Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 3: Introduction to Interrupts

As we saw in [midrange lesson 6](), the *interrupt* facility available on midrange PICs is especially useful, making it much easier to implement regular "background" tasks (such as refreshing a multiplexed display – see for example [baseline lesson 8]()) and allow programs to respond in a timely manner to external events, without having to sit in a *busy-wait*, or polling loop. Both of these applications of interrupts are demonstrated in this lesson.

This lesson revisits the material from [midrange lesson 6](), introducing external and timer interrupts (driven by Timer0) and some of their applications, such as running background tasks and switch debouncing,.

As usual, the examples are re-implemented using the "free" C compilers from HI-TECH Software: PICC-Lite and HI-TECH C[1] (in "Lite" mode), and for each, the memory usage and code length is compared with that of assembler, to demonstrate the trade-offs involved.

In summary, this lesson covers:

- Introduction to interrupts on the midrange PIC architecture
- Interrupt handling, using the HI-TECH C compilers
- Timer-driven interrupts
- Debouncing single switches with timer-driven interrupts
- External interrupts on the INT pin

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

Note that this tutorial series assumes a working knowledge of the C language; it does **not** attempt to teach C.

Interrupts

An *interrupt* is a means of interrupting the main program flow in response to an event, so that the event can be handled, or *serviced*. The event (referred to an interrupt *source*) can be internal to the PIC, such as a timer overflowing, or external, such as a change on an input pin.

When the interrupt is triggered, program execution immediately jumps to an *interrupt service routine (ISR)*, which, in the midrange PIC architecture, is always located at address 0004h (the "interrupt vector").

The HI-TECH C compilers hide this detail; if a function is defined with the qualifier '`interrupt`', the compiler considers it to be an interrupt service routine, and places it at the correct address, automatically. Of course, this means that, on midrange PICs, the '`interrupt`' qualifier can only be used with one function, as there is only one interrupt vector in the midrange architecture.

---

[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]().

The ISR must save the current processor state, or *context* (i.e. the contents of any registers which the ISR will modify, such as W and STATUS), service the interrupt, and then restore the context before returning to the main program. In this way, the main program will never "notice" that the interrupt has happened – the interrupt will be completely transparent, except for whatever action the interrupt service routine was intended to perform.

Again, the HI-TECH C compilers take care of this implementation detail, automatically adding appropriate context save and restore code to the 'interrupt' function.

Timer0 Interrupts

Timer0 can be used to regularly generate interrupts, which can be used to drive "background" tasks, such as:

- Generating a regular output; for example flashing an LED.

- Monitoring and debouncing inputs

Meanwhile, a "main program" can continue to perform other "foreground" tasks.

The examples in this section illustrate these techniques, using the circuit from lesson 2, shown on the right.
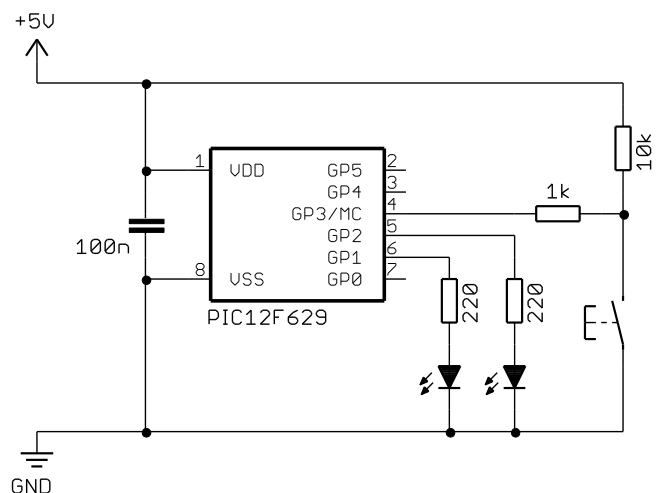
Example 1a: Flashing an LED

To begin, we'll simply flash an LED, without attempting to make it flash at exactly 1 Hz.

We saw in midrange lesson 4 that, given a 1 MHz instruction clock with maximum prescaling (1:256), the longest period that Timer0 can generate is $256 \times 256 \times 1$ μs = 65.5 ms.

Therefore, if we configured the PIC to use a 4 MHz clock, and set up Timer0 in timer mode with a 1:256 prescaler, TMR0 would *overflow* (rollover from 255 to 0) every 65.5 ms.

If we then enabled Timer0 interrupts, the interrupt would be triggered on every TMR0 overflow, i.e. every 65.5 ms. So the interrupt service routine (ISR) would be called every 65.5. ms.

If the ISR toggled an LED every time it was called, the LED would change state every 65.5 ms – it would flash with a period of 65.5 ms $\times$ 2 = 131 ms, giving a frequency of 7.6 Hz.

Having an LED flash as 7.6 Hz is not ideal, but the flashing is visible (just), and that's the slowest flash rate we can generate with the simple approach described above. So we'll start there.

The assembler code in midrange lesson 6 configured the port and Timer0, before enabling the Timer0 interrupt by setting the T0IE (Timer0 interrupt enable) and GIE (global interrupt enable) bits in the INTCON register:

```
; configure interrupts
movlw   1<<GIE|1<<T0IE  ; enable Timer0 and global interrupts
movwf   INTCON
```

The interrupt service routine began by saving the processor context, and then reset, or cleared, the Timer0 interrupt flag (T0IF) to show that this Timer0 overflow event has been handled – if this is not done, the interrupt would immediately re-trigger, as soon as the ISR has exited.

The interrupt service routine then toggled the LED, indirectly, by toggling the bit corresponding to the LED in a shadow register:

```
        movf    sGPIO,w            ; only update shadow register
        xorlw   1<<nLED
        movwf   sGPIO
```

This was done to avoid potential read-modify-write problems (described in baseline lesson 2).

Finally, the ISR restored the processor context, before exiting and returning control to the main program.

The body of the main program then had only a single task to perform – to repeatedly copy the contents of the shadow register to the GPIO port, to make the changes made within the ISR visible (literally!):

```
;***** Main loop
loop    ; continually copy shadow GPIO to port
        movf    sGPIO,w
        banksel GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop
```

HI-TECH C implementation

As mentioned above, the HI-TECH C compilers hide much of the complexity associated with handling interrupts, such as saving and restoring the processor context.

The interrupt service routine is implemented as a function, defined with the qualifier 'interrupt'.

For example:

```
void interrupt isr(void)
```

Note that the interrupt function should be declared as type void, and must not take any parameters, because it is never explicitly called from anywhere – nothing is passed to it, and nothing is returned. It just "happens", whenever an interrupt is triggered. The name of the interrupt function is not important; you don't have to call it 'isr'.

Since direct parameter passing isn't possible, any data passed between the ISR and the main program must be held in global variables (declared outside any function), so that both the interrupt function and main() (and any other functions) can access them.

In this example, the variable holding the shadow copy of GPIO is accessed by both the ISR and the main program, so it must be declared as a global variable, before main() or the interrupt function:

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

Since we don't need to worry about saving or restoring the processor context, the HI-TECH C version of the ISR can be very simple:

```
void interrupt isr(void)
{
    // handle Timer0 interrupt
    T0IF = 0;                         // clear interrupt flag

    // toggle LED
    sGPIO ^= 1<<nLED;                 // only update shadow register
}
```

In the main program, we configure the port and Timer0, as we have done before:

```
// setup ports
GPIO = 0;                           // start with LED off
sGPIO = 0;                          //   update shadow
TRISIO = ~(1<<nLED);                // configure LED pin (only) as an output
// setup Timer0
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
```

The symbol 'nLED' had been defined previously, to help make the code more maintainable:

```
// Pin assignments
#define nLED    2        // flashing LED on GP2
```

(The 'n' prefix used here indicates that this is a numeric constant; this is simply a convention, and you can choose whatever naming style works for you.)

Having configured the port and timer, we're ready to enable the Timer0 interrupt, which, as we saw above, is done by setting the T0IE and GIE bits in the INTCON register.

This could be done by:

```
// configure interrupts
T0IE = 1;                           // enable Timer0 interrupt
GIE = 1;                            // enable global interrupts
```

However, HI-TECH C defines a macro, 'ei()', which is intended to be used to enable interrupts globally, and is equivalent to writing 'GIE = 1'. Similarly, there is a 'di()' macro, used to disable all interrupts, equivalent to 'GIE = 0'.

So, in keeping with the HI-TECH C conventions, the Timer0 interrupt should be enabled by:

```
// configure interrupts
T0IE = 1;                           // enable Timer0 interrupt
ei();                               // enable global interrupts
```

Finally, we need to continually copy the shadow register to GPIO, which can be done by:

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

Complete program

Here is how these code fragments fit together, for HI-TECH C PRO or PICC-Lite:

```
*************************************************************************
*                                                                      *
*    Description:    Lesson 3, example 1a                              *
*                                                                      *
*    Demonstrates use of Timer0 interrupt to perform a background task  *
*                                                                      *
*    Flash LED at approx 7.6 Hz (50% duty cycle)                       *
*                                                                      *
```

```
/************************************************************************
 *                                                                      *
 *   Pin assignments:                                                   *
 *       GP2 - flashing LED                                             *
 *                                                                      *
 ************************************************************************/

#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 nLED    2               // flashing LED on GP2


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


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    GPIO = 0;                       // start with LED off
    sGPIO = 0;                      //   update shadow
    TRISIO = ~(1<<nLED);            // configure LED pin (only) as an output
    // setup Timer0
    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 interrupts
    T0IE = 1;                       // enable Timer0 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)
{
    // handle Timer0 interrupt
    T0IF = 0;                       // clear interrupt flag

    // toggle LED
    sGPIO ^= 1<<nLED;               // only update shadow register
}
```

Comparisons

Once again, it is worth comparing the length of the source code (ignoring comments and white space) versus program and data memory used by code generated by each language/compiler (MPASM assembler, HI-TECH PICC-Lite and C PRO in "Lite" mode), to illustrate any trade-offs between programmer efficiency and resource-usage efficiency. Longer source code implies more time spent by the programmer writing the code, and more time spent debugging or maintaining the code. Understanding these trade-offs, and the relative value of your time versus device cost (having less efficient code means that you may need a bigger, more expensive, device to hold it), is key to whether you choose to develop in C or assembler.

Flash_LED-50p-int-fast

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	47	34	3
HI-TECH PICC-Lite	16	36	5
HI-TECH C PRO Lite	16	68	7

In this case, the C source code is only around a third 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. Using interrupts with HI-TECH C is much easier than in assembler!

The code generated by the PICC-Lite compiler, with optimisation enabled, is barely any larger than the hand-written assembly version, demonstrating that there does not always have to be any real "downside" to using C – in this example we have much shorter code, fewer details to contend with, and no real penalty in code size or data memory use.

Of course, HI-TECH C PRO can always be expected to generate much larger code when running in "Lite" mode, which does not perform any optimisation.

Example 1b: Slower flashing

The LED in the last example flashed at around 7.6 Hz. Since the longest possible interval between Timer0 interrupts is 65.5 ms (with a 4 MHz processor clock), to flash the LED any slower, we can't toggle it on every interrupt; we have to skip some of them. That means counting each interrupt, and only toggling the LED when the count reaches a certain value.

A simple way to implement this, if we are not concerned with exact timing, is to use an 8-bit counter, and to let it reach 255 before toggling the LED when it overflows to 0.

If, every time an interrupt is triggered by a Timer0 overflow, the ISR increments a counter, we're essentially implementing a 16-bit timer, based on Timer0, with TMR0 as the least significant eight bits, and the counter incremented by the ISR being the most significant eight bits.

If the ISR increments the counter whenever Timer0 overflows (every 256 *ticks* of TMR0), and it toggles the LED whenever the counter overflows (every 256 interrupts), the LED is being toggled every $N \times 256 \times 256$ (where N is the prescale ratio) instruction cycles.

Assuming a 1 MHz instruction clock, LED will be toggled every $N \times 256 \times 256$ μs = $N \times 65.536$ ms.

We can make the LED flash at close to 1 Hz by choosing N = 8 (prescale ratio of 1:8). The resulting toggle period is $8 \times 256 \times 256$ μs = 524.3 ms, giving a flash rate of 0.95 Hz – close enough!

HI-TECH C implementation

To implement the Timer0 overflow counter, we'll need a variable to store it in.

Since this variable only needs to be used by the interrupt service routine, to be consistent with good modular programming practice, we should make it private to (defined within) the interrupt function:

```
void interrupt isr(void)
{
    static unsigned char    t0cnt = 0;  // counts timer0 overflows

    // (body of ISR goes here)
}
```

Note that this variable is defined as being 'static'; this is very important. The counter must retain its value between interrupts, so that it can be incremented by successive interrupts. To ensure that the counter continues to exist, preserving its value, outside the interrupt function, it must be defined as 'static'.

Note also that it the counter variable is initialised, as part of its definition. You might think that, because the definition is within the interrupt function, that this initialisation (clearing the counter) will happen every time an interrupt occurs, losing the value of the counter. But no – all static variables are initialised only once, by the start-up code generated by the C compiler, before the main() function starts executing.

We then need to add instructions to the ISR to increment this counter, and toggle the LED only when it overflows back to zero:

```
    // toggle LED every 256 interrupts (524 ms)
    ++t0cnt;                        // increment interrupt count (every 2.048 ms)
    if (t0cnt == 0)                 // if count overflow (every 524 ms),
        sGPIO ^= 1<<nLED;           //   toggle LED (using shadow register)
```

This could have been written more succinctly as:

```
    // toggle LED every 256 interrupts (524 ms)
    if (++t0cnt == 0)               // increment count; if overflow (every 524 ms),
        sGPIO ^= 1<<nLED;           //   toggle LED (using shadow register)
```

Whether you choose to sacrifice readability to save a line of source code is a question of personal style.

Here is the complete ISR, with these changes:

```
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static unsigned char    t0cnt = 0;  // counts timer0 overflows

    // handle Timer0 interrupt
    T0IF = 0;                           // clear interrupt flag

    // toggle LED every 256 interrupts (524 ms)
    ++t0cnt;                            // increment interrupt count (every 2.048 ms)
    if (t0cnt == 0)                     // if count overflow (every 524 ms),
        sGPIO ^= 1<<nLED;               //   toggle LED (using shadow register)
}
```

And finally the configuration of Timer0 needs to be changed, to select a 1:8 prescaler:

```
    // setup Timer0
    OPTION = 0b11010010;                // configure Timer0:
             //--0-----                     timer mode (T0CS = 0)
             //----0---                      prescaler assigned to Timer0 (PSA = 0)
             //-----010                      prescale = 8 (PS = 010)
                                         //  -> increment every 8 us
```

With these changes to the code in the first example, the LED will flash at a much more sedate 0.95 Hz.

Example 1c: Flashing an LED at exactly 1 Hz

What if we needed (for some reason) to flash the LED at exactly 1 Hz, given an accurate 4 MHz processor clock? As discussed in detail in midrange lesson 6, there are a number of pitfalls inherent in trying to use Timer0 to generate a cycle-exact time base.

But as we saw, these problems can be overcome, relatively easily.

To use Timer0 to provide a precise time base to drive an interrupt:

- Do not use the prescaler (assign it to the watchdog timer).

- Do not load a fixed start value into the timer.

  Instead, add an offset to the current timer value, making the timer "skip forward" by an appropriate amount, shortening the timer cycle from 256 counts to whatever period you require.

- Adjust the offset to allow for the fact that the timer is inhibited for two cycles after it is written, and that the timer increments once (if no prescaler is used) during the add instruction.

  This means that the offset to be added must be 3 cycles larger than you may expect, to achieve a given timer period.

In the example in midrange lesson 6, we used the following assembler code:

```
        movlw   .256-.250+.3    ; add value to Timer0
        banksel TMR0            ;   for overflow after 250 counts
        addwf   TMR0,f
```

to make Timer0 overflow after 250 cycles, instead of the usual 256 cycles (with no prescaler). This was done after every Timer0 overflow (i.e. within the interrupt service routine), so that the interrupt is triggered precisely every 250 instruction cycles (every 250 μs, given a 4 MHz processor clock).

Toggling the LED every 500 ms means toggling after every 500 ms ÷ 250 μs = 2000 interrupts.

This means that the ISR must be able to count to 2000, so that it can toggle the LED after 2000 interrupts.

In the assembler version, this was realised by using two 8-bit variables, one counting interrupts to create a 10 ms time base, the other counting these 10 ms intervals to generate the 500 ms period we need.

But since we're using C here, we may as well take advantage of its ability to easily work with larger quantities, and simply use a single 16-bit variable to count interrupts.

HI-TECH C implementation

Since the timer overflow counter is only accessed by the interrupt service routine, it should be defined within the interrupt function, as was done in the last example:

```
void interrupt isr(void)
{
    static unsigned int     cnt_t0 = 0;     // counts timer0 overflows

    // (body of ISR goes here)
}
```

Note that, because this variable needs to be able to count up to 2000, it is defined as a (16-bit) `int`, instead of an (8-bit) `char`.

To make the Timer0 interrupt occur every 250 cycles, instead of the usual 256, we need to add an appropriate offset to TMR0, within the ISR, as follows:

```
TMR0 += 256-250+3;                  // add value to Timer0
                                    //  for overflow after 250 counts
```

It is then a simple matter to count interrupts and toggle the LED after 500 ms (2000 counts):

```
// toggle LED every 500 ms
++cnt_t0;                           // increment interrupt count (every 250 us)
if (cnt_t0 == 500000/250) {         // if count overflow (every 500 ms),
    cnt_t0 = 0;                     //   reset count
    sGPIO ^= 1<<nLED;               //   toggle LED (using shadow register)
```

Finally, in the initialisation part of the main program, we need to configure Timer0 with no prescaler, which we could write as:

```
OPTION = 0b11001000;                // configure Timer0:
        //--0-----                   timer mode (T0CS = 0)
        //----1---                   no prescaling (PSA = 1)
                                    //  (prescaler assigned to WDT)
                                    //  -> increment every 1 us
```

But since there are only two bits being set or cleared, and HI-TECH C allows us to access them easily by name, this is more naturally written as:

```
T0CS = 0;                           // select timer mode
PSA = 1;                            // no prescaler (assigned to WDT)
                                    //  -> increment TMR0 every 1 us
```

With these modifications in place, the LED will now flash with a frequency of exactly 1 Hz, assuming that the processor clock is exactly 4 MHz (which, since we are using the internal RC oscillator, it will not be the case; it's not that accurate. Nevertheless, the LED flashes every 4,000,000 processor cycles, precisely).

Complete program

Here is how the code fragments above fit together:

```
/************************************************************************
 *    Description:    Lesson 3, example 1c                              *
 *                                                                      *
 *    Demonstrates use of Timer0 interrupt to perform a background task *
 *                                                                      *
 *    Flash LED at exactly 1 Hz (50% duty cycle)                        *
 *                                                                      *
 ************************************************************************
 *                                                                      *
 *    Pin assignments:                                                  *
 *       GP2 - flashing LED                                             *
 *                                                                      *
 ************************************************************************/
```

```c
#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 nLED    2                   // flashing LED on GP2


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


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    GPIO = 0;                           // start with LED off
    sGPIO = 0;                          //   update shadow
    TRISIO = ~(1<<nLED);                // configure LED pin (only) as an output
    // configure Timer0
    T0CS = 0;                           // select timer mode
    PSA = 1;                            // no prescaler (assigned to WDT)
                                        //  -> increment TMR0 every 1 us
    // configure interrupts
    T0IE = 1;                           // enable Timer0 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)
{
    static unsigned int    cnt_t0 = 0;     // counts timer0 overflows

    // service Timer0 interrupt
    //   TMR0 overflows every 250 clocks = 250 us
    //    (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;                 // add value to Timer0
                                       //   for overflow after 250 counts
    T0IF = 0;                          // clear interrupt flag

    // toggle LED every 500 ms
    ++cnt_t0;                          // increment interrupt count (every 250 us)
    if (cnt_t0 == 500000/250) {        // if count overflow (every 500 ms),
        cnt_t0 = 0;                    //   reset count
        sGPIO ^= 1<<nLED;              //   toggle LED (using shadow register)
    }
}
```

Comparisons

Here is the resource usage summary for the "Flash an LED at exactly 1 Hz" programs:

Flash_LED-50p-int-1Hz

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	64	49	5
HI-TECH PICC-Lite	22	52	7
HI-TECH C PRO Lite	22	92	7

Again, the C source code is only a third as long as the assembler source, while the code generated by the PICC-Lite compiler, with optimisation enabled, is barely any larger than the hand-written assembly version.

Example 2: Flash LED while responding to input

Now that we have a timer-driven interrupt flashing the LED on GP2 at 1 Hz, that flashing will continue, "on its own", independently of whatever the main program code is doing. This is the main reason for using a timer interrupt to drive a background process like this; once the process is set up, you do not need to worry about maintaining it in the main code. It may seem complex to set up the interrupt code, but, once done, it makes your main code much easier to write.

To illustrate this, we can re-implement example 2 from lesson 2, where we the LED on GP1 is lit whenever the pushbutton is pressed, while the LED on GP2 continues to flash steadily at 1 Hz.

HI-TECH C implementation

In lesson 2, we used this simple piece of code to read the pushbutton and light the LED on GP1 only when it is pressed:

```
        sGPIO &= ~(1<<1);              // assume button up -> LED off
        if (GPIO3 == 0)                // if button pressed (GP3 low)
            sGPIO |= 1<<1;             //   turn on LED on GP1

        GPIO = sGPIO;                  // update GPIO
```

In the main loop in example 1, above, we are doing nothing but copy the shadow register to GPIO:

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

    }   // repeat forever
```

All we need do, then, is to insert the pushbutton-handling code into the main loop:

```
    for (;;) {
        // check and respond to button press
        sGPIO &= ~(1<<nB_LED);         // assume button up -> LED off
        if (BUTTON == 0)               // if button pressed (GP3 low)
            sGPIO |= 1<<nB_LED;        //   turn on indicator LED

        // continually copy shadow GPIO to port
        GPIO = sGPIO;

    }   // repeat forever
```

And of course you could add any other code to the main loop, in the same way. There is no need to be "aware" of the interrupt-driven process; it runs quite independently.

The symbols used here were defined as:

```
#define nB_LED  1       // "button pressed" indicator LED on GP1
#define nF_LED  2       // flashing LED on GP2
#define BUTTON  GPIO3   // pushbutton on GP3
```

The only other change that has to be made to the code in example 1 is to configure both GP1 and GP2 as outputs:

```
    TRISIO = ~(1<<nB_LED|1<<nF_LED);  // configure LED pins (only) as outputs
```

No changes are needed within the interrupt service routine.

Complete program

Although the changes to the code in example 1 are minor, here is how they fit together:

```
/**************************************************************************
*                                                                        *
*    Description:    Lesson 3, example 2                                  *
*                                                                        *
*    Demonstrates use of Timer0 interrupt to perform a background task   *
*    while performing other actions in repsonse to changing inputs       *
*                                                                        *
*    One LED simply flashes at 1 Hz (50% duty cycle).                    *
*    The other LED is only lit when the pushbutton is pressed.           *
*                                                                        *
**************************************************************************
*                                                                        *
*    Pin assignments:                                                    *
*        GP1 - "button pressed" indicator LED                            *
*        GP2 - flashing LED                                              *
*        GP3 - pushbutton                                                *
*                                                                        *
**************************************************************************/

#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 nB_LED  1       // "button pressed" indicator LED on GP1
#define nF_LED  2       // flashing LED on GP2
#define BUTTON  GPIO3   // pushbutton on GP3


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


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

```
    GPIO = 0;                                    // start with LEDs off
    sGPIO = 0;                                   //   update shadow
    TRISIO = ~(1<<nB_LED|1<<nF_LED);   // configure LED pins (only) as outputs
    // configure Timer0
    T0CS = 0;                                    // select timer mode
    PSA = 1;                                     // no prescaler (assigned to WDT)
                                                 //  -> increment TMR0 every 1 us
    // configure interrupts
    T0IE = 1;                                    // enable Timer0 interrupt
    ei();                                        // enable global interrupts

    // Main loop
    for (;;) {
        // check and respond to button press
        sGPIO &= ~(1<<nB_LED);              // assume button up -> LED off
        if (BUTTON == 0)                       // if button pressed (GP3 low)
            sGPIO |= 1<<nB_LED;              //   turn on indicator LED

        // continually copy shadow GPIO to port
        GPIO = sGPIO;

    }   // repeat forever
}

/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static unsigned int    cnt_t0 = 0;      // counts timer0 overflows

    // service Timer0 interrupt
    //   TMR0 overflows every 250 clocks = 250 us
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;                   // add value to Timer0
                                         //   for overflow after 250 counts
    T0IF = 0;                            // clear interrupt flag

    // toggle LED every 500 ms
    ++cnt_t0;                            // increment interrupt count (every 250 us)
    if (cnt_t0 == 500000/250) {     // if count overflow (every 500 ms),
        cnt_t0 = 0;                      //   reset count
        sGPIO ^= 1<<nF_LED;          //   toggle LED (using shadow register)
    }
}
```

Comparisons

Here is the resource usage summary for the "Flash an LED while responding to a pushbutton" programs:

Flash+PB_LED-int

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	65	52	5
HI-TECH PICC-Lite	27	55	7
HI-TECH C PRO Lite	27	101	8

Once again, the C source code is less than half as long as the assembler source, while the code generated by PICC-Lite, with optimisation enabled, is barely any larger than the hand-written assembly version.

Example 3: Switch debouncing

Lesson 1 demonstrated one widely-used method of addressing the problem of switch bounce, which 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
```

The change in switch state is only accepted when the new state has been continually seen for at least some minimum period, for example 20 ms. This debounce period is measured by incrementing a count while sampling the state of the switch, at a steady rate, such as every 1 ms.

We saw in midrange lesson 6 that this counting algorithm can be readily implemented in an interrupt service routine, which regularly samples the switch and increments a counter whenever the current (or *raw*) state of the switch is different from the last accepted (or *debounced*) state.

That is, if the switch is in a different state from what it used to be, maybe it has "really" changed, or maybe this is just a glitch, or perhaps it's bouncing, so let's check a few more times to be sure. When it's been stable in the new state for some time, we accept this new state as being "real", and consider the switch to have been debounced.

Although you could have the ISR respond to and act upon switch changes, but this isn't normally done unless the event has to be responded to very quickly; it is generally best to keep the interrupt handling code short, so that the ISR finishes quickly, in case another, perhaps more important, interrupt is pending.

Instead, the ISR would normally use a flag to signal to the main program that an event (such as a change in switch state) has occurred. The main program then polls this flag and responds to the event when it is ready to do so.

In this case, we would need a 'switch state has changed' flag.

We also need a flag, or variable, to hold the "debounced", or most recently accepted state of the switch input. The ISR can then periodically compare the current "raw" switch input with the saved "debounced" input, to determine whether the switch state has changed.

This approach has the advantage that switch changes are detected quickly, while the main program does not have to respond to them immediately.

HI-TECH C implementation

In the assembler example in midrange lesson 6, the variables holding the debounced pushbutton state and the pushbutton changed flag were defined as:

```
PB_dbstate   res 1              ; bit 3 = debounced pushbutton state
                                ;   (0 = pressed, 1 = released)
PB_change    res 1              ; bit 3 = flag indicating pushbutton state change
                                ;   (1 = new debounced state)
```

This definition allocates a whole byte for each variable, even though only a single bit is needed in each case. Bit 3 was used to simplify the assembler code.

However, HI-TECH C provides a 'bit' data type, so we may as well make use of it, to simplify the C code, and to allow the compiler to pack these variables into a single byte of data memory (or not, as it sees fit – an advantage of C being that we don't have to be concerned with these implementation details[2]).

Since these variables will be updated in the ISR and accessed in the main program, they must be defined as global variables, along with the shadow copy of GPIO:

```
/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO
bit             PB_dbstate;     // debounced pushbutton state (1 = released)
bit             PB_change;      // pushbutton state change flag (1 = changed)
```

There is, however, one limitation with the way that bit variables are implemented in HI-TECH C – they cannot be initialised as part of their definition.

That is, we **cannot** write:

```
bit     PB_dbstate = 1;     // debounced pushbutton state (1 = released)
bit     PB_change = 0;      // pushbutton state change flag (1 = changed)
```

Instead, they must be initialised separately, as part of the initialisation code, before interrupts are enabled (so that they have the correct values when the ISR first runs):

```
    // initialise variables
    PB_dbstate = 1;                     // initial pushbutton state = released
    PB_change = 0;                      // clear pushbutton change flag (no change)
```

Since the debounce counter is only used within the ISR, it should be defined as being private to (within) the interrupt function, along with the timer overflow counter:

```
    static unsigned char    t0_cnt = 0;     // counts timer0 overflows
    static unsigned char    db_cnt = 0;     // debounce counter
```

Once again, these variables must be defined as being 'static', so that their values will be preserved between interrupts.

It is a good idea to define the debounce period as a constant, to make it easier to adapt the code for switches with different characteristics:

```
#define MAX_DB_CNT  20/2    // maximum debounce count =
                            //   debounce period / sample rate
                            //   (20 ms debounce period / 2 ms per sample)
```

(of course it would be cleaner still to define the debounce period and sample rate as constants, and to derive the maximum debounce count and sample timing from them – but in a short program like this it's not difficult to see how these things relate to each other, especially if it is documented in comments, as above).

---

[2] This is also a disadvantage of C – by not being aware of how the C compiler builds various constructs, we may not realize that we're doing things in an inefficient way.

The debounce routine must be run at some regular interval by the ISR. In the example in , an interval of 2 ms was used, so we'll do the same here, by incrementing and then testing a counter whenever the Timer0 interrupt is serviced:

```
// sample switch every 2 ms (8 interrupts x 250 us)
++t0_cnt;                            // increment interrupt count (every 250 us)
if (t0_cnt == 2000/250)        // until 2 ms has elapsed
{
    // debounce code goes here
}
```

Within the debounce routine, we must first determine whether the raw pushbutton state has changed since it was last debounced. Since we are using bit variables, this can be written very simply:

```
// compare raw pushbutton with current debounced state
if (BUTTON == PB_dbstate)   // if raw state matches last debounced state,
{
    // pushbutton has not changed state
}
else
{
    // pushbutton has changed state
}
```

Where previously the symbol 'BUTTON' had been defined as:

```
// Pin assignments
#define nB_LED  1            // indicator LED on GP1
#define BUTTON  GPIO3        // pushbutton on GP3 (active low)
```

Having determined whether the pushbutton's raw state has changed, we need to deal with both possibilities, as allowed for in the `if`/`else` structure above.

If the pushbutton is still in the last debounced state, all we need to do is reset the debounce counter:

```
        db_cnt = 0;                 //   reset debounce count
```

Otherwise, the pushbutton's state has changed. We need to see whether the change is stable, by counting the number of successive times we've seen it in this new state, and then check whether the maximum count has been reached, to determine whether the switch really has changed state (and has finished bouncing):

```
        ++db_cnt;                       // increment debounce count
        if (db_cnt == MAX_DB_CNT)   // when max count is reached
        {
            // accept new state as changed
        }
```

If we're accepting that the pushbutton really has changed state, we need to update the variables and flags to reflect this:

```
        PB_dbstate = !PB_dbstate;   //   toggle debounced state
        db_cnt = 0;                 //   reset debounce count
        PB_change = 1;              //   set pushbutton changed flag
```

The main program can then poll this `PB_change` flag, to see whether the button has changed state:

```
if (PB_change == 1)
{
    // pushbutton has changed state
}
```

But since this variable is a binary flag, the code can be more clearly written as:

```
if (PB_change)
{
    // pushbutton has changed state
}
```

If the button has changed state, we then need to refer to the PB_dbstate variable, to see whether it the new state is "up" or "down" (pressed); we only want to toggle the LED when the button is pressed, not when it is released, so we could write:

```
if (PB_change)
{
    // pushbutton has changed state, so check for button press
    if (PB_dbstate == 0)
    {
        // pushbutton has been pressed (low)
    }
}
```

Or, if you prefer, you can write this much more succinctly as:

```
if (PB_change && !PB_dbstate)
{
    // button state has changed and is pressed (low)
}
```

Once again, it's a question of personal style. A good C compiler will generate the same code in both cases.

Once we've determined that the button has been pressed, we can toggle the LED, using the shadow copy of GPIO, as we've done before:

```
sGPIO ^= 1<<nB_LED;            // toggle indicator LED (using shadow)
```

And finally, now that we've detected and responded to the button press, we need to clear the state change flag, to be ready for the next change:

```
PB_change = 0;                 //  clear button change flag
```

And that's all.

It's relatively complex, compared with the equivalent code in the example in lesson 2, but most of that complexity is "hidden" in the ISR; the code in the main program loop is quite simple, making it easier to do more within the main program, without having to poll and debounce switches – something that the ISR can take care of in the background.

Complete program

Here is the complete "toggle an LED on pushbutton press" program:

```
/***********************************************************************
*    Description:    Lesson 3, example 3                              *
*                                                                     *
*    Demonstrates use of Timer0 interrupt to implement               *
*    counting debounce algorithm                                      *
*                                                                     *
*    Toggles LED when the pushbutton is pressed (high -> low)         *
*                                                                     *
***********************************************************************
```

```
*    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 nB_LED  1          // indicator LED on GP1
#define BUTTON  GPIO3      // pushbutton on GP3 (active low)


/***** CONSTANTS *****/
#define MAX_DB_CNT  20/2   // max debounce count = debounce period / sample rate
                           //   (20 ms debounce period / 2 ms per sample)


/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO
bit             PB_dbstate;     // debounced pushbutton state (1 = released)
bit             PB_change;      // pushbutton state change flag (1 = changed)


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    GPIO = 0;                        // start with LED off
    sGPIO = 0;                       //   update shadow
    TRISIO = ~(1<<nB_LED);           // configure LED pin as output
    // configure Timer0
    T0CS = 0;                        // select timer mode
    PSA = 1;                         // no prescaler (assigned to WDT)
                                     //  -> increment TMR0 every 1 us
    // initialise variables
    PB_dbstate = 1;                  // initial pushbutton state = released
    PB_change = 0;                   // clear pushbutton change flag (no change)
    // configure interrupts
    T0IE = 1;                        // enable Timer0 interrupt
    ei();                            // enable global interrupts

    // Main loop
    for (;;)
    {
        // check for debounced button press
        if (PB_change && !PB_dbstate)   // if PB state changed and pressed (low)
        {
            sGPIO ^= 1<<nB_LED;         //   toggle indicator LED (using shadow)
            PB_change = 0;             //   clear button change flag
        }
        // continually copy shadow GPIO to port
        GPIO = sGPIO;

    }   // repeat forever
}
```

```
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static unsigned char    t0_cnt = 0;      // counts timer0 overflows
    static unsigned char    db_cnt = 0;      // debounce counter

    // service Timer0 interrupt
    //   TMR0 overflows every 250 clocks = 250 us
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;                  // add value to Timer0
                                        //   for overflow after 250 counts
    T0IF = 0;                           // clear interrupt flag

    // Debounce pushbutton (every 2 ms)
    //   use counting algorithm: accept change in state
    //   only if new state is seen a number of times in succession

    // sample switch every 2 ms (8 interrupts x 250 us)
    ++t0_cnt;                           // increment interrupt count (every 250 us)
    if (t0_cnt == 2000/250)             // until 2 ms has elapsed
    {
        t0_cnt = 0;                     //   reset interrupt count

        // compare raw pushbutton with current debounced state
        if (BUTTON == PB_dbstate)   // if raw state matches debounced state,
        {
            db_cnt = 0;             //   reset debounce count
        }
        else                            // else raw pushbutton has changed state
        {
            ++db_cnt;                       // increment debounce count
            if (db_cnt == MAX_DB_CNT)   // when max count is reached
            {                                   // accept new state as changed:
                PB_dbstate = !PB_dbstate;   //   toggle debounced state
                db_cnt = 0;                 //   reset debounce count
                PB_change = 1;              //   set pushbutton changed flag
                                            //   (polled and cleared in main)
            }
        }
    }
}
```

Comparisons

Here is the resource usage summary for the "toggle an LED on pushbutton press" programs:

Toggle_LED-count-int

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	87	68	7
HI-TECH PICC-Lite	39	82	10
HI-TECH C PRO Lite	39	140	8

The C source code is once again less than half as long as the assembler source. But in this example, the efficiency of hand-written assembler becomes a little more apparent, with the code generated by PICC-Lite

being 20% larger. That's not really a significant difference, but it's much bigger than in the earlier (and simpler) examples.

Example 4: Switch debouncing while flashing an LED

Since the previous example on switch debouncing was built on the framework of the earlier LED flashing examples, it's not difficult to add the LED flashing code back into the interrupt service routine, showing how a single timer-driven interrupt can be used to schedule multiple concurrent tasks.

In the assembler example in midrange lesson 6, a variable was used in the Timer0 interrupt service routine to count periods of 2 ms each (the debounce sample period), to generate a 500 ms time base, used to toggle the LED. This method (building on the existing 2 ms time base) was used in order to simplify the code, with only one additional 8-bit variable being needed.

HI-TECH C implementation

Although we could take the same approach – adding a single 8-bit variable to count 2 ms periods – the ease of handling 16-bit quantities in C means that there is little reason to do so. If you were really hard pressed to fit your variables into the available data memory, you might consider ways to save a byte here and there, although in that case, you're probably better off either using a bigger PIC or programming in assembler. We'll continue to take approaches which seem comfortable and natural from a C perspective, even if they are not necessarily the most efficient – because the emphasis when programming in C is a little different from programming in assembler.

So, as we did for the 1 Hz flashing example above, we'll define a static 16-bit variable, within the interrupt function, for the counter used to generate the 500 ms time base:

```
static unsigned char    db_t_cnt = 0;   // debounce sample timebase counter
static unsigned char    db_s_cnt = 0;   // debounce sample counter
static unsigned int     fl_t_cnt = 0;   // LED flash timebase counter
```

Note that the counter variables from the previous example have been renamed, for clarity and consistency; we now have two counters, generating two independent time bases within the same timer interrupt service routine, so it needs to be clear which is which.

And then, either before or after the debounce routine in the ISR, we need to add some code to increment the counter to generate the 500 ms time base, and flash the LED:

```
++fl_t_cnt;                         // increment interrupt count (every 250 us)
if (fl_t_cnt == 500000/250)         // until 500 ms has elapsed
{
    fl_t_cnt = 0;                   //   reset interrupt count
    sGPIO ^= 1<<nF_LED;             //   toggle LED (using shadow register)
}
```

Complete interrupt service routine

Most of the code is the same as the previous example, except for the counter variable definition and initialisation, shown above. The main loop is unchanged. But here is the new interrupt service routine, so that you can see how the LED toggling code fits in after the debounce routine:

```
/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static unsigned char    db_t_cnt = 0;   // debounce sample timebase counter
    static unsigned char    db_s_cnt = 0;   // debounce sample counter
    static unsigned int     fl_t_cnt = 0;   // LED flash timebase counter
```

```
    // service Timer0 interrupt
    //   TMR0 overflows every 250 clocks = 250 us
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;                  // add value to Timer0
                                        //   for overflow after 250 counts
    T0IF = 0;                           // clear interrupt flag

    // Debounce pushbutton (every 2 ms)
    //   use counting algorithm: accept change in state
    //   only if new state is seen a number of times in succession
    //
    // sample switch every 2 ms (8 interrupts x 250 us)
    ++db_t_cnt;                         // increment interrupt count (every 250 us)
    if (db_t_cnt == 2000/250)       // until 2 ms has elapsed
    {
        db_t_cnt = 0;               //   reset interrupt count

        // compare raw pushbutton with current debounced state
        if (BUTTON == PB_dbstate)   // if raw state matches debounced state,
        {
            db_s_cnt = 0;           //   reset debounce count
        }
        else                        // else raw pushbutton has changed state
        {
            ++db_s_cnt;                     // increment debounce count
            if (db_s_cnt == MAX_DB_CNT)     // when max count is reached
            {                               //   accept new state as changed:
                PB_dbstate = !PB_dbstate;   //     toggle debounced state
                db_s_cnt = 0;               //     reset debounce count
                PB_change = 1;              //     set pushbutton changed flag
                                            //     (polled and cleared in main)
            }
        }
    }

    // Flash LED (toggle every 500 ms)
    //
    ++fl_t_cnt;                         // increment interrupt count (every 250 us)
    if (fl_t_cnt == 500000/250)     // until 500 ms has elapsed
    {
        fl_t_cnt = 0;               //   reset interrupt count
        sGPIO ^= 1<<nF_LED;         //   toggle LED (using shadow register)
    }
}
```

Comparisons

Here is the resource usage summary for the "flash LED while toggling on pushbutton press" programs:

Flash+Toggle_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	98	77	8
HI-TECH PICC-Lite	45	104	12
HI-TECH C PRO Lite	45	164	8

The C source code continues to be less than half as long as the assembler source. However, the efficiency of hand-written assembler is becoming more apparent as the code becomes more complex, with the code generated by PICC-Lite now being 35% larger than the assembler version.

However, it's interesting to note that the HI-TECH C PRO compiler generates code which uses no more data memory than the assembler version, even when running in "Lite" mode – despite the comments above about not necessarily choosing to the most efficient approach for variable size in the C version.

External Interrupts

Although polling input pins for changes is effective in many cases, especially in user interfaces, where the human user won't notice a delay of a few milliseconds before a button press is responded to, some situations require a more immediate response.

For a very fast response to a digital signal, the external interrupt, INT (which shares its pin with GP2) can be used. This pin is *edge-triggered*, meaning that an interrupt will be triggered (if enabled) by a rising or falling transition of the input signal.

Example 5: Using a pushbutton to trigger an external interrupt

To demonstrate how to implement external interrupts, we can toggle an LED whenever the external interrupt is trigged by a pushbutton press, using the circuit from midrange lesson 6, shown on the right.

As explained in that lesson, the capacitor connected across the switch is used, in conjunction with the two resistors, to debounce the pushbutton, because it is difficult to implement software debouncing for an edge-triggered interrupt, while retaining a fast response.

This simple RC filter approach can be used because the 12F629's INT input is a Schmitt trigger type, as explained in baseline lesson 4.
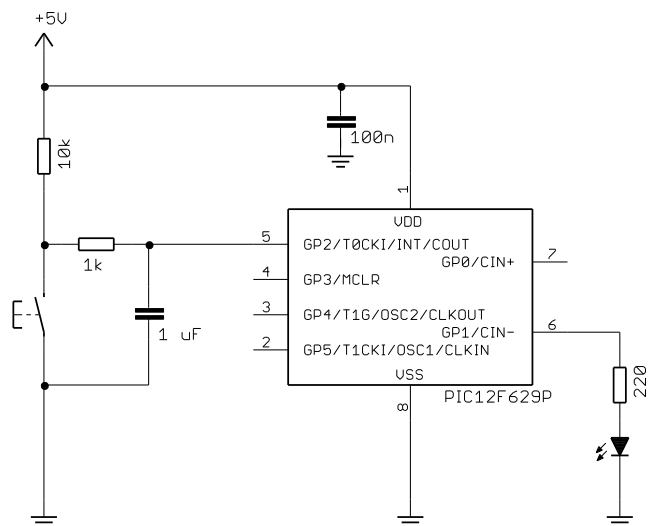
The assembler code in midrange lesson 6 configured the external interrupt, so that it would be triggered by a falling edge (high → low transition) on the INT pin (caused by the pushbutton being pressed), by clearing the INTEDG bit in the OPTION register:

```
; configure external interrupt
banksel OPTION_REG
bcf     OPTION_REG,INTEDG   ; trigger on falling edge
```

It then enabled the external interrupt, by setting the INTE bit in the INTCON register:

```
; configure interrupts
movlw   1<<GIE|1<<INTE        ; enable external and global interrupts
movwf   INTCON
```

(and we must also set GIE, as always, to globally enable interrupts)

Within the ISR, the only actions which needed to be taken were to clear the INTF interrupt flag (to indicate that the external interrupt has been serviced) and to toggle the LED on GP1:

```
        bcf     INTCON,INTF         ; clear interrupt flag
        ; toggle LED
        movlw   1<<nB_LED           ; toggle indicator LED
        xorwf   sGPIO,f             ;   using shadow register
```

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, being very similar to what we have done before.

Firstly, to select the type of transition to trigger the external interrupt:

```
    // configure external interrupt
    INTEDG = 0;                         // trigger on falling edge
```

Then to enable the external interrupt:

```
    // configure interrupts
    INTE = 1;                           // enable external interrupt
    ei();                               // enable global interrupts
```

And finally to service the external interrupt:

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

    // toggle LED
    sGPIO ^= 1<<nB_LED;                 // only update shadow register
```

Complete program

Here is how these code fragments (along with code from the previous examples) fit together:

```
/****************************************************************************
*    Description:    Lesson 3, example 5                                    *
*                                                                          *
*    Demonstrates use of external interrupt (INT pin)                      *
*                                                                          *
*    Toggles LED on GP1                                                    *
*    when pushbutton on INT is pressed (high -> low transition)            *
*                                                                          *
****************************************************************************
*                                                                          *
*    Pin assignments:                                                      *
*        GP1 - indicator LED                                               *
*        INT - pushbutton (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


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

```
/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    GPIO = 0;                           // start with LED off
    sGPIO = 0;                          //   update shadow
    TRISIO = ~(1<<nB_LED);              // configure LED pin as output
    // configure external interrupt
    INTEDG = 0;                         // trigger on falling edge
    // configure interrupts
    INTE = 1;                           // enable external 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 external interrupt
    //   Triggered on high -> low transition on INT pin
    //   caused by externally debounced pushbutton press
    //
    INTF = 0;                           // clear interrupt flag

    // toggle LED
    sGPIO ^= 1<<nB_LED;                 // only update shadow register
}
```

Comparisons

Here is the resource usage summary for the "toggle LED on external interrupt" programs:

Toggle_LED-ext_int

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	45	32	3
HI-TECH PICC-Lite	16	35	5
HI-TECH C PRO Lite	16	67	7

Once again, for a simple interrupt-based program, the C source code is only about a third as long as the assembler source, while the code generated by PICC-Lite continues to be barely any larger than the hand-written assembly version.

Example 6: Multiple interrupt sources

So far we've only used a single interrupt source, but it is common for more than one source to be active; for example, one or more timers scheduling background tasks, while servicing events such as external interrupts.

To demonstrate this, we can combine the two interrupt sources used in this lesson, with a Timer0 interrupt flashing one LED, while the external interrupt is used to toggle another LED.

This means adding an LED to the circuit in the previous example, as shown on the right.

We'll flash the LED on GP0 at 1 Hz, and toggle the LED on GP1 whenever the pushbutton is pressed, as we did in midrange lesson 6.

The program in the example in midrange lesson 6 was put together by re-using routines from the previous LED flashing and external interrupt examples.

Of course, both interrupt sources had to be enabled:

```
        ; enable interrupts
        movlw   1<<GIE|1<<T0IE|1<<INTE  ; enable external, Timer0
        movwf   INTCON                  ;   and global interrupts
```

And code had to be added to the interrupt service routine, checking the interrupt flags to determine which source had triggered the interrupt, and then branching to the appropriate service handler:

```
        ; *** Identify interrupt source
        btfsc   INTCON,INTF     ; external
        goto    ext_int
        btfsc   INTCON,T0IF     ; Timer0
        goto    t0_int
        goto    isr_end         ; none of the above, so exit
```

In this way, only one interrupt source will be serviced, each time an interrupt was triggered. If more than one interrupt is pending (more than one interrupt flag is set), another interrupt will triggered, immediately after the ISR exits, and the next interrupt source will be serviced the next time the ISR is run.

Since only one source was to be serviced when an interrupt was triggered, a 'goto' instruction was added to the end of each service handler, to skip to the end of the ISR:

For example:

```
ext_int ; *** Service external interrupt
        ;     Triggered on high -> low transition on INT pin
        ;     caused by externally debounced pushbutton press
        ;
        bcf     INTCON,INTF         ; clear interrupt flag
        ; toggle "button pressed" LED
        movlw   1<<nB_LED           ; toggle indicator LED
        xorwf   sGPIO,f             ;   using shadow register
        goto    isr_end
```
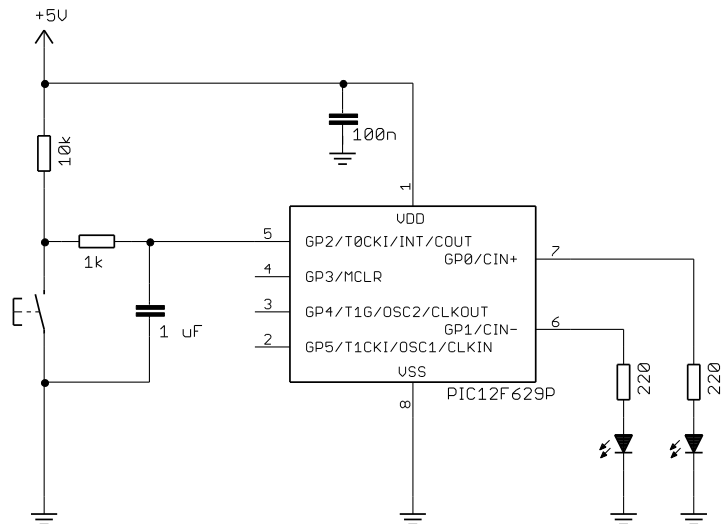
HI-TECH C implementation

When checking for multiple interrupt sources, using C, it seems most natural to use a series of 'if' statements, each testing an interrupt flag, and executing the corresponding service handler if that interrupt flag is set.

For example:

```
// Service all triggered interrupt sources

if (INTF)
{
    // External interrupt handler goes here
}

if (T0IF)
{
    // Timer0 interrupt handler goes here
}
```

With this structure, every pending interrupt source will be serviced when an interrupt is triggered. This is different from the assembly version given above, where only one source is serviced per interrupt.

The C version is perhaps clearer and has slightly less overhead (since fewer interrupts may be triggered overall), but in practice the difference is negligible.

In both approaches, the highest priority interrupt source should be serviced first – in this case we consider an external interrupt to more important (should be serviced more quickly) than a timer overflow, but that's something only you can decide, in the context of your application.

The actual interrupt handlers are the same as before, so they are easy to "plug in" to this framework.

The only other addition needed is to enable all the interrupt sources:

```
// configure interrupts
T0IE = 1;                             // enable Timer0
INTE = 1;                             // enable external interrupt
ei();                                 // enable global interrupts
```

Complete program

Here is the complete "toggle LED via external interrupt while flashing LED via timer interrupt" program, so that you can see how these pieces fit together:

```
*****************************************************************************
*    Description:    Lesson 3, example 6                                    *
*                                                                           *
*    Demonstrates handling of multiple interrupt sources                    *
*                                                                           *
*    Toggles LED on GP1 when pushbutton on INT is pressed                   *
*    (high -> low transition triggering external interrupt)                 *
*    while LED on GP0 flashes at 1 Hz (driven by Timer0 interrupt)          *
*                                                                           *
*****************************************************************************
*    Pin assignments:                                                       *
*        GP0 - flashing LED                                                 *
*        GP1 - indicator LED                                                *
*        INT - pushbutton (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 nF_LED  0               // flashing LED on GP0
#define nB_LED  1               // indicator LED on GP1


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


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    GPIO = 0;                           // start with LEDs off
    sGPIO = 0;                          //   update shadow
    TRISIO = ~(1<<nB_LED|1<<nF_LED);    // configure LED pins as output
    // configure Timer0
    T0CS = 0;                           // select timer mode
    PSA = 1;                            // no prescaler (assigned to WDT)
                                        //  -> increment TMR0 every 1 us
    // configure external interrupt
    INTEDG = 0;                         // trigger on falling edge
    // configure interrupts
    T0IE = 1;                           // enable Timer0 interrupt
    INTE = 1;                           // enable external 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)
{
    static unsigned int     fl_t_cnt = 0;   // LED flash timebase counter

    // Service all triggered interrupt sources

    if (INTF)
    {
        // External interrupt
        //    Triggered on high -> low transition on INT pin
        //    caused by externally debounced pushbutton press
        //
        INTF = 0;                       // clear interrupt flag

        // toggle LED
        sGPIO ^= 1<<nB_LED;         // only update shadow register
    }
```

```
    if (T0IF)
    {
        // Timer0 interrupt
        //   TMR0 overflows every 250 clocks = 250 us
        //    (only Timer0 interrupts are enabled)
        //
        TMR0 += 256-250+3;              // add value to Timer0
                                        //   for overflow after 250 counts
        T0IF = 0;                       // clear interrupt flag

        // Flash LED (toggle every 500 ms)
        //
        ++fl_t_cnt;                     // increment interrupt count (every 250 us)
        if (fl_t_cnt == 500000/250)     // until 500 ms has elapsed
        {
            fl_t_cnt = 0;               //   reset interrupt count
            sGPIO ^= 1<<nF_LED;         //   toggle LED (using shadow register)
        }
    }
}
```

Comparisons

Here is the resource usage summary for the "toggle LED via external interrupt while flashing LED"
programs:

Flash+Toggle_LED-ext_int

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	77	61	5
HI-TECH PICC-Lite	29	62	7
HI-TECH C PRO Lite	29	107	7

In this example, the code generated by PICC-Lite continues to be remarkably efficient, being barely any
larger than the hand-written assembly version, while the C source code remains less than half as long as the
assembler source.

Summary

These examples have demonstrated that the HI-TECH C compilers can be used to implement interrupts, in a
very straightforward way. Because the compilers take care of many of the details, such as saving and
restoring processor context, transparently, the C source code can be quite simple and succinct, as illustrated
by the code length comparisons:

Source code (lines)

Assembler / Compiler	Ex 1a	Ex 1c	Ex 2	Ex 3	Ex 4	Ex 5	Ex 6
Microchip MPASM	47	64	65	87	98	45	77
HI-TECH PICC-Lite	16	22	27	39	45	16	29
HI-TECH C PRO Lite	16	22	27	39	45	16	29

The C source code is significantly less than half the length of the assembler source in each example.

The PICC-Lite compiler was able to generate highly optimised code for the simpler examples, coming very close to the efficiency of hand-written assembler in many cases:

Program memory (words)

Assembler / Compiler	Ex 1a	Ex 1c	Ex 2	Ex 3	Ex 4	Ex 5	Ex 6
Microchip MPASM	34	49	52	68	77	32	61
HI-TECH PICC-Lite	36	52	55	82	104	35	62
HI-TECH C PRO Lite	68	92	101	140	164	67	107

However, the resource efficiency of assembler was more apparent in the examples involving more complex algorithms (the debouncing code in examples 3 and 4), where PICC-Lite generated code 20 – 35% larger than the assembler equivalent.

Once again, we see that the programs generated by the C compilers consistently use more data memory than the assembler versions:

Data memory (bytes)

Assembler / Compiler	Ex 1a	Ex 1c	Ex 2	Ex 3	Ex 4	Ex 5	Ex 6
Microchip MPASM	3	5	5	7	8	3	5
HI-TECH PICC-Lite	5	7	7	10	12	5	7
HI-TECH C PRO Lite	7	7	8	8	8	7	7

Of course this is not an important issue in these small examples, where, even in example 4, the C programs are using a maximum of 12 out of 64 bytes of data memory available on the 12F629.

The next lesson covers "interrupt on change", sleep mode, and the watchdog timer.