripherals. Read the device datasheet to see what peripherals are available and what they can do.

Microcontroller Programs

Microcontroller programs usually follow a standard flow. Interrupts are first disabled and then all of the peripherals (such as timers, UARTs, and clocks) are configured. Once the microcontroller is set up, interrupts are re-enabled and the microcontroller enters the embedded loop. This loop should never terminate and is typically done through a while(1) loop. The embedded system loop is the main bulk of the program. It is where any polling and data processing should take place. All time-intensive tasks should be done within the embedded loop.

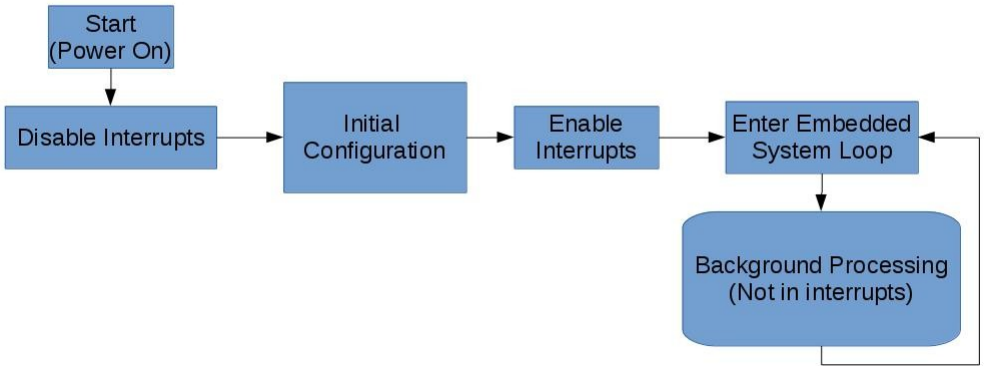

Figure 1: Typical Microcontroller Program Flow

Configuring Inputs and Outputs

Inputs and outputs can be configured by utilizing the TRISx register. The x should be replaced by the port that should be modified (either A or B). By setting a bit to a 1 in this register, a pin is configured to be an input.

For example:

```
ANSELA = 0;
TRISA = 1<<0;
```

This will set pin A0 as a digital input. The first line configures all port A pins as digital by writing 0 to the analog selection register. The next line configures A0 as an input by writing a 1 to position 0 in TRISA. Digital input pins **must** be configured by writing to the ANSELx register.

When configuring outputs, the TRISx register must be set to a 0 and the default output value must be set in the LATx register. For example:

```
ANSELA = 0;
TRISA = 0;
LATA |= 1 << 2;
```

This will set pin A2 as an output of digital value 1. The first line sets the pin to digital and the second configures it as an output by writing 0 to position 2 of TRISA. The third line specifies what value to output on A2 by writing 1 to position 2 of the latch register.

For more information about inputs, outputs, and reading values, read the Coding Cheatsheet.

Procedures:

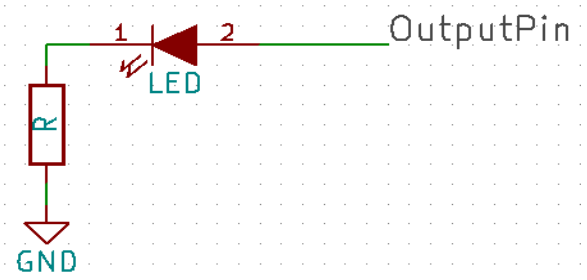

Figure 2: Output LED Schematic

Part 1:
Configure the microcontroller to have two LED outputs. To determine the pin you are using, look at the white label on the board. Place current limiting resistors (100 Ohm) in series with the LED to ensure that they are drawing the proper amount of current. Refer to *Figure 2*.

Tip: To read about how to configure inputs and outputs, please read the Coding Cheatsheet document. For an example using pins A0 and A1, please refer to *Figure 4*.

Next, place two push button inputs on two other pins of the microcontroller. These buttons should have one end connected to the pin of the microcontroller and the other connected to ground. These buttons will 'pull' the pin down to ground when they are pressed. Add very large resistors (10k Ohm or above) with one lead connected to 3.3V and the other connected to the pin of the microcontroller. Because the pins take no current input, there is no voltage drop across the resistor when the button is not pressed and the pin is at the same voltage as the high side of the resistor. When the button is pressed, the pin is connected to ground and current flows through the resistor. This is

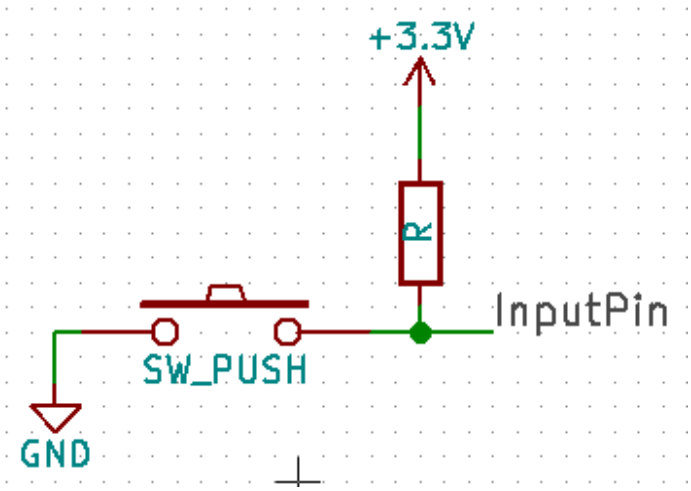

Figure 3: Button Input Schematic

why a large resistance value was selected – the larger the resistance value, the less current that flows when the button is pressed. Refer to *Figure 3*.

Now, write two functions to blink each LED five times. One function should blink the first LED and the other should blink the second. To make an LED blink, periodically write an output of 1 or 0 to the respect LATx register. Use empty for loops to delay between operations. An example of one of the functions can be found in *Figure 4*. You will need to write the second function for the other LED.

Once the functions are written and the inputs and outputs are configured, program the embedded system loop – the loop beginning with "while (1)". Write code to check if either button one or two has been pressed. If they have, call the respective blink function.

> **Tip**: See the Coding Cheatsheet for help with reading input pin status. Refer to *Figure 4* below for an example of most of the code you will need.

Try your program out. What happens when you try to press both buttons at once?

Post-Lab

The code you wrote is limited. What you have implemented is commonly referred to as a 'polling' architecture. The pin status is checked in the embedded loop and a function executes if a state is found. However, only one thing can be done at a time because of this. Polling can limit the abilities of the microcontroller. If the microcontroller needs to do more than one thing at a time, the design of the program will need to be changed toward what is referred to as an interrupt-driven design.

Beware that both polling and interrupt-driven designs have advantages and disadvantages. Polling designs are simpler and often quicker to implement, but tend to have reduced functionality. Interrupt-driven designs tend to be able to do more tasks simultaneously, but can be difficult to design and understand. Additionally, with interrupt driven designs, it can be harder to guarantee that tasks will be accomplished when they are scheduled. There will be more discussion on the trade-offs between interrupt and polling architectures in later labs.

- Can both LEDs blink at the same time? If not, why?
- What are some ways of implementing this lab to have both LEDs blink simultaneously?
- Where did the magic variables ANSELA, ANSELB, TRISA, etc. come from and what do they actually represent? Do you have to use them in your code to modify those registers?

Try replacing all of the magic variables with your own. Read the datasheet to get the addresses of each of the registers. In Table11-3 of the datasheet, it is shown that TRISA resides at memory location 0xBF880_6010. For an example, the following would replace TRISA:

```
int *myTRISA = (int *)0xBF886010; //Declare your own TRISA
*myTRISA = 1; //Write to your TRISA
```

```c
#include "sublibinal.h"

void blinkOne()
{
    int i = 0, j = 0;
    int count = 300000;
    for (j = 0; j < 5; j++)
    {
        LATA |= (1<<0); //Set LED high
        for (i = 0; i < count; i++);

        LATA &= ~(1<<0); //Set LED low
        for (i = 0; i < count; i++);
    }
}

void blinkTwo()
{
    //Write your own similar to
    //blinkOne, except use a new output
}

int main()
{
    //Disable all analog pins
    ANSELA = 0;
    ANSELB = 0;

    //Configure A0 and A1 as outputs
    TRISA &= ~(1<<0); //Set the TRISx to 0
    TRISA &= ~(1<<1);
    LATA &= ~(1<<0); //Set initial output to 0
    LATA &= ~(1<<1);

    /* Set up your own input pins for the buttons! */

    enable_Interrupts();

    while (1)
    {
        if (/*Button 1 is pressed*/) blinkOne();
        if (/*Button 2 is pressed*/) blinkTwo();
    }
}
```

Figure 4: Example Program.