Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital I/O

The Baseline PIC C Programming tutorial series demonstrated how to program baseline PICs (such as the 12F509 and 16F506) in C, to perform the tasks covered in the Baseline PIC Assembler lessons: from flashing LEDs and reading switches, through to implementing a simple light meter with smoothed output on a multiplexed 7-segment LED display. The baseline C programming series used "free" compilers from HI-TECH Software and Custom Computer Services (CCS) bundled with MPLAB, which fully support all current baseline PICs. We saw in that series that, although these C compilers were perfectly adequate for the simplest tasks, they faltered when it came to the more complex applications involving arrays, reflecting the difficulty of implementing a C compiler for the baseline PIC architecture.

This tutorial series revisits this material, along with other topics covered in the Midrange PIC Assembler lessons, using midrange devices such as the 12F629 and 16F690. It will become apparent that the midrange architecture is much more suitable for C programming than the baseline architecture, especially for applications which need to access more than one bank of data memory. But we will also see that, although it is often easier to program in C, in the sense that programs are shorter and more easily expressed, assembler remains the most effective way to make the most of the limited resources on these small devices, even for midrange PICs. Nevertheless, we will see that C is a very practical alternative for most applications.

Unfortunately, the CCS PCB compiler bundled with MPLAB (as of version 8.40) only supports baseline PICs, so for programming midrange PICs in C we will only use the "free" compilers from HI-TECH Software: PICC-Lite and HI-TECH C.

This lesson covers basic digital I/O: flashing LEDs, responding to and debouncing switches, as covered in lessons 1 to 3 of the midrange 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 C and PICC-Lite compilers
- Digital input and output
- Programmed delays
- Switch debouncing
- Using internal (weak) pull-ups

with examples for both 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 the HI-TECH C Compilers

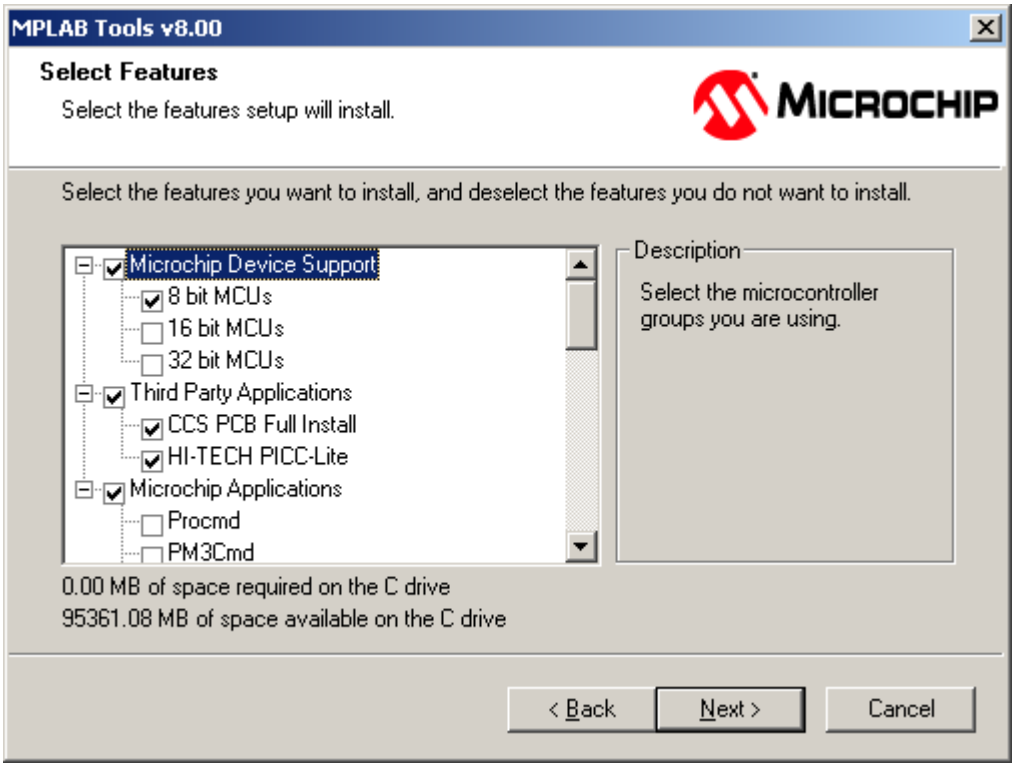
HI-TECH's "PICC-Lite" C compiler was bundled with version 8.10 of Microchip's MPLAB. It is free to use, integrated with MPLAB, fully supports the entire range of baseline (12-bit) PICs, but supports only a small number of midrange (14-bit) PICs. And for most of these "supported" midrange PICs, PICC-Lite limits the amount of data and program memory that can be used. Luckily the PIC12F629 used in this lesson is fully supported by PICC-Lite, with no memory restrictions. But to make full use of the other "supported" midrange PICs, such as the 16F690, you must upgrade (for a price!) to the full "HI-TECH C" compiler, which supports all baseline and midrange PICs, with no memory restrictions.

HI-TECH C can be used for free, when running in "Lite mode". In this mode, all compiler optimisation is turned off, making the generated code around twice the size of that generated by PICC-Lite. However, HI-TECH Software has (as of June 2009) retired PICC-Lite, and no longer supports it. Versions 8.15 and later of MPLAB have instead included HI-TECH C[1], giving 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 since we are using the 12F629 in this lesson, it is better to continue to use PICC-Lite (if you are able to locate a copy – it is no longer available on www.htsoft.com), since it will generate much more efficient code, while allowing all the memory on the 12F629 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.

Whichever HI-TECH compiler is bundled with the version of MPLAB you are using, you should select it as an option 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 7.

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, if all the defaults are chosen during the install process.

However it is worth downloading and installing the latest version (patch level) of the HI-TECH C compiler from www.htsoft.com, instead of using the version bundled with MPLAB. Although PICC-Lite will no longer be updated, HI-TECH will continue to find and correct bugs in HI-TECH C, so it's sensible to use a more recent version if you can.

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 HI-TECH C and, for comparison, the size of the same data types in CCS PCB:

Type	HI-TECH	CCS PCB
bit	1	-
char	8	8
short	16	1
int	16	8
long	32	16
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.

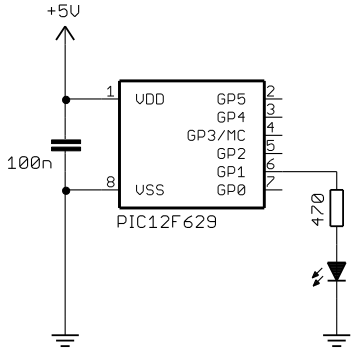
The "standard" 'int' type is 16 bits in HI-TECH C, but 8 bits in CCS PCB.

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.

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. 32 bits 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 midrange 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 PIC12F629.

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

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

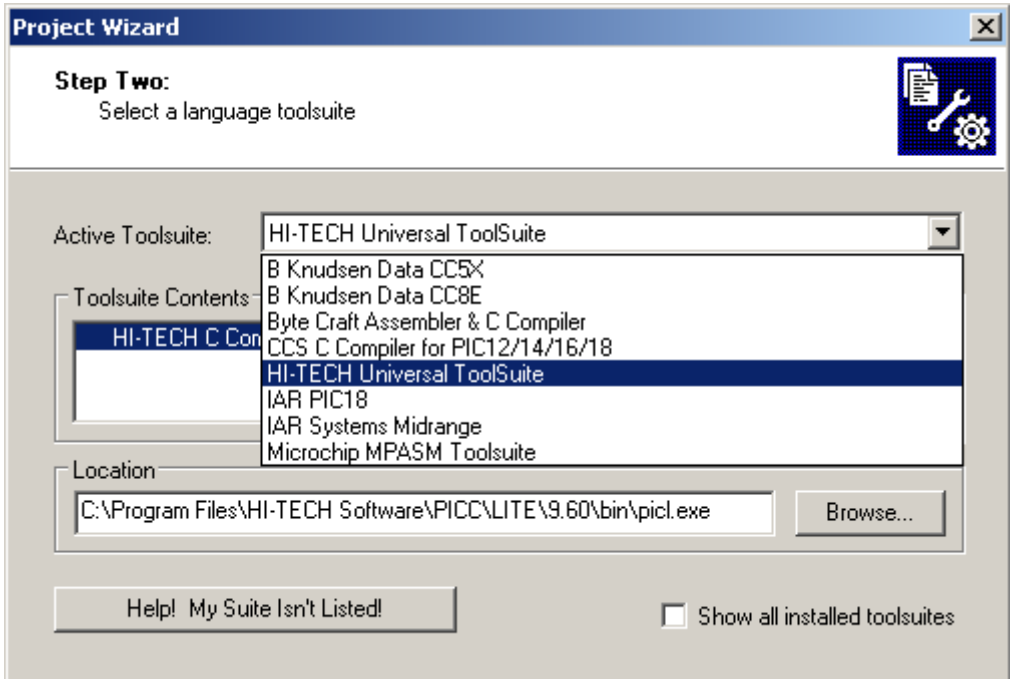
Finally, the end of the program consisted of an infinite loop ('`goto $`'), to leave the LED turned on.

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

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


RESET   CODE    0x0000          ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL

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

        ; turn on LED
        banksel GPIO
        bsf     GPIO,GP1        ; set GP1 high

        goto    $               ; loop forever
```

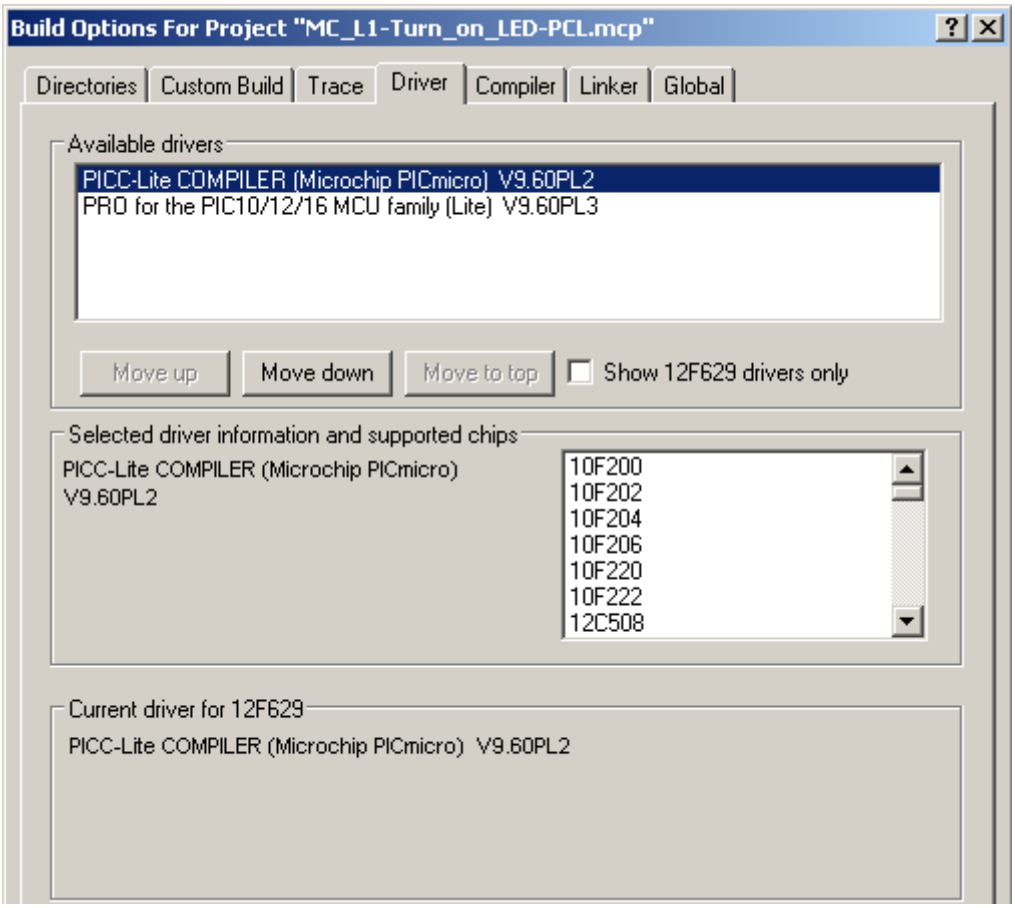
Creating a HI-TECH C Project

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 PRO 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:       MC_L1-Turn_on_LED-PCL.c                             *
*   Date:           5/8/08                                              *
*   File Version:   1.0                                                 *
*                                                                       *
*   Author:         David Meiklejohn                                    *
*   Company:        Gooligum Electronics                                *
```

```
*                                                                             *
*******************************************************************************
*                                                                             *
*   Architecture:  Midrange PIC                                               *
*   Processor:     12F629                                                     *
*   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 brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);
```

Note that the configuration symbols used are different to those defined in the MPASM include files. For example, '`MCLREN`' instead of '`_MCLRE_ON`', and '`INTIO`' instead of '`_INTRC_OSC_NOCLKOUT`'. 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 12F629), these symbols are defined in the "`pic126x.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 TRISIO register with 111101b, it is simply a matter of:

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

Alternatively this could be expressed as:

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

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

```
GPIO1 = 1;              // set GP1 high
```

Note that this is different from the 12F509 version of this program, presented in baseline C lesson 1, where GP1 was accessed through a variable called '`GP1`'. The HI-TECH C include files are inconsistent; some refer to the GPIO bits as '`GPn`', others as '`GPIOn`' (where '`n`' is the bit number). The only way to be sure of which variable names to use for a given PIC is to check the appropriate include file.

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
    ;
}
```

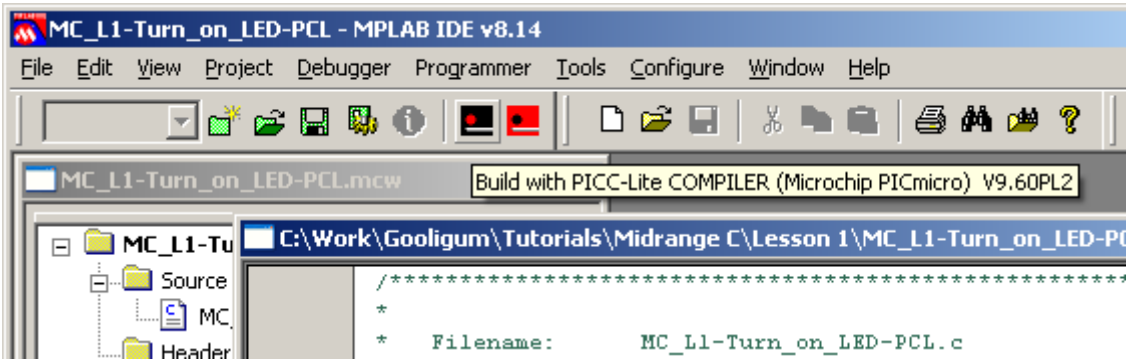
Complete program

Here is the complete code to turn on a 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 brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);
```

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

    GPIO1 = 1;              // set GP1 high

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

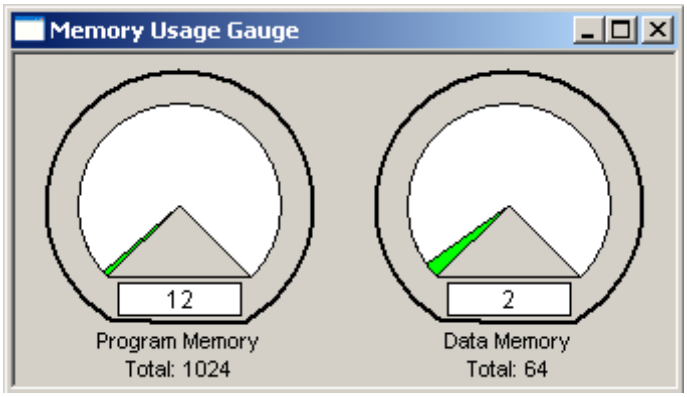
To compile the source code, click on the "Build project" icon (or simply press F10). This is equivalent to the assembler "Make" option, compiling all the source files which have changed, and linking the resulting object files and any library functions, creating an output '.hex' file, which is programmed into the PIC as normal:



Comparisons

It is apparent that this source code is much shorter than the assembler version, making it quicker to write and easier to understand. We don't need to bother with such details as loading OSCCAL and register banking; the C compiler takes care of that for us. And this program is almost identical to the 12F509 version, from baseline C lesson 1 – only the register variable names have changed ('`TRISIO`' instead of '`TRIS`', '`GPIO1`' instead of '`GP1`'), demonstrating that it is generally much easier to migrate C code between processors, than it is for assembler.

So is there any downside to using C? We can find out by checking the compiled program's resource usage, using the memory usage gauge available in MPLAB ("View" → "Memory Usage Gauge"):



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

It shows that 12 out of the 1024 available words of program memory have been used, and 2 bytes out of the 64 bytes of data memory available on the 12F629 has been allocated.

The table on the next page summarises the resource usage of our "Turn on a LED" assembler, PICC-Lite and HI-TECH C PRO[2] (in "Lite mode") programs, as well as the number of lines of source code (excluding

[2] Version 9.60PL3 of HI-TECH C PRO, current at the time of writing (August 2008), contains a bug which affects code generation for the 12F629, when run in Lite mode. A patch is available on request from HI-TECH Software.

comment, white-space and single-brace lines). The resource usage of the baseline (PIC12F509) versions of this example, from baseline C lesson 1, is also given for comparison:

Turn_on_LED

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	15	13	12	10	0	0
HI-TECH PICC-Lite	6	6	12	9	2	4
HI-TECH C PRO Lite	6	6	13	11	2	2

As expected, the assembler source code is much longer than the equivalent C code.

We'd normally expect that a hand-written assembler program would be smaller than that generated by C compiler, but for this simple example, the PICC-Lite compiler manages to generate code the same size as the assembler version.

Although 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.

Also note that the 12F629 versions are all a little larger than the corresponding 12F509 version, reflecting the need for bank selection instructions for special function register access in the midrange architecture.

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

In midrange lesson 1, we used the same circuit as above, but made the LED flash by toggling the GP1 output. The delay was created by an in-line busy-wait loop.

Midrange lesson 2 showed how to move the delay loop into a subroutine, and to generalise it, so that the delay is passed as a parameter to the routine, in W. This was demonstrated by a program which flashed the LED at 1 Hz, with a duty cycle of 20%, by turning it on for 200 ms and then off for 800 ms, before repeating.

Here is the main loop from the assembler code from midrange lesson 2:

```
flash
        movlw   1<<GP1          ; turn on LED
        movwf   GPIO
        movlw   .20             ; stay on for 0.2s:
        call    delay10         ;   delay 20 x 10ms = 200ms
        clrf    GPIO            ; turn off LED
        movlw   .80             ; stay off for 0.8s:
        call    delay10         ;   delay 80 x 10ms = 800ms

        goto    flash           ; repeat forever
```

HI-TECH PICC-Lite

We saw above how to turn the LED on, with:

```
        GPIO1 = 1;              // turn on LED on GP1
```

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

```
        GPIO1 = 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 between 0 and 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 a 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 brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);
```

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

    // Main loop
    for (;;) {
        // turn on LED on GP1
        GPIO1 = 1;

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

        // turn off LED on GP1
        GPIO1 = 0;

        // stay off for 800ms
        DelayMs(250);
        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, two function (or macro) calls are needed to create the initial 200 ms delay:

```
        // 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 if you have more than a few 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
        GPIO1 = 1;

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

        // turn off LED on GP1
        GPIO1 = 0;

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

Complete program

Here is the complete HI-TECH C PRO version of the "flash a LED with 20% duty cycle" program:

```
/************************************************************************
*                                                                       *
*   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  4000000     // oscillator frequency for _delay()

// Config: ext reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);


void main()
{
    unsigned char   dcnt;       // delay counter
```

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

    // Main loop
    for (;;) {
        // turn on LED on GP1
        GPIO1 = 1;

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

        // turn off LED on GP1
        GPIO1 = 0;

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

Comparisons

In addition to the source code length and memory usage, we can also consider 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)	
	12F629	12F509	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	35	42	29	34	3	3	0.15%	0.15%
HI-TECH PICC-Lite	14	14	40	78	5	9	1.6%	1.6%
HI-TECH C PRO Lite	13	13	60	58	5	6	0.011%	0.011%

The assembler code presented in midrange lesson 2 included a delay subroutine, adding 11 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 (unlike PICC-Lite); we had to write the delay routine from scratch.

The C source is again significantly shorter than the assembler source, but we are now starting to see that the assembler version is more efficient (using less program and data memory) than even the optimised code generated by PICC-Lite – although in this case the un-optimised HI-TECH C PRO in "Lite" mode actually generated smaller code than PICC-Lite – probably because the use of an efficient built-in delay function.

We can also see that PICC-Lite is able to generate much more efficient code for the 12F629 than the 12F509 in this example. This is because the program includes several function calls; it is difficult for C compilers to implement function calls efficiently on baseline devices, which have a stack only two levels deep. The midrange architecture has an eight-level stack, making it practical for the compiler to generate simple `call` instructions, instead of the elaborate jump-table system that PICC-Lite implements on baseline devices.

And finally, the HI-TECH C PRO compiler's built-in delay function is amazingly accurate!

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

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

The read-modify-write problem revisited

As discussed in that lesson, any operation which reads an output (or part-output) port, modifies the value read, and then writes it back, can lead to unexpected results. This is because, when a port is read, it is the value at the pins that is read, not necessarily the value that was written to the output latches. And that's a problem if, for example, you have written a '1' to an output pin, which, because it is being externally loaded (or, more usually, it hasn't finished going high yet, because of a 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 '`GPIO1 = 1`' in HI-TECH C, the compiler translates those statements into corresponding bit set or clear instructions, which may lead to read-modify-write problems.

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

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

But you should be aware that a sequence such as:

```
GPIO1 = 1;
GPIO2 = 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 midrange lesson 2:

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

        clrf    sGPIO           ; start with shadow GPIO zeroed

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

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

        ; repeat forever
        pagesel flash
        goto    flash
```

HI-TECH C Implementation

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

```
GPIO1 = ~GPIO1;
```

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

You can, however, use:

```
GPIO1 = !GPIO1;
```

This statement is also supported:

```
GPIO1 = GPIO1 ? 0 : 1;
```

It works because bit variables, such as `GPIO1`, 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 ^= 1<<1;          // flip shadow bit corresponding to GP1
        GPIO = sGPIO;           // write to GPIO
```

Complete program

Here is how the HI-TECH C PRO code to flash a 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 brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);


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

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

    // Main loop
    for (;;) {
        // toggle GP1
        sGPIO ^= 1<<1;          // 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);
```

Comparisons

Here is the resource usage and accuracy summary for the "Flash a LED at 50% duty cycle" programs:

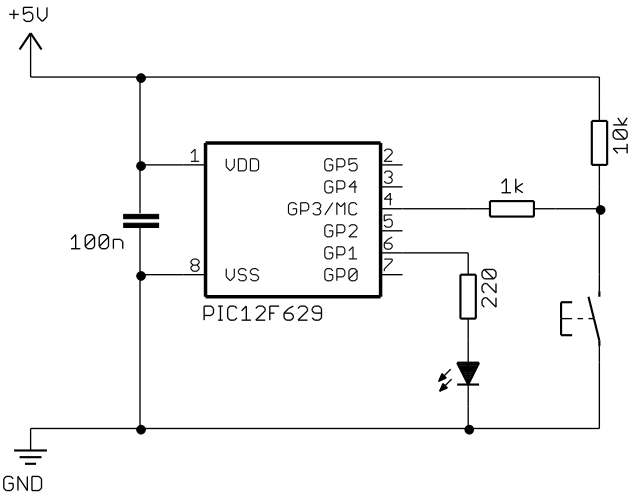
Flash_LED-50p

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)		Delay accuracy (timing error)	
	12F629	12F509	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	26	28	30	34	4	4	0.15%	0.15%
HI-TECH PICC-Lite	12	12	39	60	6	10	1.6%	1.6%
HI-TECH C PRO Lite	12	12	49	47	6	6	0.015%	0.015%

The pattern here is the same as in the last example: the C source code is much shorter, but the assembler version is more efficient. And the PICC-Lite compiler is again able to generate much more efficient code for the 12F629 than the 12F509, because the midrange architecture better supports C programming.

Example 4: Reading Digital Inputs

Midrange lesson 3 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:

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

        banksel GPIO            ; select bank for GPIO access
loop
        clrf    sGPIO           ; assume button up -> LED off
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     sGPIO,GP1       ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

HI-TECH C Implementation

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

```
        GPIO1 = GPIO3;          // 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:

```
        GPIO1 = !GPIO3;         // copy !GP3 to GP1
```

or

```
        GPIO1 = GPIO3 ? 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 (GPIO3 == 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 = GPIO3 ? 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 a 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 brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);

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

```
    // Main loop
    for (;;) {
        // set GP1 if GP3 is low (button pressed), else clear GP1
        GPIO = GPIO3 ? 0 : 1<<1;

    }   // 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.

Comparisons

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

PB_LED

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	21	17	16	13	1	1
HI-TECH PICC-Lite	6	6	16	15	2	4
HI-TECH C PRO Lite	6	6	25	23	3	3

At only six 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 is able to generate code as small as the hand-crafted assembly version.

Example 5: Switch Debouncing

Midrange lesson 3 included a discussion of the switch contact bounce problem, and various hardware and software approaches to addressing it.

The problem was illustrated by an example application, using the circuit from example 4 (above), where the LED is toggled each time the pushbutton is pressed. If the switch is not debounced, the LED toggles on every contact bounce, making it difficult to control.

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

The algorithm was expressed in pseudo-code as:

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

It was implemented in assembler as follows:

```
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,GP3        ; 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 Implementation

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 (GPIO3 == 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 (GPIO3 == 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 (GPIO3 == 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 = (GPIO3 == 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 a 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 brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);


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
    TRISIO = ~(1<<1);               // 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 (GPIO3 == 1)         // if button up (GP3 high)
                db_cnt = 0;         //   restart count
        }                           // until button down for 10 successive reads

        // toggle LED on GP1
        sGPIO ^= 1<<1;              // 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 (GPIO3 == 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)`'.

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)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	42	41	35	34	3	3
HI-TECH PICC-Lite	21	21	64	77	7	11
HI-TECH C PRO Lite	20	20	94	92	5	3

Note that the C source code is around half the length of the assembler source, while even the (optimised) PICC-Lite program uses around twice as much program memory as the assembler version.

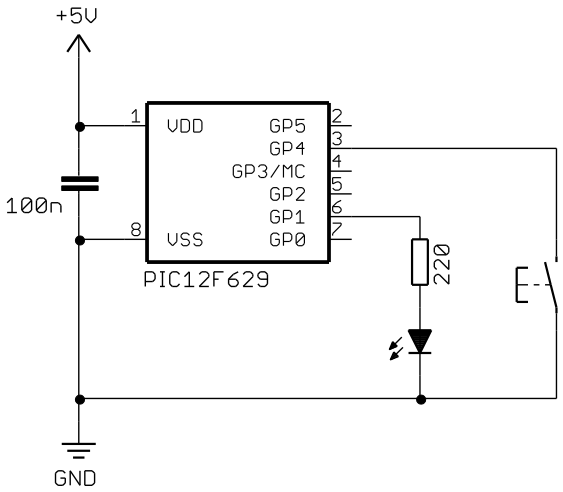
Example 6: Internal (Weak) Pull-ups

As discussed in midrange lesson 3, many PICs include internal "weak pull-ups", which can be used to pull floating inputs (such as an open switch) high.

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

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

Unfortunately, one the 12F629, internal pull-ups are available on every I/O pin *except* GP3, so we need to choose another pin to demonstrate their use – GP4 in this example, as shown in the circuit diagram.



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

In the example assembler program from midrange lesson 3, this was done by:

```
bcf     OPTION_REG,NOT_GPPU     ; enable global pull-ups
```

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

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

This was done, in the assembler example in midrange lesson 3, by:

```
movlw   1<<GP4          ; select pull-up on GP4 only
movwf   WPU
```

HI-TECH C Implementation

The HI-TECH C compilers make the individual bits in the OPTION register available as single-bit variables, so to clear $\overline{\text{GPPU}}$, without affecting any of the other OPTION bits, we can simply write:

```
GPPU = 0;                       // enable global pull-ups
```

Note again you should look at the include file for your PIC ("`pic126x.h`" in this case) to check the name of the variable associated with the register bit you wish to access. It is not necessarily the same as the symbol defined in the assembler include file – for example, the HI-TECH include file defines the variable '`GPPU`', while the MPASM include file defines the symbol '`NOT_GPPU`', both referring to the $\overline{\text{GPPU}}$ bit.

Similarly, the WPU register is accessible through a variable named '`WPU`', so to select the pull-up on GP4 we can write:

```
WPU = 1<<4;                     // select pull-up on GP4 only
```

With these additions, the initialisation code then becomes:

```
// Initialisation
GPPU = 0;                       // enable global pull-ups
WPU = 1<<4;                     // select pull-up on GP4 only
GPIO = 0;                       // start with LED off
sGPIO = 0;                      //   update shadow
TRISIO = ~(1<<1);               // configure GP1 (only) as an output
```

The rest of the program, whether for PICC-Lite or HI-TECH C PRO, is then the same as before (example 5, above).

Comparisons

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

Toggle_LED+WPU

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	47	43	40	36	3	3
HI-TECH PICC-Lite	23	22	68	79	7	11
HI-TECH C PRO Lite	22	21	99	94	5	3

Since we have only added a couple of lines, the source code is barely longer than that in example 5. And the assembler and HI-TECH C programs use only four or five more words of program memory, since they have added only the few instructions needed to clear the $\overline{\text{GPPU}}$ bit and load WPU.

Summary

Overall, we have seen that basic digital I/O operations can be expressed succinctly using C, 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	15	35	26	21	42	47
HI-TECH PICC-Lite	6	14	12	6	21	23
HI-TECH C PRO Lite	6	13	12	6	20	22

The C code is typically only half, or less, as long as the corresponding assembler source code.

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

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

However, 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	12	29	30	16	35	40
HI-TECH PICC-Lite	12	40	39	16	64	68
HI-TECH C PRO Lite	13	60	49	25	94	99

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	2	5	6	2	7	7
HI-TECH C PRO Lite	2	5	6	3	5	5

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

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