Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 4: Reading Switches

The previous lessons introduced simple digital output, by turning on or flashing an LED. That's more useful than it seems, because, with some circuit changes (such as adding transistors and relays), it can be readily adapted to turning on and off almost any electrical device.

But most systems need to interact with their environment in some way; to respond according to user commands or varying inputs. The simplest form of input is an on/off switch. This lesson shows how to read and respond to a simple pushbutton switch, or, equivalently, a slide or toggle switch, or even something more elaborate such as a mercury tilt switch, or a sensor with a digital output – anything that makes or breaks a single connection, or is "on" or "off", "high" or "low".

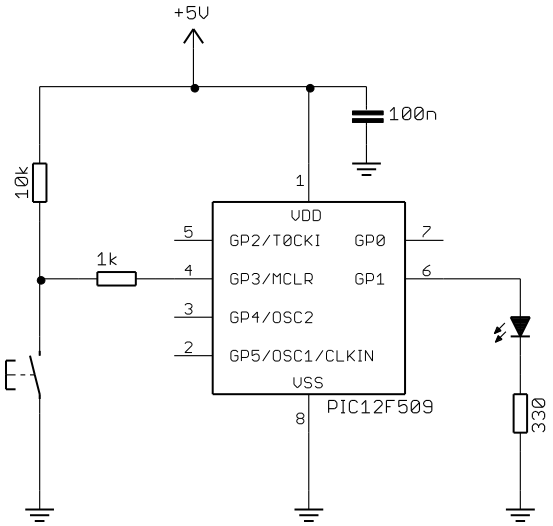
This lesson covers:

- Reading digital inputs
- Conditional branching
- Using internal pull-ups
- Hardware and software approaches to switch debouncing

Example Circuit

To show how to read a pushbutton switch, we'll need a circuit with a pushbutton!

The Gooligum baseline training board provides tact switches connected to pins GP2 and GP3, while the Microchip Low Pin Count demo board only has a tact switch connected to GP3, as in the circuit shown below, so we'll use this circuit in this lesson.



We'll continue to use a LED on GP1, as in the previous lessons. If you're using the Gooligum training board, you should connect jumper JP3, to bring the 10 kΩ resistor into the circuit.

The pushbutton is connected to GP3 via a 1 kΩ resistor. This is good practice, but not strictly necessary. Such resistors are used to provide some isolation between the PIC and the external circuit, for example to limit the impact of over- or under-voltages on the input pin, or to provide some protection against an input pin being inadvertently programmed as an output. If the switch was to be pressed while the pin was mistakenly configured as an output and set "high", the result is

likely to be a dead PIC – unless there is a resistor to limit the current flowing to ground.

In this case, that scenario is impossible, because, as mentioned in lesson 1, GP3 can only ever be an input. So why the resistor? It is necessary to allow the PIC to be safely and successfully programmed.

You might recall, from lesson 0, that the PICkit 2and PICkit 3 are In-Circuit Serial Programming (ICSP) programmers. The ICSP protocol allows the PICs that support it to be programmed while in-circuit. That is, they don't have to be removed from the circuit and placed into a separate, stand-alone programmer. That's very convenient, but it does place some restrictions on the circuit. The programmer must be able to set appropriate voltages on particular pins, without being affected by the rest of the circuit. That implies some isolation, and often a simple resistor, such as the 1 kΩ resistor here, is all that is needed.

To place a PIC such as the 12F509 into programming mode, a high voltage (around 12V) is applied to pin 4 – the same pin that is used for GP3. Imagine what would happen if, while the PIC was being programmed, with 12V applied to pin 4, that pin was grounded by someone pressing a pushbutton connected directly to it! The result in this case wouldn't be a dead PIC; it would be a dead PICkit 2 or PICkit 3 programmer… Or suppose that a sensor with a digital output was connected to the GP3 input. That sensor may not react well to having 12 V applied directly to its output!

But, if you are sure that you know what you are doing and understand the risks, you can leave out isolation or protection resistors, such as the 1 kΩ resistor on GP3.

The 10 kΩ resistor holds GP3 high while the switch is open. How can we be sure? According to the PIC12F509 data sheet, the "input leakage current" flowing into GP3 can be up to 5 µA (parameter D061). That equates to a voltage drop of up to 55 mV across the 10 kΩ and 1 kΩ resistors in series (5 µA × 11 kΩ), so, with the switch open, the voltage at GP3 will be a minimum of V_{DD} – 55 mV. The minimum supply voltage is 2.0 V (parameter D001), so in the worst case, the voltage at GP3 = 2.0 – 55 mV = 1.945 V. The lowest input voltage guaranteed to be read as "high" is given as 0.25 V_{DD} + 0.8 V (parameter D040A). For V_{DD} = 2.0 V, this is 0.25 × 2.0 V + 0.8 V = 1.3 V. That's well below the worst-case "high" input to GP3 of 1.945 V, so with these resistors, the pin is guaranteed to read as "high", over the allowable supply voltage range.

In practice, you generally don't need to bother with such detailed analysis. As a rule of thumb, 10 kΩ is a good value for a *pull-up* resistor like this. But, it's good to know that the rule of thumb is supported by the characteristics specified in the data sheet.

When the switch is pressed, the pin is pulled to ground through the 1 kΩ resistor. Now the input leakage current flows out of GP3, giving a voltage drop across the 1 kΩ resistor of up to 5 mV (5 µA × 1 kΩ), so with the switch closed, the voltage at GP3 will be a maximum of 5 mV. The highest input voltage guaranteed to be read as a "low" is 0.15 V_{DD} (parameter D030A). For V_{DD} = 2.0 V (the worst case), this is 0.15 × 2.0 V = 300 mV. That's above the maximum "low" input to GP3 of 5 mV, so the pin is guaranteed to read as "low" when the pushbutton is pressed.

Again, that's something you come to know as a rule of thumb. With just a little experience, you'll look at a circuit like this and see immediately that GP3 is normally held high, but is pulled low if the pushbutton is pressed.

Interference from $\overline{MCLR}$

There is a potential problem with using a pushbutton on GP3; as we have seen, the same pin can instead be configured (using the PIC's configuration word) as the processor reset, $\overline{MCLR}$.

This is potentially a problem because, by default, as we saw in lesson 1, MPLAB provides for control of the $\overline{MCLR}$ line through the "Release from Reset" and "Hold in Reset" menu items (MPLAB 8 only) and toolbar buttons (MPLAB 8 and MPLAB X).

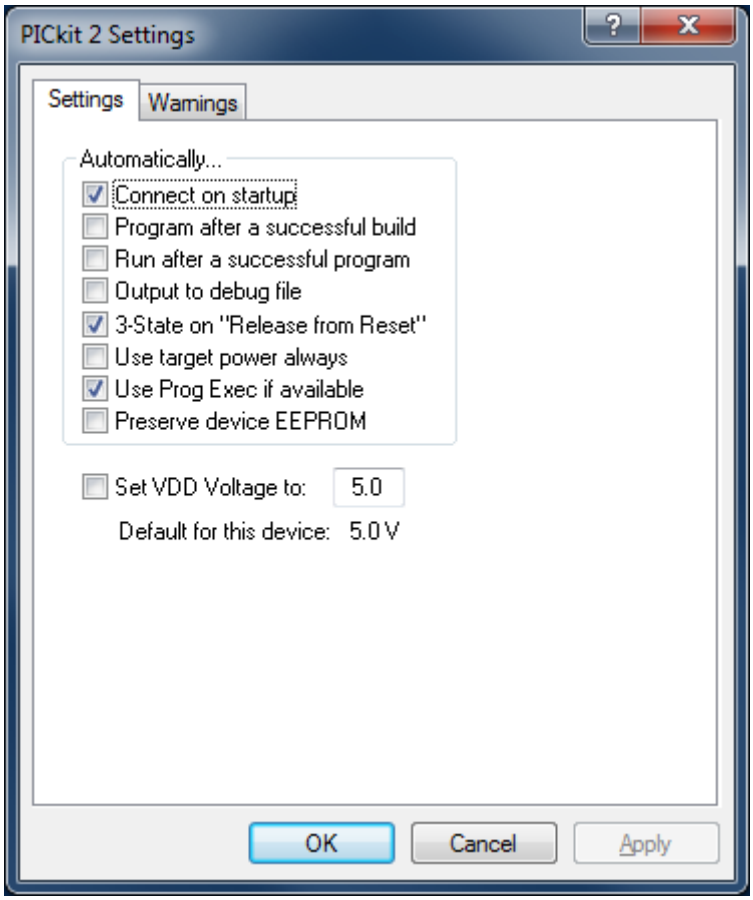
That's not actually a problem if you're using a PICkit 3, because when the PICkit 3 is used a programmer[1], its $\overline{\text{MCLR}}$ output is disconnected ("tri-stated") immediately after programming, meaning that the PICkit 3 won't affect the PIC's $\overline{\text{MCLR}}$ / GP3 input. The pull-up resistor and pushbutton are able to pull GP3 high and low, as described above.

It's different with the PICkit 2 where, by default, the PICkit 2 continues to assert control over the $\overline{\text{MCLR}}$ line and, because of the 1 kΩ isolation resistor, the 10 kΩ pull-up resistor and the pushbutton cannot overcome the PICkit 2's control of that line.

> *When the PICkit 2 is used as a programmer with MPLAB, it will, by default, assert control of the $\overline{\text{MCLR}}$ line, overriding any pushbutton switch on the PIC's $\overline{\text{MCLR}}$ / GP3 input.*

If you are using MPLAB 8, this problem can be overcome by changing the PICkit 2 programming settings, to tri-state the PICkit 2's $\overline{\text{MCLR}}$ output (effectively disconnecting it) when it is not being used to hold the PIC in reset.

To do this, select the PICkit 2 as a programmer (using the "Programmer → Select Programmer" submenu) and then use the "Programmer → Settings" menu item to display the PICkit 2 Settings dialog window, shown on the right. Select the '3-State on "Release from Reset"' option in the Settings tab and then click "OK".



After using the PICkit 2 to program your device, it will hold $\overline{\text{MCLR}}$ low, holding the GP3 input low, overriding the pull-up resistor.

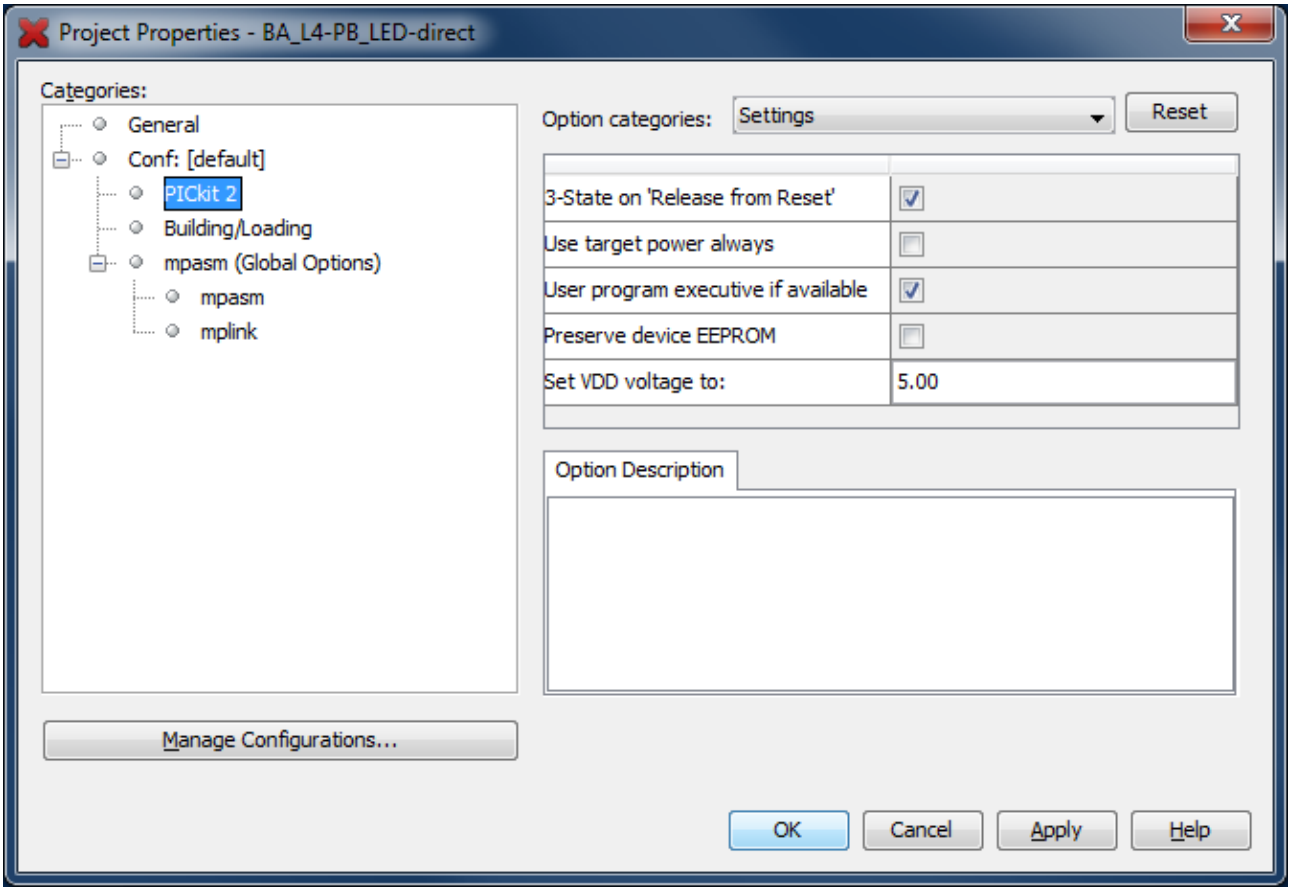
When you now click on the on the  button in the programming toolbar, or select the "Programmer → Release from Reset" menu item, the PICkit 2 will release control of $\overline{\text{MCLR}}$, allowing GP3 to be driven high or low by the pull-up resistor and pushbutton.

MPLAB X also allows you to prevent the PICkit 2 asserting control over $\overline{\text{MCLR}}$, in much the same way.

[1] as opposed to being used as a debugger; see lesson 0

To do this, open the Project Properties window, by selecting the "File → Project Properties" menu item, or right-clicking the project name in the Projects window and selecting "Properties", or simply clicking on the "Project Properties" button to the left of the Dashboard.

If you click on "PICkit 2", you will see the settings shown below:



Select "3-State on 'Release from Reset'", and then click "OK".

Now, when you build and run your project, the PICkit 2's $\overline{\text{MCLR}}$ output will be tri-stated, making it possible for you to use a pushbutton on GP3.

Reading Digital Inputs

In general, to read a pin, we need to:

- Configure the pin as an input
- Read or test the bit corresponding to the pin

Recall, from lesson 1, that the pins on the PICs we've seen so far, including the 12F509, are all only *digital* inputs or outputs. When configured as outputs, they can be turned on or off, but nothing in between. Similarly, when configured as inputs, they can only read a voltage as being "high" or "low".

As mentioned above, the data sheet defines input voltage ranges where the pin is guaranteed to read as "high" or "low". For voltages between these ranges, the pin might read as either; the input behaviour for intermediate voltages is *undefined*.

As you might expect, a "high" input voltage reads as a '1', and a "low" reads as a '0'.

To configure a pin as an input, set the corresponding TRIS bit to '1'.

However, as we've seen, GP3 is a special pin. If it is not configured as $\overline{\text{MCLR}}$, it can only be an input. It is not possible to use GP3 as an output. So, when using GP3 as an input, there's no need to set its TRIS bit. Although for clarity, you may as well do so.

Reading an input isn't much use unless we're able to respond to that input – to do something different, depending on whether the input is high or low.

This is where *bit test* instructions are useful. There are two:

'`btfsc f,b`' tests bit 'b' in register 'f'. If it is '0', the following instruction is skipped – "**b**it **t**est **f**ile register, **s**kip if **c**lear".

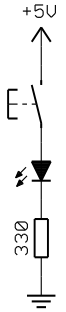
'`btfss f,b`' tests bit 'b' in register 'f'. If it is '1', the following instruction is skipped – "**b**it **t**est **f**ile register, **s**kip if **s**et".

Their use is illustrated in the following example.

Example 1: Read a switch

We'll start by simply lighting the LED only while the pushbutton is pressed.

Of course, that's a waste of a microcontroller. To get the same effect, you could leave the PIC out and build the circuit shown on the right!



But, this simple example avoids having to deal with the problem of switch contact bounce, which we'll look at later.

Here's some code that will do this:

```
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO            ; (GP3 is an input)

        clrf    GPIO            ; start with GPIO clear (GP1 low)
loop
        btfss   GPIO,3          ; if button pressed (GP3 low)
        bsf     GPIO,1          ;   turn on LED
        btfsc   GPIO,3          ; if button up (GP3 high)
        bcf     GPIO,1          ;   turn off LED

        goto    loop            ; repeat forever
```

Note that the logic seems to be inverse; the LED is turned on if GP3 is clear, yet the '`btfss`' instruction tests for the GP3 bit being set. The bit test instructions skip the next instruction if the bit test condition is met, so the instruction following a bit test is executed only if the condition is *not* met. Often, following a bit test instruction, you'll place a '`goto`' or '`call`' to jump to a block of code that is to be executed if the bit test condition is not met. In this case, there is no need, as the LED can be turned on or off with single instructions:

'`bsf f,b`' sets bit 'b' in register 'f' to '1' – "**b**it **s**et **f**ile register".

'`bcf f,b`' clears bit 'b' in register 'f' to '0' – "**b**it **c**lear **f**ile register".

Previously, we have set, cleared and toggled bits by operating on the whole GPIO port at once.

That is what these bit set and clear instructions are doing behind the scenes; they read the entire port, set or clear the designated bit, and then rewrite the result. They are examples of 'read-modify-write' instructions, as discussed in lesson 2, and their use can lead to unintended effects – you may find that bits other than the designated one are also being changed.

This unwanted effect often occurs when sequential bit set/clear instructions are performed on the same port. Trouble can be avoided by separating sequential '`bsf`' and '`bcf`' instructions with a '`nop`'.

Although unlikely to be necessary in this case, since the bit set/clear instructions are not sequential, a shadow register (see lesson 2) could be used as follows:

```
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO            ; (GP3 is an input)

        clrf    GPIO            ; start with GPIO clear (LED off)
        clrf    sGPIO           ; update shadow copy

loop
        btfss   GPIO,3          ; if button pressed (GP3 low)
        bsf     sGPIO,1         ;   turn on LED
        btfsc   GPIO,3          ; if button up (GP3 high)
        bcf     sGPIO,1         ;  turn off LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

It's possible to optimise this a little. There is no need to test for button up as well as button down; it will be either one or the other, so we can instead write a value to the shadow register, assuming the button is up, and then test just once, updating the shadow if the button is found to be down.

It's also not really necessary to initialise GPIO at the start; whatever it is initialised to, it will be updated the first time the loop completes, a few µs later – much too fast to see.

If setting the initial values of output pins correctly is important, to avoid power-on glitches that may affect circuits connected to them, the correct values should be written to the port registers before configuring the pins as outputs, i.e. initialise GPIO before using `tris` to configure the port.

But when dealing with human perception, it's not important, so the following code is acceptable:

```
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO            ; (GP3 is an input)

loop
        clrf    sGPIO           ; assume button up -> LED off
        btfss   GPIO,3          ; if button pressed (GP3 low)
        bsf     sGPIO,1         ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

If you didn't use a shadow register, but tried to take the same approach – assuming a state (such as "button up"), setting GPIO, then reading the button and changing GPIO accordingly – it would mean that the LED would be flickering on and off, albeit too fast to see. Using a shadow register is a neat solution that avoids this problem, as well as any read-modify-write concerns, since the physical register (GPIO) is only ever updated with the correctly determined value.

Complete program

Here's the "light an LED when button pressed" code again, along with the other parts we need to make a complete working program. Note that the comments include the statement that the pushbutton switch is "active low", meaning that it's connected such that when it is pressed (activated), the PIC input is driven low. It makes it easier to maintain your code if you are clear about any assumptions like that.

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 4, example 1b                        *
;                                                                *
;   Demonstrates reading a switch                               *
;   (using shadow register to update port)                      *
;                                                                *
;   Turns on LED when pushbutton on GP3 is pressed              *
;                                                                *
;*****************************************************************
;                                                                *
;   Pin assignments:                                            *
;       GP1 = LED                                               *
;       GP3 = pushbutton switch (active low)                    *
;                                                                *
;*****************************************************************

    list        p=12F509
    #include    <p12F509.inc>


;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC


;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO
```

```
;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR ****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration


;***** MAIN PROGRAM ****************************************************

;***** Initialisation
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO            ; (GP3 is an input)


;***** Main loop
main_loop
        ; turn on LED only if button pressed
        clrf    sGPIO           ; assume button up -> LED off
        btfss   GPIO,3          ; if button pressed (GP3 low)
        bsf     sGPIO,1         ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        ; repeat forever
        goto    main_loop


        END
```

Debouncing

In most applications, you want your code to respond to transitions; some action should be triggered when a button is pressed or a switch is toggled.

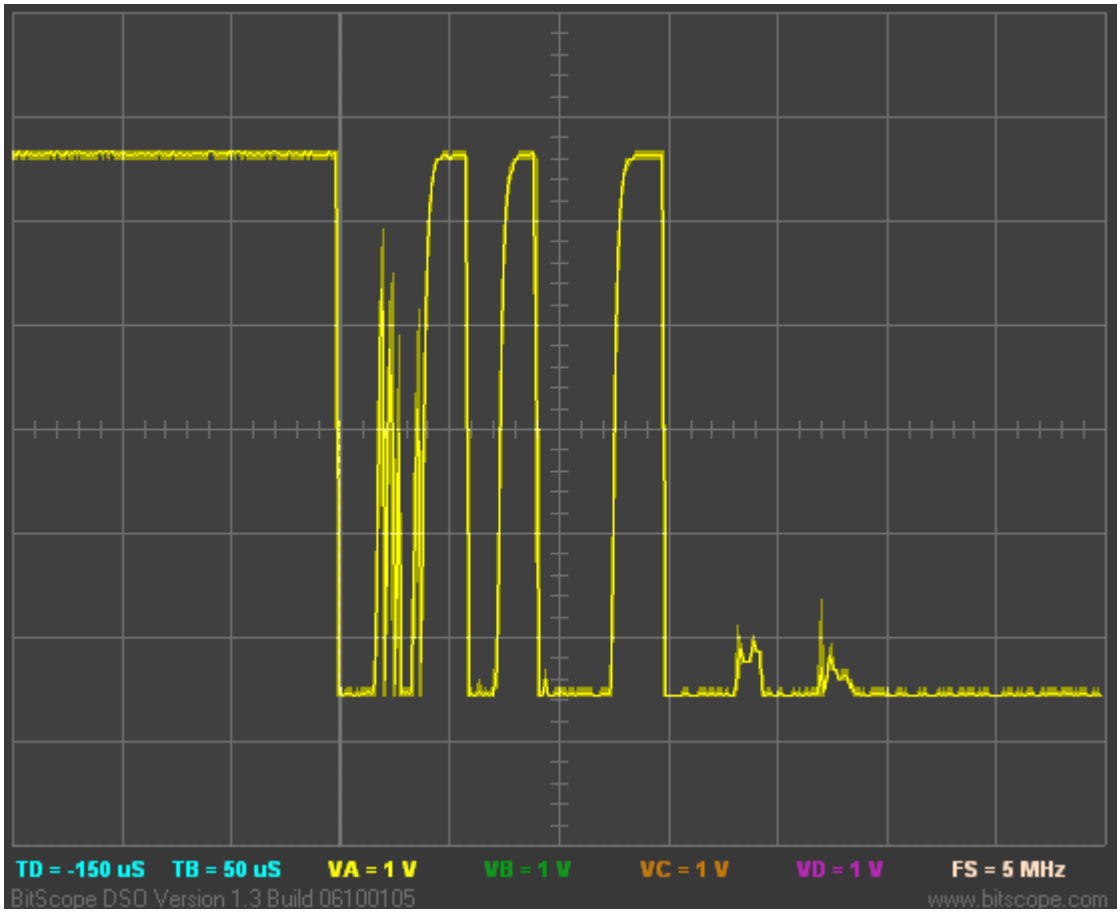
This presents a problem when interacting with real, physical switches, because their contacts *bounce*.

When most switches change, the contacts in the switch will make and break a number of times before settling into the new position. This contact bounce is generally too fast for a human to perceive, but microcontrollers are fast enough to react to each of these rapid, unwanted transitions.

Overcoming this problem is called switch *debouncing*.

There are many possible ways to address the problem of switch bounce, some of which we'll look at in this section.

The picture at the top of the next page is a recording of an actual switch bounce, using a common pushbutton switch:



The switch transitions several times before settling into the new state (low), after around 250 µs.

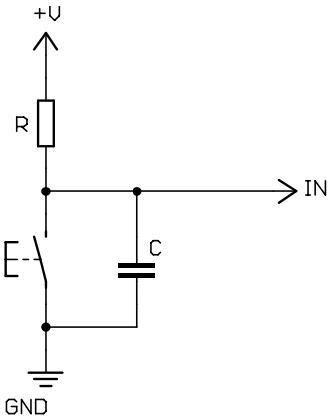
A similar problem can be caused by *electromagnetic interference (EMI)*. Unwanted spikes may appear on an input line, due to electromagnetic noise, especially (but not only) when switches or sensors are some distance from the microcontroller. But any solution which deals effectively with contact bounce will generally also remove or ignore input spikes caused by EMI.

Hardware debouncing

Debouncing is effectively a filtering problem; you want to filter out fast transitions, leaving only the slower changes that result from intentional human input.

That suggests a low-pass filter; the simplest of which consists of a resistor and a capacitor (an "*RC filter*").

To debounce a normally-open pushbutton switch, pulled high by a resistor, the simplest hardware solution is to place a capacitor directly across the switch, as shown at right.



In theory, that's all that's needed. The idea is that, when the switch is pressed, the capacitor is immediately discharged, and the input will go instantly to 0 V. When the contacts bounce open, the capacitor will begin to charge, through the resistor. The voltage at the input pin is the voltage across the capacitor:

$$V_{in} = V_{DD}\left(1 - e^{-t/RC}\right)$$

This is an exponential function with a *time constant* equal to the product RC.

The general I/O pins on the PIC12F509 act as TTL inputs: given a 5 V power supply, any input voltage between 0 V and 0.8 V reads as a '0'. As long as the input remains below 0.8 V, the PIC will continue to read '0', which is what we want, to avoid transitions to '1' due to switch bounce.

Solving the above equation for V_{DD} = 5.0 V and V_{in} = 0.8 V gives t = 0.174RC.

This is the maximum time that the capacitor can charge, before the input voltage goes higher than that allowed for a logical '0'. That is, it's the longest 'high' bounce that will be filtered out.

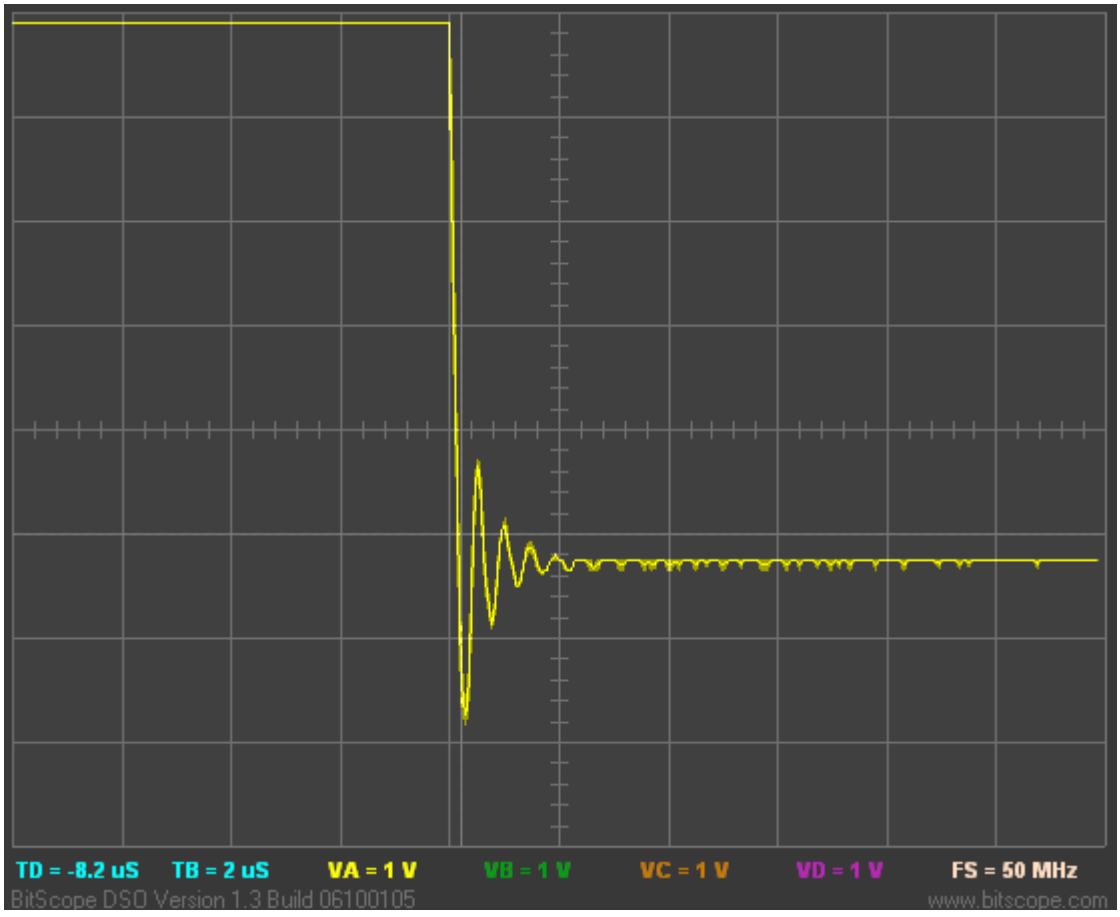
In the pushbutton press recorded above, the longest 'high' bounce is approx. 25 µs. Assuming a pull-up resistance of 10 kΩ, as in the original circuit, we can solve for C = 25 µs ÷ (0.174 × 10 kΩ) = 14.4 nF. So, in theory, any capacitor 15 nF or more could be used to effectively filter out these bounces.

In practice, you don't really need all these calculations. As a rule of thumb, if you choose a time constant several times longer than the maximum settling time (250 µs in the switch press above), the debouncing will be effective. So, for example, 1 ms would be a reasonable time constant to aim for here – it's a few times longer than the settling time, but still well below human perception (no one will notice a 1 ms delay after pressing a button).

To create a time constant of 1 ms, you can use a 10 kΩ pull-up resistor with a 100 nF capacitor:

10 kΩ × 100 nF = 1 ms

Testing the above circuit, with R = 10 kΩ, C = 100 nF and using the same pushbutton switch as before, gave the following response:



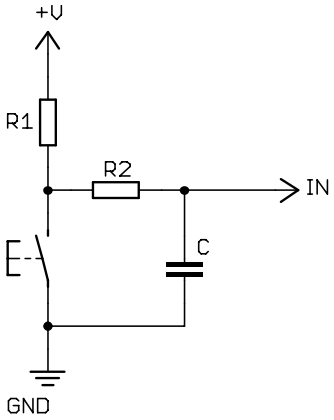
Sure enough, the bounces are all gone, but there is now an overshoot – a ringing at around 2 MHz, decaying in around 2 µs. The problem is that the description above is idealised. In the real world, capacitors and switches and the connections between them all have resistance, so the capacitor cannot discharge instantly. More significantly, every component and interconnection has some inductance. The combination of

inductance and capacitance leads to oscillation (the 2 MHz ringing). Inductance has the effect of maintaining current flow. When the switch contacts are closed, a high current rapidly discharges the capacitor. The inductance causes this current flow to continue beyond the point where the capacitor is fully discharged, slightly charging in the opposite direction, making V_{in} go (briefly) negative. Then it reverses, the capacitor discharging in the opposite direction, overshooting again, and so on – the oscillation losing energy to resistance and quickly dying away.

So – is this a problem? Yes!

According to the PIC12F509 data sheet, the absolute minimum voltage allowed on any input pin is -0.3 V. But the initial overshoot in the pushbutton press response, shown above, is approx. -1.5 V. That means that this simple debounce circuit is presenting voltages outside the 12F509's absolute maximum ratings. You might get away with it. Or you might end up with a fried PIC!

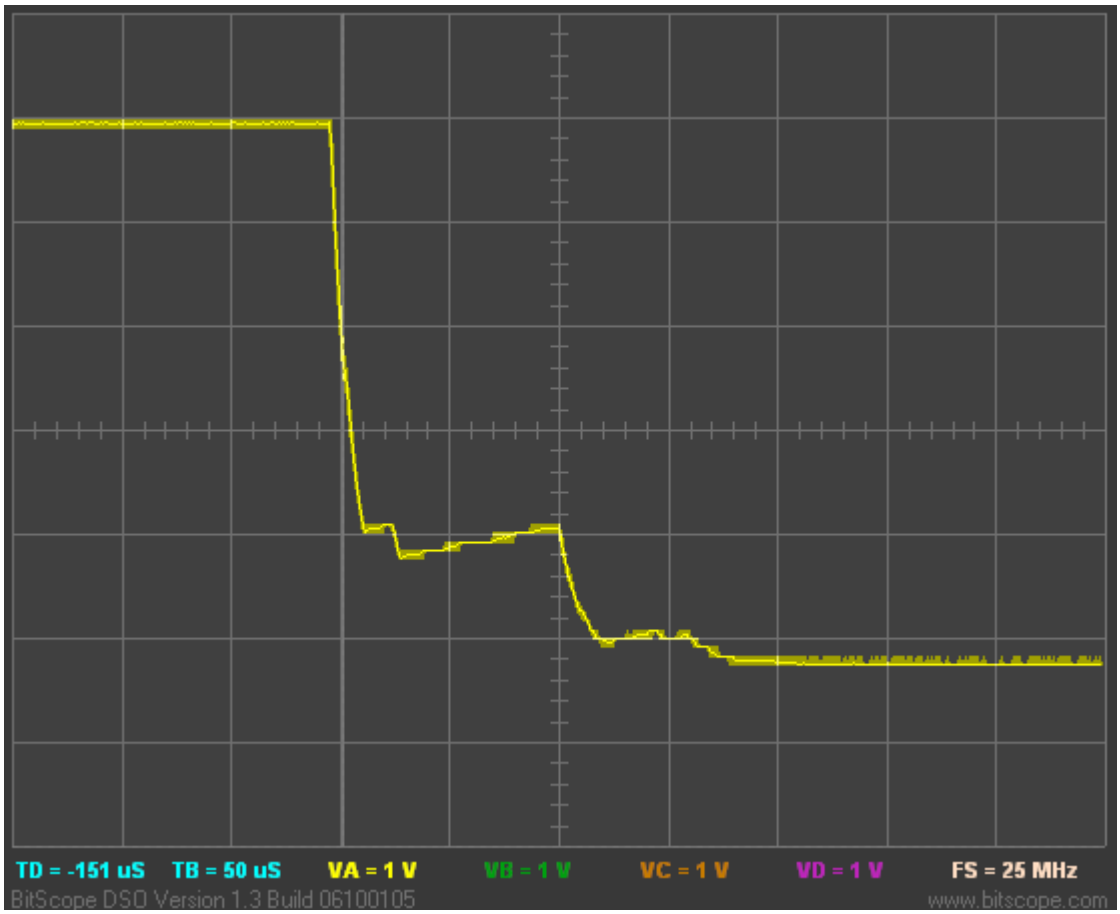
To avoid this, we need to limit the discharge current from the capacitor, since it is the high discharge current that is working through stray inductance to drive the input voltage to a dangerously low level.



In the circuit shown at right, the discharge current is limited by the addition of resistor R2.

We still want the capacitor to discharge much more quickly than it charges, since the circuit is intended to work essentially the same way as the first – a fast discharge to 0 V, followed by much slower charging during 'high' bounces. So we should have R2 much smaller than R1.

The following oscilloscope trace shows the same pushbutton response as before, with R1 = 10 kΩ, R2 = 100 Ω and C = 100 nF:



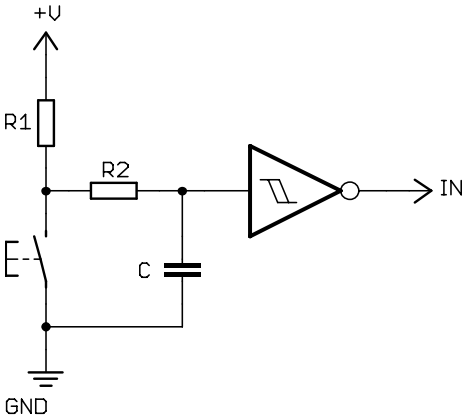
The ringing has been eliminated.

Instead of large steps from low to high, the bounces show as "ramps", of up to 75 µs, where the voltage rises by up to 0.4 V.

This effect could be reduced, and the decline from high to low made smoother, by adjusting the values of R1, R2 and C. But small movements up and down, of a fraction of a volt, will never be eliminated. And the fact that the high → low transition takes time to settle can present a problem.

With a 5 V power supply, according to the PIC12F509 data sheet, a voltage between 0.8 V and 2.0 V on a TTL-compatible input (any of the general I/O pins) is undefined. Voltages between 0.8 V and 2.0 V could read as either a '0' or a '1'. If we can't guarantee what value will be read, we can't say that the switch has been debounced; it's still an unknown.

An effective solution to this problem is to use a Schmitt trigger buffer, such as a 74HC14 inverter, as shown in the circuit on the right.



A Schmitt trigger input displays *hysteresis*; on the high → low transition, the buffer will not change state until the input falls to a low voltage threshold (say 0.8 V). It will not change state until the input rises to a high voltage threshold (say 1.8 V).

That means that, given a slowly changing input signal, which is generally falling, with some small rises on the way down (as in the trace above), only a single transition will appear at the buffer's output. Similarly, a Schmitt trigger will clean up slowly rising, noisy input signal, producing a single sharp transition, at the correct TTL levels, suitable for interfacing directly to the PIC.

Of course, if you use a Schmitt trigger inverter, as shown here, you must reverse your program's logic: when the switch is pressed, the PIC will see a '1' instead of a '0'.

Note that when some of the PIC12F509's pins are configured for special function inputs, instead of general purpose inputs, they use Schmitt trigger inputs. For example, as we've seen, pin 4 of the 12F509 can be configured as an external reset line ($\overline{\text{MCLR}}$) instead of GP3. When connecting a switch for external $\overline{\text{MCLR}}$ you only need an RC filter for debouncing; the Schmitt trigger input is built into the reset circuitry on the PIC.

Software debouncing

One of the reasons to use microcontrollers is that they allow you to solve what would otherwise be a hardware problem, in software. A good example is switch debouncing.

If the software can ignore input transitions due to contact bounce or EMI, while responding to genuine switch changes, no external debounce circuitry is needed. As with the hardware approach, the problem is essentially one of filtering; we need to ignore any transitions too short to be 'real'.

But to illustrate the problem, and provide a base to build on, we'll start with some code with no debouncing at all.

Suppose the task is to toggle the LED on GP1, once, each time the button on GP3 is pressed.

In pseudo-code, this could be expressed as:

```
do forever
      wait for button press
      toggle LED
      wait for button release
end
```

Note that it is necessary to wait for the button to be released before restarting the loop. This ensures that the LED will toggle only once per button press. If we didn't wait for the button to be released before continuing, the LED would continue to toggle as long as the button was held down; not the desired behaviour.

This pseudo-code could be implemented as:

```
loop
        ; wait for button press
wait_dn btfsc   GPIO,3          ; wait until GP3 low
        goto    wait_dn

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ;   in shadow register
        movwf   GPIO            ; and write to GPIO

        ; wait for button release
wait_up btfss   GPIO,3          ; wait until GP3 high
        goto    wait_up

        ; repeat forever
        goto    loop
```

If you build this into a complete program[2] and test it, you will find that it is difficult to reliably change the LED when you press the button; sometimes it will change, other times not. This is due to contact bounce.

Debounce delay

A simple approach to software is to estimate the maximum time the switch could possibly take to settle, and then merely wait at least that long, after detecting the first transition. If the wait time, or delay, is longer than the maximum possible settling time, you can be sure that after this delay the switch will have finished bouncing.

It's only a matter of adding a suitable debounce delay, after each transition is detected, as in the following pseudo-code:

```
do forever
      wait for button press
      toggle LED
      delay debounce_time
      wait for button release
      delay debounce_time
end
```

Note that the LED is toggled immediately after the button press is detected. There's no need to wait for debouncing. By acting on the button press as soon as it is detected, the user will experience as fast a response as possible.

But it is important to ensure that the "button pressed" state is stable (debounced), before waiting for button release. Otherwise, the first bounce after the button press would be seen as a release.

The necessary minimum delay time depends on the characteristics of the switch. For example, the switch tested above was seen to settle in around 250 µs. Repeated testing showed no settling time greater than 1 ms, but it's difficult to be sure of that, and perhaps a different switch, say that used in production hardware, rather than the prototype, may behave differently. So it's best to err on the safe side, and use the longest

[2] You'd need to add processor configuration, reset vector, initialisation code etc., and declare the `sGPIO` variable, as we've done before. Or download the complete source code from www.gooligum.com.au.

delay we can get away with. People don't notice delays of 20 ms or less (flicker is only barely perceptible at 50 Hz, corresponding to a 20 ms delay), so a good choice is probably 20 ms.

As you can see, choosing a suitable debounce delay is not an exact science!

The previous code can be modified to call the 10 ms delay module we developed in lesson 3, as follows:

```
main_loop
        ; wait for button press
wait_dn btfsc   GPIO,3          ; wait until GP3 low
        goto    wait_dn

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ;   in shadow register
        movwf   GPIO            ; and write to GPIO

        ; delay to debounce button press
        movlw   .2
        pagesel delay10
        call    delay10         ; delay 2 x 10 ms = 20 ms
        pagesel $

        ; wait for button release
wait_up btfss   GPIO,3          ; wait until GP3 high
        goto    wait_up

        ; delay to debounce button press
        movlw   .2
        pagesel delay10
        call    delay10         ; delay 2 x 10 ms = 20 ms
        pagesel $

        ; repeat forever
        goto    main_loop
```

If you build and test this code, you should find that the LED now reliably changes state every time you press the button.

Counting algorithm

There are a couple of problems with using a fixed length delay for debouncing.

Firstly, the need to be "better safe than sorry" means making the delay as long as possible, and probably slowing the response to switch changes more than is really necessary, potentially affecting the feel of the device you're designing.

More importantly, the delay approach cannot differentiate between a glitch and the start of a switch change. As discussed, spurious transitions can be caused be EMI, or electrical noise – or a momentary change in pressure while a button is held down.

A commonly used approach, which avoids these problems, is to regularly read (or *sample*) the input, and only accept that the switch is in a new state, when the input has remained in that state for some number of times in a row. If the new state isn't maintained for enough consecutive times, it's considered to be a glitch or a bounce, and is ignored.

For example, you could sample the input every 1 ms, and only accept a new state if it is seen 10 times in a row; i.e. high or low for a continuous 10 ms.

To do this, set a counter to zero when the first transition is seen. Then, for each sample period (say every 1 ms), check to see if the input is still in the desired state and, if it is, increment the counter before checking

again. If the input has changed state, that means the switch is still bouncing (or there was a glitch), so the counter is set back to zero and the process restarts. The process finishes when the final count is reached, indicating that the switch has settled into the new state.

The algorithm can be expressed in pseudo-code as:

```
count = 0
while count < max_samples
      delay sample_time
      if input = required_state
            count = count + 1
      else
            count = 0
end
```

Here is the modified "toggle an LED" loop, illustrating the use of this counting debounce algorithm:

```
main_loop
        banksel db_cnt          ; select data bank for this section

        ; wait for button press
db_dn   clrf    db_cnt          ; wait until button pressed (GP3 low)
        clrf    dc1             ; debounce by counting:
dn_dly  incfsz  dc1,f           ;   delay 256x3 = 768 us.
        goto    dn_dly
        btfsc   GPIO,3          ;   if button up (GP3 high),
        goto    db_dn           ;       restart count
        incf    db_cnt,f        ;   else increment count
        movlw   .13             ;   max count = 10ms/768us = 13
        xorwf   db_cnt,w        ;   repeat until max count reached
        btfss   STATUS,Z
        goto    dn_dly

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ;   in shadow register
        movwf   GPIO            ; and write to GPIO

        ; wait for button release
db_up   clrf    db_cnt          ; wait until button released (GP3 high)
        clrf    dc1             ; debounce by counting:
up_dly  incfsz  dc1,f           ;   delay 256x3 = 768 us.
        goto    up_dly
        btfss   GPIO,3          ;   if button down (GP3 low),
        goto    db_up           ;       restart count
        incf    db_cnt,f        ;   else increment count
        movlw   .13             ;   max count = 10ms/768us = 13
        xorwf   db_cnt,w        ;   repeat until max count reached
        btfss   STATUS,Z
        goto    up_dly

        ; repeat forever
        goto    main_loop
```

There are two debounce routines here; one for the button press, the other for button release. The program first waits for a pushbutton press, debounces the press, and then toggles the LED before waiting for the pushbutton to be released, and then debouncing the release.

The only difference between the two debounce routines is the input test: '`btfsc GPIO,3`' when testing for button up, versus '`btfss GPIO,3`' to test for button down.

The above code demonstrates one method for counting up to a given value (13 in this case):

The count is zeroed at the start of the routine.

It is incremented within the loop, using the '`incf`' instruction – "**inc**rement **f**ile register". As with many other instructions, the incremented result can be written back to the register, by specifying '`,f`' as the destination, or to W, by specifying '`,w`' – but normally you would use it as shown, with '`,f`', so that the count in the register is incremented.

The baseline PICs also provide a '`decf`' instruction – "**dec**rement **f**ile register", which is similar to '`incf`', except that it performs a decrement instead of increment.

We've seen the '`xorwf`' instruction before, but not used in quite this way. The result of exclusive-oring any binary number with itself is zero. If any dissimilar binary numbers are exclusive-ored, the result will be non-zero. Thus, XOR can be used to test for equality, which is how it is being used here. First, the maximum count value is loaded into W, and then this maximum count value in W is xor'd with the loop count. If the loop counter has reached the maximum value, the result of the XOR will be zero. Note that we do not care what the result of the XOR actually is, only whether it is zero or not. And we certainly do not want to overwrite the loop counter with the result, so we specify '`,w`' as the destination of the '`xorwf`' instruction – writing the result to W, effectively discarding it.

To check whether the result of the XOR was zero (which will be true if the count has reached the maximum value), we use the '`btfss`' instruction to test the zero flag bit, Z, in the STATUS register.

Alternatively, the debounce loop could have been coded by initialising the loop counter to the maximum value at the start of the loop, and using '`decfsz`' at the end of the loop, as follows:

```
        ; wait for button press, debounce by counting:
db_dn   movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
dn_dly  incfsz  dc1,f           ; delay 256x3 = 768 us.
        goto    dn_dly
        btfsc   GPIO,3          ; if button up (GP3 high),
        goto    db_dn           ;   restart count
        decfsz  db_cnt,f        ; else repeat until max count reached
        goto    dn_dly
```

That's two instructions shorter, and at least as clear, so it's a better way to code this routine.

But in some situations it is better to count up to a given value, so it's also worth knowing how to do that, including the use of XOR to test for equality, as shown above.

Complete program

Substituting this new debounce routine into the previous "toggle an LED" loop, and wrapping it in the other pieces we need to create a full working program, we get:

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 4, example 2d                         *
;                                                                *
;   Demonstrates use of counting algorithm for debouncing        *
;                                                                *
;   Toggles LED when pushbutton is pressed then released,        *
;   using a counting algorithm to debounce switch                *
;   (alternative version using decfsz in debounce loop)          *
;                                                                *
;*****************************************************************
;                                                                *
;   Pin assignments:                                             *
```

```
;       GP1 = LED                                                           *
;       GP3 = pushbutton switch (active low)                                *
;                                                                           *
;***************************************************************************

    list        p=12F509
    #include    <p12F509.inc>


;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC


;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO

        UDATA
db_cnt  res 1                   ; debounce counter
dc1     res 1                   ; delay counter


;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR ****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration


;***** MAIN PROGRAM ****************************************************

;***** Initialisation
start
        clrf    GPIO            ; start with LED off
        clrf    sGPIO           ;   update shadow
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO            ; (GP3 is an input)

;***** Main loop
main_loop
        banksel db_cnt          ; select data bank for this section

        ; wait for button press, debounce by counting:
db_dn   movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
dn_dly  incfsz  dc1,f           ; delay 256x3 = 768 us.
        goto    dn_dly
        btfsc   GPIO,3          ; if button up (GP3 high),
        goto    db_dn           ;   restart count
        decfsz  db_cnt,f        ; else repeat until max count reached
        goto    dn_dly

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ;   in shadow register
        movwf   GPIO            ; and write to GPIO
```

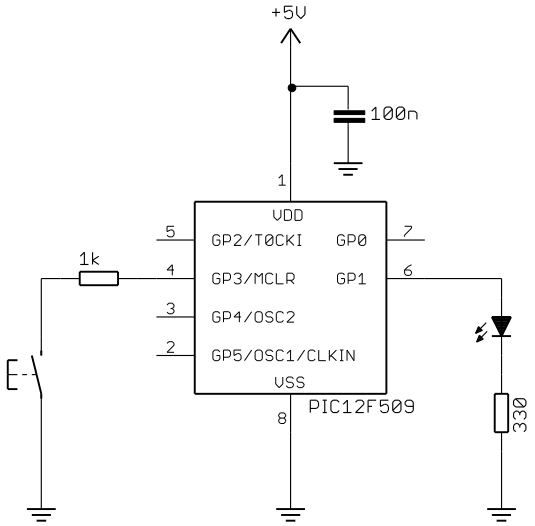
```
        ; wait for button release, debounce by counting:
db_up   movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
up_dly  incfsz  dc1,f           ; delay 256x3 = 768 us.
        goto    up_dly
        btfss   GPIO,3          ; if button down (GP3 low),
        goto    db_up           ;   restart count
        decfsz  db_cnt,f        ; else repeat until max count reached
        goto    up_dly

        ; repeat forever
        goto    main_loop


        END
```

Internal Pull-ups

The use of pull-up resistors is so common that most modern PICs make them available internally, on at least some of the pins.

By using internal pull-ups, we can do away with the external pull-up resistor, as shown in the circuit on the right.

Strictly speaking, the internal pull-ups are not simple resistors. Microchip refer to them as "weak pull-ups"; they provide a small current which is enough to hold a disconnected, or *floating*, input high, but does not strongly resist any external signal trying to drive the input low. This current is typically 250 µA on most input pins (parameter D070), or up to 30 µA on GP3, when configured with internal pull-ups enabled.



The internal weak pull-ups are controlled by the $\overline{\text{GPPU}}$ bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION	$\overline{\text{GPWU}}$	$\overline{\text{GPPU}}$	T0CS	T0SE	PSA	PS2	PS1	PS0

The OPTION register is used to control various aspects of the PIC's operation, including the timer (which will be introduced in the next lesson), as well as weak pull-ups.

Like TRIS, OPTION is not directly addressable, is write-only, and can only be written using a special instruction: '`option`' – "load **option** register".

By default (after a power-on or reset), $\overline{\text{GPPU}}$ = 1 and the internal pull-ups are disabled.
To enable internal pull-ups, clear $\overline{\text{GPPU}}$.

Assuming no other options are being set (leaving all the other bits at the default value of '1'), internal pull-ups are enabled by:

```
        movlw   b'10111111'     ; enable internal pull-ups
                ; -0------          pullups enabled (/GPPU = 0)
        option
```

Note the way that this has been commented: the line with '-0------' makes it clear that only bit 6 ($\overline{\text{GPPU}}$) is relevant to enabling or disabling the internal pull-ups, and that they are enabled by clearing this bit.

The initialisation code from the last example now becomes:

```
;***** Initialisation
start
        movlw   b'10111111'     ; enable internal pull-ups
                ; -0------          pullups enabled (/GPPU = 0)
        option
        clrf    GPIO            ; start with LED off
        clrf    sGPIO           ;   update shadow
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO            ; (GP3 is an input)
```

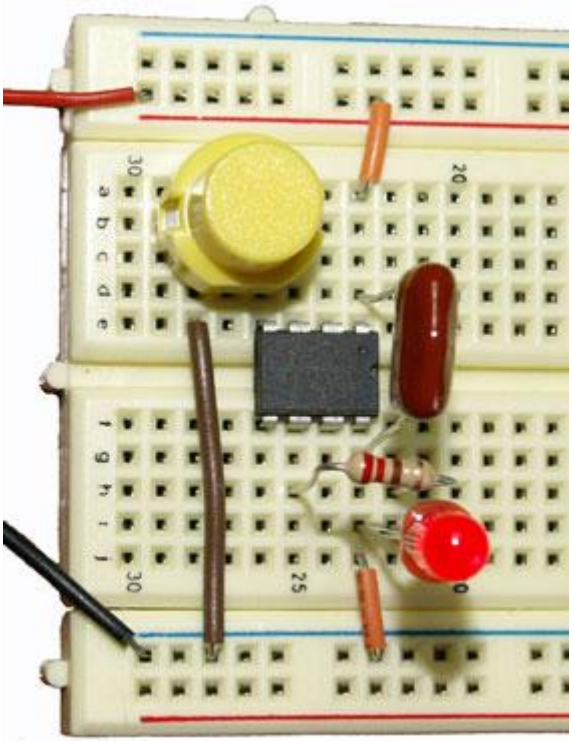
The rest of the program remains the same as before – the only difference is that the internal weak pull-ups have been enabled.

In the PIC12F509, internal pull-ups are only available on GP0, GP1 and GP3.

Internal pull-ups on the baseline PICs are not selectable by pin; they are either all on, or all off. However, if a pin is configured as an output, the internal pull-up is disabled for that pin (preventing excess current being drawn).

To test that the internal pull-ups are working, you will need to remove the 10 kΩ external pull-up resistor from the circuit we used previously.

If you have the Gooligum baseline training board, simply remove jumper JP3 and the pull-up resistor is disconnected from the pushbutton on GP3.



If you're using the Microchip Low Pin Count Demo Board, there's no easy way to disconnect the pull-up resistor from the pushbutton on that board.

One option is to build the above circuit (without a pull-up resistor), using prototyping breadboard, as illustrated on the left.

You would use the LPC demo board to program the 12F509, as usual, but then, when the PIC is programmed, remove it from the demo board and place into your breadboarded circuit.

Note that, unlike the circuit above, the prototype illustrated here does not include a current-limiting resistor between GP3 and the pushbutton. As discussed earlier, that's generally ok, but to be safe, it's good practice to include a current limiting resistor, of around 1 kΩ, between the PIC pin and the pushbutton.

But as this example illustrates, functional PIC-based circuits really can need very few external components!

When you have a version of the circuit without an external pull-up resistor, you should try testing it with the program from the previous example. You will find that the program no longer responds to the pushbutton.

However, if you modify the initialisation code, adding the instructions to enable the weak pull-ups, you will find that the program now responds correctly again – even with no external pull-up resistor!

Conclusion

You should now be able to write programs which read and respond to simple switches or other digital inputs, and be able to effectively debounce switch or other noisy inputs.

As an exercise, you could try modifying the examples in this lesson, to use a different pin as an input, instead of GP3. Hint: change the bit in GPIO being tested by the bit test instructions.

If you have the Gooligum baseline training board, you already have a pushbutton switch on GP2, making it easy to use as an input: jumper JP7 is used to connect a pull-up resistor to this switch. Note however that, because there is no internal weak pull-up available for GP2, you can't use GP2 for the final example. To test weak pull-ups, without using GP3, you would need to add a pushbutton switch to the training board's breadboard area, and connect it to GP0 (the only other pin with weak pull-ups is GP1, but you're already using that to drive the LED…)[3].

That's it for reading switches for now. There's plenty more to explore, of course, such as reading keypads and debouncing multiple switch inputs – topics to explore later.

But in the next lesson we'll look at the PIC12F509's timer module.

[3] If you use a switch with a weak pull-up on GP0 or GP1, you will have to unplug the PICkit 2 or PICkit 3 from the training board, and power the board externally while testing, because the PICkit 2 or 3 puts too much load on those pins, more than the internal weak pull-ups can overcome.