Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 4: Interrupt-on-change, Sleep Mode and the Watchdog Timer

This lesson revisits material from midrange lesson 7, looking at the midrange PIC architecture's power-saving sleep mode, its ability to generate interrupts and/or wake from sleep when an input changes state and the watchdog timer – generally used to automatically restart a crashed program, but also useful for periodically waking the PIC from sleep, for low-power operation.

One again, selected examples from that lesson are re-implemented using "free" C compilers from HI-TECH Software: PICC-Lite and HI-TECH C[1] (in "Lite" mode), introduced in lesson 1.

In summary, this lesson covers:

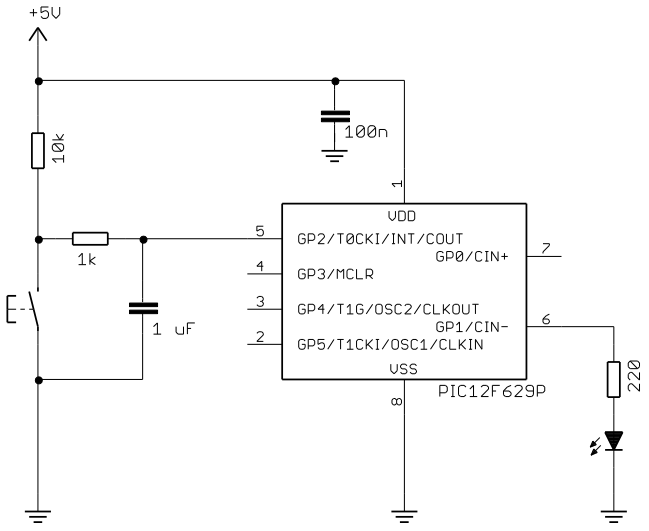
- Interrupt-on-change
- Sleep mode (power down)
- Wake-up on change (power up on input change)
- The watchdog timer
- Periodic wake from sleep

with examples for HI-TECH C and PICC-Lite.

Interrupt-on-change

As we saw in midrange lesson 7, midrange PICs provide a port change interrupt facility, which, on the 12F629, is available on every GPIO pin.

This feature is similar to the external interrupt facility covered in lesson 3, except that a port change interrupt will be triggered by any change (not just one type of transition) on any of the pins for which it is enabled. This makes it more flexible (being available on more pins), but also more difficult to deal with correctly, as we shall see in the examples in this section.



The first example uses the circuit on the right to demonstrate how to use interrupt-on-change to respond to a single, externally debounced input.

GP2 is used here because, on the 12F629, it has a Schmitt-trigger input, allowing the simple RC

[1] PICC-Lite was bundled with versions of MPLAB up to 8.10. HI-TECH C (earlier known as "HI-TECH C PRO") was bundled with MPLAB 8.15 and later, although you should download the latest version from www.htsoft.com.

filter to provide effective hardware debouncing, as explained in baseline lesson 4.

To enable a pin for interrupt-on-change, the corresponding bit must be set in the IOC register. This was done in midrange lesson 7 by:

```
banksel IOC             ; enable interrupt-on-change
bsf     IOC,nBUTTON     ;   on pushbutton input
```

(where '`nBUTTON`' is a constant which has been set to '2')

Before actually enabling port change interrupt, it is necessary to either read or write to the port to clear any existing *mismatch condition*, to prevent any false triggering. The port change interrupt can then enabled by setting the GPIE bit in the INTCON register.

This was done in assembler by:

```
banksel GPIO            ;   (write to GPIO will clear any mismatch)
clrf    GPIO            ; start with all LEDs off
clrf    sGPIO           ;   update shadow
; configure interrupts
movlw   1<<GIE|1<<GPIE  ; enable port change and global interrupts
movwf   INTCON
```

(note that global interrupts are also being enabled here, by setting the GIE bit)

In the interrupt handler, we must clear the port mismatch condition which triggered this interrupt. In midrange lesson 7 this was done reading the port. And (as for all interrupts), we must also clear the corresponding interrupt flag, GPIF:

```
banksel GPIO
movf    GPIO,w              ; clear mismatch condition
bcf     INTCON,GPIF         ; clear interrupt flag
```

Since we want to toggle the LED on GP1 each time the pushbutton is pressed, but not when it is released, we need to check whether the switch is up or down (this is different from the situation with external interrupts, which are only triggered on one type of transition).

This was implemented in assembler as:

```
; toggle LED only on button press
btfsc   GPIO,nBUTTON        ; is button down?
goto    isr_end
movlw   1<<nB_LED           ; if so, toggle indicator LED
xorwf   sGPIO,f             ;   using shadow register
```

(where '`nB_LED`' is a constant which has been set to '1')

The shadow register was copied to GPIO in the main loop, as in the earlier examples.

HI-TECH C implementation

Implementing these steps using HI-TECH C is quite straightforward, using techniques we have seen before.

Firstly, to enable interrupt-on-change on GP2:

```
IOCB |= 1<<nBUTTON;         // enable IOC on pushbutton input
```

(where '`nBUTTON`' is a symbol which has been defined as the numeric constant '2')

Note that HI-TECH C refers to the IOC register as '`IOCB`'.

You should always check what names (symbols) the compiler uses for registers and bits, because, as in this case, it can be different from the "official" name used by Microchip in the device's data sheet. These symbolic names are defined in the include, or header (*.h), files located in the "`include`" directory within the compiler installation directory. In this case, for both the PICC-Lite and HI-TECH C PRO compilers, the symbols for the PIC12F629 are defined in the '`pic12f6x.h`' file.

If you look in this header file, you will see that symbols are defined for individual bits in most registers. So we could for example have equivalently written:

```
IOCB2 = 1;                  // enable IOC on pushbutton input
```

This approach may seem clearer, but it has the disadvantage that, if the pushbutton is ever moved to a different pin, you would need to find all the lines of code affected by the change, including ones like this, where it may not be obvious that it relates to GP2. The first form depends only on a single definition of the symbol '`nBUTTON`' at the start of the program, and changing that definition will "automatically" update all the instructions which reference it.

There is no code size penalty for using either form; the HI-TECH C compilers are clever enough to generate the same code (a simple '`bsf`' instruction) in both cases. The choice is purely a matter of personal style.

Next, we enable the port change and global interrupts:

```
// configure interrupts
GPIE = 1;                   // enable port change interrupt
ei();                       // enable global interrupts
```

In the interrupt handler, it is best to explicitly clear the mismatch condition (by reading or writing GPIO) at the start of the routine, instead of relying on this occurring as a side-effect of statements in the body of the handler, which may be changed later.

The problem is that there is no way to simply read a register (or variable) with 'C'. Most statements do something, and there is a danger that any "do nothing" statements will be optimised out of the generated code.

However, if we use the form:

```
GPIO;                           // read GPIO to clear mismatch condition
```

which is an expression which evaluates to the value of the contents of GPIO, but does nothing, both HI-TECH C compilers appear to recognise what we're trying to do, and instead of optimising this statement out, they generate a '`movf GPIO,w`' instruction to read GPIO, which is exactly what's wanted.

We must also clear the port interrupt flag, to indicate that this interrupt has been serviced:

```
GPIF = 0;                       // clear interrupt flag
```

Finally, we need to check the status of the pushbutton, and toggle the LED only if the button is pressed (meaning that GP2 is low):

```
// toggle LED only on button press
if (!(GPIO & 1<<nBUTTON))       // if button is down
{
    sGPIO ^= 1<<nB_LED;         //   toggle LED using shadow register
}
```

Again, although '`if (!GPIO2)`' might have been clearer, using an expression involving the '`nBUTTON`' symbol makes the code more maintainable. The HI-TECH C compilers recognise that both forms are simple

bit tests, and generate an appropriate '`btfsc`' instruction in each case, so there is no code size penalty either way; this continues to be a matter of personal style.

Complete program

Here is how these code fragments fit into a working program:

```
/************************************************************************
*   Description:    Lesson 4, example 1                                 *
*                                                                       *
*   Demonstrates use of interrupt-on-change interrupts                  *
*   (without software debouncing)                                       *
*                                                                       *
*   Toggles LED on GP1                                                  *
*   when pushbutton on GP2 is pressed (high -> low transition)          *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       GP1 - indicator LED                                             *
*       GP2 - pushbutton (externally debounced, active low)             *
*                                                                       *
************************************************************************/

#include <htc.h>

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

// Pin assignments
#define nB_LED  1               // indicator LED on GP1
#define nBUTTON 2               // externally debounced pushbutton on GP2


/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    TRISIO = ~(1<<nB_LED);      // configure LED pin (only) as output
    IOCB |= 1<<nBUTTON;         // enable IOC on pushbutton input
    GPIO = 0;                   // start with all LEDs off
                                //   (write to GPIO will clear any IOC mismatch)
    sGPIO = 0;                  //   update shadow

    // configure interrupts
    GPIE = 1;                   // enable port change interrupt
    ei();                       // enable global interrupts

    // Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO;

    }   // repeat forever
}
```

```
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    // Service interrupt-on-change
    //   Triggered on any transition on IOC-enabled input pin
    //   caused by externally debounced pushbutton press
    //
    GPIO;                           // read GPIO to clear mismatch condition
    GPIF = 0;                       // clear interrupt flag

    // toggle LED only on button press
    if (!(GPIO & 1<<nBUTTON))       // if button is down
    {
        sGPIO ^= 1<<nB_LED;         //   toggle LED using shadow register
    }
}
```

Comparisons

Here is the source code length (excluding comments and white space) and resource (memory) use for this example by both HI-TECH C compilers and the corresponding assembler program from midrange lesson 7:

Toggle_LED-IOC

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	50	36	3
HI-TECH PICC-Lite	19	37	5
HI-TECH C PRO Lite	19	72	7

The C source code is less than half as long as the assembler source, reflecting the extent to which HI-TECH C is able to make some of the details of implementing interrupts (saving and restoring context, reset code jumping past the interrupt vector) transparent, as well as the succinctness of C expressions and statements.

The code generated by the PICC-Lite compiler, with optimisation enabled, is barely any larger than the hand-written assembly version, demonstrating again that C can be almost as efficient as assembler.

As expected, HI-TECH C PRO generated much larger code, because it is running in “Lite” mode, which does not perform any optimisation.

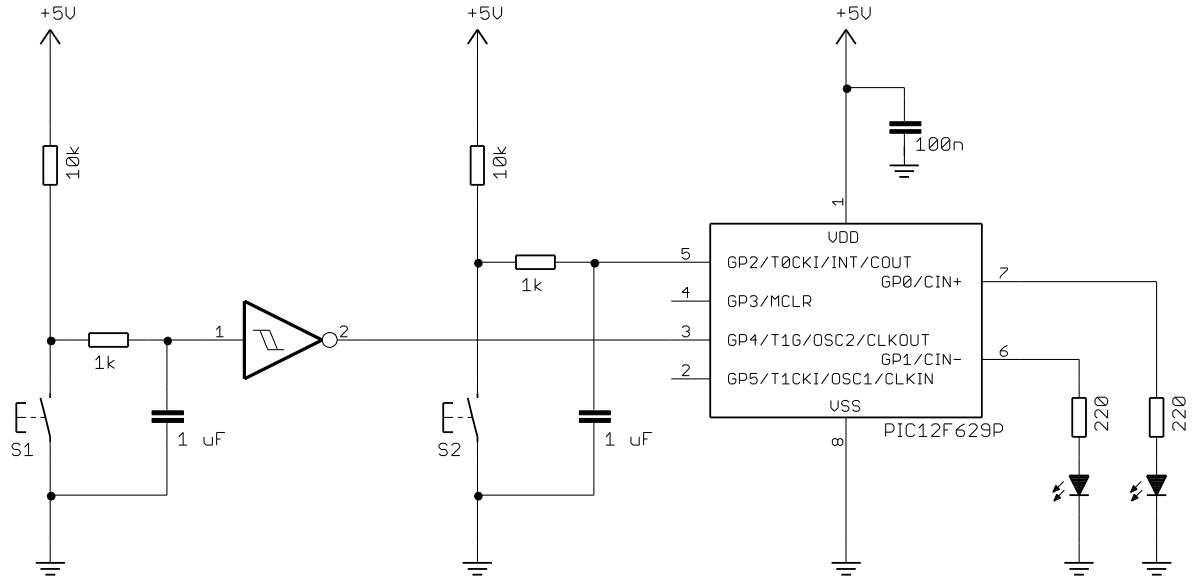
Example 2: Interrupt-on-change (multiple inputs)

This example demonstrates how to handle the situation where interrupt-on-change is enabled on more than one input pin.

The basic difficulty with handling this situation is that there are no flags to indicate which input has changed; the GPIF flag can tell you that at least one pin enabled for IOC has happened, but not which pin, or pins, it was.

So when a port change interrupt occurs, we need to deduce which pin(s) have changed, by reading GPIO and comparing the current state to the last recorded state. That means that the ISR (where the port change interrupt is handled) needs to keep track of the state of GPIO, and update that “last state” record, every time a change is detected, to be ready for the next time.

We'll use the circuit from the corresponding example in midrange lesson 7, shown below:



Each pushbutton toggles an LED: S1 controls the LED on GP1, and S2 controls the LED on GP0.

To simplify the software, both buttons are externally debounced, and since the only Schmitt-trigger GP input on the 12F629 is GP2, an external Schmitt-trigger inverter (such as a 74HC14) is used to drive GP4.

Thus, the operation of S1 is inverted, with respect to S2; the software will have to take this difference into account.

HI-TECH C implementation

As in the last example, to make the code more maintainable, it is good practice to define symbols represent the pins being used:

```
// Pin assignments
#define nB1_LED 0           // "button 1 pressed" indicator LED on GP0
#define nB2_LED 1           // "button 2 pressed" indicator LED on GP1
#define nPB1    2           // pushbutton 1 (ext debounce, active low) on GP2
#define nPB2    4           // pushbutton 2 (ext debounce, active high) on GP4
```

Given that we must keep track of the "last state" of GPIO, to compare with the current state when a port change is detected, and that this state will need to be initialised in the main code, but accessed and updated in the ISR, we need to declare it as a global variable (along with the shadow copy of GPIO, which is also accessed in both the ISR and the main code):

```
/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO
unsigned char   lGPIO;          // last state of GPIO (for change detection)
```

The two LED pins have to be configured as outputs, and interrupt-on-change has to be enabled on the two pushbutton inputs, so in the initialisation routine at the start of the main code, we have:

```
    // setup ports
    TRISIO = ~(1<<nB1_LED|1<<nB2_LED);  // configure LED pins as outputs
    IOCB |= 1<<nPB1|1<<nPB2;            // enable IOC on pushbuttons 1 and 2
```

As before, we start with both LEDs off:

```
    GPIO = 0;                   // start with all LEDs off
    sGPIO = 0;                  //   update shadow
```

To accurately detect pin changes, it is important to insure that the "last state" variable matches the current state of GPIO, before the port change interrupt is enabled, so we have:

```
    lGPIO = GPIO;               // update last state, for pin change detection
                                //   (GPIO read will clear any IOC mismatch)
```

Why bother reading GPIO, when we just cleared it?

Why not just write:

```
    GPIO = 0;                   // start with all LEDs off
    sGPIO = 0;                  //   update shadow
    lGPIO = 0;                  //   and last state (NOTE: THIS WILL NOT WORK!)
```

This approach will not, in general, work, because the value read from an input pin depends on the external signal applied to the pin; if an input pin is being held high externally, clearing the port register (GPIO) won't have any effect – it will still read as a '1'.

So the only way to be sure of the current state of GPIO is to read it.

Note that a useful side-effect of reading GPIO is that is clears any existing IOC mismatch condition, so we can now go ahead and enable the port change interrupt:

```
    // configure interrupts
    GPIE = 1;                   // enable port change interrupt
    ei();                       // enable global interrupts
```

As usual, the main loop does nothing more than continually update GPIO from the shadow register:

```
    // Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO;
    }
```

Meanwhile, the ISR updates the shadow copy of GPIO, whenever a port change occurs (triggering an interrupt).

Within the ISR, it is best to take a "snapshot" of the current state of GPIO, and use this to determine which pins have changed, instead of referring back continually to GPIO itself, in case an input changes while the interrupt handler is running (leading to inconsistent results).

So we declare a variable within the ISR function, so hold this current state:

```
void interrupt isr(void)
{
    unsigned char   cGPIO;      // current state of GPIO (used by IOC handler)

    // IOC handler goes here…
}
```

When servicing the port change interrupt, we begin by clearing the interrupt flag, as usual:

```
    GPIF = 0;                       // clear interrupt flag
```

There is no need to explicitly clear the IOC mismatch here, because GPIO is read in the very next statement:

```
    cGPIO = GPIO;                   // save current GPIO state
```

Next we need to determine which pin(s) have changed, by comparing the current state of GPIO with the last recorded state.

This can be done by XORing the current and last states. Since an XOR results in a '1' only where the inputs differ, the result will be all '0's, except for those bits corresponding to any pins which have changed.

We could write this as:

```
changes = lGPIO ^ cGPIO     // XOR current with last state to detect changes
```

but there is actually no need to introduce another variable; the only time we need to reference the last state (`lGPIO`) is here, to deduce which pins have changed, and, having done so, there is no need to refer back to `lGPIO` again, until it is updated at the end of the ISR.

So, to save data memory, it is possible to write the XOR result back to `lGPIO` with:

```
lGPIO ^= cGPIO;             // XOR current with last state to detect changes
```

The `lGPIO` variable will now contain '1's only in bit positions where the current state differs from the last state, corresponding to pins that have changed.

We can then use this to check whether each pushbutton input has changed, for example:

```
if (lGPIO & 1<<nPB1)            // if button 1 changed
{
    // handle button 1 input change…
}
```

But since we only want to toggle the LED when the pushbutton has pressed, we must check not only that the pushbutton input has changed, but that the button is down, which we can do with nested `if` statements:

```
// toggle LED 1 only on button 1 press (active low)
if (lGPIO & 1<<nPB1)            // if button 1 changed
    if (!(cGPIO & 1<<nPB1))     //  and if button 1 is down (low)
    {
        sGPIO ^= 1<<nB1_LED;    //   toggle LED 1 using shadow register
    }
```

Alternatively, this can be written as a single `if` statement, using a logical AND expression:

```
// toggle LED 1 only on button 1 press (active low)
if ((lGPIO & 1<<nPB1)           // if button 1 changed
    && (!(cGPIO & 1<<nPB1)))    //  and button 1 is down (low)
    {
        sGPIO ^= 1<<nB1_LED;    //   toggle LED 1 using shadow register
    }
```

Either form is acceptable; both generate the same (efficient) code, so which you use only a question of personal programming style.

Looking at these logical expressions, you may conclude that it would also be possible to replace the logical AND ('`&&`') with a bitwise AND ('`&`') and condense the expression to:

```
if (lGPIO & cGPIO & 1<<nPB1)    // if button 1 changed and down
{
    sGPIO ^= 1<<nB1_LED;        //   toggle LED 1 using shadow register
}
```

However, although this works, in terms of program logic (the expressions are, after all, logically the same), it is less clear and generates less efficient code. This is a case where writing more obscure code is counter-productive – it's simply a bad idea.

We can then write a very similar construct for the second pushbutton, but with the logic for testing "button down" inverted because this signal is active high, not low:

```
// toggle LED 2 only on button 2 press (active high)
if ((lGPIO & 1<<nPB2)           // if button 2 changed
    && (cGPIO & 1<<nPB2))       //  and button 2 is down (high)
    {
        sGPIO ^= 1<<nB2_LED;    //   toggle LED 2 using shadow register
    }
```

Finally, before exiting the interrupt handler, we must save the current state of GPIO as the new "last state", so that the next input change can be properly detected:

```
// update last GPIO state (for next time)
lGPIO = cGPIO;                  // new "last state" = current
```

Complete program

Here is how these code fragments fit together, to form the complete "interrupt-on-change with multiple inputs" example program:

```
************************************************************************
*                                                                      *
*   Description:    Lesson 4, example 2                                *
*                                                                      *
*   Demonstrates use of interrupt-on-change interrupts                 *
*   (without software debouncing)                                      *
*                                                                      *
*   Toggles LED on GP0 when pushbutton on GP2 is pressed               *
*   (high -> low transition)                                           *
*   and LED on GP1 when pushbutton on GP4 is pressed                   *
*   (low -> high transition)                                           *
*                                                                      *
************************************************************************
*                                                                      *
*   Pin assignments:                                                   *
*       GP0 - indicator LED 1                                          *
*       GP1 - indicator LED 2                                          *
*       GP2 - pushbutton 1 (externally debounced, active low)          *
*       GP4 - pushbutton 2 (externally debounced, active high)         *
*                                                                      *
************************************************************************/

#include <htc.h>

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

// Pin assignments
#define nB1_LED 0           // "button 1 pressed" indicator LED on GP0
#define nB2_LED 1           // "button 2 pressed" indicator LED on GP1
#define nPB1    2           // pushbutton 1 (ext debounce, active low) on GP2
#define nPB2    4           // pushbutton 2 (ext debounce, active high) on GP4


/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO
unsigned char   lGPIO;          // last state of GPIO (for change detection)
```

```
/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // configure and initialise ports
    TRISIO = ~(1<<nB1_LED|1<<nB2_LED);  // configure LED pins as outputs
    IOCB |= 1<<nPB1|1<<nPB2;            // enable IOC on pushbuttons 1 and 2
    GPIO = 0;                   // start with all LEDs off
    sGPIO = 0;                  //   update shadow
    lGPIO = GPIO;               // update last state, for pin change detection
                                //   (GPIO read will clear any IOC mismatch)

    // configure interrupts
    GPIE = 1;                   // enable port change interrupt
    ei();                       // enable global interrupts

    // Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO;

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    unsigned char   cGPIO;      // current state of GPIO (used by IOC handler)

    // Service interrupt-on-change
    //   Triggered on any transition on IOC-enabled input pin
    //   caused by externally debounced pushbutton press
    //
    GPIF = 0;                       // clear interrupt flag

    // determine which pins have changed
    cGPIO = GPIO;                   // save current GPIO state
                                    //   (GPIO read clears mismatch condition)
    lGPIO ^= cGPIO;                 // XOR current with last to detect changes

    // toggle LED 1 only on button 1 press (active low)
    if ((lGPIO & 1<<nPB1)           // if button 1 changed
        && (!(cGPIO & 1<<nPB1)))    //  and button 1 is down (low)
        {
            sGPIO ^= 1<<nB1_LED;    //   toggle LED 1 using shadow register
        }

    // toggle LED 2 only on button 2 press (active high)
    if ((lGPIO & 1<<nPB2)           // if button 2 changed
        && (cGPIO & 1<<nPB2))       //  and button 2 is down (high)
        {
            sGPIO ^= 1<<nB2_LED;    //   toggle LED 2 using shadow register
        }

    // update last GPIO state (for next time)
    lGPIO = cGPIO;                  // new "last state" = current
}
```

Comparisons

Here is the resource usage summary for the "IOC with multiple inputs" programs:

2xToggle_LED-IOC

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	69	51	5
HI-TECH PICC-Lite	30	52	7
HI-TECH C PRO Lite	30	108	9

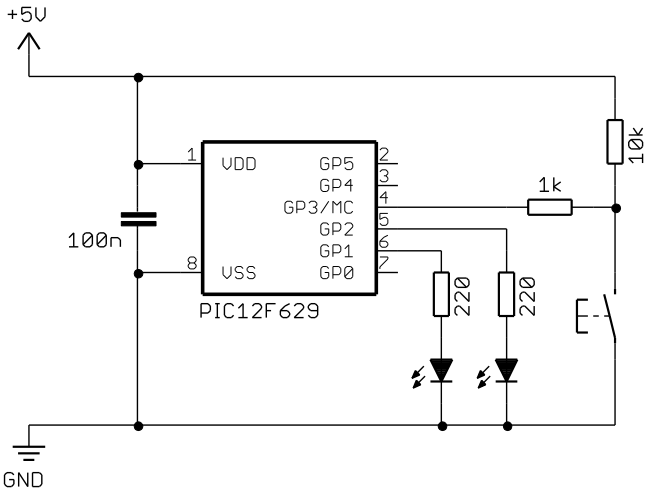
Again, the C source code is less than half as long as the assembler source, while the code generated by the PICC-Lite compiler is remarkably efficient – only one word larger than the hand-written assembly version!

Sleep Mode

As explained in midrange lesson 7, the midrange PICs can be placed into a power-saving standby, or sleep mode, using the assembler instruction '`sleep`'.

In this mode, the PIC12F629 will typically draw only a few nanoamps (or less), when all of the power-consuming facilities have been disabled and the output pins are not supplying any current.

This was demonstrated using the circuit on the right, which, if you wish to verify the low power consumption yourself, you will need to build in a way that allows you to measure the supply current.



The LED on GP1 is initially turned on, and then when the pushbutton is pressed, the LED is turned off (reducing power consumption) before placing the PIC permanently into sleep mode (effectively shutting it down)

The following assembler code was used:

```
        movlw   ~(1<<GP1)       ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO

        banksel GPIO
        bsf     GPIO,GP1        ; turn on LED

waitlo  btfsc   GPIO,GP3        ; wait for button press (low)
        goto    waitlo

        sleep                   ; enter sleep mode

        goto    $               ; (this instruction should never run)
```

HI-TECH C implementation

To place the PIC into sleep mode, HI-TECH C provides a '`SLEEP()`' macro.

It is defined in the "`pic.h`" header file (called from the "`htc.h`" file we've included at the start of each program), as:

```
#define     SLEEP()           asm("sleep")
```

'`asm()`' is a HI-TECH C statement which embeds a single assembler instruction, in-line, in the C source code. But since '`SLEEP()`' is provided as a standard macro, it makes sense to use it, instead of the '`asm()`' statement.

Complete program

The following program shows how this HI-TECH C '`SLEEP()`' macro is used:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, example 3                                 *
*                                                                       *
*   Demonstrates sleep mode                                             *
*                                                                       *
*   Turn on LED, wait for button pressed, turn off LED, then sleep      *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 - indicator LED                                             *
*       GP3 - pushbutton (active low)                                   *
*                                                                       *
************************************************************************/

#include <htc.h>

/***** CONFIGURATION *****/
//      int reset, no code or data protect, no brownout detect,
//      no watchdog, power-up timer enabled, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);

// Pin assignments
#define LED     GPIO1           // indicator LED on GP1
#define nLED    1               //   (LED on pin #1)
#define BUTTON  GPIO3           // pushbutton on GP3 (active low)


/***** MAIN PROGRAM *****/
void main()
{
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as output

    LED = 1;                    // turn on LED

    while (BUTTON == 1)         // wait for button press (low)
        ;

    LED = 0;                    // turn off LED

    SLEEP();                    // enter sleep mode

    for (;;)                    // (this loop should never execute)
        ;
}
```

Comparisons

Here is the resource usage summary for the "sleep after pushbutton press" programs, along with the baseline (PIC12F509) versions of this example, from baseline C lesson 3, for comparison:

Sleep_LED_off

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	22	20	16	13	0	0
HI-TECH PICC-Lite	12	12	16	13	2	4
HI-TECH C PRO Lite	12	12	20	18	2	2

The C source code is around half the length of the assembler version, while the PICC-Lite compiler continues to generate very efficient code, being the same size as the assembler version.

Note that the code sizes are slightly larger for the 12F629, compared with the 12F509 (baseline) versions. This reflects the need for bank selection instructions in the midrange architecture.

Wake-up from sleep

As discussed in midrange lesson 7, midrange PICs can be woken from sleep mode in a number of ways:

- Any device reset, such as an external reset signal on the $\overline{\text{MCLR}}$ pin (if enabled)
- Watchdog timer timeout (see the section on the watchdog timer, later in this lesson)
- Any enabled interrupt source which can set its interrupt flag while in sleep mode

Since the PIC's oscillator (clock) does not run in sleep mode, interrupt sources which require the clock to function, such as Timer0, cannot be used wake the device from sleep. However, external (INT pin) and port change interrupts (and others that we will see in later lessons) can be used to wake up a midrange PIC.

The following example looks at how to use the port change interrupt to wake a PIC from sleep mode; the method for using an external interrupt is essentially the same, but is of course limited to the INT pin.

Example 4: Using interrupt-on-change for wake-up from sleep

In baseline lesson 7, we saw that the baseline architecture includes a "wake-up on change" feature. Its midrange equivalent is the interrupt-on-change facility, introduced above.

"Interrupt-on-change" can be used to wake the device from sleep, even if interrupts are not enabled. If port change interrupts are enabled (GPIE = 1), but global interrupts are disabled (GIE = 0), then the device will wake from sleep when an IOC-enabled input changes, but no interrupt will occur. Program execution simply continues with the instruction following the `sleep` instruction, or, if using HI-TECH C, the statement following the '`SLEEP()`' macro.

If port change interrupts are enabled (GPIE = 1) and global interrupts are enabled (GIE = 1), if an IOC-enabled input changes while the PIC is in sleep mode, the device will wake from sleep, execute the instruction following `sleep`, and then enter the interrupt service routine.

If you want the PIC to execute the ISR immediately after it wakes from sleep, you need to enable interrupts and place a `nop` ("do nothing" – available in HI-TECH C as a '`NOP()`' macro) instruction immediately following the `sleep` instruction.

If you are using other interrupts (such as Timer0) in your program, but don't want to have to deal with executing the ISR as the device wakes from sleep, simply disable interrupts (clear GIE – which can be done in HI-TECH C using the '`di()`' macro) before entering sleep mode.

In any case, if GPIE = 1, the PIC will wake if the value of any IOC-enabled input changes while it is in sleep mode.

It is important to clear the GPIF flag before entering sleep mode, or else the PIC will wake immediately.

> *Note: You should read the input pins configured for interrupt-on-change just prior to entering sleep mode, and clear GPIF. Otherwise, if the value at an IOC-enabled pin had changed since the last time it was read, the PIC will wake immediately upon entering sleep mode, as the input value would be seen to be different from that last read.*

It is also important to ensure that any input which will be used to trigger a wake-up is stable before entering sleep mode.

This means that any switch used as a "soft" on/off switch must be debounced both as soon as the PIC has been restarted (in case the switch is still bouncing) and prior to entering sleep mode (in case a bounce causes the PIC to wake).

In this example, we want to wake-up the PIC and turn on an LED when the button is pressed, and then turn off the LED and place the PIC into sleep mode when the button is pressed again.

The necessary sequence is:

```
do
    turn on LED
    wait for stable button high
    wait for button low
    turn off LED
    wait for stable button high
    clear GPIF
    sleep
forever     // repeat from the beginning
```

HI-TECH C implementation

The following code, which uses the debounce macro defined in lesson 2, implements the sequence of steps given above:

```
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as output

    // configure Timer0 (for DbnceHi() macro)
    OPTION = 0b11010111;            // configure Timer0:
             //--0-----                  timer mode (T0CS = 0)
             //----0---                  prescaler assigned to Timer0 (PSA = 0)
             //-----111                  prescale = 256 (PS = 111)
                                     //  -> increment every 256 us
    // configure interrupt-on-change
    IOCB |= 1<<nBUTTON;         // enable IOC on pushbutton input
    GPIE = 1;                   // enable wake-up (interrupt) on port change
```

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

        DbnceHi(BUTTON);            // wait for stable button high
                                    //   (in case restarted after button press)

        while (BUTTON == 1)         // wait for button press (low)
            ;

        LED = 0;                    // turn off LED

        DbnceHi(BUTTON);            // wait for stable button release

        GPIF = 0;                   // clear port change interrupt flag

        SLEEP();                    // enter sleep mode
    }
```

(the labels '`LED`', '`nLED`', '`BUTTON`' and '`nBUTTON`' are defined earlier in the program, as usual)

This code does essentially the same thing as the "toggle an LED" programs developed in lessons 1 and 2, except that in this case, then the LED is off, the PIC is drawing negligible power.

Comparisons

Here is the resource usage summary for the "wake-up on change demo" programs:

Wakeup

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	32	39	0
HI-TECH PICC-Lite	20	37	2
HI-TECH C PRO Lite	20	67	2

In this example, the PICC-Lite is able to generate code which requires even less program memory than the hand-written assembler version! The assembler code could be optimised by removing redundant bank selection directives, but one of the advantages of using 'C' is that the compiler keeps track of bank selection for us, and will insert bank selection instructions only as necessary.

Watchdog Timer

As described in midrange lesson 7, the watchdog timer is free-running counter which, if enabled, operates independently of the program running on the PIC. It is typically used to avoid program crashes, where your application enters a state it will never return from, such as a loop waiting for a condition that will never occur. If the watchdog timer overflows, the PIC is reset, restarting your program – hopefully allowing it to recover and operate normally.

To avoid this "WDT reset" from occurring, your program must periodically reset, or clear, the watchdog timer before it overflows.

This watchdog time-out period on the midrange PICs is nominally 18 ms, but can be extended to a maximum of 2.3 seconds by assigning the prescaler to the watchdog timer (in which case the prescaler is no longer available for use with Timer0).

The watchdog timer can also be used to regularly wake the PIC from sleep mode, perhaps to sample and log an environmental input (say a temperature sensor), for low power operation.

Example 5a: Enabling the watchdog timer and detecting WDT resets

To illustrate how the watchdog timer allows the PIC to recover from a crash, we'll use a simple program which turns on an LED for 1.0 s, turns it off again, and then enters an endless loop (simulating a crash).

If the watchdog timer is disabled, the loop will never exit and the LED will remain off. But if the watchdog timer is enabled, with a period of 2.3 s, the program should restart itself after 2.3s, and the LED will flash: on for 1.0 s and off for 1.3 s (approximately).

We saw in midrange lesson 7 that the watchdog timer is controlled by the WDTE bit in the processor configuration word: setting WDTE to '1' enables the watchdog timer.

Since the configuration word cannot be accessed by programs running on the PIC (it can only be written to when the PIC is being programmed), **the watchdog timer cannot be enabled or disabled at runtime**. It can only be configured to be 'on' or 'off' when the PIC is programmed.

The assembler examples in that lesson included the following construct, to make it easy to select whether the watchdog timer is enabled or disabled when the code is built:

```
    #define     WATCHDOG        ; define to enable watchdog timer

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

To set the watchdog time-out period to the maximum of 2.3 seconds, the prescaler was assigned to the watchdog timer, with a prescale ratio of 1:128 (18 ms × 128 = 2.3 s), by:

```
        movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
        banksel OPTION_REG      ; -> WDT timeout = 2.3 s
        movwf   OPTION_REG
```

If you want your program to behave differently when restarted by a watchdog time-out, test the $\overline{\text{TO}}$ flag in the STATUS register: it is cleared to '0' only when a WDT reset has occurred.

The example in midrange lesson 7 used this approach to turn on an "error" LED, to indicate if a restart was due to a WDT reset:

```
;***** Initialisation
        ; configure port
        movlw   ~(1<<nF_LED|1<<nW_LED)  ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO
        ; configure watchdog timer prescaler
        movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
        banksel OPTION_REG      ; -> WDT timeout = 2.3 s
        movwf   OPTION_REG
```

```
;***** Main code
        banksel GPIO
        btfss   STATUS,NOT_TO   ; if WDT timeout has occurred,
        bsf     GPIO,nW_LED     ;   turn on "WDT" LED

        bsf     GPIO,nF_LED     ; turn on "flashing" LED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off "flashing" LED
        bcf     GPIO,nF_LED

        goto    $               ; wait forever
```

HI-TECH PICC-Lite implementation

Since the watchdog timer is controlled by a configuration bit, the only change we need to make to enable it is to use a different `__CONFIG()` statement, with the symbol '`WDTEN`' replacing '`WDTDIS`'.

A construct very similar to that in the assembler example can be used to select between processor configurations:

```
#define     WATCHDOG            // define to enable watchdog timer

#ifdef WATCHDOG
    // Config: ext reset, no code or data protect, no brownout detect,
    //         watchdog, power-up timer enabled, 4MHz int clock
    __CONFIG(MCLREN & UNPROTECT & BORDIS & WDTEN & PWRTEN & INTIO);
#else
    // Config: ext reset, no code or data protect, no brownout detect,
    //         no watchdog, power-up timer enabled, 4MHz int clock
    __CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);
#endif
```

Assigning the prescaler to the watchdog timer and selecting a prescale ratio of 128:1 is done by:

```
    OPTION = 0b11001111;            // set WDT timeout:
             //----1---                 prescaler assigned to WDT (PSA = 1)
             //-----111                 prescale = 128 (PS = 111)
                                    //  -> WDT timeout = 2.3 s
```

To check for a WDT timeout reset, the $\overline{TO}$ flag can be tested directly, using:

```
    if (!TO)                    // if WDT timeout has occurred,
    {
        W_LED = 1;              //   turn on "WDT" LED
    }
```

Note that the test condition is inverted, using '!', since this flag is "active" when clear[2].

[2] The problem with the default PICC-Lite runtime start-up code, which prevented this test from working correctly in the corresponding example in baseline C lesson 3, does not appear to affect midrange PICs.

Complete program

Here is the complete program, showing how the above code fragments are used:

```
************************************************************************
*                                                                      *
*   Description:    Lesson 4, example 5a                               *
*                                                                      *
*   Demonstrates watchdog timer                                        *
*       plus differentiation of WDT time-out from POR reset            *
*                                                                      *
*   Turn on LED for 1s, turn off, then enter endless loop              *
*   If WDT enabled, processor resets after 2.3s                        *
*   Turns on WDT LED to indicate WDT reset                             *
*                                                                      *
************************************************************************
*                                                                      *
*   Pin assignments:                                                   *
*       GP1 - "flashing" LED                                           *
*       GP2 - WDT LED (indicates WDT time-out reset)                   *
*                                                                      *
************************************************************************/

#include <htc.h>

#include "stdmacros-PCL.h"      // DelayS() - delay in seconds


/***** CONFIGURATION *****/
#define     WATCHDOG            // define to enable watchdog timer

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

// Pin assignments
#define F_LED   GPIO1           // "flashing" LED on GP1
#define nF_LED  1               //   (pin 1)
#define W_LED   GPIO2           // WDT LED to indicate WDT time-out reset
#define nW_LED  2               //   (pin 2)


/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/
    // configure port
    TRISIO = ~(1<<nF_LED|1<<nW_LED);    // configure LED pins as outputs

    // configure watchdog timer
    OPTION = 0b11001111;            // set WDT timeout:
             //----1---                 prescaler assigned to WDT (PSA = 1)
             //-----111                 prescale = 128 (PS = 111)
                                    //  -> WDT timeout = 2.3 s
```

```
    /*** Main code ***/
    if (!TO)                    // if WDT timeout has occurred,
    {
        W_LED = 1;              //   turn on "WDT" LED
    }

    F_LED = 1;                  // turn on "flash" LED

    DelayS(1);                  // delay 1s

    F_LED = 0;                  // turn off "flash" LED

    for (;;)                    // wait forever
        ;
}
```

HI-TECH C PRO Lite implementation

Although HI-TECH C PRO provides a `__delay_ms()` library function, which is instead of the delay routines provided with PICC-Lite, this difference is hidden by the `DelayS()` macro, introduced in lesson 2 and used in the PICC-Lite example above.

However, to use the `__delay_ms()` library function, we must define the oscillator frequency, and include the correct macro definition, toward the start of the program:

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

#include "stdmacros-HTC.h"      // DelayS() - delay in seconds
```

Comparisons

Here is the resource usage summary for the "watchdog timer demo" programs, including those from the corresponding example in baseline C lesson 3:

WDTdemo+LED

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	34	34	32	36	3	3
HI-TECH PICC-Lite	21	20	43	48	6	10
HI-TECH C PRO Lite	22	21	47	50	5	4

In this case, the HI-TECH C PRO compiler generates reasonably efficient code (better than CCS), even when running in non-optimised "Lite" mode, because of its library delay function.

Example 5b: Clearing the watchdog timer

Normally, you will want to prevent watchdog timer overflows; a WDT reset should only happen when something has gone wrong.

To avoid WDT resets, the watchdog timer has to be regularly cleared. This is typically done by inserting a '`clrwdt`' instruction within the program's "main loop", and within any subroutine which may, in normal operation, not complete within the watchdog timer period.

To demonstrate the effect of clearing the watchdog timer, a '`clrwdt`' instruction was added into the endless loop in the example in midrange lesson 7:

```
;***** Main code
        banksel GPIO            ; turn on LED
        bsf     GPIO,LED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off LED
        bcf     GPIO,LED

loop    clrwdt                  ; clear watchdog timer
        goto    loop            ;   repeat forever
```

With the '`clrwdt`' instruction in place, the watchdog timer never overflows, so the PIC is never restarted by a WDT reset, and the LED remains turned off (until the power is cycled), whether the watchdog timer is enabled or not.

HI-TECH C implementation

HI-TECH C provides a '`CLRWDT()`' macro, defined in the "`pic.h`" header file as:

```
#define     CLRWDT()    asm("clrwdt")
```

That is, the '`CLRWDT()`' macro simply inserts a '`clrwdt`' instruction into the code.

Using this macro, the assembler code above can be implemented with HI-TECH C as follows:

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

    DelayS(1);                  // delay 1s

    LED = 0;                    // turn off LED

    for (;;)                    // repeat forever
    {
        CLRWDT();               //   clear watchdog timer
    }
```

Example 6: Periodic wake from sleep

As explained in midrange lesson 7, the watchdog timer is can also be used to periodically wake the PIC from sleep mode, typically to read some inputs, take some action and then return to sleep mode, saving power. This can be combined with wake-up on pin change, allowing immediate response to some inputs, such as a button press, while periodically checking others.

To illustrate this, the example in midrange lesson 7 converted the main code in the first watchdog timer example into a loop, incorporating the '`sleep`' instruction:

```
loop
        banksel GPIO            ; turn on LED
        bsf     GPIO,LED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off LED
        bcf     GPIO,LED

        sleep                   ; enter sleep mode (until WDT time-out)

        goto    loop            ; repeat forever
```

With the watchdog timer enabled, with a period of 2.3 s, the LED is on for 1 s, and then off for 1.3 s, as in the earlier example. But this time the PIC is in sleep mode while the LED is off, conserving power.

HI-TECH C implementation

In a similar way, we can convert the main code in example 5, above, into a loop – dropping the WDT timeout test, and adding a `SLEEP()` macro:

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

        DelayS(1);                  // delay 1s

        LED = 0;                    // turn off LED

        SLEEP();                    // enter sleep mode (until WDT time-out)

    }   // repeat forever
```

Complete program

Here is how this new main loop fits into the code:

```
;*****************************************************************************
;*                                                                           *
;*   Description:    Lesson 4, example 6                                     *
;*                                                                           *
;*   Demonstrates periodic wake from sleep, using the watchdog timer         *
;*                                                                           *
;*   Turn on LED for 1s, turn off, then sleep                                *
;*       LED stays off if watchdog not enabled,                              *
;*       flashes (1s on, 2.3s off) if WDT enabled                            *
;*                                                                           *
;*****************************************************************************
;*                                                                           *
;*   Pin assignments:                                                        *
;*       GP1 - indicator LED                                                 *
;*                                                                           *
;*****************************************************************************/

#include <htc.h>

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

#include "stdmacros-HTC.h"      // DelayS() - delay in seconds


/***** CONFIGURATION *****/
#define     WATCHDOG            // define to enable watchdog timer

#ifdef WATCHDOG
    // Config: ext reset, no code or data protect, no brownout detect,
    //         watchdog, power-up timer enabled, 4MHz int clock
    __CONFIG(MCLREN & UNPROTECT & BORDIS & WDTEN & PWRTEN & INTIO);
#else
    // Config: ext reset, no code or data protect, no brownout detect,
    //         no watchdog, power-up timer enabled, 4MHz int clock
    __CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);
#endif
```

```
// Pin assignments
#define LED     GPIO1           // indicator LED on GP1
#define n_LED   1               //   (pin 1)


/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/
    // configure port
    TRISIO = ~(1<<n_LED);           // configure LED pin (only) as output

    // configure watchdog timer
    OPTION = 0b11001111;            // set WDT timeout:
             //----1---                prescaler assigned to WDT (PSA = 1)
             //-----111                prescale = 128 (PS = 111)
                                     //  -> WDT timeout = 2.3 s

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

        DelayS(1);                  // delay 1s

        LED = 0;                    // turn off LED

        SLEEP();                    // enter sleep mode (until WDT time-out)

    }   // repeat forever
}
```

Summary

Overall, we have seen that the interrupt-on-change, sleep mode, wake-up on change, and watchdog timer features of the midrange PIC architecture can be configured and used effectively in C programs, using either of the HI-TECH compilers.

All of the examples could be expressed succinctly in C, as illustrated by the code length comparisons:

Source code (lines)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5a
Microchip MPASM	50	69	22	32	34
HI-TECH PICC-Lite	19	30	12	20	21
HI-TECH C PRO Lite	19	30	12	20	22

As in the previous lessons, the C source code is significantly shorter than the corresponding assembler source, particularly in examples 1 and 2, where it is less than half as long, reflecting the ease with which interrupts can be implemented using HI-TECH C.

We would normally expect hand-written assembly code to be more efficient, in terms of program and data memory use, than that generated by a C compiler, even with good optimisation, but this wasn't really true for the examples in this lesson:

Program memory (words)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5a
Microchip MPASM	36	51	16	39	32
HI-TECH PICC-Lite	37	52	16	37	43
HI-TECH C PRO Lite	72	109	20	67	47

In all but example 5, the PICC-Lite compiler was able to generate code almost as small, and in the case of example 4, smaller, than the hand-written equivalent.

The programs generated by the C compilers do, however, use more data memory than the assembler versions:

Data memory (bytes)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5a
Microchip MPASM	3	5	0	0	3
HI-TECH PICC-Lite	5	7	2	2	6
HI-TECH C PRO Lite	7	9	2	2	5

This is not a problem for such small programs, given the 64 bytes of data memory available on the 12F629, but it could become an issue for larger programs.

The next lesson revisits material from midrange lesson 8, briefly covering some of the hardware-related features of the 12F629 (also included in most other midrange PICs), such as brown-out detection and the available oscillator (clock) options.