Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital I/O

The Baseline PIC C Programming tutorial series demonstrated how to program baseline PICs (such as the 12F509 and 16F506) in C, to perform the tasks covered in the Baseline PIC Assembler lessons: from flashing LEDs and reading switches, through to implementing a simple light meter with smoothed output on a multiplexed 7-segment LED display. The baseline C programming series used the "free" CCS PCB compiler bundled with MPLAB[1], and Microchip's XC8 compiler (running in "Free mode"). We saw in that series that, although these C compilers were perfectly adequate for the simplest tasks, they faltered when it came to the more complex applications involving arrays, reflecting the difficulty of implementing a C compiler for the baseline PIC architecture.

This tutorial series revisits this material, along with other topics covered in the Mid-range PIC Assembler lessons, using mid-range devices such as the 12F629 and 16F684. It will become apparent that the mid-range architecture is much more suitable for C programming than the baseline architecture, especially for applications which need to access more than one bank of data memory. But we will also see that, although it is often easier to program in C, in the sense that programs are shorter and more easily expressed, assembler remains the most effective way to make the most of the limited resources on these small devices, even for mid-range PICs. Nevertheless, we will see that C is a very practical alternative for most applications.

Microchip's XC8 compiler supports all mid-range PICs, and can be operated in "Free mode" (with all optimisations disabled) for free – making it a good choice for use in these lessons.

This lesson covers basic digital I/O: flashing LEDs, responding to and debouncing switches, as covered in lessons 1 to 3 of the mid-range assembler tutorial series. You may need to refer to those lessons while working through this one.

In summary, this lesson covers:

- Introduction to the Microchip XC8 compiler
- Digital input and output
- Programmed delays
- Switch debouncing
- Using internal (weak) pull-ups

These tutorials assume a working knowledge of the C language; they **do not** attempt to teach C.

[1] CCS PCB is bundled with MPLAB 8 only, and only supports baseline PICs, so cannot be used with mid-range PICs.

Introducing the XC8 Compiler

Up until version 8.10, MPLAB was bundled with HI-TECH's "PICC-Lite" compiler, which supported all the baseline (12-bit) PICs available at that time, including those used in this tutorial series, with no restrictions. It also supports a small number of the mid-range (14-bit) PICs – although, for most of the mid-range devices it supported, PICC-Lite limited the amount of data and program memory that could be used, to provide an incentive to buy the full compiler. Microchip have since acquired HI-TECH Software, and no longer supply or support PICC-Lite. As such, PICC-Lite will not be covered in these tutorials.

XC8 supports the whole 8-bit PIC10/12/16/18 series in a single edition, with different licence keys unlocking different levels of code optimisation – "Free" (free, but no optimisation), "Standard" and "PRO" (most expensive and highest optimisation).

Microchip XC compilers are also available for the PIC24, dsPIC and PIC32 families.

XC8's "Free mode" supports all 8-bit (including baseline and mid-range) PICs, with no memory restrictions. However, in this mode, most compiler optimisation is turned off, making the generated code around twice the size of that generated by PICC-Lite.

This gives those developing for baseline and mid-range PICs easy access to a free compiler supporting a much wider range of devices than PICC-Lite does, without memory usage restrictions, albeit at the cost of much larger generated code. And XC8 will continue to be maintained, supporting new baseline and mid-range devices over time.

But if you are using Windows and developing code for a supported mid-range PIC, it is quite valid to continue to use PICC-Lite (if you are able to locate a copy – by downloading MPLAB 8.10 from the archives on www.microchip.com, for example), since it will generate much more efficient code. It can be installed alongside XC8. But to repeat – PICC-Lite won't be described in these lessons.

Regardless of which version of MPLAB you are using, the XC8 installer (for Windows, Linux or Mac) has to be downloaded separately from www.microchip.com.

When you run the XC8 installer, you will be asked to enter a license activation key. Unless you have purchased the commercial version, you should leave this blank. You can then choose whether to run the compiler in "Free mode", or activate an evaluation license. We'll be using "Free mode" in these lessons, but it's ok to use the evaluation license (for 60 days) if you choose to.

See baseline assembler lesson 1 for more detail on installing and using MPLAB 8 or MPLAB X.

Data Types

One of the problems with implementing ANSI-standard C on microcontrollers is that there is often a need to work with individual bits, while the smallest data-type included in the ANSI standard is '`char`', which is normally considered to be a single byte, or 8 bits. Another problem is the length of a standard integer ('`int`') is not defined, being implementation-dependent. Whether an '`int`' is 16 or 32 bits is an issue on larger systems, but it makes a much more significant difference to code portability on microcontrollers. Similarly, the sizes of '`float`', '`double`', and the effect of the modifiers '`short`' and '`long`' is not defined by the standard.

So various compilers use different sizes for the "standard" data types, and for microcontroller implementations it is common to add a single-bit type as well – generally specific to that compiler.

Here are the data types and sizes supported by XC8 and, for comparison, the size of the same data types in CCS PCB:

Type	XC8	CCS PCB
bit	1	-
char	8	8
short	16	1
int	16	8
short long	24	-
long	32	16
float	24 or 32	32
double	24 or 32	-

You'll see that very few of these line up; the only point of agreement is that 'char' is 8 bits!

XC8 defines a single '`bit`' type, unique to XC8.

The "standard" '`int`' type is 16 bits in XC8, but 8 bits in CCS PCB.

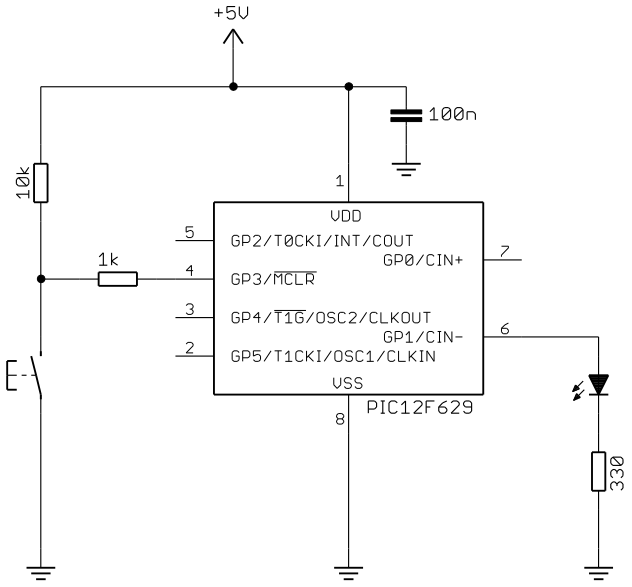
But by far the greatest difference is in the definition of '`short`': in XC8, it is a synonym for '`int`', with '`short`', '`int`' and '`short int`' all being 16-bit quantities, whereas in CCS PCB, '`short`' is a single-bit type.

Finally, note that '`double`' floating-point variables in XC8 can be either 24 or 32 bits; this is set by a compiler option. 32 bits may be a higher level of precision than is needed in most applications for small applications, so XC8's ability to work with 24-bit floating point numbers can be useful.

To make it easier to create portable code, XC8 provides the '`stdint.h`' header file, which defines the C99 standard types such as '`uint8_t`' and '`int16_t`'.

Example 1: Turning on an LED

We saw in mid-range assembler lesson 1 how to turn on a single LED, and leave it on; the (very simple) circuit is shown below:



The LED, with a current-limiting resistor, is connected to the GP1 pin of a PIC12F629.

The pushbutton acts as a reset switch, and the 10 kΩ *pull-up* resistor holds $\overline{\text{MCLR}}$ high while the switch is open[2].

If you are using the Gooligum training board, plug your PIC12F629 into the top section of the 14-pin IC socket – the section marked '12F'[3]. Close jumpers JP3 and JP12 to bring the 10 kΩ resistor into the circuit and to connect the LED to GP1, and ensure that every other jumper is disconnected.

If you have the Microchip Low Pin Count Demo Board, refer back to baseline assembler lesson 1 to see how to build this circuit.

[2] This external pull-up resistor wasn't needed in the baseline PIC examples, because the baseline PICs, and indeed most mid-range PICs, include an internal weak pull-up (see example 6, later in this lesson) on $\overline{\text{MCLR}}$ which is automatically enabled whenever the device is configured for external reset.

[3] Note that, although the PIC12F629 comes in an 8-pin package, **it will not work** in the 8-pin '10F' socket. You must install it in the '12F' section of the 14-pin socket.

To turn on the LED, we loaded the TRISIO register with 111101b, so that only GP1 is set as an output, and then set bit 1 of GPIO, setting GP1 high, turning the LED on.

At the start of the program, the PIC's configuration was set, and the OSCCAL register was loaded with the factory calibration value.

Finally, the end of the program consisted of an infinite loop ('`goto $`'), to leave the LED turned on.

Here are the key parts of the assembler code from mid-range assembler lesson 1:

```
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4 Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT


;***** RESET VECTOR ****************************************************
RESET   CODE    0x0000          ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL


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

;***** Initialisation
start
        ; configure port
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO
        movwf   TRISIO

;***** Main code
        ; turn on LED
        banksel GPIO
        bsf     GPIO,GP1        ; set GP1 high

        ; loop forever
        goto    $
```

XC8 implementation

You should start by creating a new XC8 project, as we did in baseline C lesson 1.

When you run the Project Wizard (or "New Project wizard" if you are using MPLAB X), select the PIC12F629 as the device, and XC8 as the compiler ("Microchip XC8 ToolSuite" in MPLAB 8, or "XC8" in MPLAB X).

Create an empty '.c' source file in your project folder, in the same was as in baseline C lesson 1.

Open it in the MPLAB text editor, and you're ready to start coding!

As usual, you should include a comment block at the start of each program or module. Most of the information in the comment block should be much the same, regardless of the programming language used, since it relates to what this application is, who wrote it, dependencies and the assumed environment, such as pin assignments. However, when writing in C, it is a good idea to state which compiler has been used, since, as we have seen for data types, C code for microcontrollers is not necessarily easily portable.

So we might use something like:

```
/************************************************************************
*                                                                       *
*   Filename:       MC_L1-Turn_on_LED-HTC.c                             *
*   Date:           8/6/12                                              *
*   File Version:   1.2                                                 *
*                                                                       *
*   Author:         David Meiklejohn                                    *
*   Company:        Gooligum Electronics                                *
*                                                                       *
************************************************************************
*                                                                       *
*   Architecture:   Mid-range PIC                                       *
*   Processor:      12F629                                              *
*   Compiler:       MPLAB XC8 v1.00 (Free mode)                         *
*                                                                       *
************************************************************************
*                                                                       *
*   Files required: none                                                *
*                                                                       *
************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 1                                 *
*                                                                       *
*   Turns on LED.  LED remains on until power is removed.               *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*                                                                       *
************************************************************************/
```

Note that, as we did our previous assembler code, the processor architecture and device are specified in the comment block. This is important for the XC8 compiler, as there is no way to specify the device in the code; i.e. there is no equivalent to the MPASM '`list p=`' or '`processor`' directives. Instead, the processor is specified in the IDE (MPLAB), or as a command-line option.

Most of the symbols relevant to specific processors are defined in header files. But instead of including a specific file, as we would do in assembler, it is normal to include a single "catch-all" file: "`xc.h`" (or "`htc.h`"). This file identifies the processor being used, and then calls other header files as appropriate. So our next line, which should be at the start of every XC8 program, is:

```
#include <xc.h>
```

The processor can be configured with a series of "*configuration pragmas*", such as:

```
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
#pragma config MCLRE = ON, CP = OFF, CPD = OFF, BOREN = OFF, WDTE = OFF
#pragma config PWRTE = OFF, FOSC = INTRCIO
```

Or, you can use the '`__CONFIG`' macro, in a very similar way to the `__CONFIG` directive in MPASM:

```
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);
```

The symbols are the same in both, but note that the pragma uses '=' (with optional spaces) between each setting, such as 'MCLRE', and its value, such as 'ON', while the macro uses '_' (with no spaces)[4]. To see which symbols to use for a given PIC, you need to consult the "pic_chipinfo.html" file, in the "docs" directory within the compiler install directory.

As with most C compilers, the entry point for "user" code is a function called 'main()'.

So an XC8 program will look like:

```
void main()
{
    ;   // user code goes here
}
```

Declaring main() as void isn't strictly necessary, since any value returned by main() is only relevant when the program is being run by an operating system which can act on that return value, but of course there is no operating system here. Similarly it would be more "correct" to declare main() as taking no parameters (i.e. main(void)), since there is no operating system to pass any parameters to the program. How you declare main() is really a question of personal style.

At the start of our assembler programs, we've always loaded the OSCCAL register with the factory calibration value (although it is only necessary when using the internal RC oscillator). There is no need to do so when using XC8; the default start-up code, which runs before main(), loads OSCCAL for us.

XC8 makes the PIC's special function registers, such as TRISIO, available as variables.

To load the TRISIO register with 111101b, it is simply a matter of:

```
    TRISIO = 0b111101;      // configure GP1 (only) as an output
```

Alternatively this could be expressed as:

```
    TRISIO = ~(1<<1);       // configure GP1 (only) as an output
```

Individual bits, such as GP1, can be accessed through bit-fields defined in the header files.

For example, the "pic12f629.h" file header file defines a union called GPIObits, containing a structure with bit-field members GP0, GP1, etc.

So, to set GP1 to '1', we can write:

```
    GPIObits.GP1 = 1;       // set GP1 high
```

Baseline assembler lesson 2 explained that setting or clearing a single pin in this way is a "read-modify-write" ("rmw") operation, which may lead to problems.

To avoid any potential for such rmw problems, we can load the value 000010b into GPIO (setting bit 1, and clearing all the other bits), with:

```
    GPIO = 0b000010;        // set GP1 high
```

That's not likely to be necessary in this simple example, where only one pin on the GPIO port is being used as an output, and where it is not being changed rapidly – so for simplicity we'll go ahead and set the pin individually, with "GPIObits.GP1 = 1" in this example. Nevertheless it's best to be aware of the potential for rmw issues, and later examples will illustrate ways to avoid it.

---

[4] Although the '__CONFIG' macro is now (as of XC8 v1.10) considered to be a "legacy" feature, it is still supported and we will continue to use it in these tutorials (the examples were originally written for HI-TECH C).

Finally, we need to loop forever. There are a number of C constructs that could be used for this, but one that's as good as any is:

```
    for (;;)
    {                       // loop forever
        ;
    }
```

Complete program

Here is the complete code to turn on an LED on GP1:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 1                                 *
*                                                                       *
*   Turns on LED.  LED remains on until power is removed.               *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*                                                                       *
************************************************************************/

#include <xc.h>


/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRISIO = ~(1<<1);       // configure GP1 (only) as an output
    GPIObits.GP1 = 1;       // set GP1 high


    //*** Main loop
    for (;;)
    {                       // loop forever
        ;
    }
}
```

Building the project

Whether you use MPLAB 8 or MPLAB X, the process of compiling and linking your code (making or building your project) is essentially the same as for an assembler project.

To compile the source code in MPLAB 8, select "Project → Build", press F10, or click on the "Build" toolbar button. This compiles all the source files which have changed, and links the resulting object files and any library functions, creating an output '.hex' file, which can then be programmed into the PIC as normal

(see baseline assembler lesson 1). The other Project menu item or toolbar button, "Rebuild", is equivalent to the MPASM "Build All", recompiling all your source files, regardless of whether they have changed.

Building an XC8 project in MPLAB X is exactly the same as for a MPASM assembler project: click on the "Build" or "Clean and Build" toolbar button, or select the equivalent items in the "Run" menu, to compile and link your code. When it builds without errors and you are ready to program your code into your PIC, select the "Run → Run Main Project" menu item, click on the "Make and Program Device" toolbar button, or simply press F6.

Example 2: Flashing an LED (20% duty cycle)

In mid-range assembler lesson 1, we used the same circuit as above, but made the LED flash by toggling the GP1 output. The delay was created by an in-line busy-wait loop. Mid-range assembler lesson 2 showed how to move the delay loop into a subroutine, and to generalise it, so that the delay is passed as a parameter to the routine, in W. This was demonstrated by a program which flashed the LED at 1 Hz, with a duty cycle of 20%, by turning it on for 200 ms and then off for 800 ms, before repeating.

Here is the main loop from the assembler code from that lesson:

```
main_loop
        ; turn on LED
        banksel GPIO
        movlw   1<<GP1          ; set GP1
        movwf   GPIO
        ; delay 0.2 s
        movlw   .20             ; delay 20 x 10 ms = 200 ms
        call    delay10
        ; turn off LED
        clrf    GPIO            ; (clearing GPIO clears GP1)
        ; delay 0.8 s
        movlw   .80             ; delay 80 x 10ms = 800ms
        call    delay10

        ; repeat forever
        goto    main_loop
```

XC8 implementation

We've seen how to turn on the LED on GP1, with:

```
        GPIObits.GP1 = 1;       // set GP1
```

or

```
        GPIO = 0b000010;        // set GP1 (bit 1 of GPIO)
```

And of course, to turn the LED off, it is simply:

```
        GPIObits.GP1 = 0;       // clear GP1
```

or

```
        GPIO = 0;               // (clearing GPIO clears GP1)
```

These statements can easily be placed within an endless loop, to repeatedly turn the LED on and off. All we need to add is a delay.

XC8 provides a built-in function, '`_delay(n)`', which creates a delay '`n`' instruction clock cycles long. The maximum possible delay depends on which PIC you are using, but it is a little over 50,000,000 cycles. With a 4 MHz processor clock, corresponding to a 1 MHz instruction clock, that's a maximum delay of a little over 50 seconds.

The compiler also provides two macros: '`__delay_us()`' and '`__delay_ms()`', which use the '`_delay(n)`' function create delays specified in µs and ms respectively. To do so, they reference the symbol "`_XTAL_FREQ`", which you must define as the processor oscillator frequency, in Hertz.

Since our PIC12F629 is running at 4 MHz, we have:

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

Then, to generate a 200 ms delay, we can write:

```
        __delay_ms(200);        // stay on for 200 ms
```

Complete program

Putting these delay macros into the main loop, we have:

```
/************************************************************************
*   Description:    Lesson 1, example 2                                 *
*                                                                       *
*   Flashes an LED at approx 1 Hz, with 20% duty cycle                  *
*   LED continues to flash until power is removed.                      *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       GP1 = flashing LED                                              *
*                                                                       *
************************************************************************/

#include <xc.h>

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


//***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRISIO = 0b111101;          // configure GP1 (only) as an output


    //*** Main loop
    for (;;)
    {
        GPIO = 0b000010;        // turn on LED on GP1 (bit 1)

        __delay_ms(200);        // stay on for 200 ms

        GPIO = 0;               // turn off LED (clearing GPIO clears GP1)

        __delay_ms(800);        // stay off for 800 ms

    }                           // repeat forever
}
```

Example 3: Flashing an LED (50% duty cycle)

The LED flashing example in mid-range assembler lesson 1 used an XOR operation to flip the GP1 bit every 500 ms, creating a 1 Hz flash with a 50% duty cycle.

The read-modify-write problem revisited

As discussed in that lesson, any operation which reads an output (or part-output) port, modifies the value read, and then writes it back, can lead to unexpected results. This is because, when a port is read, it is the value at the pins that is read, not necessarily the value that was written to the output latches. And that's a problem if, for example, you have written a '1' to an output pin, which, because it is being externally loaded (or, more usually, it hasn't finished going high yet, because of a capacitive load on the pin), it reads back as a '0'. When the operation completes, that output bit would be written back as a '0', and the output pin sent low instead of high – not what it is supposed to be.

This can happen with any instruction which reads the current value of a register when updating it. That includes logic operations such as XOR, but also arithmetic operations (add, subtract), rotate instructions, and increment and decrement operations. And crucially, it also includes the bit set and clear instructions.

You may think that the instruction '`bsf GPIO,1`' will only affect GP1, but in fact that instruction reads the whole of GPIO, sets bit 1, and then writes the whole of GPIO back again.

Consider the sequence:

```
bsf     GPIO,1
bsf     GPIO,2
```

Assuming that GP1 and GP2 are both initially low, the first instruction will attempt to raise the GP1 pin high. However, the first instruction writes to GPIO at the end of the instruction cycle, while the second instruction reads the port pins toward the start of the following instruction cycle. That doesn't leave much time for GP1 to be pulled high, against whatever capacitance is loading the pin. If it hasn't gone high enough by the time the second '`bsf`' instruction reads the pins, it will read as a '0', and it will then be written back as a '0' when the second '`bsf`' writes to GPIO. The potential result is that, instead of both GP1 and GP2 being set high, as you would expect, it is possible that only GP2 will be set high, while the GP1 pin remains low, and the GP1 bit holds a '0'.

This problem is sometimes avoided by placing '`nop`' instructions between successive read-modify-write operations, but as we've seen in the assembler lessons, a more robust solution is to use a shadow register.

So why revisit this topic, in a lesson on C programming?

When you use a statement like '`GPIObits.GP1 = 1`' in XC8, the compiler translates those statements into corresponding bit set or clear instructions, which may lead to read-modify-write problems.

> *Note: Any C statements which directly modify individual port bits may be subject to read-modify-write considerations.*

There was no problem with using these types of statements in the examples above, where only a single pin is being used and there are lengthy delays between changes.

But you should be aware that a sequence such as:

```
GPIObits.GP1 = 1;
GPIObits.GP2 = 1;
```

may in fact result in GP1 being cleared and only GP2 being set high.

To avoid such problems, shadow variables can be used in C programs, in much the same way that they are used in assembly language.

Here is the main code from the program presented in mid-range assembler lesson 2:

```
        ; configure port
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO
        movwf   TRISIO

        clrf    sGPIO           ; start with shadow GPIO zeroed

;***** Main loop
main_loop
        ; toggle LED
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   1<<GP1          ; toggle bit corresponding to GP1
        movwf   sGPIO           ;   in shadow register
        banksel GPIO            ; and write to GPIO
        movwf   GPIO


        ; delay 500 ms -> 1 Hz flashing at 50% duty cycle
        movlw   .50
        pagesel delay10         ; delay 50 x 10 ms = 500 ms
        call    delay10

        ; repeat forever
        pagesel main_loop
        goto    main_loop
```

XC8 implementation

To toggle GP1, you could use the statement:

```
    GPIObits.GP1 = ~GPIObits.GP1;
```

or:

```
    GPIObits.GP1 = !GPIObits.GP1;
```

This statement is also supported:

```
    GPIObits.GP1 = GPIObits.GP1 ? 0 : 1;
```

It works because single-bit bit-fields, such as `GP1`, hold either a '0' or '1', representing 'false' or 'true' respectively, and so can be used directly in a conditional expression like this.

However, since these statements modify individual bits in GPIO, to avoid potential read-modify-write issues we'll instead use a shadow variable, which can be declared and initialised with:

```
uint8_t     sGPIO = 0;          // shadow copy of GPIO
```

This makes it clear that the variable is an unsigned, eight-bit integer. We could have declared this as an '`unsigned char`', or simply '`char`' (because '`char`' is unsigned by default), but you can make your code clearer and more portable by using the C99 standard integer types defined in the "`stdint.h`" header file.

To define these standard integer types, add this line toward the start of your program:

```
#include <stdint.h>
```

The variable declaration could be placed within the `main()` function, which is what you should do for any variable that is only accessed within `main()`. However, a variable such as a shadow register may need to be accessed by other functions. For example, it's quite common to place all of your initialisation code into a function called `init()`, which might initialise the shadow register variables as well as the ports, and your `main()` code may also need to access them. It is often best to define such variables as global (or "external") variables toward the start of your code, before any functions, so that they can be accessed throughout your program.

But remember that, to make your code more maintainable and to minimise data memory use, you should declare any variable which is only used by one function, as a local variable within that function.

We'll see examples of that later, but in this example we'll define `sGPIO` as a global variable.

Flipping the shadow copy of GP1 and updating GPIO, can then be done by:

```
        sGPIO ^= 1<<1;          // flip shadow bit corresponding to GP1
        GPIO = sGPIO;           // write to GPIO
```

Complete program

Here is how the XC8 code to flash a LED on GP1, with a 50% duty cycle, fits together:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 3                                 *
*                                                                       *
*   Flashes a LED at approx 1 Hz.                                       *
*   LED continues to flash until power is removed.                      *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = flashing LED                                              *
*                                                                       *
************************************************************************/

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

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


//***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);


/***** GLOBAL VARIABLES *****/
uint8_t     sGPIO = 0;          // shadow copy of GPIO


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRISIO = ~(1<<1);           // configure GP1 (only) as an output
```

```
    //*** Main loop
    for (;;)
    {
        // toggle LED on GP1
        sGPIO ^= 1<<1;          // flip shadow bit corresponding to GP1
        GPIO = sGPIO;           // write to GPIO

        // delay 500 ms
        __delay_ms(500);

    }   // repeat forever
}
```

Comparisons

Although this is a very small, simple application, it is instructive to compare the source code size (lines of code[5]) and resource utilisation (program and data memory usage) for this C version with the assembler version of this example from mid-range lesson 2.

Source code length is a rough indication of how difficult or time-consuming a program is to write. We expect that C code is easier and quicker to write than assembly language, but that a C compiler (especially one with optimisation disabled, as XC8 is, when running in "Free mode") will produce code that is bigger or uses memory less efficiently than hand-crafted assembly. But is this true?

It's also interesting to see whether the delay function provided by the C compiler generates accurately-timed delays, and how its accuracy compares with our assembler version.

MPLAB correctly reports the memory usage for assembler and XC8 projects, and the MPLAB simulator[6] can be used to accurately measure the time between LED flashes – ideally it would be exactly 1.000000 seconds, and the difference from that gives us the overall timing error.

Here is the resource usage and accuracy summary for the "Flash an LED at 50% duty cycle" programs. The resource usage and accuracy of the baseline (12F509) versions of this example, from baseline assembler lesson 3 and baseline C lesson 2, is also given for comparison:

Flash_LED-50p

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)		Delay accuracy (timing error)	
	12F629	12F509	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	27	28	29	34	4	4	0.15%	0.15%
XC8 (Free mode)	11	11	35	36	4	4	0.0024%	0.0024%

The assembler version called the delay routine as an external module, so it's quite comparable with the C programs which make use of built-in delay functions. Nevertheless, the assembly language source code is around three times as long as the C version! This illustrates how much more compact C code can be.

As for C being less efficient – the XC8 version is only a little larger than the assembler version, despite having most compiler optimisations disabled in "Free mode". This is largely because the built-in delay code (which, as we can see is highly accurate!) is optimised, but it does show that C is not necessarily inherently inefficient.

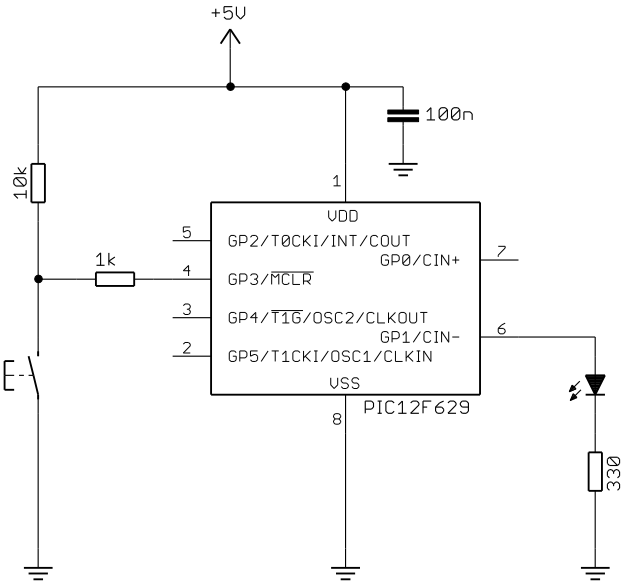
---

[5] ignoring whitespace, comments, and "unnecessary" lines such as the redefinition of pin names in the CCS C examples

[6] a topic for a future tutorial?

Example 4: Reading Digital Inputs

Mid-range assembler lesson 3 introduced digital inputs, using a pushbutton switch in the simple circuit shown on the right.



It's the same circuit as in the earlier examples (you can leave your board configured the same way as before), but now we'll use the pushbutton to drive a digital input (GP3), instead of as a reset switch.

The 10 kΩ resistor normally holds the GP3 input high, 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 mid-range assembler lesson 3, 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 assembler code we used to implement this, using a shadow register, was:

```
        ; configure port
        movlw   ~(1<<GP1)       ; configure GP1 (only) as an output
        banksel TRISIO          ; (GP3 is an input)
        movwf   TRISIO

;***** Main loop
main_loop
        ; turn on LED only if button pressed
        clrf    sGPIO           ; assume button up -> LED off
        banksel GPIO
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     sGPIO,GP1       ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    main_loop       ; repeat forever
```

XC8 implementation

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 a LED when a pushbutton is pressed:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 4                                 *
*                                                                       *
*   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 brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);
```

```
/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    TRISIO = ~(1<<1);           // 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, 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)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	21	18	16	13	1	1
XC8 (Free mode)	6	6	29	29	2	2

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

Example 5: Switch Debouncing

Mid-range lesson 3 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 4 (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,GP3        ; 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 implementation

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 1, example 5                                 *
*                                                                       *
*   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 (active low)                            *
*                                                                       *
************************************************************************/

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

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


/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);


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

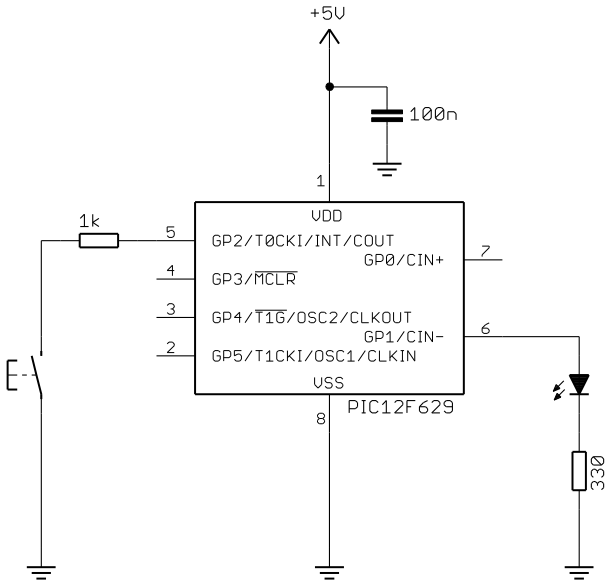
```
/***** MAIN PROGRAM *****/
void main()
{
    uint8_t     db_cnt;             // debounce counter

    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with LED off
    sGPIO = 0;                      //   update shadow
    TRISIO = ~(1<<1);               // 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

    }   // repeat forever
}
```

Example 6: Internal (Weak) Pull-ups

As discussed in mid-range lesson 3, 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 only supplying a small current; not enough to present a problem when a closed switch grounds the input.

This means that, on pins with a weak pull-up, it is possible to directly connect switches between an input pin and ground, as shown on the right.

Unfortunately, there is no internal pull-up on the 12F629’s GP3 pin, so to demonstrate their use we need to use a different input pin, which is why the switch is connected to GP2 here.



The Gooligum training board already has a pushbutton switch connected to GP2 as shown, but you should ensure that jumper JP7 is not closed, so that there is no external pull-up in place.

If you are using the Microchip demo board, you will need to supply your own pushbutton and connect it between GP2 (pin 9 of the 14-pin header) and ground (pin 14 on the header).

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

In the example assembler program from mid-range lesson 3, this was done by:

```
bcf     OPTION_REG,NOT_GPPU     ; enable weak pull-ups (global)
```

Unlike the baseline PICs, the weak pull-ups on mid-range devices individually selectable; to enable a pull-up, the corresponding bit in the WPU register must be set.

By default (after a power-on reset) every bit in WPU is set, so to enable a pull-up on only a single pin, the remaining bits in WPU must be cleared.

This was done, in the assembler example in mid-range lesson 3, by:

```
movlw   1<<GP2          ; select pull-up on GP2 only
movwf   WPU
```

XC8 implementation

The XC8 compiler makes the individual bits in the OPTION register available as bit-fields, so to clear $\overline{\text{GPPU}}$, without affecting any of the other OPTION bits, we can simply write:

```
OPTION_REGbits.nGPPU = 0;       // enable weak pull-ups (global)
```

Note again you should look at the header file for your PIC ("`pic12f629.h`" in this case) to check the name of the bit-field associated with the register bit you wish to access. It is not necessarily the same as the symbol defined in the assembler include file – for example, the XC8 include file defines the bit-field '`nGPPU`', while the MPASM include file defines the symbol '`NOT_GPPU`', both referring to the $\overline{\text{GPPU}}$ bit.

Similarly, the WPU register is accessible through a variable named '`WPU`', so to enable the pull-up on GP2 (and disable all the other pull-ups) can write:

```
WPU = 1<<2;                     // select pull-up on GP2 only
```

With these additions, the initialisation code then becomes:

```
// configure port
OPTION_REGbits.nGPPU = 0;       // enable weak pull-ups (global)
WPU = 1<<2;                     // enable pull-up on GP2 only
GPIO = 0;                       // start with LED off
sGPIO = 0;                      //   update shadow
TRISIO = ~(1<<1);               // configure GP1 (only) as an output
```

The rest of the program is then the same as before (example 5, above), except for testing GP2 instead of GP3.

Comparisons

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

Toggle_LED+WPU

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	47	43	40	36	3	3
XC8 (Free mode)	23	22	97	94	3	3

Although the difference is less pronounced than in the simpler, earlier examples, the C source code continues to be less than half the length of the assembly version, while the (unoptimised) code generated by the XC8 compiler continues to be more than twice the size of the hand-written assembly language version.

Summary

Overall, we have seen that basic digital I/O operations can be expressed succinctly using C, leading to significantly shorter source code than assembly language, as illustrated by the comparisons we have done in this lesson: the C code is typically only half, or less, as long as the corresponding assembler source code.

It could be argued that, because the C code is significantly shorter than corresponding assembler code, with the program structure more readily apparent, C programs are more easily understood, faster to write, and simpler to debug, than assembler.

So why use assembler? One argument is that, because assembler is closer to the hardware, the developer benefits from having a greater understanding of exactly what the hardware is doing; there are no unexpected or undocumented side effects, no opportunities to be bitten by bugs in built-in or library functions. But that argument applies more to other compilers than it does to XC8, which exposes all the PIC's registers as variables, and the programmer has to directly modify the register contents in much the same way as would be done in assembler.

However, C compilers usually generate code which occupies more program memory and uses more data memory than for corresponding hand-written assembler programs[7].

Since the C compilers consistently use more resources than assembler (for equivalent programs), there comes a point, as programs grow, that a C program will not fit into a particular PIC, while the same program would have fit if it had been written in assembler. In that case, the choice is to write in assembler, or use a more expensive PIC. For a one-off project, a more expensive chip probably makes sense, whereas for volume production, using resources efficiently by writing in assembler may be the right choice.

In the next lesson we'll see how to use XC8 to configure and access Timer0.

[7] It's a little unfair to draw this conclusion, based on a compiler (XC8) running with optimisation disabled (in "Free mode"). However, this statement remains true when the PICC-Lite compiler (sadly, no longer available), with full optimisation enabled, is used to implement these examples.