Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital I/O

Although assembler is a common choice when 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 lesson demonstrates, it is quite possible to use C for simple programs on baseline PICs. Nevertheless, it is true that C is less suited to implementing more complex applications, where we need to get the most out of these small devices, as we will see in later lessons.

This lesson introduces the "free" compilers from HI-TECH Software and Custom Computer Services (CCS) bundled with MPLAB, both of which fully support all current baseline PICs. As we'll see, the HI-TECH 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 I/O: flashing LEDs, responding to and debouncing switches, as covered in lessons 1 to 4 of the baseline assembler tutorial series. You may need to refer to those lessons while working through this one.

In summary, this lesson covers:

- Introduction to the HI-TECH PICC-Lite, HI-TECH C and CCS PCB compilers
- Digital input and output
- Programmed delays
- Switch debouncing
- Using internal (weak) pull-ups

with examples for all three compilers, and comparisons with assembler (resource usage versus code length).

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

Introducing HI-TECH C and CCS PCB

Microchip's MPLAB version 8.10 was bundled with two C compilers: HI-TECH's "PICC-Lite" and CCS's "PCB". They are free to use, integrated with MPLAB, and support all current baseline (12-bit) PICs, including those used so far in this tutorial series, with no restrictions. PICC-Lite also supports a small number of the midrange (14-bit) PICs – although, for most of the midrange devices it supports, PICC-Lite limits the amount of data and program memory that can be used, to provide an incentive to buy the full compiler. However, in 2009, HI-TECH Software retired PICC-Lite, in favour of their new "HI-TECH C" compiler, and no longer supply or support PICC-Lite.

"HI-TECH C" supports all baseline and midrange PICs, with no memory restrictions. It can be used for free, when running in "Lite mode". However, in this mode, all compiler optimisation is turned off, making the generated code around twice the size of that generated by PICC-Lite.

Versions 8.15 and later of MPLAB have included HI-TECH C[1], instead of PICC-Lite. This gives those developing for midrange 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 HI-TECH C will continue to be maintained, supporting new baseline and midrange devices over time.

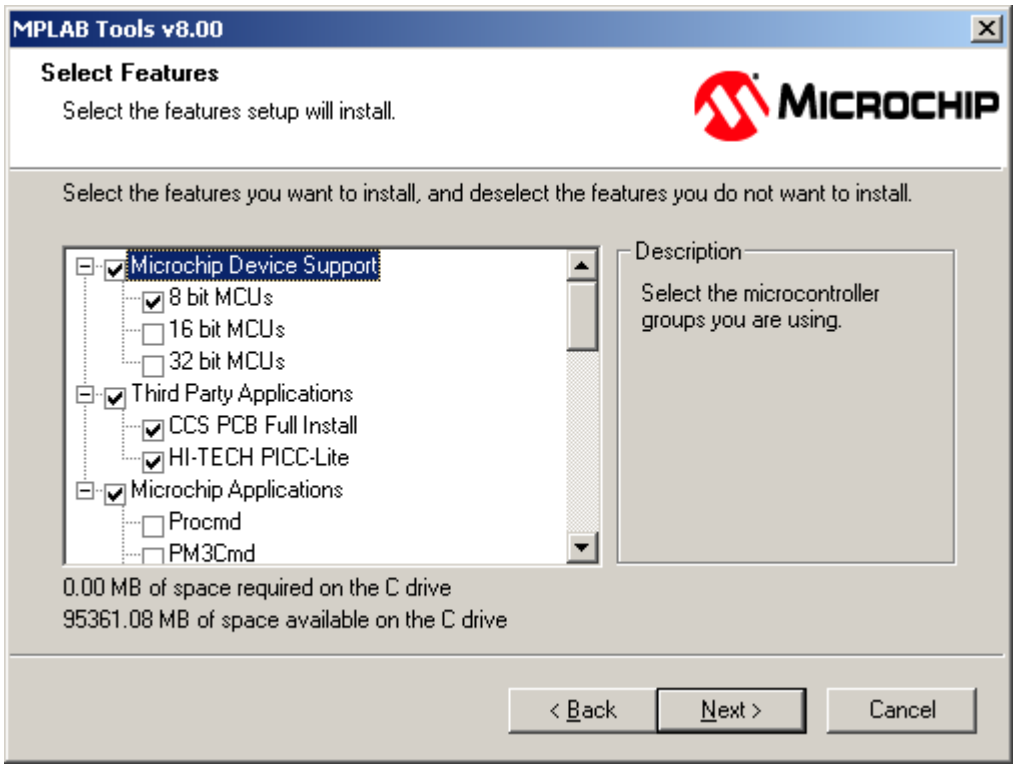
But if you are using a supported baseline device, it is better to continue to use PICC-Lite (if you are able to locate a copy), since it will generate much more efficient code, while allowing all the (limited) memory on your baseline PIC to be used. It can be installed alongside HI-TECH C.

For comparison with PICC-Lite, the size of each example program compiled by HI-TECH C, in "Lite" mode, is given in these lessons, illustrating the difference between optimised and non-optimised code.

Why would HI-TECH Software and CCS be prepared to give away what are effectively full, unrestricted versions of their C compilers, for the baseline PICs?

As we'll see, programs written using either of these compilers use more resources (program and data memory) than an equivalent assembly program normally would, and these baseline PICs do not have much memory available. These very small MCUs tend to be used in applications where it is generally not too onerous to develop effectively in assembly. So the market for C compilers for these devices is quite small. HI-TECH and CCS would not lose many sales by offering these products for free. But by introducing designers to their products in this way, they encourage sales of their compilers for the larger PICs (and indeed other, non-Microchip MCUs – one of the advantages of using a higher level language such as C is that it becomes easier – although not trivial – to port code to different MCUs).

To install these compilers, you should select them as options when installing MPLAB, to ensure that the integration with the MPLAB IDE will be done correctly:



[1] Versions 8.15 to 8.40 of MPLAB were bundled with a compiler called "HI-TECH C PRO" – essentially an earlier version of the current "HI-TECH C" compiler, with a similar "Lite" mode. It was used in lessons 1 to 5.

A separate installer will be launched for PICC-Lite or HI-TECH C. There is no need install Hi-Tide (HI-TECH's own IDE) if prompted, as the HI-TECH compilers can be used effectively from within MPLAB, as long as all the defaults are chosen during the install process.

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 July 2008) almost all the PIC processors from the 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 midrange (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 the Windows command-line compilers to be integrated into MPLAB (this plug-in is installed when CCS PCB is selected as part of the MPLAB installation).

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 July 2008, the advertised costs range from US$50 for PCB (but it is available for free with MPLAB), through US$150 for PCM, US$350 for PCW, to US$600 for the full PCWHD suite.

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).

HI-TECH Software "PICC-Lite" and "HI-TECH C"

HI-TECH Software (www.htsoft.com) was purchased by Microchip in 2009, and now only support Microchip MCUs. Until recently (late 2009), they also offered an IDE ("HI-TIDE"), available for both Windows and Linux, which includes a "C-aware" editor, and debugger/simulator. However, HI-TIDE is being de-emphasised in favour of MPLAB integration, and although still available for download[2], technical support is no longer available.

The "HI-TECH C" supports the whole PIC10/12/16 series in a single edition, with different licence keys unlocking different levels of code optimisation – "Lite" (free, but no optimisation), "Standard" and "PRO" (most expensive and highest optimisation).

HI-TECH Software also offers compilers for the PIC18, PIC24, dsPIC and PIC32 families.

PICC-Lite is a cut-down version of the old PICC STD compiler (no longer available). It supports almost all baseline PICs[3], with no limitations. It also supports the 14-bit 12F629, 12F675, 16C84 and 16F84A PICs, with no limitations. The 14-bit 16F627, 16F627A, 16F684, 16F690, 16F877, 16F877A, 16F887 and 16F917 PICs are also supported, but with limitations on the amount of program and data memory that can be used.

---

[2] as of early 2010

[3] devices released prior to February 2008

The HI-TECH compilers are more expensive than those from CCS: as of January 2009, the advertised costs range from US$495 for the PIC10/12/16 and PIC18 compilers in "Standard" mode, through US$1195 for these compilers in "PRO" mode, to $3795 for the HI-TECH C Enterprise Edition (a bundle supporting all PIC MCUs).

The HI-TECH approach is to expose 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 HI-TECH compilers, 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 similar in "spirit" to the sample code provided with each. For example, although it is possible to map registers into variables in the CCS compilers, the examples use the 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 compilers and the original assembler version.

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 CCS PCB and HI-TECH C:

Type	HI-TECH	CCS PCB
bit	1	-
int1	-	1
char	8	8
int8	-	8
short	16	1
int	16	8
int16	-	16
long	32	16
int32	-	32
float	24	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!

HI-TECH C defines a single 'bit' type, unique to HI-TECH C.

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).

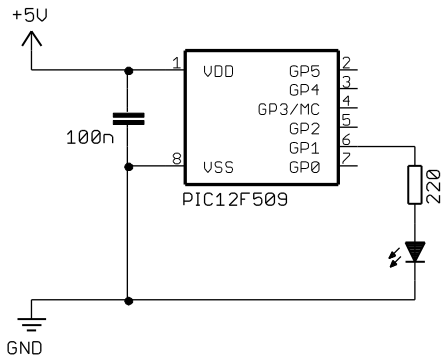
The "standard" 'int' type is 8 bits in CCS PCB, but 16 bits in HI-TECH C.

But by far the greatest difference is in the definition of 'short': in HI-TECH C, 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, 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.

Finally, note that 'double' floating-point variables in HI-TECH C 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 HI-TECH C's ability to work with 24-bit floating point numbers can be useful.

Example 1: Turning on an LED

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



The LED is connected to the GP1 pin on a PIC12F509.

To turn on the LED, we loaded the TRIS 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.

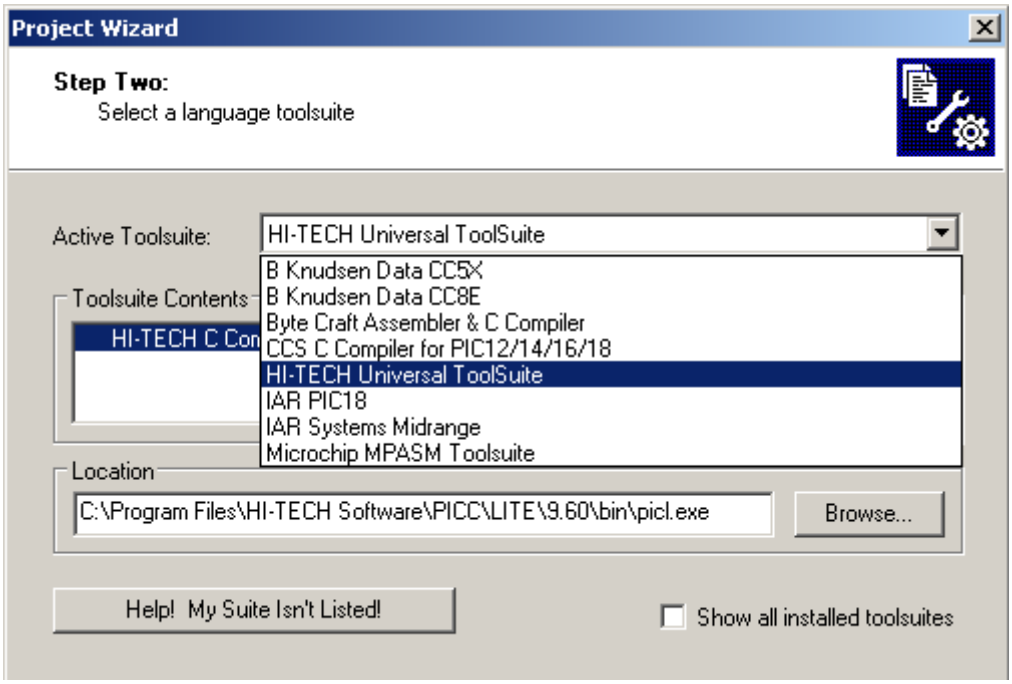
Finally, the end of the program consisted of an infinite loop ('`goto $`'), to leave the LED turned on.

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

```
                ; ext reset, no code protect, no watchdog, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC

MAIN    CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO
        movlw   b'000010'       ; set GP1 high
        movwf   GPIO

        goto    $               ; loop forever
```

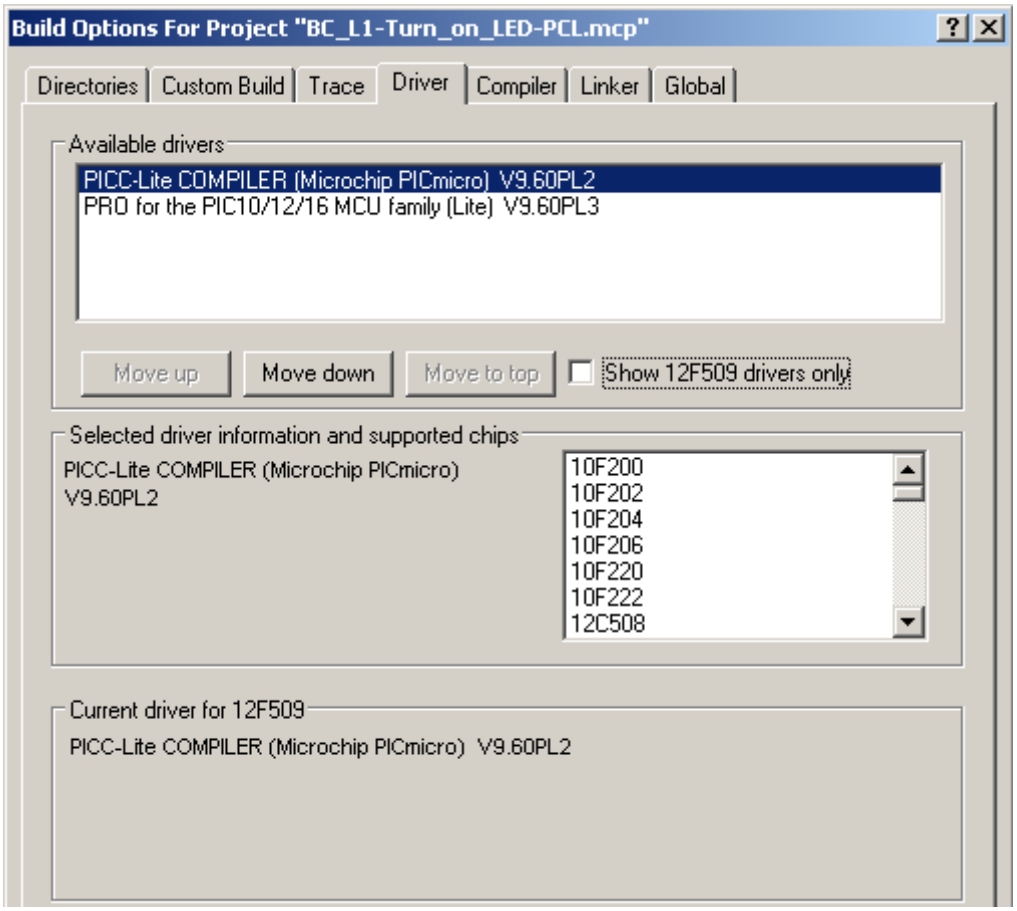
HI-TECH C or PICC-Lite

When creating a new project for one of the HI-TECH compilers, using the project wizard, you should select the "HI-TECH Universal ToolSuite" when you come to "Step Two: Select a language toolsuite":



If you have installed both the HI-TECH C and PICC-Lite compilers, you need to tell the HI-TECH toolsuite which compiler, or *driver*, to use.

To do this, open the project build options window (Project → Build Options… → Project) then select the "Driver" tab:



To select the compiler to use, move it to the top of the list of available drivers, by selecting it then 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 can then proceed to create a new project, in the same way that you would for an assembler project, except that your source code file should end in '.c' instead of '.asm'.

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:       BC_L1-Turn_on_LED-PCL.c                             *
*   Date:           6/12/07                                             *
*   File Version:   1.1                                                 *
```

```
*                                                                               *
*   Author:         David Meiklejohn                                            *
*   Company:        Gooligum Electronics                                        *
*                                                                               *
********************************************************************************
*                                                                               *
*   Architecture:   Baseline PIC                                                *
*   Processor:      12F508/509                                                  *
*   Compiler:       Hi-Tech PICC-Lite v9.60PL2                                  *
*                                                                               *
********************************************************************************
*                                                                               *
*   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 HI-TECH compilers, 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 Hi-TIDE), or as a command-line option.

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

```
#include <htc.h>
```

To set the processor configuration, a macro called '`__CONFIG(x)`' is used, in a very similar way to the `__CONFIG` directive in MPASM:

```
// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRC);
```

Note that the configuration symbols used are different to those defined in the MPASM include files. For example, '`UNPROTECT`' instead of '`_CP_OFF`', and '`INTRC`' instead of '`_IntRC_OSC`'. To see which symbols to use for a given MCU, you need to look in the appropriate include file. For example, in this case (for the 12F509), these symbols are defined in the "`pic125xx.h`" file, found in the "include" directory within the compiler install directory.

As with most C compilers, the entry point for "user" code is a function called '`main()`'. So a HI-TECH C 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 HI-TECH C; the default start-up code, which runs before `main()` is entered, loads OSCCAL for us.

HI-TECH C makes the PIC's registers available as variables, so to load the TRIS register with 111101b, it is simply a matter of:

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

Individual bits within registers, such as GP1, are also mapped as variables, so that to set GP1 to '1', we can write:

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

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:

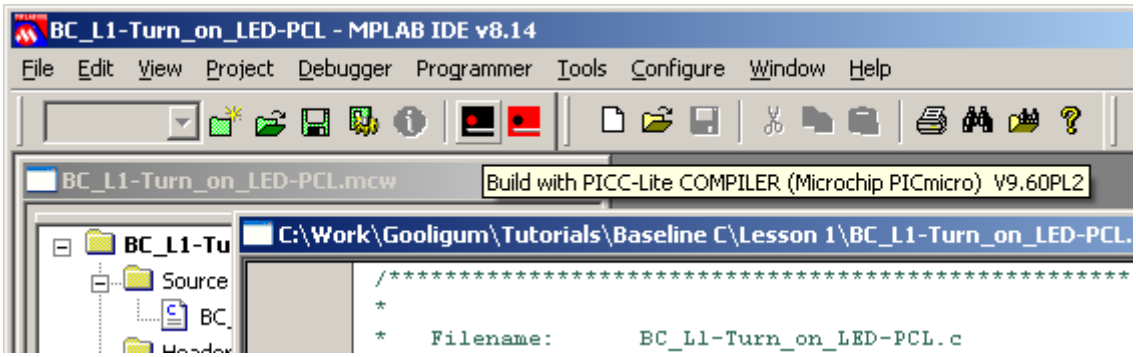
```
    for (;;) {              // loop forever
        ;
    }
```

Complete program

Here is the complete code to turn on an LED on GP1, for HI-TECH C:

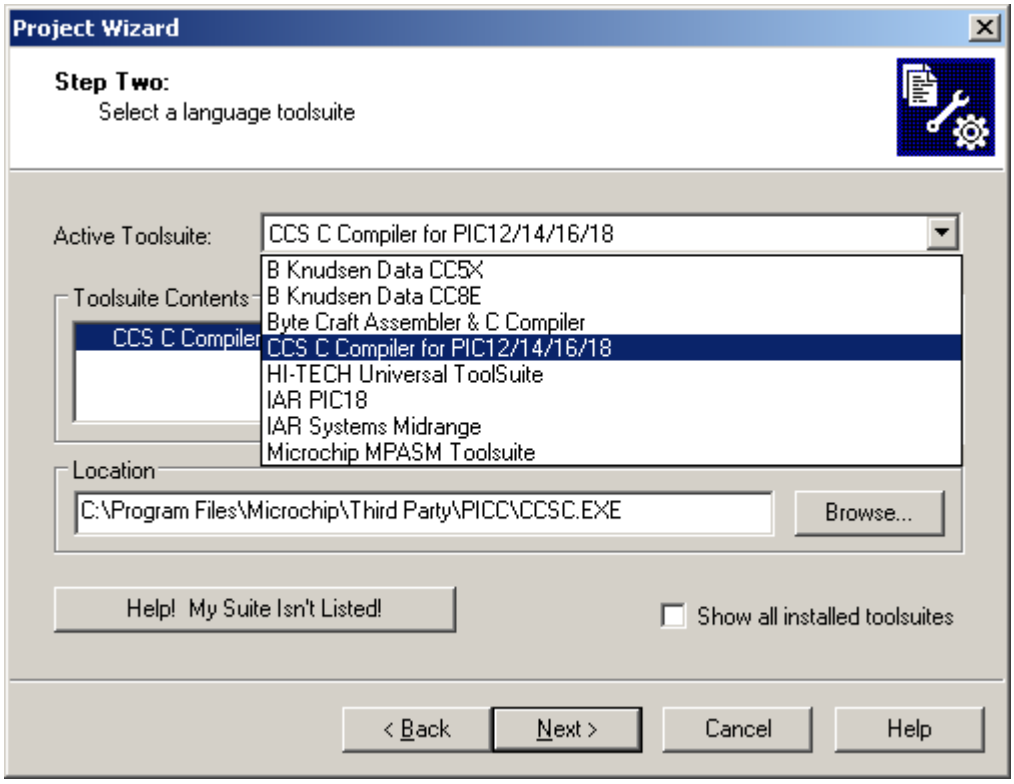
```
/*****************************************************************
*   Description:    Lesson 1, example 1                          *
*                                                                *
*   Turns on LED.  LED remains on until power is removed.        *
*                           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

#include <htc.h>
// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRC);

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

    GP1 = 1;                // set GP1 high

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

To compile the source code, click on the "Build project" icon (or simply press F10). 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 is programmed into the PIC as normal:



CCS PCB

The process to create a new project using the CCS PCB compiler is that same as that for HI-TECH C, except that you need to select the "CCS C Compiler for PIC12/14/16/18" toolsuite:



The comment block at the start of CCS PCB programs can of course be similar to that for any other C compiler (including HI-TECH C), 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 HI-TECH C, CCS PCB provides a '`#device`' directive, used to specify which processor the code is to be compiled for.

Also unlike HI-TECH C, there is no "catch-all" include file, so you are expected to identify the appropriate include 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 include the appropriate '`#device`' directive, so you would not normally include that directive separately in your source code. Instead, at the start of every CCS PCB program, you should include a line such as:

```
#include <12F509.h>
```

However, you will find that this file, for the 12F509, 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 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
```

To set the processor configuration, a directive called '#fuses' is used:

```
// Config: ext reset, no code protect, no watchdog, 4MHz int 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 MCU, you need to look in the include file.

In the same way as HI-TECH C, the user program starts with main():

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

And, as with HI-TECH C, the default start-up code, run before main() is entered, loads OSCCAL for us, so there is no need to write code to do that.

As mentioned above, 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, the code is simply:

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

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 includes a '#use fast_io' directive and 'set_tris_X()' functions to override this slower "standard I/O", but we'll stay with the default here.

To loop forever, we could use the 'for(;;) {}' code used in the HI-TECH example above, but since the standard CCS PCB include files define the symbol 'TRUE' (and HI-TECH C doesn't), we can use:

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

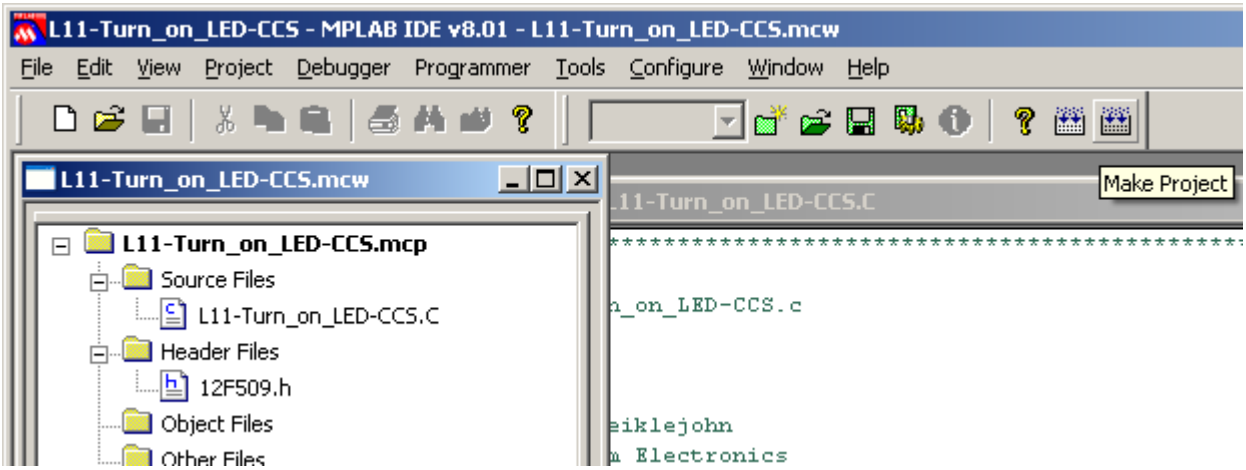
Complete program

Here is the complete 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 <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

// Config: ext reset, no code protect, no watchdog, 4MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC


void main()
{
   output_high(GP1);    // configure GP1 (only) as an output and set high

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

To compile the source code, click on the "Make Project" icon (or press F10). This is the CCS equivalent to the HI-TECH "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 usual:



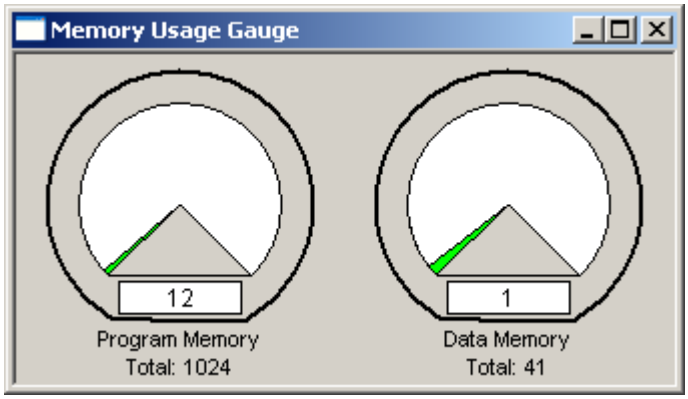
Comparisons

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

The HI-TECH C code shows a closer correspondence to the assembler version, with the TRIS register being explicitly written to, and GP1 being set by writing to a variable.

On the other hand, in the CCS PCB example, GP1 is configured as an output and set high, through a single built-in function that performs both operations, effectively hiding the existence of the TRIS register from the programmer.

To compare the memory used by each compiler, we can use the memory usage gauge available in MPLAB ("View" → "Memory Usage Gauge"):



This is the gauge as shown after building the CCS example above.

It shows that 12 out of the 1024 available words of program memory have been used, and 1 byte out of the 41 bytes of data memory available on the 12F509 has been allocated.

Although this gauge is accurate for MPASM assembler and HI-TECH C projects, it does not indicate all the data memory used by CCS PCB, which does not inform MPLAB of all the memory it allocates.

However, when a CCS PCB build completes, it reports the memory used. For example, for the "Turn on an LED" example, the report is:

Memory usage: ROM=1% RAM=12% - 12%

"ROM=1%" agrees with the program memory usage shown above, since 12 ÷ 1024 = 1.2%.

However, the "RAM=12% - 12%" (which means data memory usage will vary between 12% and 12% as the program runs, i.e. always 12% in this case) seems very high.

This is explained in the "Common Questions and Answers" section of the CCS online help, which reads in part, under "Why does the compiler show less RAM than there really is?":

Some devices make part of the RAM much more ineffective to access than the standard RAM. In particular, the 509, 57, 66, 67,76 and 77 devices have this problem.

By default, the compiler will not automatically allocate variables to the problem RAM and, therefore, the RAM available will show a number smaller than expected.

Since we are using a 12F509, it is one of these devices with "problem RAM", which appears to be the general purpose registers in bank 1. That means that the CCS PCB compiler only considers the first register bank to be usable, restricting data memory to 9 shared and 16 "bank 0" GPRs – a total of 25 bytes of usable data memory.

The online help suggests a couple of ways to use all available data memory, including:

*You can switch to larger pointers for full RAM access (this takes more ROM). In PCB add *=8 to the #device.*

We don't need to worry about that for now, since RAM use is "only" 12%.

This is 12% of 25 bytes, so data memory use in this case is 3 bytes.

The following table summarises the resource usage by our "Turn on an LED" assembler, HI-TECH and CCS PCB C programs, as well as the number of lines of source code (excluding comment, white-space and single-brace lines). For the CCS PCB program, the additional '`#define`' statements have also been excluded, since they are not strictly necessary. For the HI-TECH version, the same source code was compiled with both HI-TECH C PRO (in "Lite mode") and PICC-Lite, for comparison:

Turn_on_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	11	9	0
HI-TECH PICC-Lite	6	9	4
HI-TECH C PRO Lite	6	11	2
CCS PCB	5	12	3

In general, assembler source code will be longer than equivalent C code – around twice as long as the C versions of this example.

Although you may not expect that a C complier would be able to optimise code as well as a human programmer, in this trivially short example, the C compilers are able to generate code as small as, or only slightly larger than, the hand-written assembler version.

The HI-TECH C PRO compiler, when running in the free "Lite mode", would normally be expected to generate inefficient (non-optimised) code, this inefficiency is not apparent in this example.

We would also generally expect that CCS source code will be shorter than for equivalent HI-TECH C programs, because the CCS built-in functions can perform an action that would require a number of instructions in HI-TECH C to express.

As we will see, the differences become more apparent in longer, more complex programs.

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

In baseline lesson 2, 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.

Baseline lesson 3 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 baseline lesson 3:

```
flash
        movlw   b'000010'       ; set bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; write to GPIO to turn on LED
        movlw   .20             ; stay on for 0.2s:
        pagesel delay10
        call    delay10         ;   delay 20 x 10ms = 200ms
        clrf    GPIO            ; clear GPIO to turn off LED
        movlw   .80             ; stay off for 0.8s:
        call    delay10         ;   delay 80 x 10ms = 800ms
        pagesel flash
        goto    flash           ; repeat forever
```

HI-TECH PICC-Lite

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

```
GP1 = 1;                // turn on LED on GP1
```

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

```
GP1 = 0;                // turn off LED on GP1
```

These statements can easily be placed within the infinite '`for (;;) {}`' loop, to repeatedly turn the LED on and off. All we need to add is a delay.

PICC-Lite comes with a number of code examples, found in the "samples" directory within the PICC-Lite install directory. The sample code includes some useful delay functions, defined in the files "delay.h" and "delay.c", found in the "delay" directory (within "samples").

These functions are '`DelayUs()`' (technically not a function, but a macro), which provides a delay in µs, and '`DelayMs()`', which provides a delay in ms. Both take a parameter from 0 to 255.

To use these functions in your own program, you could copy the code into your source code, perhaps customising it as appropriate. But it is probably easier to treat them as library functions, by copying the "delay.h" and "delay.c" files into your project directory, and adding those files to your project (using for example the "Project → Add Files to Project…" menu item).

You then need to include the "delay.h" file at the start of your program, so that it can reference the delay functions. But first you must define the processor clock speed, so that the delay functions perform the correct number of loops. This is done by defining the symbol "`XTAL_FREQ`". For example:

```
#define XTAL_FREQ   4MHZ     // oscillator frequency for DelayMs()
#include "delay.h"           // defines DelayMs()
```

("`MHZ`" and "`KHZ`" are defined in "delay.h")

To create a 200 ms delay, it is then simply a matter of using:

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

But since the '`DelayMs()`' function has a single-byte parameter, i.e. 0 – 255, to create an 800 ms delay, we need a series of function calls, such as:

```
DelayMs(250);           // stay off for 800ms
DelayMs(250);
DelayMs(250);
DelayMs(50);
```

Complete program

Here then is the complete code to flash an LED on GP1, with a 20% duty cycle, using PICC-Lite:

```
/************************************************************************
*   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 <htc.h>

#define XTAL_FREQ   4MHZ     // oscillator frequency for DelayMs()
#include "delay.h"           // defines DelayMs()

// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRC);


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

    // Main loop
    for (;;) {
        GP1 = 1;                // turn on LED on GP1

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

        GP1 = 0;                // turn off LED on GP1

        DelayMs(250);           // stay off for 800ms
        DelayMs(250);
        DelayMs(250);
        DelayMs(50);
    }                           // repeat forever
}
```

HI-TECH C PRO Lite

Unfortunately, the sample delay code included with PICC-Lite doesn't work correctly when HI-TECH C PRO is used in the free "Lite" mode. The delay timing assumes that the code will be compiled with full optimisation, while optimisation is disabled in HI-TECH C PRO's Lite mode. So although the code compiles, the delay code takes much longer (around three times) to execute than it should.

Luckily, the HI-TECH C PRO compiler provides a built-in function, '`_delay(n)`', which creates a delay '`n`' instruction clock cycles long, up to a maximum of 197120 cycles. With a 4 MHz processor clock, corresponding to a 1 MHz instruction clock, that's a maximum delay of 197.12 ms.

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 is used in much the same way as "`XTAL_FREQ`" was with the PICC-Lite delay functions, except that the symbols "`MHZ`" and "`KHZ`" are not predefined and so, unless you define them yourself, cannot be used.

So to define the processor frequency, we have:

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

And note that there is no need to include the "delay.h" header file from PICC-Lite; it's not used.

Since the PIC will be running at 4 MHz, the maximum delay we can generate with a single built-in delay function or macro is 197.12 ms, so the initial 200 ms delay must be split into two function (or macro) calls:

```
        // stay on for 200ms
        __delay_ms(100);
        __delay_ms(100);
```

To create the 800 ms delay, we could repeat '`__delay_ms(100)`' eight times, but once you get up to three or four repetitions, it's more efficient to write it as a loop:

```
        // stay off for 800ms
        for (dcnt = 0; dcnt < 8; dcnt++) {
            __delay_ms(100);
```

The delay counter variable had previously been defined as:

```
    unsigned char   dcnt;       // delay counter
```

The main loop then becomes:

```
    // Main loop
    for (;;) {
        // turn on LED on GP1
        GP1 = 1;

        // stay on for 200ms
        __delay_ms(100);
        __delay_ms(100);

        // turn off LED on GP1
        GP1 = 0;

        // stay off for 800ms
        for (dcnt = 0; dcnt < 8; dcnt++) {
            __delay_ms(100);
        }
    }   // repeat forever
```

CCS PCB

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

```
        output_high(GP1);       // turn on LED on GP1
```

Similarly, the LED can be turned off by:

```
        output_low(GP1);        // turn off LED on GP1
```

In a similar way to HI-TECH C PRO, 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 either an 8-bit variable (0-255) or a 16-bit constant (0-65535) 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. For example:

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

This specifies the clock speed in Hz (4 MHz in this case).

To create a 200 ms delay, we can write:

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

And since the '`delay_ms()`' function can take a 16-bit constant as a parameter, to create an 800 ms delay we can use:

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

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

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

// Config: ext reset, no code protect, no watchdog, 4MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC

void main()
{
    while (TRUE) {
        output_high(GP1);       // turn on LED on GP1

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

        output_low(GP1);        // turn off LED on GP1

        delay_ms(800);          // stay off for 800ms
    }                           // repeat forever
}
```

Comparisons

In addition to the source code length and memory usage, another relevant comparison for this example is the accuracy of the delays – how close is the LED flash period to the intended 1 second (assuming a clock rate of exactly 4 MHz)? This can be precisely determined by using the stopwatch facility provided by the MPLAB SIM simulator – a topic for a future lesson.

Flash_LED_20p

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)	Delay accuracy (timing error)
Microchip MPASM	40	33	3	0.15%
HI-TECH PICC-Lite	14	78	9	1.6%
HI-TECH C PRO Lite	13	58	6	0.011%
CCS PCB	9	45	5	0.0063%

The assembler code presented in baseline lesson 3 included a delay subroutine, adding 13 lines to the source code. It seems appropriate to include the delay subroutine in the source code length in the table above, because MPASM does not come with sample delay code that could be included in the source (as was done with PICC-Lite); we had to write the delay routine from scratch.

The C source is significantly shorter than assembler, and shorter for CCS PCB than HI-TECH C.

The CCS compiler produces much more efficient code than PICC-Lite in this example, although neither can match the efficiency of assembler.

And in this case the HI-TECH C PRO Lite compiler is not as inefficient as expected, actually generating smaller code than PICC-Lite – probably because of the use of an efficient built-in delay function.

The HI-TECH C PRO and CCS compilers also generate amazingly accurate programmed delays!

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

The first LED flashing example in baseline lesson 2 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 capacitative 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 on a port, but as discussed in lesson 2, 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 '`GP1 = 1`' in HI-TECH C, or '`output_high(GP1)`' in CCS PCB, the compilers translate 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:

```
GP1 = 1;
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 the same way that shadow registers are used in assembler.

Here is the main code from the program presented in baseline lesson 3:

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

        clrf    sGPIO           ; start with shadow GPIO zeroed

;***** Main loop
flash
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   b'000010'       ; flip bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; write to GPIO
        movwf   sGPIO           ; and update shadow copy
        movlw   .50
        pagesel delay10
        call    delay10         ; delay 500ms -> 1Hz at 50% duty cycle

        pagesel flash
        goto    flash           ; repeat forever
```

HI-TECH C PRO or PICC-Lite

You might expect that to toggle GP1, you could use the statement:

```
GP1 = ~GP1;
```

Unfortunately, HI-TECH C doesn't support that, reporting "illegal operation on bit variable".

You can, however, use:

```
GP1 = !GP1;
```

This statement is also supported:

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

It works because bit variables, 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.

The HI-TECH compilers are able to translate either of these statements into an XOR instruction, as you would do in assembler. [To see what assembly code is produced for each C statement, use "View → Disassembly Listing" in MPLAB, or look at the listing file ("*.lst") in the project directory, after compilation.]

But since this statement reads and modifies GPIO, we'll use a shadow variable, which can be declared and initialised with:

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

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

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

Complete program

Here is how the HI-TECH C PRO 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 <htc.h>

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

// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRC);


void main()
{
    unsigned char   sGPIO = 0;  // shadow copy of GPIO
    unsigned char   dcnt;       // delay counter

    // Initialisation
    TRIS = 0b111101;            // configure GP1 (only) as an output

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

        // delay 500 ms
        for (dcnt = 0; dcnt < 5; dcnt++) {
            __delay_ms(100);
        }
    }   // repeat forever
}
```

Note that when using PICC-Lite, instead of using a loop to generate the 500 ms delay, you would write:

```
        DelayMs(250);           // delay 500ms
        DelayMs(250);
```

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

This is of course the same declaration as for HI-TECH C; not surprising, given that both are ANSI C compilers. Although with CCS PCB, we could also have declared this 8-bit variable as an 'int'. Since the HI-TECH and CCS compilers use different size 'int's, declaring 8-bit variables as 'char' is more portable.

Toggling the shadow copy of GP1 is also the same:

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

However, to write the result to GPIO, we need to use another built-in function:

```
output_b(sGPIO);        // write to GPIO ("port B")
```

The '`output_b()`' function writes a whole byte to port B. Similarly there is an '`output_a()`' which writes to port A, '`output_c()`' which writes to port C, etc.

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

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>

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

// Config: ext reset, no code protect, no watchdog, 4MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC


void main()
{
   unsigned char    sGPIO = 0;  // shadow copy of GPIO

   while (TRUE) {
        sGPIO ^= 0b000010;      // flip shadow bit corresponding to GP1
        output_b(sGPIO);        // write to GPIO ("port B")

        delay_ms(500);          // delay 500ms
   }                            // repeat forever
}
```

Comparisons

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	26	33	4	0.15%
HI-TECH PICC-Lite	12	60	10	1.6%
HI-TECH C PRO Lite	12	47	6	0.015%
CCS PCB	9	43	6	0.0076%

In this case, the assembler program in baseline lesson 3 called the delay routine as an external module – something a developer might keep in their library, after having written it once. So the assembler source code length given above reflects only the main program, including the external reference to the delay module, but not the source for the module itself. But the C sources are still much shorter.

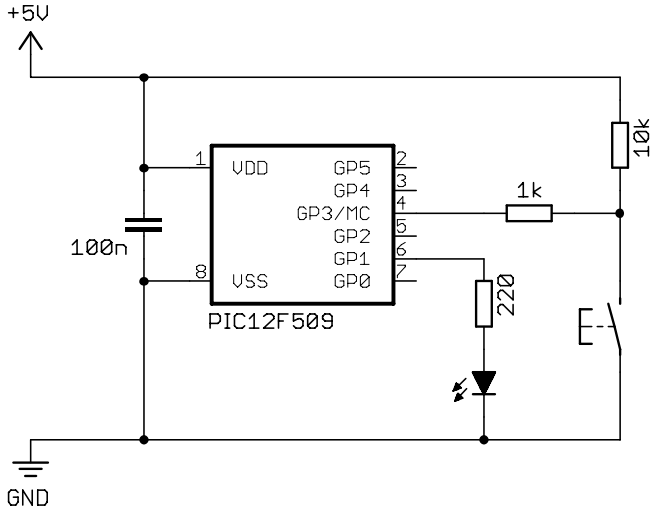
Example 4: Reading Digital Inputs

Baseline lesson 4 introduced digital inputs, using a pushbutton switch in a simple circuit (as shown on the right) to illustrate the principles involved.

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:

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

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

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

HI-TECH C PRO or PICC-Lite

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

```
GP1 = 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:

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

or

```
GP1 = 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 (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 = GP3 ? 0 : 0b000010;
```

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 HI-TECH C code to turn on an LED when a pushbutton is pressed:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 4                                 *
*                                                                       *
*   Demonstrates reading a switch                                       *
*                                                                       *
*   Turns on LED when pushbutton is pressed (active low)                *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 - indicator LED                                             *
*       GP3 - pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <htc.h>

// Config: int reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & WDTDIS & INTRC);


void main()
{
    TRIS = 0b111101;            // configure GP1 (only) as an output
```

```
    for (;;) {
        // set GP1 if GP3 is low (button pressed), else clear GP1
        GPIO = GP3 ? 0 : 0b000010;
    }   // repeat forever
}
```

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

CCS PCB

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

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

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

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

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

```
output_b(input(GP3) ? 0 : 0b000010);
```

Complete program

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

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 4                                 *
*                                                                       *
*   Demonstrates reading a switch                                       *
*                                                                       *
*   Turns on LED when pushbutton is pressed (active low)                *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 - indicator LED                                             *
*       GP3 - pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <12F509.h>

// Config: int reset, no code protect, no watchdog, 4MHz int clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC


void main()
{
    while (TRUE) {
        // set GP1 if GP3 is low (button pressed), else clear GP1
        output_b(input(GP3) ? 0 : 0b000010);
    }   // repeat forever
}
```

Again, 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)
Microchip MPASM	16	12	1
HI-TECH PICC-Lite	6	15	4
HI-TECH C PRO Lite	6	23	3
CCS PCB	5	22	4

It's clear that, at only 5 or 6 lines, the C source code is amazingly succinct – thanks mainly to the use of C's conditional expression (?:). And yet the HI-TECH PICC-Lite compiler in particular is able to generate code nearly as efficient in program memory use as hand-crafted assembly.

Example 5: Switch Debouncing

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

The problem was illustrated by an example application, using the circuit from example 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:

```
db_dn   ; wait until button pressed (GP3 low), debounce by counting:
        movlw   .13             ; max count = 10ms/768us = 13
        movwf   db_cnt
        clrf    dc1
dn_dly  incfsz  dc1,f           ; delay 256x3 = 768us.
        goto    dn_dly
        btfsc   GPIO,3          ; if button up (GP3 set),
        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.

HI-TECH C PRO or PICC-Lite

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

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

Whether you modify this 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 (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 (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 = (GP3 == 0) ? db_cnt+1 : 0;
```

However the previous version seems easier to understand.

Complete program

Here is the complete HI-TECH C PRO 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                                 *
*                                                                       *
*   Toggles LED when pushbutton is pressed (low) then released (high)   *
*   Uses counting algorithm to debounce switch                          *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       GP1 - indicator LED                                             *
*       GP3 - pushbutton switch                                         *
************************************************************************/
```

```
#include <htc.h>

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

// Config: int reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & WDTDIS & INTRC);


void main()
{
    unsigned char   sGPIO;          // shadow copy of GPIO
    unsigned char   db_cnt;         // debounce counter

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

    // Main loop
    for (;;) {
        // wait until button pressed (GP3 low), debounce by counting:
        for (db_cnt = 0; db_cnt <= 10; db_cnt++) {
            __delay_ms(1);          // sample every 1 ms
            if (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;          // flip shadow GP1
        GPIO = sGPIO;               // write to GPIO

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

    }   // repeat forever
}
```

Note that the PICC-Lite version would use '`DelayMs(1)`' instead of '`__delay_ms(1)`'.

CCS PCB

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

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

Complete program

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

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 1, example 5                                 *
```

```
*                                                                             *
*   Toggles LED when pushbutton is pressed (low) then released (high)         *
*   Uses counting algorithm to debounce switch                                *
*                                                                             *
*******************************************************************************
*                                                                             *
*   Pin assignments:                                                          *
*       GP1 - indicator LED                                                   *
*       GP3 - pushbutton switch                                               *
*                                                                             *
*******************************************************************************/

#include <12F509.h>
#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

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

// Config: int reset, no code protect, no watchdog, 4MHz int clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC


void main()
{
    unsigned char   sGPIO;          // shadow copy of GPIO
    unsigned char   db_cnt;         // debounce counter

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

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

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

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

    }   // repeat forever
}
```

As before, the processor configuration in both the HI-TECH and CCS programs 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 "toggle an LED when a pushbutton is pressed" programs:

Toggle_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	39	33	3
HI-TECH PICC-Lite	21	77	11
HI-TECH C PRO Lite	20	92	3
CCS PCB	19	73	6

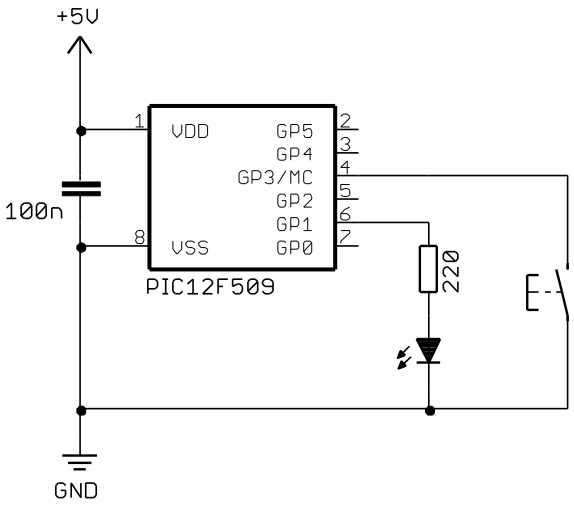
Note that in all three cases, the C source code is around half the length of the assembler source, while the (optimised) C programs use a little more than twice as much program memory as the assembler version.

Example 6: Internal (Weak) Pull-ups

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

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

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



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

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

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

```
movlw   b'10111111'     ; enable internal pull-ups
option
```

HI-TECH C PRO or PICC-Lite

To load the OPTION register in HI-TECH C, simply assign a value to the variable `OPTION`.

For example:

```
OPTION = 0b10111111;
```

But since the supplied include files define symbols to represent register bits, including those in OPTION, you can write:

```
OPTION = ~GPPU;
```

to turn off only the $\overline{\text{GPPU}}$ bit, making the code more readable.

To enable weak pull-ups in the "toggle an LED" program from example 5, simply add this "`OPTION =`" line to the initialisation code. For example:

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

CCS PCB

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

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

Note: use SETUP_COUNTERS on PCB parts

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

To figure this out, we need to go digging in the include files. The "12F509.h" header file includes the following lines:

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

And here, finally, is a clue.

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

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

Therefore, for the baseline PICs, the '`SETUP_COUNTERS()`' function also controls the weak pull-up and wake-up on change functions, in addition to setting timer and watchdog options.

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

For example:

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

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

For example:

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

To enable weak pull-ups in the "toggle an LED" program from example 5, add this '`SETUP_COUNTERS()`' line to the initialisation code. For example:

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

Comparisons

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

Toggle_LED+WPU

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	41	35	3
HI-TECH PICC-Lite	22	79	11
HI-TECH C PRO Lite	21	94	3
CCS PCB	20	81	6

Since we have only added a line (for C) or two (for assembler), the source code is barely longer than that in example 5. And the assembler and HI-TECH programs use only two more words of program memory, since they have added only the two instructions needed to load the OPTION register.

However, the CCS program has grown from 73 to 81 words long, because the '`SETUP_COUNTERS()`' function has added a number of instructions for setting up the timers, which are not actually needed in this application.

Summary

Overall, we have seen that, although HI-TECH and CCS take quite different approaches, basic digital I/O operations can be expressed succinctly using either C compiler, leading to significantly shorter source code, as illustrated by the code comparisons we have done in this lesson:

Source code (lines)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
Microchip MPASM	11	40	26	16	39	41
HI-TECH PICC-Lite	6	14	12	6	21	22
HI-TECH C PRO Lite	6	13	12	6	20	21
CCS PCB	5	9	9	5	19	20

The CCS source code is consistently shorter than that for HI-TECH C, reflecting the availability of built-in functions in CCS PCB which perform operations which may take a number of statements in HI-TECH C to accomplish. Whether this approach is better is a matter of personal style, although we saw in example 6 that the use of built-in functions does not necessarily make the code easier to follow; the operation of a built-in function may not always be clear.

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 HI-TECH C, 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.

However, we have also seen that both C compilers generate code which occupies significantly more program memory and uses more data memory than for corresponding hand-written assembler programs:

Program memory (words)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
Microchip MPASM	9	33	33	12	33	35
HI-TECH PICC-Lite	9	78	60	15	77	79
HI-TECH C PRO Lite	11	58	47	23	92	94
CCS PCB	12	45	43	22	73	81

Data memory (bytes)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
Microchip MPASM	0	3	4	1	3	3
HI-TECH PICC-Lite	4	9	10	4	11	11
HI-TECH C PRO Lite	2	6	6	3	3	3
CCS PCB	3	5	6	4	6	6

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 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 is the right choice.

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

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