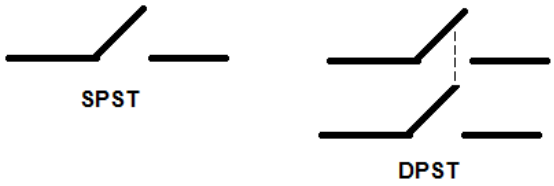
Chapter 7: Using the I/O pins as Inputs.

In addition to working as outputs and being able to turn the I/O pins on and off, these same pins can be used as inputs. In this mode the PIC is able to determine whether there is a voltage present on an individual pin or not. One application of this is to allow you to connect a pushbutton to an I/O pin and determine whether the button has been pressed or not. In its simplest form, a push button has two pins on it. When the button is pressed, the two contacts are connected to complete the circuit that runs through the pushbutton. The simplest pushbutton is called a **single pole, single throw** or **SPST** pushbutton. The single pole refers to the fact that only one circuit is being switched. A **double pole, single throw** (**DPST**) button would allow two circuits to be switched on or off with a single button press.



Most of the small pushbutton switches that you see that have four leads are actually not DPST switches. Instead they are SPST switches with two pins connected to each side. When you use these switches you must be careful to select a pair of pins that are not permanently connected together. You could by using an inexpensive multimeter that will measure the resistance between two points. If you find there is zero resistance when you are not pushing the button, you have the wrong pair of pins. Alternatively, if you use any pair of pins that are diagonal from each other, you will be sure that they will work properly.

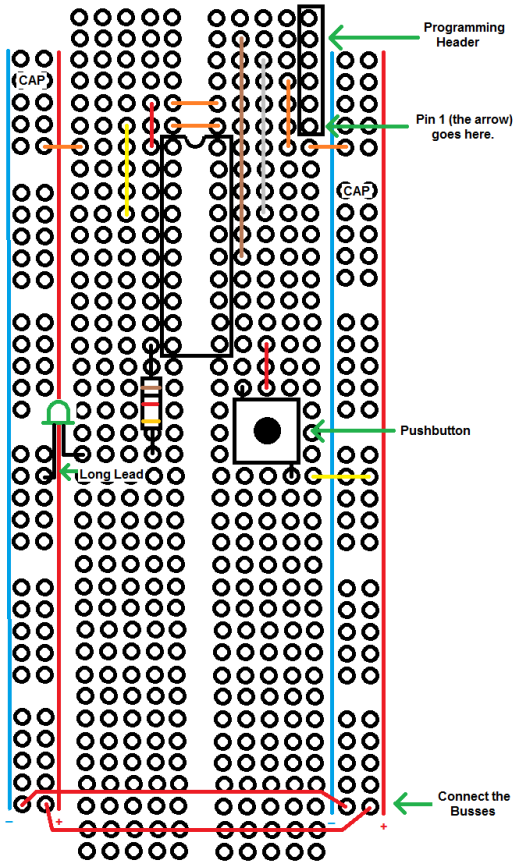


Figure 1: Pushbutton without Pull-up.

In connecting a pushbutton to a PIC pin, the idea is to either provide 5 volts when the button is pressed and zero volts when it is not pressed or, alternatively, to provide 0 volts when the button is pressed and 5 volts when the button is not pressed.

You might think that the easiest way to do this is simply to hook one side of the button up to the PIC pin and the other side up to the positive power rail. This would certainly provide 5 volts when the button is pressed. The problem with this approach is that when button is not pressed the pin is simply left floating; it is not solidly

connected to ground. You can't simply solve the problem by connecting the pin to the ground bus (and then have the button connect it to the 5 volt bus when pressed) because if you did that, pushing the button would short the two power rails together. Instead, the way to solve this problem is to use something called a "pull-up resistor". Connect the pull up resistor (typically 10,000 ohms) between the PIC pin and the 5 volt bus. This provides a solid 5 volts on the PIC pin, but doesn't result in much current flow. Connect the other side of the push button to the ground bus (instead of the positive bus). When the button is not pressed the PIC pin will see 5 volts. When the button is pressed, the pin will be connected directly to ground and the PIC pin will see 0 volts. While the button is being pressed, the two buses will be connected, but it won't be a short because the current will have to flow through the resistor to get from one to the other. A small amount of current will flow, but not enough to do any damage. (try this: use Ohm's law to calculate how much current will flow.) When a resistor is used in this way it is called a "weak pull-up" because it connects the pin to the rail through a resistor rather than directly.

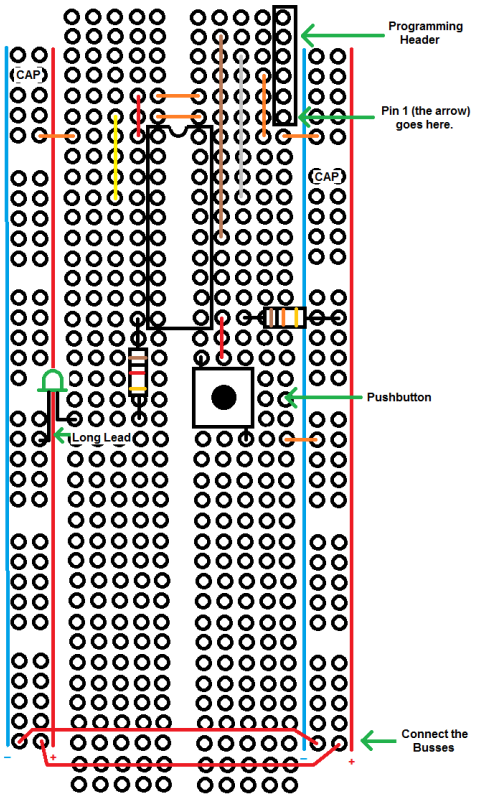
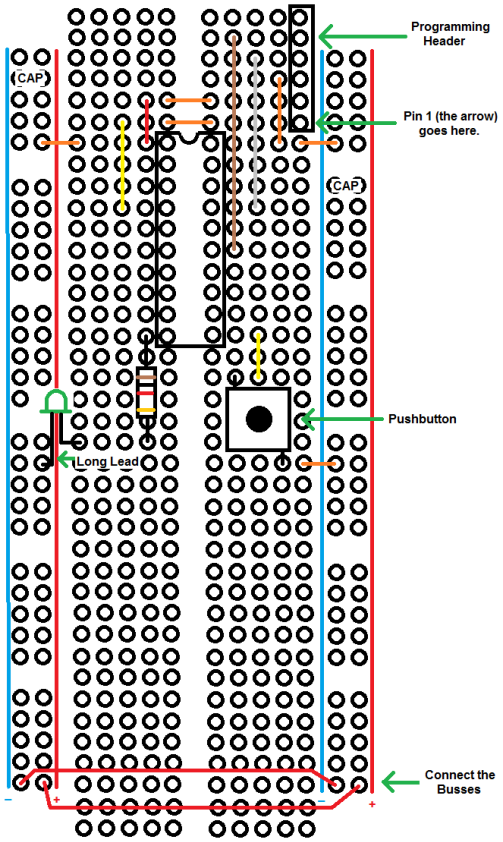


Figure 2: Pushbutton Using External Pull-up Resistor.



This strategy would work quite well, but is unnecessary because the PIC chip contains pull-up resistors that can be enabled in code. All you need to do is tell the chip to turn it on for the pin you will be using with your button and you won't need the external pull-up resistor. This approach is shown in Figure 3.

Recall that the PIC register that determines whether the pin is an input or an output is the TRIS register. In Figure 3 the pushbutton is connected to pin 6 of port B. Thus bit 6 of TRISB must be set to a 1 to make this pin an input.

To turn on the weak pull-up resistor we need to do two things. There is a global switch that can allow weak pull-ups to be used that is located in a register called OPTION_REG. It is the WPUEN bit (for global **w**eak **p**ull-**u**p **en**able) and is bit number 7. The OPTION_REG is located in Bank 1 *(not bank 0!).* It may be somewhat counter-intuitive, but to turn ON the global pull-ups, you clear this bit (make it a 0). To disable global pull-ups, you set the bit (make it a 1).

Figure 3: Pushbutton Using Internal Pull-up.

In addition to clearing this bit, you must also turn on the pull-up for pin 6 of port B. This is done in the WPUB register that is located in bank 4. To turn on the interrupt for this pin, after you switch to bank 4, set bit 6.

```
bsf   WPUB, 6
```

In order to check to see whether the button has been pressed, we must be able to "read" bit 6 on port B. Here are a pair of instructions that will facilitate this:

```
btfss  PORTB,6

btfsc  PORTB,6
```

The first instruction skips the line of code that follows it if bit 6 of the PORTB register is a 1. The mnemonic is **b**it **t**est **f**ile **s**kip if **s**et. The second line skips the line of code that follows it if bit 6 of the PORTB register is a 0. The mnemonic is **b**it **t**est **f**ile **s**kip if **c**lear. Remember that the pin is high (reading +5 volts) when the button is **NOT** pressed because of the pull-up resistor. When the button is pressed, the pin is connected to ground and the pin will be low (reading 0 volts). So if the button is pressed, bit 6 of PORTB is a 0 and if the button is not pressed the bit is a 1. This is somewhat counter-intuitive, but it stems from the resistor being a "pull-up" not a "pull-down". To complicate matters, also remember that when bit 7 of PORTB is low the LED is on, and when bit 7 is high the LED is off.

This may sound like a lot of double negatives, and it can get a bit confusing. Think about it first in English and then translate it into assembly language.

When the button is not pressed turn the light off. Translates to:

When pin 6 of PORTB is high set bit 7 of PORTB to high. Translates to:

When pin 6 of PORTB is high DON'T set bit 7 of PORTB to low (because of the "skip" we want to know what NOT to do)

The assembly code for this is:

```
btfss   PORTB, 6
bcf     PORTB, 7
```

Similarly we also want:

When the button is pressed turn the light on. Translates to:

When pin 6 of PORTB is low set bit 7 of PORTB to low. Translates to:

When pin 6 of PORTB is low DON'T set bit 7 of PORTB to high (because of the "skip" we want to know what NOT to do)

The assembly code for this is:

```
btfsc   PORTB, 6
bsf     PORTB, 7
```

Here is a program that will cause the LED to be turned on when the button is pressed and turned off when the button is released:

```
    radix hex
    include "p16f1459.inc"
    __CONFIG _CONFIG1, _FOSC_INTOSC & _WDTE_OFF & _PWRTE_OFF & _MCLRE_ON & _CP_OFF &
                    _BOREN_ON & _CLKOUTEN_OFF & _IESO_OFF & _FCMEN_OFF
   __CONFIG _CONFIG2, _WRT_OFF & _CPUDIV_CLKDIV6 & _USBLSCLK_48MHz & _PLLMULT_3x & _PLLEN_DISABLED
                    & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF
    errorlevel -302
    org 0

  call INITIAL

LOOP
  btfss PORTB, 6
  bcf  PORTB, 7
  btfsc PORTB, 6
  bsf  PORTB, 7
  goto LOOP

INITIAL
  movlb 1
  movlw b'01111111'    ; bits 0 through 6 are all set to inputs.  Bit 6 must be a 1, bit 7 must be a 0.
  movwf TRISB
  movlw b'00110100'
  movwf OSCCON
  bcf    OPTION_REG, 7    ;clear bit 7 to turn on global weak pull-ups
  movlb 4
  bsf   WPUB, 6              ;turn on weak pull-up for RB6
  movlb 3
  clrf ANSELB
  movlb 0
  return

  end
```

The block of code that begins with the radix hex statement and ends with org 0 is exactly the same as the code we wrote for the program in Chapter 5. Refer to that chapter if you need any review on what it means. Then once again we have an initialization section that sets up the PIC so it will work properly. In summary:

Register	Value	Bank	Comments
TRISB	01111111	1	Pin 7 (the LED) is an output; Pin 6 (button) is an input.
OSCCON	00110100	1	Set oscillator to run at 4 MHz.
OPTION_REG	pin 7 =0	1	Turn on global weak pull-up capability.
WPUB	pin 6 = 1	4	Turn on the weak pull-up on pin 6 of port B.
ANSELB	00000000	3	All PORTB pins should be digital, not analog.

After this initialization we'll enter an infinite loop (label LOOP) that implements the logic discussed above.

Button Debouncing

Let's look at a potential problem that sometimes occurs when dealing with buttons. Suppose we want to write program that changes the state of the light when you press the button. Push the button to turn it off; push it again to turn it on. This is what happens, for example, on your TV remote control. The same button turns the device on and off. The configuration code at the top of the program will be the same as in the previous program as will the initialization code, so it won't be reproduced here. First, let's bring back the BLINK code from the program in Chapter 5:

```
BLINK
        movlw B'10000000'
        xorwf PORTB,F
        return
```

Recall that in this section we used the xorwf instruction to flip the state of the LED. This is exactly what we want to happen in the current program, but we only want it to happen when we have pressed the button. So consider translating this again from English to assembly language:

When the button is pressed, change the state of the light. Translates to:

When the button is pressed, call BLINK. Translates to:

When the button is not pressed, skip the instruction that calls BLINK. Translates to:

```
        btfss  PORTB,6
        call BLINK
```

So the infinite loop will be:

```
LOOP
        btfss  PORTB,6
        call BLINK
        goto LOOP
```

Looks good, right? Try this: Put this code in your chip and press the button and see what happens. It will probably not run the way you expect. Sometimes when you press the button the light will change, sometimes it won't. There are two reasons for this. One is that it only takes a few microseconds for this loop to execute. You may be fast, but you probably can't get your finger off that button before the loop has executed literally dozens of times. If it happens to execute an odd number of times, the light will change state. If it executes an even number of times the light won't change.

But that's not the only problem; the other problem is mechanical. Even if you were extremely quick in pressing the button, the button itself will "bounce". That is, it will make and break

contact several times while you pressing it once. This phenomenon is called "button bounce" and in order to fix it, you need to "de-bounce" the button. The simplest way to do this is to bring back your delay routine from Chapter 5 and have that delay run after the button has changed state. So now the BLINK routine will look like this:

```
BLINK
        movlw B'10000000'
        xorwf PORTB,F
        call DELAY
        return
```

Recall that the DELAY routine paused for about 200 ms before proceeding. That's most likely enough time to get your finger off the button. If the only issue is the button bounce issue itself, 10 ms would have been enough time.

But now you have another odd behavior. Push the button and hold it down. It will cause the LED to flash just like it did in the program in Chapter 5. The reason is that the BLINK routine is executing constantly (because the button is held down) and BLINK has a delay built in before it is allowed to continue.

TRY THIS: Figure out a way to cause the LED to change state once and only once if you hold the button down.

TRY THIS: Add a button to the traffic intersection you simulated at the end of Chapter 6. This button will be a crosswalk button. When the button is pressed, the next time the lights change they should be held so that both directions are red for 3 seconds before proceeding. Note, the lights should not immediately go red (this would kill someone!); instead, you should wait until the next time they would have changed and pause at both being red before proceeding.