Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital Output

Although assembly language is commonly used to programming small microcontrollers, it is less appropriate for complex applications on larger MCUs; it can become unwieldy and difficult to maintain as programs grow longer. A number of higher-level languages are used in embedded systems development, including BASIC, Forth and even Pascal. But the most commonly used "high level" language is C.

C is often considered to be inappropriate for very small MCUs, such as the baseline PICs we have examined in the baseline assembler tutorial series, because they have limited resources and their architecture is not well suited to C code. However, as this tutorial series will demonstrate, it is quite possible to use C for simple programs on baseline PICs – although it is true that C may not be able to make the most efficient use of the limited memory on these small devices, as we will see in later lessons.

This lesson introduces the "free" Custom Computer Services (CCS) compiler bundled with MPLAB[1], and Microchip's XC8 compiler (running in "Free mode"), both of which fully support all current baseline PICs[2]. As we'll see, the XC8 and CCS compilers take quite different approaches to many implementation tasks. Most other PIC C compilers take a similar approach to one or the other, or fall somewhere in between, making these compilers a good choice for an introduction to programming PICs in C.

This lesson covers basic digital output, through that standby of introductory microcontroller courses: flashing LEDs – although the concepts can be applied to anything which can be controlled by a high/low, on/off signal, including power MOSFET switches and relays.

It is assumed that you are already familiar with the material covered in those baseline assembler lessons 1 to 3. If not, you should review those lessons while working through this one. Specifically, this lesson **does not** provide a detailed overview of the baseline PIC architecture, installing and using MPLAB or programmers such as the PICkit 2. Instead, this lesson explains how to create C projects in MPLAB, and how to implement the examples from the assembler lessons, in C.

In summary, this lesson covers:

- Introduction to the Microchip XC8 and CCS PCB compilers
- Using MPLAB 8 and MPLAB X to create C projects
- Simple control of digital output pins
- Programmed delays

with examples for both compilers.

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

---

[1] CCS PCB is bundled with MPLAB 8 only.

[2] at the time of writing (September 2012)

Introducing XC8 and CCS PCB

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.

Microchip have developed the former HI-TECH C compiler into their own "MPLAB XC8" compiler. It is available for download at www.microchip.com.

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, 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 baseline 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, while allowing all the (limited) memory on your baseline PIC to be used. It can be installed alongside XC8. But to repeat – PICC-Lite won't be described in these lessons.

MPLAB 8 includes a free copy of CCS's PCB C compiler for Windows, which supports most baseline PICs, including those used in these tutorials. Although it's now a little date (at the time of writing, the version bundled with MPLAB was 4.073, while the latest commercially available version was 4.135), it remains useful and so is used in these tutorials.

If you are using MPLAB 8, you should select the CCS compiler as an option when installing MPLAB, to ensure that the integration with the MPLAB IDE will be done correctly.

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.

Custom Computer Services (CCS) "PCB"

CCS (www.ccsinfo.com) specialises in PIC development, offering hardware development platforms, as well as a range of C compilers supporting (as of February 2012) almost all the PIC processors from the baseline 10Fs through to the 16-bit PIC24Fs and dsPICs. They also offer an IDE, including a "C-aware" editor, and debugger/simulator.

"PCB" is the command-line compiler supporting the baseline (12-bit) PICs.

A separate command-line compiler, called "PCM", supports the mid-range (14-bit) PICs, including most PIC16s. Similarly, "PCH" supports the 16-bit instruction-width, 8-bit data width PIC18 series, while "PCD" supports the 24-bit instruction-width, 16-bit data width PIC24 and dsPIC series. These command-line compilers are available for both Windows and Linux. A plug-in allows these compilers to be integrated into both MPLAB 8 and MPLAB X[3].

[3] Note that the free version of the CCS PCB compiler, supplied with MPLAB 8, **will not** work with MPLAB X.

CCS also offer a Windows IDE, called "PCW", which incorporates the PCB and PCM compilers. "PCWH" extends this to include PCH for 18F support, while "PCWHD" supports the full suite of PICs. A lower-cost IDE, called "PCDIDE" includes only the PCD compiler.

The CCS compilers and IDEs are relatively inexpensive: as of February 2012, the advertised costs range from US$50 for PCB, through US$150 for PCM, US$350 for PCW, to US$600 for the full PCWHD suite.

As we'll see, the CCS approach is to provide a large number of PIC-specific inbuilt functions, such as `read_adc()`, which make it easy to access or use PIC features, without having to be aware of and specify all the registers and bits involved. That means that the CCS compilers can be used without needing a deep understanding of the underlying hardware, which can be a two-edged sword; it is easier to get started and less-error prone (in that the compiler can be expected to set up the registers correctly), but can be less flexible and more difficult to debug when something is wrong (especially if the bug is in the compiler's implementation, and not your code).

Microchip "MPLAB XC8"

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.

The XC8 compiler is more expensive than those from CCS: as of August 2012, the advertised costs include US$495 for the "Standard" mode, and US$995 for this compiler in "PRO" mode.

We'll see that XC8 exposes the PIC's registers as variables, to be accessed "directly" by the developer, in much the same way that they would be in assembler, instead of via built-in functions. This means that, to effectively use the XC8 compiler, you need a strong understanding of the underlying PIC hardware, equivalent to that needed for programming in assembler.

These differing approaches are highlighted in the examples below. Instead of trying to force either compiler into a particular style, the examples for each compiler are written in a style similar in "spirit" to the sample code provided with each. Although it is possible to map registers into variables in the CCS compilers, the examples in these tutorials use the CCS built-in functions where that seems reasonable, since that is how that compiler was intended to be used. However, identical comments are used where reasonable, to highlight the correspondence between both C compilers and the original assembler version of each example.

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 different compilers use various 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 CCS PCB and XC8:

Type	XC8	CCS PCB
bit	1	-
int1	-	1
char	8	8
int8	-	8
short	16	1
int	16	8
int16	-	16
short long	24	-
long	32	16
int32	-	32
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.

CCS PCB defines '`int1`', '`int8`', '`int16`' and '`int32`' types, which make it easy to be explicit about the size of a data element (such as a variable).

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

But by far the greatest difference is in the definition of '`short`': in XC8, it is a synonym for '`int`' and is a 16-bit type, whereas in CCS PCB, '`short`' is a single-bit type, the same as an '`int1`'. That could be very confusing when porting code from CCS PCB to another compiler, so for clarity it is probably best to use '`int1`' when defining single-bit variables.

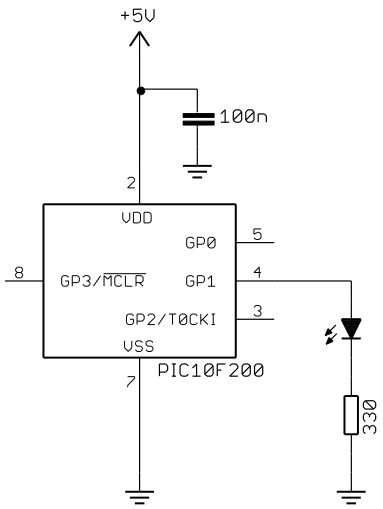
XC8 also offers the non-standard 24-bit '`short long`' type. And note that floating-point variables in XC8 can be either 24 or 32 bits; this is set by a compiler option. The only floating-point representation available in CCS PCB is 32-bit, which may be a higher level of precision than is needed in most 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`'.

Unfortunately, CCS PCB does not come with a version of '`stdint.h`', although the size of CCS types such as '`int8`', and '`int16`' is clear.

Example 1: Turning on an LED

In baseline assembler lesson 1 we saw how to turn on a single LED, and leave it on; the (very simple) circuit, using a PIC10F200, is shown below:



This circuit is intended for use with the Gooligum baseline training board, where you can simply plug the 10F200 into the '10F' socket, and connect jumper JP12.

If you have the Microchip Low Pin Count Demo board, which does not support PIC10F devices, you will have to substitute a 12F508 or 12F509 and connect an LED to pin GP1: see baseline assembler lesson 1 for details.

To turn on the LED on GP1, we must clear bit 1 of the TRIS, configuring GP1 as an output, and then set bit 1 of GPIO, setting GP1 high, turning the LED on.

At the start of the assembly language program, the PIC was configured, and the internal RC oscillator was calibrated, by loading the factory calibration value into the OSCCAL register.

Finally, the end of the program consisted of an infinite loop, to leave the LED turned on.

Here are the key parts of the 10F200 version of the assembler code from baseline lesson 1:

```
                ; ext reset, no code protect, no watchdog
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF

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

        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO
        movlw   b'000010'       ; set GP1 high
        movwf   GPIO

        goto    $               ; loop forever
```

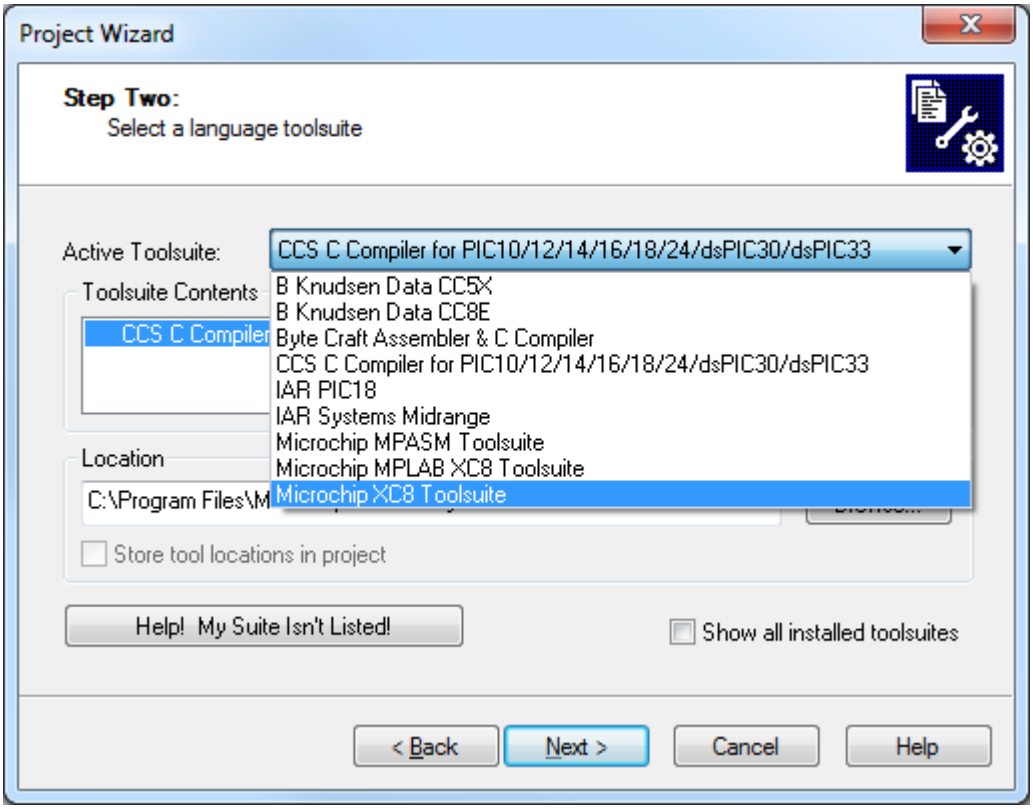
XC8

XC8 projects in MPLAB are created in a similar way to assembler projects, but as we saw in baseline lesson 1, the details depend on which version of the MPLAB IDE you are using.

MPLAB 8.xx

You should use the project wizard to create a new project, as before.

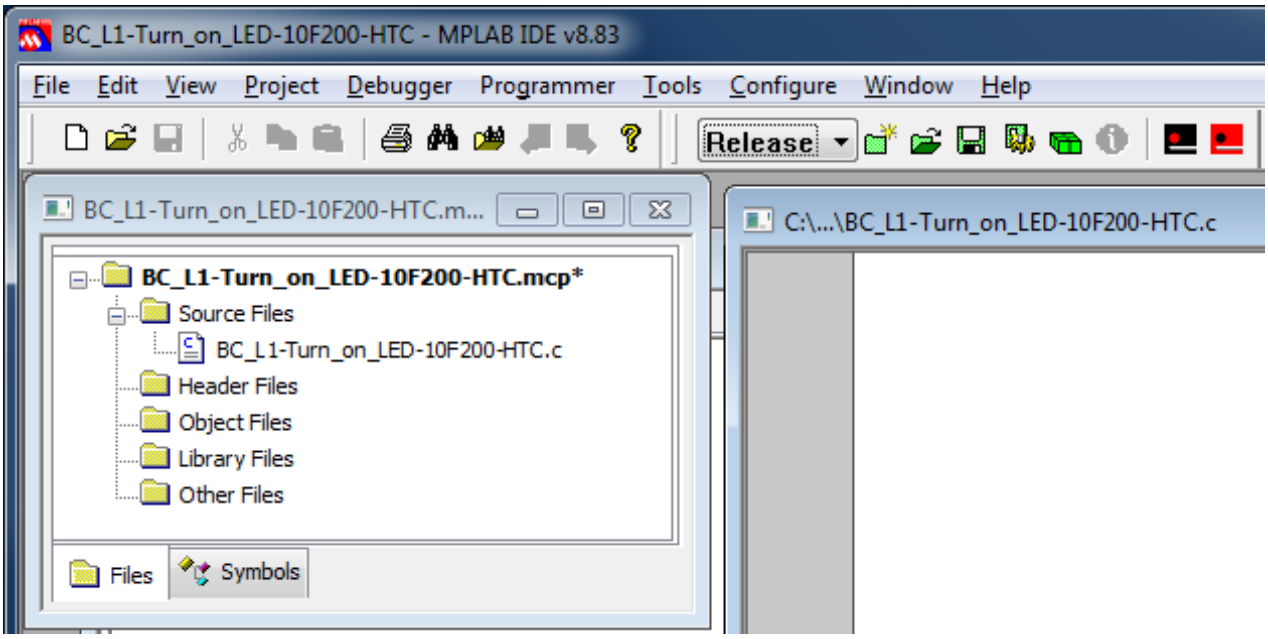
To specify that this is an XC8 project, select the "Microchip XC8 ToolSuite" when you reach "Step Two: Select a language toolsuite":



When completing the wizard, note that there is no need to add existing files to your project, unless you wish to make use of some existing code, perhaps from a previous project. There is no equivalent to the MPASM "template files" we saw in baseline lesson 1.

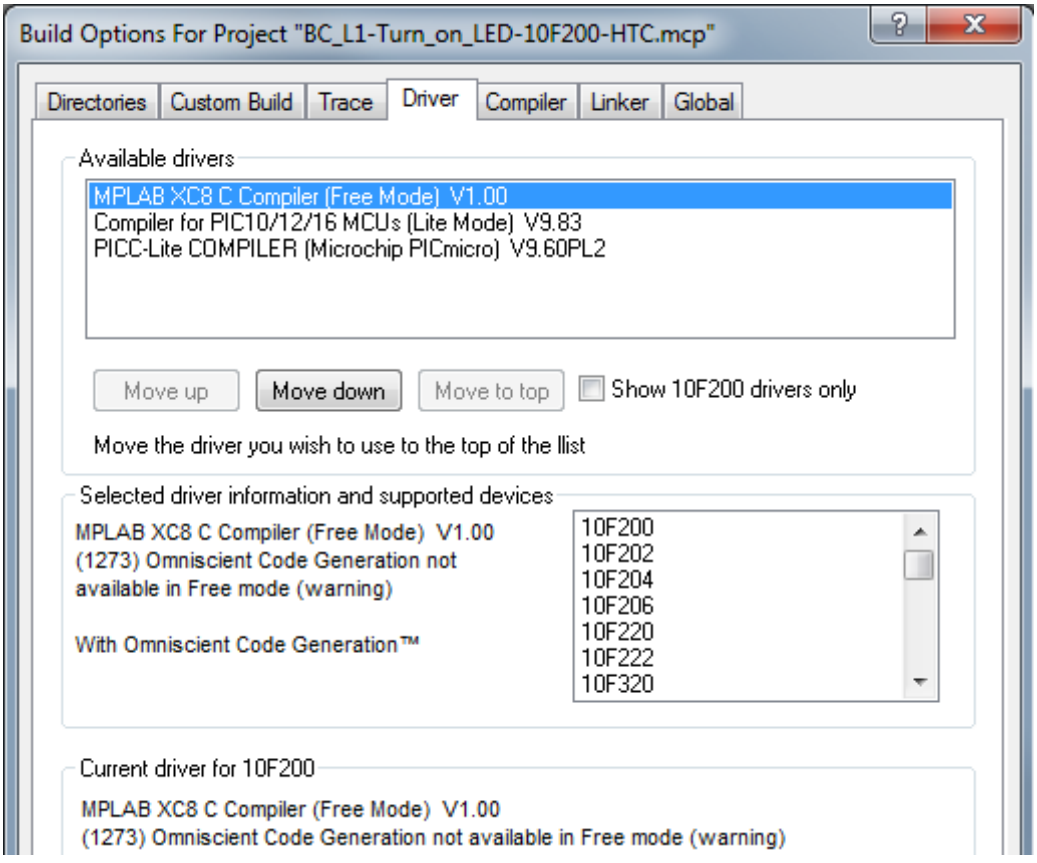
After finishing the project wizard, you can use the "File → Add New File to Project…" menu item to create a '.c' source file in your project folder.

It should appear under "Source Files" in the project window, as usual:



If you have installed more than one HI-TECH or XC8 compiler, you need to tell the XC8 toolsuite which compiler, or *driver*, to use.

Open the project build options window (Project → Build Options… → Project) then select the "Driver" tab, as shown on the right.



To select the compiler you wish to use, move it to the top of the list of available drivers, by using the "Move up" button). The "Current driver"

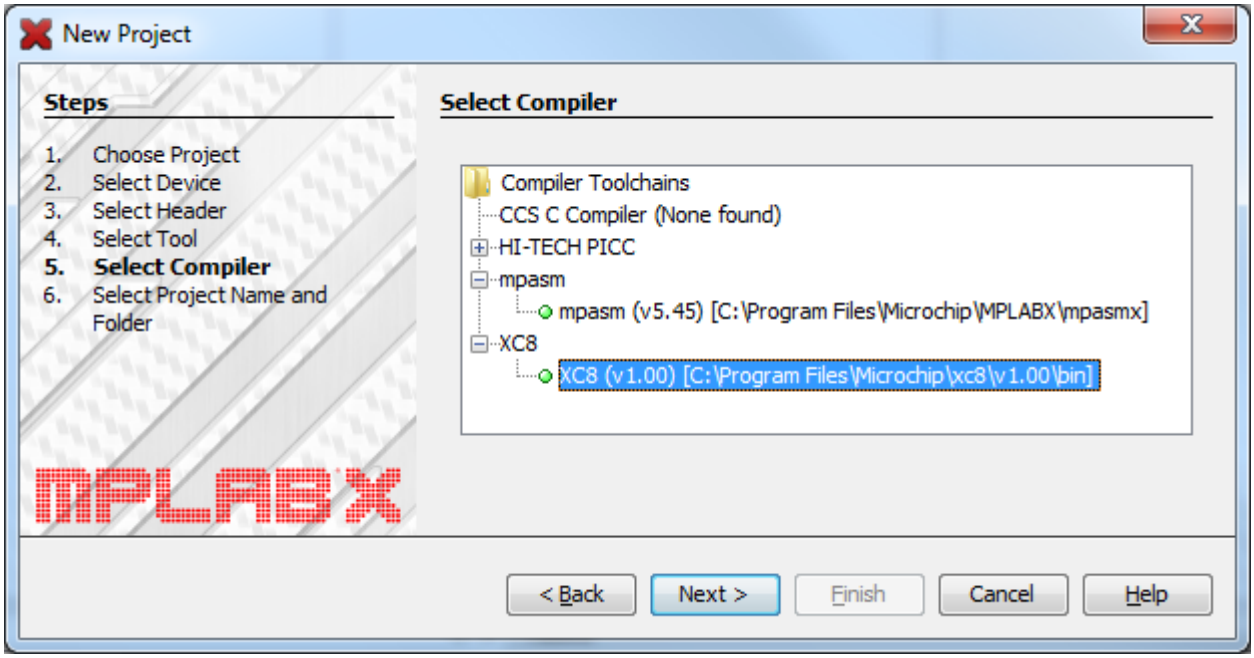
panel shows which compiler will be used for your device; since not every compiler supports every PIC, the toolsuite selects the first driver in the list which supports the device you are compiling for. When you have selected the compiler you wish to use, click "OK" to continue.

You are now ready to start coding!

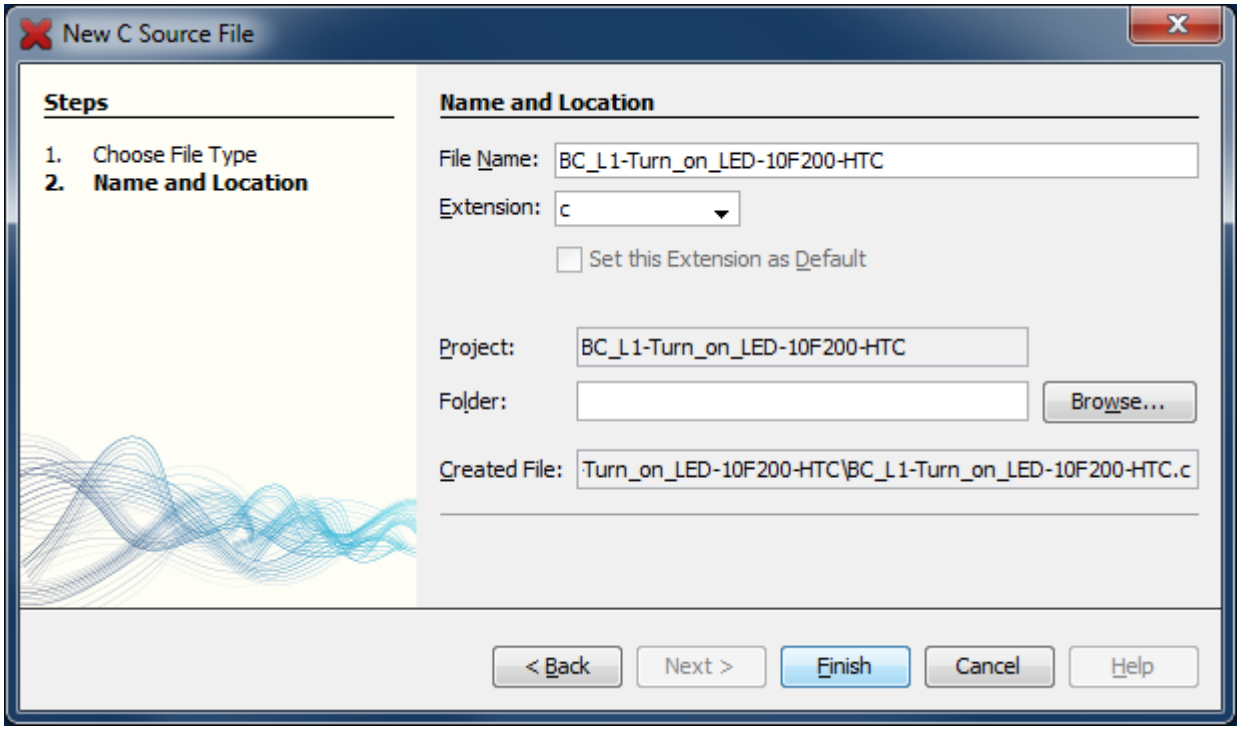
MPLAB X

You should use the New Project wizard to create your new project, as usual.

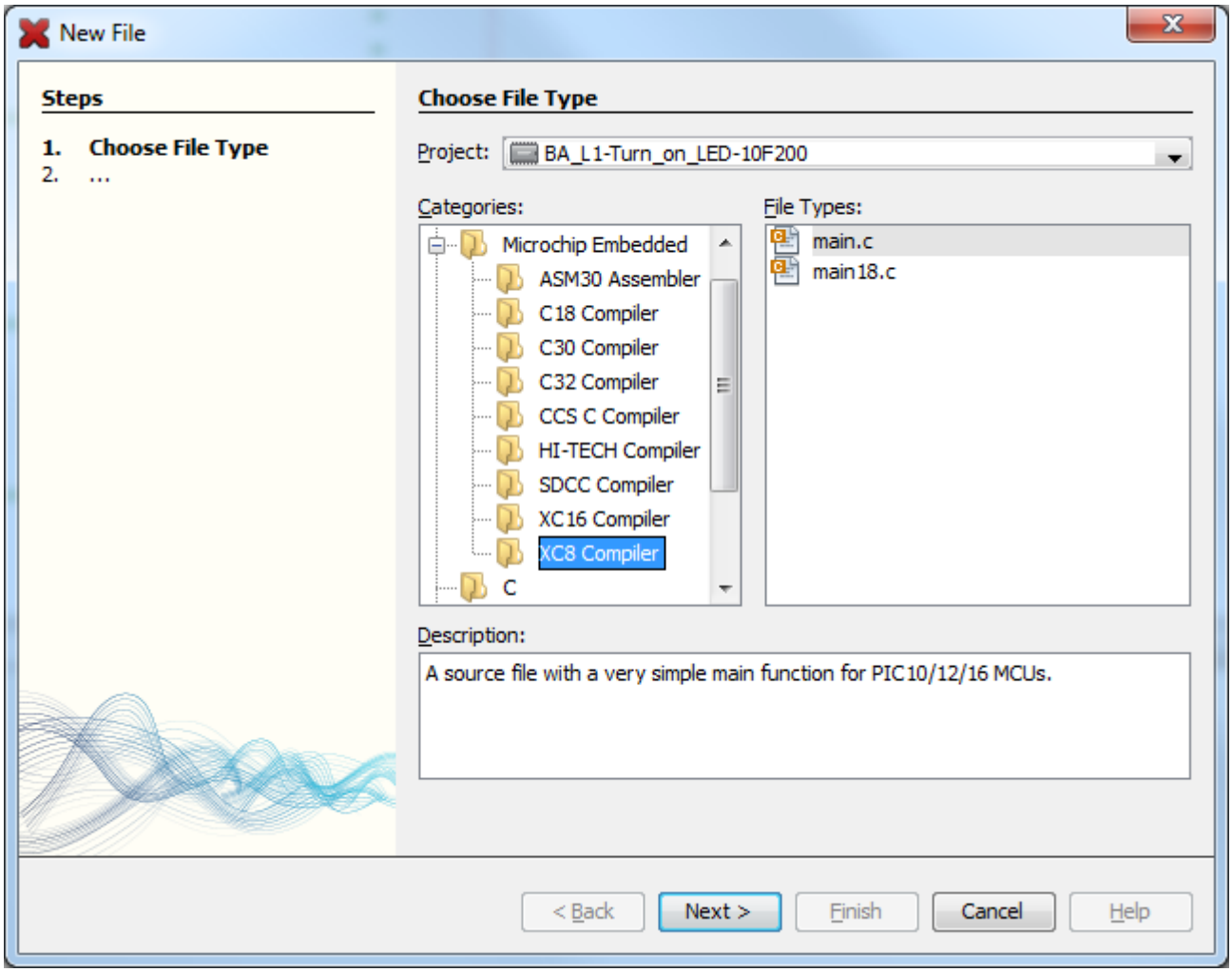
When you reach step 5, "Select Compiler", select "XC8" (taking care to select the version you wish to use, if you have more than one XC8 compiler installed), to specify that this is an XC8 project:



After completing the wizard, right-click 'Source Files' in the project tree within the Projects window, and select "New → C Source File…" to create a '.c' source file in your project folder:

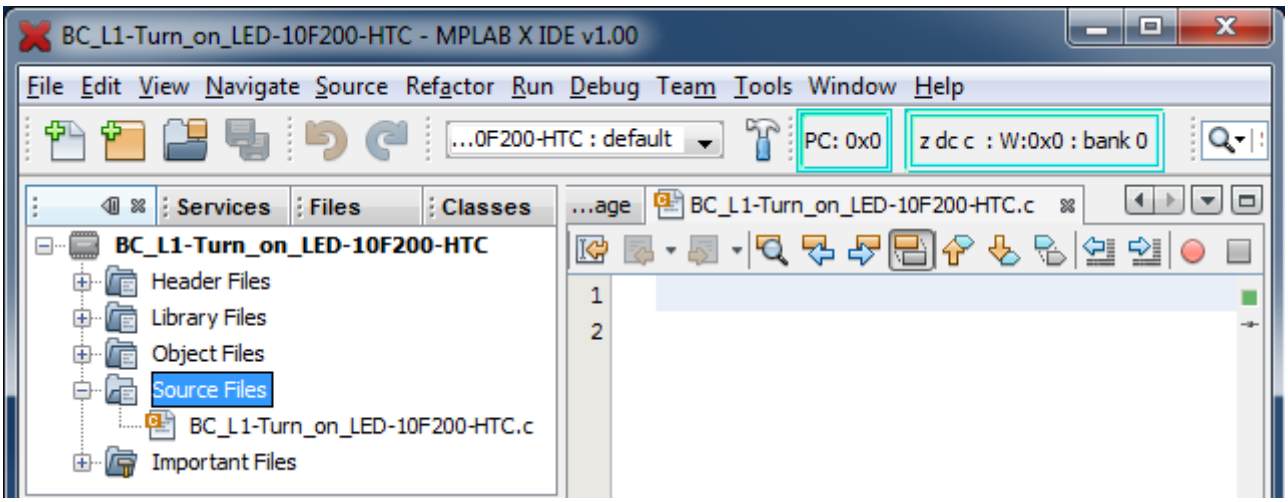


Note that, when you right-click 'Source Files' and select the "New" sub-menu, to create a new file and add it to your project, you are presented with a number of options, including any file types you have recently used, and "Other". This leads you to the "New File" dialog, which allows you to base your new file on an existing template. You'll find some templates for use with XC8 under 'Microchip Embedded':



The "main.c" template is a reasonable start, but it's slightly different from the C source code style used in these tutorials, so we won't use it here. Nevertheless, these code templates are a helpful feature of MPLAB X, and as you become more experienced, you can even develop your own.

When you have created a blank C source file, it should appear in the project tree, as usual:



You can now use the editor to start coding!

XC8 Source code

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 because, as we have seen for data types, C code for microcontrollers is not necessarily easily portable.

So we might use something like:

```
/************************************************************************
*                                                                       *
*   Filename:       BC_L1-Turn_on_LED-10F200-HTC.c                      *
*   Date:           7/6/12                                              *
*   File Version:   1.1                                                 *
*                                                                       *
*   Author:         David Meiklejohn                                    *
*   Company:        Gooligum Electronics                                *
*                                                                       *
************************************************************************
*                                                                       *
*   Architecture:   Baseline PIC                                        *
*   Processor:      10F200                                              *
*   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>
```

Next, we need to configure the processor.

This can be done with a "*configuration pragma*".

For the 10F200 version of this example, we would have:

```
// ext reset, no code protect, no watchdog
#pragma config MCLRE = ON, CP = OFF, WDTE = OFF
```

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

```
__CONFIG(MCLRE_ON & CP_OFF & WDTE_OFF);
```

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.

For the 12F508 or 12F509 version[5], we have:

```
// ext reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_IntRC);
```

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)), given that 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 available as variables defined in the header files.

Loading the TRIS register with 111101b (clearing bit 1, configuring GP1 as an output) is simply:

```
    TRIS = 0b111101;        // 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 "pic10f200.h" file header file defines a union called GPIObits, containing a structure with bit-field members GP0, GP1, etc.

---

[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).

[5] The source code for the 12F508 and 12F509 is exactly the same.

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, even though setting GP1 individually, as above, will almost certainly work in this case.

To avoid any potential for 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
```

Finally, we need to loop forever. There are a number of C constructs that could be used for this, but the one used in most of the XC8 sample code (and it's as good as any) is:

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

Complete program

Here is the complete 10F200 version of the code to turn on an LED on GP1, for XC8:

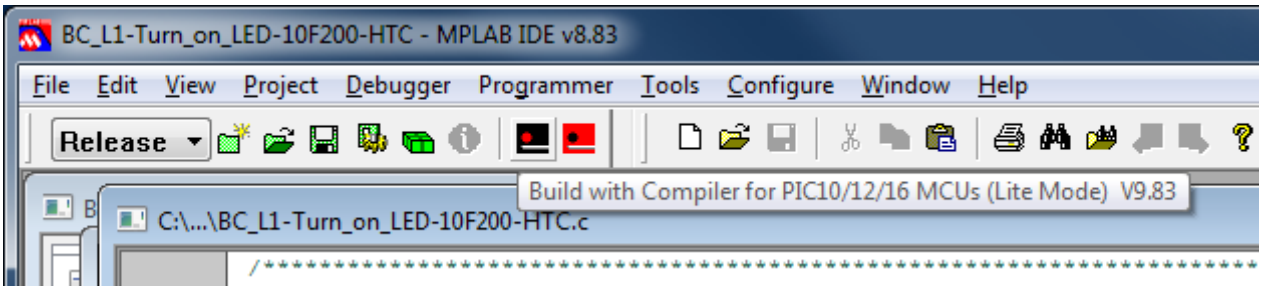
```
/************************************************************************
*   Description:    Lesson 1, example 1                                 *
*                                                                       *
*   Turns on LED.  LED remains on until power is removed.               *
*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

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog
__CONFIG(MCLRE_ON & CP_OFF & WDTE_OFF);


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

    GPIO = 0b000010;        // set GP1 high

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

The 12F508/509 version is the same, except for the different `__CONFIG` line, given earlier.

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 (see baseline assembler lesson 1).

To compile the source code in MPLAB 8, select "Project → Build", press F10, or click on the "Build" toolbar button:

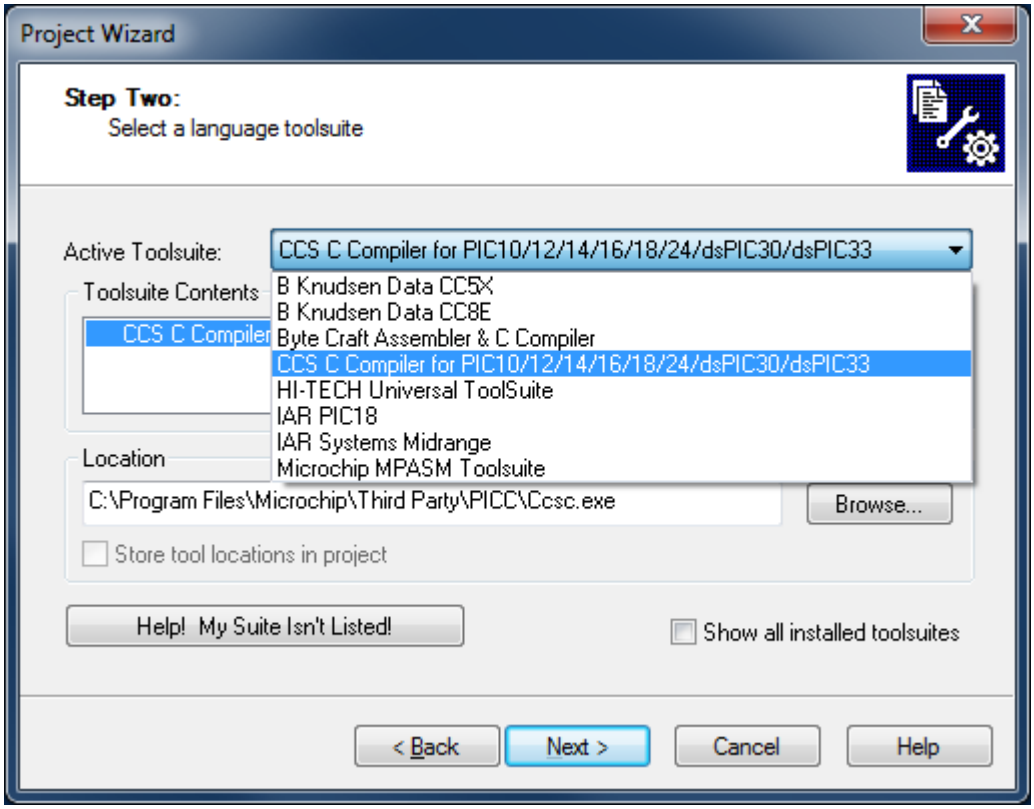


This is equivalent to the assembler "Make" option, compiling all the source files which have changed, and linking the resulting object files and any library functions, creating an output '.hex' file, which can then be programmed into the PIC as normal. The other Project menu item or toolbar button, "Rebuild", 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.

CCS PCB

The process of creating a new CCS PCB project in MPLAB 8 is the same as that for XC8, except that you need to select the "CCS C Compiler" toolsuite, in the project wizard, as shown on the right.



Note that the free version of CCS PCB, bundled with MPLAB 8, cannot be used with MPLAB X. However, if you purchase the most recent version of MPLAB X, you can use MPLAB C to create and build CCS C projects, in the same way as for XC8.

CCS PCB Source code

The comment block at the start of CCS PCB programs can of course be similar to that for any other C compiler (including XC8), but the comments should state that this code is for the CCS compiler.

It's not as important for the comments to state which processor is being used, since, unlike XC8, CCS PCB requires a '`#device`' directive, used to specify which processor the code is to be compiled for.

Also unlike XC8, there is no "catch-all" header file, so you are expected to include the appropriate '.h' file (found in the "devices" directory within the CCS PCB install directory), which defines all the symbols relevant to the processor you are using. This file will incorporate the appropriate '`#device`' directive, so you would not normally place that directive separately in your source code. Instead, at the start of every CCS PCB program, you should include a line such as:

```
#include <10F200.h>
```

However, you will find that this file, for the 10F200, defines the pins as `PIN_B0`, `PIN_B1`, etc., instead of the more commonly-used `GP0`, `GP1`, etc. This is true for the other 10F and 12F PICs as well. So to be able to use the normal symbols, we can add these lines at the start of our code, when working with 10F or 12F PICs:

```
#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
```

(if you're using a 10F device, you only need to define `GP0` to `GP3`, because the 10Fs only have four pins)

The '`#fuses`' directive is used to configure the processor.

For the 10F200, we have:

```
// ext reset, no code protect, no watchdog
#fuses MCLR,NOPROTECT,NOWDT
```

While for the 12F508 or 12F509, we also need to configure the oscillator:

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

Again, although this is similar to the `__CONFIG` directive we know from MPASM, the configuration symbols are different. For example, '`NOPROTECT`' instead of '`_CP_OFF`', and '`INTRC`' instead of '`_IntRC_OSC`'.

To see which symbols to use for a given PIC, you need to consult the header file for that device.

In the same way as XC8, the user program starts with `main()`:

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

And, as with XC8, the default start-up code, run before `main()` is entered, loads the factory calibration value into OSCCAL, so there is no need to write code to do that.

As mentioned earlier, the approach taken by CCS PCB is to make much of the PIC functionality available through built-in functions, reducing the need to access registers directly.

The '`output_high()`' function loads the TRIS register to configure the specified pin as an output, and then sets that output high. So to make GP1 an output and set it high, we can use simply:

```
output_high(GP1);         // configure GP1 (only) as an output and set high
```

However, as explained for the XC8 version and in detail in baseline assembler lesson 2, setting or clearing a single pin may lead to potential "read-modify-write" problems, and it is safer to write a value to the whole port (GPIO) in a single operation.

CCS C, we can do this with the '`output_x()`' built-in function, which outputs an entire byte to port x:

```
output_b(0b000010);     // configure GPIO as all output and set GP1 high
```

Note that, as far as CCS is concerned, on a 10F200 or 12F508/509, GPIO is port 'b'.

Also note that the '`output_x()`' function configures the port with every pin as an output, before outputting the specified value.

This behaviour of loading TRIS every time an output is made high (or low) makes the code simpler to write, but can be slower and use more memory, so CCS PCB provides a '`#use fast_io`' directive and '`set_tris_X()`' functions to override this slower "standard I/O", but for simplicity, and in the spirit of CCS C programming style, we'll keep this default behaviour for now.

To loop forever, we could use the '`for(;;) {}`' loop used in the XC8 example above, but since the standard CCS PCB header files define the symbol '`TRUE`' (and XC8 doesn't), we can use:

```
while (TRUE)
{                         // loop forever
    ;
}
```

Complete program

Here is the complete 10F200 version of the code to turn on an LED on GP1, using CCS PCB:

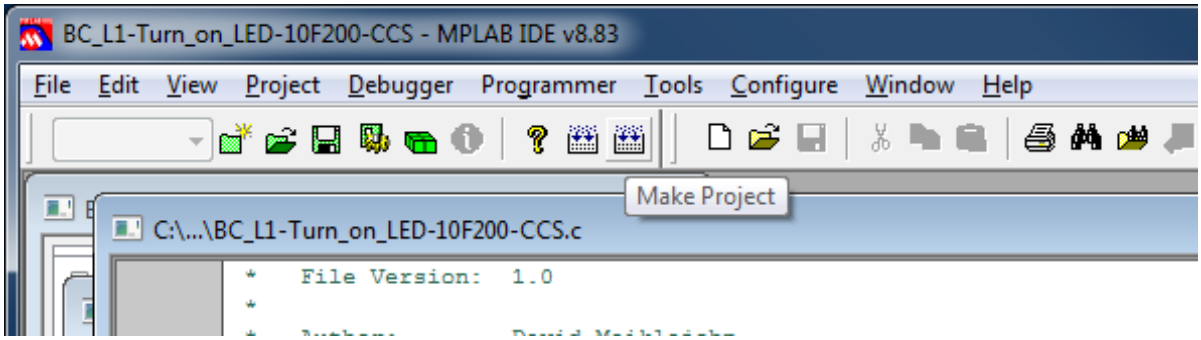
```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 1                                 *
*                                                                       *
*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

#include <10F200.h>

#define GP0 PIN_B0      // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog
#fuses MCLR,NOPROTECT,NOWDT
```

```
/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    output_b(0b000010);     // configure GPIO as all output and set GP1 high

    // Main loop
    while (TRUE)
    {                       // loop forever
        ;
    }
}
```

The 12F508/509 versions are the same, except that you need to `#include` a different header (`12F508.h` or `12F509.h`, instead of `10F200.h`), and change the `#fuses` directive, as shown earlier.

Building the project

To compile the source code in MPLAB 8, select "Project → Build", press F10, or click on the "Make Project" toolbar button:



This is the CCS equivalent to the XC8 "Build project" option, compiling all changed source files and linking the object files and library functions to create an output '.hex' file, which can be programmed into the PIC as normal.

The other Project menu item or toolbar button, "Build All", is equivalent to the XC8 "Rebuild" or the MPASM "Build All", recompiling your entire project, regardless of what's changed.

Comparisons

Even in an example as simple as turning on a single LED, the difference in approach between XC8 and CCS PCB is apparent.

The XC8 code shows a closer correspondence to the assembler version, with the TRIS register being explicitly written to.

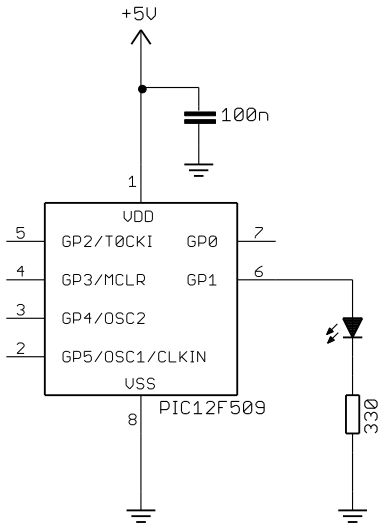
On the other hand, in the CCS PCB example, GPIO is configured as all outputs and written to, through a single built-in function that performs both operations, effectively hiding the existence of the TRIS register from the programmer.

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

In baseline lesson 2, we used the same circuits as above, but made the LED flash by toggling the GP1 output. The delay was created by an in-line busy-wait loop.

Baseline lesson 3 showed how to move the delay loop into a subroutine, and to generalise it, so that the delay, as a multiple of 10 ms, is passed as a parameter to the routine, in W.

This was demonstrated on the PIC12F509, using the circuit shown on the right. If you are using the Gooligum baseline training board, remember to remove the 10F200 from the '10F' socket, before plugging the 12F509 into the '12F' section of the 14-pin socket.



The example program 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, and continually repeating.

Here is the main loop from the assembler code from baseline lesson 3:

```
main_loop
        ; turn on LED
        movlw   b'000010'       ; set GP1 (bit 1)
        movwf   GPIO
        ; delay 0.2 s
        movlw   .20             ; delay 20 x 10 ms = 200 ms
        pagesel delay10
        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
        pagesel main_loop
        goto    main_loop
```

XC8

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 PIC 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>


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

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


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRIS = 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
}
```

CCS PCB

In the previous example, we turned on the LED with:

```
        output_high(GP1);       // set GP1
```

or

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

Similarly, the LED can be turned off by:

```
        output_low(GP1);        // clear GP1
```

or

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

In a similar way to XC8, CCS PCB provides built-in delay functions: '`delay_us()`' and '`delay_ms()`', which create delays of a specified number of µs and ms respectively. They accept a 16-bit unsigned value (0-65535) as a parameter. Unlike the XC8 macros, which can only generate a constant delay, the CCS delay functions accept either a variable or a constant as a parameter.

Since the functions are built-in, there is no need to include any header files before using them. But you must still specify the processor clock speed, so that the delays can be created correctly.

This is done using the '`#use delay`' pre-processor directive to the processor oscillator frequency, in Hertz.

For example, since our PIC is running at 4 MHz, we have:

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

To create a 200 ms delay, we can then use:

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

Complete program

Here is the complete code to flash an LED on GP1, with a 20% duty cycle, using CCS PCB:

```
/************************************************************************
*                                                                       *
*   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 <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 *****/
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC

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


/***** MAIN PROGRAM *****/
void main()
{
    // Main loop
    while (TRUE)
    {
        output_b(0b000010);     // turn on LED on GP1 (bit 1)

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

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

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

    }                           // repeat forever
}
```

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

The first LED flashing example in baseline assembler lesson 2 used an XOR operation to flip the GP1 bit every 500 ms, creating a 1 Hz flash with a 50% duty cycle.

In that example, we used a *shadow register* to maintain a copy of the port register (GPIO), and flipped the shadow bit corresponding to GP1, instead of working on port directly. As that lesson explained, this was to avoid potential problems due to read-modify-write operations on the port bits. If you're a little hazy on this concept, it would be a good idea to review that section of baseline assembler lesson 2.

As noted earlier, when you use a statement like '`GPIObits.GP1 = 1`' in XC8, or '`output_high(GP1)`' in CCS PCB, the compilers translate those statements into bit set or clear instructions, acting directly on the port registers, 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.*

To avoid such problems, shadow variables can be used in C programs, in the same way that shadow registers are used in assembler programs.

To demonstrate this, we can continue to use the circuit from the previous example, and model our code on the corresponding example from baseline assembler lesson 3:

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

        clrf    sGPIO           ; start with shadow GPIO zeroed

;***** Main loop
```

```
main_loop
        ; toggle LED on GP1
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   b'000010'       ; toggle bit corresponding to GP1 (bit 1)
        movwf   sGPIO           ;   in shadow register
        movwf   GPIO            ; and write to GPIO
        ; delay 0.5 s
        movlw   .50             ; delay 50 x 10 ms = 500 ms
        pagesel delay10         ;   -> 1 Hz flashing at 50% duty cycle
        call    delay10

        ; repeat forever
        pagesel main_loop
        goto    main_loop
```

XC8

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>
```

This 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 ^= 0b000010;      // toggle shadow bit corresponding to GP1
        GPIO = sGPIO;           // write to GPIO
```

Complete program

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

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

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


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

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


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


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

    // Main loop
    for (;;)
    {
        // toggle LED on GP1
        sGPIO ^= 0b000010;      // toggle shadow bit corresponding to GP1
        GPIO = sGPIO;           // write to GPIO

        // delay 500 ms
        __delay_ms(500);

    }   // repeat forever
}
```

CCS PCB

CCS PCB provides a built-in function specifically for toggling an output pin: '`output_toggle()`'. To toggle GP1, all that is needed is:

```
output_toggle(GP1);
```

But since this function performs a read-modify-write operation on GPIO, we'll use a shadow variable, which can be declared and initialised with:

```
unsigned int8   sGPIO = 0;      // shadow copy of GPIO
```

CCS PCB doesn't come with an equivalent to '`stdint.h`', so we can't use the C99 standard 'uint8_t' type, as we did with XC8. We could have declared this variable as a 'char', and that would be ok, but by declaring it as an 'unsigned int8', it's very clear that this variable is an unsigned, eight-bit integer.

And, as in the XC8 example, we'll define this as a global variable, outside `main()`, so that it can be accessed by any other functions you add to the code in future.

Toggling the shadow copy of GP1 is then the same as for XC8:

```
        sGPIO ^= 0b000010;      // toggle shadow bit corresponding to GP1
```

To write the result to GPIO, we can use the `output_b()` built-in function, as before:

```
        output_b(sGPIO);        // write to GPIO
```

[Recall that CCS PCB refers to GPIO on the 10F and 12F PICs as port B.]

Complete program

Here is the complete CCS PCB code to flash an LED on GP1, with a 50% duty cycle:

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

#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 *****/
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,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()
{
   // Main loop
   while (TRUE)
   {
        // toggle LED on GP1
        sGPIO ^= 0b000010;      // toggle shadow bit corresponding to GP1
        output_b(sGPIO);        // write to GPIO

        // delay 500ms
        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[6]) and resource utilisation (program and data memory usage) for the two compilers and the assembler version of this example from baseline lesson 3.

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 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 functions provided by the C compilers generate accurately-timed delays, and how their accuracy compares with our assembler version.

Memory usage is reported correctly by MPLAB for assembler and XC8 projects, but note that the CCS PCB compiler does not accurately report data memory usage to MPLAB. We can get it from the '*.lst' file generated by the CCS compiler.

The MPLAB simulator[7] 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:

Flash_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)	Delay accuracy (timing error)
Microchip MPASM	28	34	4	0.15%
XC8 (Free mode)	11	36	4	0.0024%
CCS PCB	9	43	6	0.0076%

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

[7] a topic for a future tutorial?

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 versions! This illustrates how much more compact C code can be.

As for C being less efficient – the XC8 version is barely larger than the assembler version, despite having most compiler optimisations disabled in "Free" mode. This is largely because the built-in delay code is highly optimised, but it does show that C is not necessarily inherently inefficient – at least for simple applications like this.

On the other hand, the CCS compiler is noticeably less efficient in this example, generating code 26% bigger than the assembler version.

Finally, note that the time delays in both C versions are amazingly accurate!

Summary

Overall, we have seen that, although XC8 and CCS PCB take quite different approaches, basic digital output operations can be expressed succinctly using either C compiler.

We saw that the CCS approach is to use built-in functions to perform operations which may take a number of statements in XC8 to accomplish (such as configuring pin direction and outputting a value in a single statement). Whether this approach is better is largely a matter of personal style, although having so many built-in functions available can make development much easier.

Whichever compiler you use, 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. This argument may apply to CCS PCB, which as we have seen, tends to hide details of the hardware from the programmer; it is not always apparent what the program is always doing "behind the scenes". But it doesn't really apply to XC8, which exposes all the PIC's registers as variables, and the programmer has to modify the register contents in the same way as would be done in assembler.

Although it's not really apparent in the comparison table above, 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 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 assembly is the right choice. And if you need to write a really tight loop, where every instruction cycle counts, assembly may be the only viable choice. Although again, using a faster, but more expensive chip may be a better solution, unless your application is high-volume.

If this is a hobby for you, then it's purely a question of personal preference, because as we have seen, both the Microchip and CCS "free" C compilers, as well as assembler, are viable options.

In addition to providing an output (such as a blinking LED), PIC applications usually have to respond sensors and/or user input.

In the next lesson we'll see how to use our C compilers to read and respond to switches, such as pushbuttons.

And since real switches "bounce", which can be a problem for microcontroller applications, we'll look at ways to "debounce" them, in software.