Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 2: Reading Switches

The previous lesson introduced simple digital output, by flashing an LED. That's more useful than it may seem, because, with appropriate circuit changes, the same principles can be readily adapted to turning on and off almost any electrical device.

But most systems also need to respond to user commands or sensor inputs. The simplest form of input is an on/off switch – an example of a digital input: anything that makes or breaks a single connection, or is "on" or "off", "high" or "low".

This lesson revisits the material from baseline assembler lesson 4 (which, if you are not familiar with, you should review before you start), showing how to read and respond to a simple pushbutton switch, and handle the inevitable "bouncing" of mechanical switch contacts.

The examples are re-implemented using Microchip's XC8 compiler (running in "Free mode") and CCS PCB[1], introduced in lesson 1.

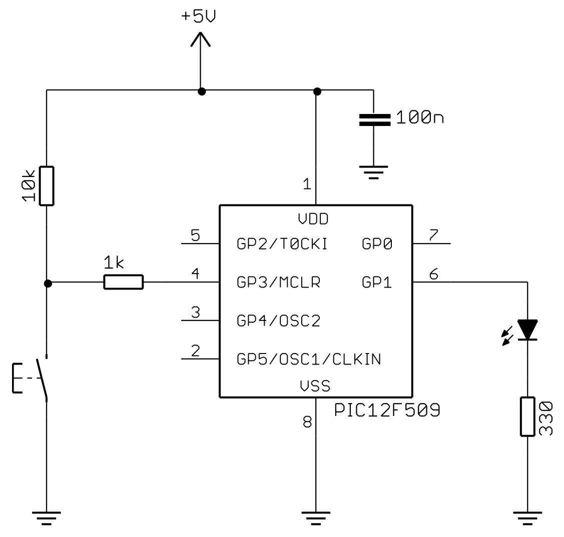
This lesson covers:

- Reading digital inputs
- Using internal pull-ups
- Switch debouncing (using a counting algorithm)

with examples for both compilers.

This tutorial assumes a working knowledge of the C language; it does **not** attempt to teach C.

Example 1: Reading Digital Inputs



Baseline assembler lesson 4 introduced digital inputs, using a pushbutton switch in the simple circuit shown on the right.

If you're using the Gooligum baseline training board, you should connect jumper JP3, to bring the 10 kΩ resistor into the circuit, and JP12 to enable the LED on GP1.

The Microchip Low Pin Count Demo Board has a pushbutton, 10 kΩ pull-up resistor and 1 kΩ isolation resistor connected to GP3, as shown. But if you are using that board, you will need to connect an LED to GP1, as described in baseline assembler lesson 1.

The 10 kΩ resistor normally holds the GP3 input high,

[1] XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

until the pushbutton is pressed, pulling the input low.

Note: if you are using a PICkit 2 programmer, you must enable '3-State on "Release from Reset"', as described in baseline assembler lesson 4, to allow the pushbutton to pull GP3 low when pressed.

As an initial example, the pushbutton input was copied to the LED output, so that the LED was on, whenever the pushbutton is pressed.

In pseudo-code, the operation is:

```
do forever
      if button down
            turn on LED
      else
            turn off LED
end
```

The assembly code we used to implement this, using a shadow register, was:

```
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
```

XC8

To copy a value from one bit to another, e.g. GP3 to GP1, using XC8, can be done as simply as:

```
    GPIObits.GP1 = GPIObits.GP3;            // copy GP3 to GP1
```

But that won't do quite what we want; given that GP3 goes low when the button is pressed, simply copying GP3 to GP1 would lead to the LED being on when the button is up, and on when it is pressed – the opposite of the required behaviour.

We can address that by inverting the logic:

```
    GPIObits.GP1 = !GPIObits.GP3;           // copy !GP3 to GP1
```

or

```
    GPIObits.GP1 = GPIObits.GP3 ? 0 : 1;    // copy !GP3 to GP1
```

This works well in practice, but to allow a valid comparison with the assembly source above, which uses a shadow register, we should not use statements which modify individual bits in GPIO. Instead we should write an entire byte to GPIO at once.

For example, we could write:

```
    if (GPIObits.GP3 == 0)    // if button pressed
        GPIO = 0b000010;      //   turn on LED
    else
        GPIO = 0;             // else turn off LED
```

However, this can be written much more concisely using C's conditional expression:

```
    GPIO = GPIObits.GP3 ? 0 : 0b000010; // if GP3 high, clear GP1, else set GP1
```

It may seem a little obscure, but this is exactly the type of situation the conditional expression is intended for.

Complete program

Here is the complete XC8 code to turn on an LED when a pushbutton is pressed:

```
/************************************************************************
*   Description:    Lesson 2, example 1                                 *
*                                                                       *
*   Demonstrates reading a switch                                       *
*                                                                       *
*   Turns on LED when pushbutton is pressed                             *
*                                                                       *
*************************************************************************
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <xc.h>


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRIS = 0b111101;            // configure GP1 (only) as an output

    // Main loop
    for (;;)
    {
        // turn on LED only if button pressed
        GPIO = GPIObits.GP3 ? 0 : 0b000010;     // if GP3 high, clear GP1
                                                //else set GP1
    }
}
```

Note that the processor configuration has been changed to disable the external $\overline{\text{MCLR}}$ reset, to allow us to use GP3 as an input.

CCS PCB

Reading a digital input pin with CCS PCB is done through the '`input()`' built-in function, which returns the state of the specified pin as a '0' or '1'.

To output a single bit, we could use the '`output_bit()`' function. For example:

```
        output_bit(GP1, ~input(GP3));
```

This would set GP1 to the inverse of the value on GP3, which is exactly what we want.

But once again, statements like this, which change only one bit in a port, are potentially subject to read-modify-write issues. We should instead use code which writes an entire byte to GPIO (or, as CCS would have it, port B) at once:

```
        output_b(input(GP3) ? 0 : 0b000010);    // if GP3 high, clear GP1
                                                //   else set GP1
```

Again, using the '?:' conditional expression makes this seem a little obscure, but this is very concise and, when you are familiar with these expressions, clear.

Complete program

Here is the complete CCS PCB code to turn on an LED when a pushbutton is pressed:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 1                                 *
*                                                                       *
*   Demonstrates reading a switch                                       *
*                                                                       *
*   Turns on LED when pushbutton is pressed                             *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <12F509.h>

#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC


/***** MAIN PROGRAM *****/
void main()
{
    // Main loop
    while (TRUE)
    {
        // turn on LED only if button pressed
        output_b(input(GP3) ? 0 : 0b000010);    // if GP3 high, clear GP1
                                                //   else set GP1
    }   // repeat forever
}
```

Note again that the processor configuration has been changed to disable the external $\overline{\text{MCLR}}$ reset, so that GP3 is available as an input.

Comparisons

Here is the resource usage summary for the "Turn on LED when pushbutton pressed" programs:

PB_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	18	13	1
XC8 (Free mode)	6	29	2
CCS PCB	5	22	4

At only 5 or 6 lines, the C source code is amazingly succinct – thanks mainly to the use of C's conditional expression ('?:').

Example 2: Switch Debouncing

Baseline lesson 4 included a discussion of the switch contact bounce problem, and various hardware and software approaches to addressing it.

The problem was illustrated by an example application, using the circuit from example 1 (above), where the LED is toggled each time the pushbutton is pressed. If the switch is not debounced, the LED toggles on every contact bounce, making it difficult to control.

The most sophisticated software debounce method presented in that lesson was a counting algorithm, where the switch is read (*sampled*) periodically (e.g. every 1 ms) and is only considered to have definitely changed state if it has been in the new state for some number of successive samples (e.g. 10), by which time it is considered to have settled.

The algorithm was 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
```

It was implemented in assembler 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
```

This code waits for the button to be pressed (GP3 being pulled low), by sampling GP3 every 768 µs and waiting until it has been low for 13 times in succession – approximately 10 ms in total.

XC8

To implement the counting debounce algorithm (above) using XC8, the pseudo-code can be translated almost directly into C:

```
        db_cnt = 0;
        while (db_cnt < 10)
        {
            __delay_ms(1);
            if (GPIObits.GP3 == 0)
                db_cnt++;
            else
                db_cnt = 0;
        }
```

where the debounce counter variable has been declared as:

```
    uint8_t     db_cnt;             // debounce counter
```

Note that, because this variable is only used locally (other functions would never need to access it), it should be declared within `main()`.

Whether you modify this code to make it shorter is largely a question of personal style. Compressed C code, using a lot of "clever tricks" can be difficult to follow.

But note that the `while` loop above is equivalent to the following `for` loop:

```
        for (db_cnt = 0; db_cnt < 10;)
        {
            __delay_ms(1);
            if (GPIObits.GP3 == 0)
                db_cnt++;
            else
                db_cnt = 0;
        }
```

That suggests restructuring the code into a traditional `for` loop, as follows:

```
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            __delay_ms(1);
            if (GPIObits.GP3 == 1)
                db_cnt = 0;
        }
```

In this case, the debounce counter is incremented every time around the loop, regardless of whether it has been reset to zero within the loop body. For that reason, the end of loop test has to be changed from '<' to '<=', so that the number of iterations remains the same.

Alternatively, the loop could be written as:

```
        for (db_cnt = 0; db_cnt < 10;)
        {
            __delay_ms(1);
            db_cnt = (GPIObits.GP3 == 0) ? db_cnt+1 : 0;
        }
```

However the previous version seems easier to understand.

Complete program

Here is the complete XC8 code to toggle an LED when a pushbutton is pressed, including the debounce routines for button-up and button-down:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 2                                 *
*                                                                       *
*   Demonstrates use of counting algorithm for debouncing               *
*                                                                       *
*   Toggles LED when pushbutton is pressed then released,               *
*   using a counting algorithm to debounce switch                       *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*       GP3 = pushbutton switch                                         *
*                                                                       *
************************************************************************/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for _delay()


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);


/***** GLOBAL VARIABLES *****/
uint8_t     sGPIO;                  // shadow copy of GPIO


/***** MAIN PROGRAM *****/
void main()
{
    uint8_t     db_cnt;             // debounce counter

    // Initialisation
    GPIO = 0;                       // start with LED off
    sGPIO = 0;                      //   update shadow
    TRIS = 0b111101;                // configure GP1 (only) as an output

    // Main loop
    for (;;)
    {
        // wait for button press, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            __delay_ms(1);          // sample every 1 ms
            if (GPIObits.GP3 == 1)  // if button up (GP3 high)
                db_cnt = 0;         //   restart count
        }                           // until button down for 10 successive reads

        // toggle LED on GP1
        sGPIO ^= 0b000010;          // toggle shadow GP1
        GPIO = sGPIO;               // write to GPIO
        // wait for button release, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            __delay_ms(1);           // sample every 1 ms
            if (GPIObits.GP3 == 0)  // if button down (GP3 low)
                db_cnt = 0;         //   restart count
        }                           // until button up for 10 successive reads
    }
}
```

CCS PCB

To adapt the debounce routine to CCS PCB, the only change needed is to use the `input()` function to read GP3, and to use the `delay_ms()` delay function:

```
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            delay_ms(1);
            if (input(GP3) == 1)
                db_cnt = 0;
        }
```

where the debounce counter variable has been declared as:

```
    unsigned int8   db_cnt;         // debounce counter
```

Once again, because this variable is only used locally, it should be declared within `main()`.

Complete program

This debounce routine fits into the “toggle an LED when a pushbutton is pressed” program, as follows:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 2                                 *
*                                                                       *
*   Demonstrates use of counting algorithm for debouncing               *
*                                                                       *
*   Toggles LED when pushbutton is pressed then released,               *
*   using a counting algorithm to debounce switch                       *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*       GP3 = pushbutton switch                                         *
*                                                                       *
************************************************************************/

#include <12F509.h>

#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5


/***** CONFIGURATION *****/
```

```
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

#use delay (clock=4000000)      // oscillator frequency for delay_ms()


/***** GLOBAL VARIABLES *****/
unsigned int8   sGPIO = 0;          // shadow copy of GPIO


/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8   db_cnt;         // debounce counter

    // Initialisation
    output_b(0);                    // start with LED off
    sGPIO = 0;                      //   update shadow

    // Main loop
    while (TRUE)
    {
        // wait for button press, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            delay_ms(1);            // sample every 1 ms
            if (input(GP3) == 1)    // if button up (GP3 high)
                db_cnt = 0;         //   restart count
        }                           // until button down for 10 successive reads

        // toggle LED on GP1
        sGPIO ^= 0b000010;          // toggle shadow GP1
        output_b(sGPIO);            // write to GPIO

        // wait for button release, debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++)
        {
            delay_ms(1);            // sample every 1 ms
            if (input(GP3) == 0)    // if button down (GP3 low)
                db_cnt = 0;         //   restart count
        }                           // until button up for 10 successive reads

    }   // repeat forever
}
```

As before, the processor configuration in both the XC8 and CCS programs has been changed to disable the external $\overline{\text{MCLR}}$ reset, so that GP3 is available as an input.

Example 3: Internal (Weak) Pull-ups

As we saw in baseline assembler lesson 4, many PICs include internal "weak pull-ups", which can be used to pull floating inputs (such as an open switch) high.

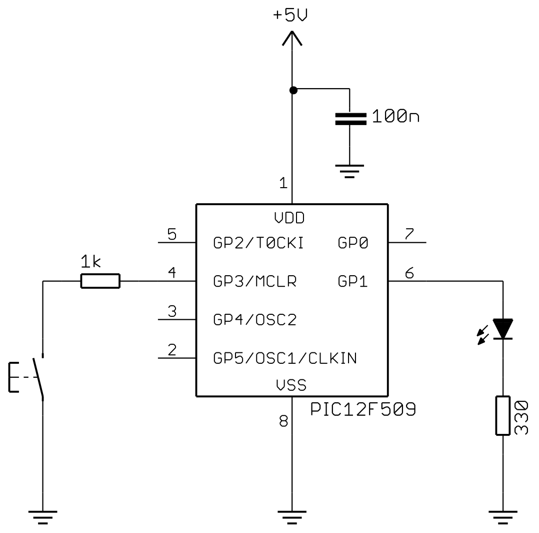
They perform the same function as external pull-up resistors, pulling an input high when a connected switch is open, but supplying only a small current; not enough to present a problem when a closed switch grounds the input.

This means that, on pins where weak pull-ups are available, it is possible to directly connect switches between an input pin and ground, as shown on the right.

To build this circuit, 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, you can simply remove jumper JP3 to disconnect the pull-up resistor from the pushbutton on GP3.

If you're using the Microchip Low Pin Count Demo Board, there's no easy way to take the pull-up resistor on that board out of the circuit. One option is to build the circuit using prototyping breadboard, as shown in baseline assembler lesson 4.



In the baseline (12-bit) PICs, such as the 12F509, the weak pull-ups are not individually selectable; they are either all on, or all off.

To enable the weak pull-ups, clear the $\overline{\text{GPPU}}$ bit in the OPTION register.

In the example assembler program from baseline lesson 4, this was done by:

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

XC8

To load the OPTION register in XC8, simply assign a value to the variable `OPTION`.

For example:

```
        OPTION = 0b10111111;    // enable internal pull-ups
                 //-0------        pullups enabled (/GPPU = 0)
```

Note that this is commented in a similar way to the assembler version, with '`-0------`' making it clear we are concerned with the value of bit 6 ($\overline{\text{GPPU}}$), and that clearing it enables pull-ups.

However, if we use the symbols for register bits, defined in the header files, we can write instead:

```
    OPTION = ~nGPPU;                // enable weak pull-ups (/GPPU = 0)
```

To enable weak pull-ups in the "toggle an LED" program from the previous example, simply add this "`OPTION =`" line into the initialisation routine.

The new initialisation code becomes:

```
    // Initialisation
    OPTION = ~nGPPU;                // enable weak pull-ups (/GPPU = 0)
    GPIO = 0;                       // start with LED off
    sGPIO = 0;                      //   update shadow
    TRIS = 0b111101;                // configure GP1 (only) as an output
```

CCS PCB

Enabling the internal weak pull-ups using CCS PCB is a little obscure, and not well documented.

The CCS compiler provides a built-in function for enabling pull-ups, ‘`PORT_x_PULLUPS()`’, but the documentation (in the online help) for this function states that it is only available for 14-bit (midrange) and 16-bit (18F) PICs. For baseline PICs, we are told:

Note: use SETUP_COUNTERS on PCB parts

However, the documentation for the built-in ‘`SETUP_COUNTERS()`’ function makes does not mention the weak pull-ups at all.

To figure this out, we need to go digging in the header files. “12F509.h” includes the following lines:

```
// Timer 0 (AKA RTCC)Functions: SETUP_COUNTERS() or SETUP_TIMER_0(),
…
#define RTCC_INTERNAL   0
…
#define RTCC_DIV_1      8
#define RTCC_DIV_2      0
…
// Constants used for SETUP_COUNTERS() are the above
// constants for the 1st param and the following for
// the 2nd param:
…
// Watch Dog Timer Functions: SETUP_WDT() or SETUP_COUNTERS() (see above)
…
#define WDT_18MS        0x8008
…
#define DISABLE_PULLUPS             0x40  // for 508 and 509 only
#define DISABLE_WAKEUP_ON_CHANGE    0x80  // for 508 and 509 only
```

And here, finally, is a clue.

As explained in baseline assembler lesson 5, the OPTION register in the baseline PICs is mainly used for selecting Timer0 options, including prescaler assignment and prescale ratio. And since the prescaler is shared with the watchdog timer (see baseline assembler lesson 7), some of these OPTION bits are also used to select watchdog timing options.

That is why the Timer0 and watchdog options are both being set by the ‘`SETUP_COUNTERS()`’ function, the use of which is being de-emphasised by CCS, in favour of more specialised built-in functions. But as well as Timer0 and watchdog options, the OPTION register on the baseline PICs also controls the weak pull-up and wake-up on change (see baseline assembler lesson 7) functions.

Therefore, for the baseline PICs, the ‘`SETUP_COUNTERS()`’ function also controls the weak pull-up and wake-up on change functions, in addition to setting timer and watchdog options. It’s just not documented very well!

It is not possible to simply enable the weak pull-ups. Instead, we must configure Timer0 (something we’ll look at in more detail in the next lesson); the pull-ups are implicitly enabled by default.

For example:

```
        setup_counters(RTCC_INTERNAL,RTCC_DIV_1);
```

To setup the timer without enabling the pull-ups, you explicitly disable them by ORing the ‘`DISABLE_PULLUPS`’ symbol with the second parameter.

For example:

```
        setup_counters(RTCC_INTERNAL,RTCC_DIV_1|DISABLE_PULLUPS);
```

To enable weak pull-ups in the "toggle an LED" program from the last example, add this 'SETUP_COUNTERS()' line to the initialisation routine.

Our new initialisation code is:

```
    // Initialisation
    setup_counters(RTCC_INTERNAL,RTCC_DIV_1);   // enable weak pull-ups
    output_b(0);                        // start with LED off
    sGPIO = 0;                          //   update shadow
```

Comparisons

Here is the resource usage summary for the "toggle an LED using weak pull-ups" programs:

Toggle_LED+WPU

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	43	36	3
XC8 (Free mode)	21	94	3
CCS PCB	20	82	6

The C programs are less than half as long as the assembler versions, but even the CCS compiler, which has optimisations enabled (unlike the XC8 compiler in "Free mode"), generates code more than twice the size of the hand-written assembler version.

Summary

This lesson has shown that basic digital input operations can readily be performed in C, using either the XC8 or CCS compiler, despite their quite different approaches.

However, we also saw, in example 3, that the use of CCS's built-in functions does not necessarily make the code easier to follow; the operation of a built-in function may not always be clear, or well-documented. Sometimes, the XC8 approach of directly accessing the PIC registers is actually easier to follow.

In the next lesson we'll see how to use these C compilers to configure and access Timer0.