Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 3: Introduction to Interrupts

As we saw in mid-range lesson 6, the *interrupt* facility available on mid-range PICs is especially useful, making it much easier to implement regular "background" tasks (such as refreshing a multiplexed display – see for example baseline C lesson 5) 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 mid-range 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 Microchip's XC8 compiler[1] (running in "Free mode"), introduced in lesson 1.

In summary, this lesson covers:

- Introduction to interrupts on the mid-range PIC architecture
- Interrupt handling, using XC8
- Timer-driven interrupts
- Debouncing single switches with timer-driven interrupts
- External interrupts on the INT pin

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 mid-range PIC architecture, is always located at address 0004h (the "interrupt vector").

The XC8 compiler hides 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 mid-range PICs, the '`interrupt`' qualifier can only be used with one function, as there is only one interrupt vector in the mid-range architecture.

[1] Available as a free download from www.microchip.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 XC8 compiler takes care of this implementation detail, automatically adding appropriate context save and restore code to the '`interrupt`' function.

Timer0 Interrupts

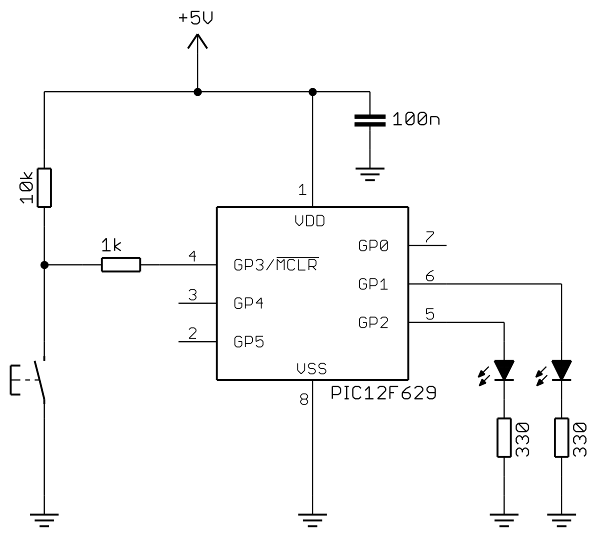
Timer0 can be used to regularly generate interrupts, which can drive "background" tasks, such as:

- 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.

If you have the Gooligum training board, close jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.



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 mid-range lesson 4 that, given a 1 MHz instruction clock with maximum prescaling (1:256), the longest period that Timer0 can generate is 256 × 256 × 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 × 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 mid-range 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:

```
; enable 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 assembler 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
        ; continually copy shadow GPIO to port
        movf    sGPIO,w
        banksel GPIO
        movwf   GPIO

        ; repeat forever
        goto    main_loop
```

XC8 implementation

As mentioned above, the XC8 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.

Additionally, any global variable which may be modified by the ISR must be declared as '`volatile`', to warn the compiler from eliminating apparently redundant references to those variables in the main program.

We'll continue to use the union construct introduced in the previous lesson for the shadow copy of GPIO. Since it is accessed by both the ISR and the main program, it must be declared as a global variable, before `main()` or the interrupt function, and qualified as '`volatile`':

```
/***** GLOBAL VARIABLES *****/
volatile union {                        // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;
```

Since we don't need to worry about saving or restoring the processor context, the XC8 version of the ISR can be very simple:

```
void interrupt isr(void)
{
    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 65.5 ms
    //
    //  Flashes LED at ~7.6 Hz by toggling on each interrupt
    //      (every ~65.5 ms)
    //
    //  (only Timer0 interrupts are enabled)
    //
    INTCONbits.T0IF = 0;            // clear interrupt flag

    // toggle LED
    sF_LED = ~sF_LED;               // (via shadow register)
}
```

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

```
// Pin assignments
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)
```

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

```
    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = ~(1<<2);               // configure GP2 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 0;         // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b111;      // prescale = 256
                                    // -> increment every 256 us
```

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:

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

However, XC8 defines a macro, '`ei()`', which is intended to be used to enable interrupts globally, and is equivalent to '`INTCONbits.GIE = 1`'.

Similarly, there is a '`di()`' macro, used to disable all interrupts, equivalent to '`INTCONbits.GIE = 0`'.

So, in keeping with the XC8 conventions, the Timer0 interrupt should be enabled by:

```
    // enable interrupts
    INTCONbits.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.port;

    }   // repeat forever
```

Complete program

Here is how these code fragments fit together:

```
/************************************************************************
*                                                                       *
*   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 <xc.h>
#include <stdint.h>


/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation
    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = ~(1<<2);               // configure GP2 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 0;         // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b111;      // prescale = 256
                                    // -> increment every 256 us

    // enable interrupts
    INTCONbits.T0IE = 1;            // enable Timer0 interrupt
    ei();                           // enable global interrupts


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

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 65.5 ms
    //
    //  Flashes LED at ~7.6 Hz by toggling on each interrupt
    //      (every ~65.5 ms)
    //
    //  (only Timer0 interrupts are enabled)
    //
    INTCONbits.T0IF = 0;            // clear interrupt flag

    // toggle LED
    sF_LED = ~sF_LED;               // (via shadow register)
}
```

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 × 256 × 256 (where N is the prescale ratio) instruction cycles.

Assuming a 1 MHz instruction clock, LED will be toggled every N × 256 × 256 µs = N × 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 × 256 × 256 µs = 524.3 ms, giving a flash rate of 0.95 Hz – close enough!

XC8 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 uint8_t  cnt_t0 = 0;       // counts timer0 overflows

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

Note that this variable is declared 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 declared 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)
    ++cnt_t0;                     // increment interrupt count (every 2.048 ms)
    if (cnt_t0 == 0)              // if count overflow (every 256 interrupts),
        sF_LED = ~sF_LED;         //   toggle LED (via shadow register)
```

This could have been written more succinctly as:

```
    // toggle LED every 256 interrupts (524 ms)
    if (++cnt_t0 == 0)            // increment count; if overflow (every 524 ms),
        sF_LED = ~sF_LED;         //   toggle LED (via 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:

```
void interrupt isr(void)
{
    static uint8_t  cnt_t0 = 0;       // counts timer0 overflows

    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 2.048 ms
    //
    //  Flashes LED at ~0.95 Hz by toggling on every 256th interrupt
    //      (every ~524 ms)
    //   (only Timer0 interrupts are enabled)
    //
    INTCONbits.T0IF = 0;            // clear interrupt flag

    // toggle LED every 256 interrupts (524 ms)
    ++cnt_t0;                     // increment interrupt count (every 2.048 ms)
    if (cnt_t0 == 0)              // if count overflow (every 256 interrupts),
        sF_LED = ~sF_LED;         //   toggle LED (via shadow register)
}
```

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

```
    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 0;         // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b010;      // prescale = 8
                                    // -> 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 mid-range 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 mid-range 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.

XC8 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 uint16_t  cnt_t0 = 0;    // counts timer0 overflows

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

Note again that, because this variable needs to be able to count up to 2000, it is defined as a 16-bit integer (`uint16_t`), instead of the 8-bit type (`uint8_t`) we used in the previous example.

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
        sF_LED = ~sF_LED;           //   toggle LED (via shadow register)
```

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

```
    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 1;         // no prescaler (assigned to WDT)
                                    // -> increment 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                                              *
*                                                                       *
************************************************************************/
```

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


/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = ~(1<<2);               // configure GP2 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 1;         // no prescaler (assigned to WDT)
                                    // -> increment every 1 us

    // enable interrupts
    INTCONbits.T0IE = 1;            // enable Timer0 interrupt
    ei();                           // enable global interrupts


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

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static uint16_t  cnt_t0 = 0;    // counts timer0 overflows

    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 250 clocks = 250 us
    //  Flashes LED at 1 Hz by toggling on every 2000th interrupt
    //      (every 500 ms)
    //
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;              // add value to Timer0
                                    //   for overflow after 250 counts
    INTCONbits.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
        sF_LED = ~sF_LED;           //   toggle LED (via shadow register)
    }
}
```

Comparisons

As we've done before, we can compare the length of the source code (ignoring comments and white space) versus program and data memory utilisation for this XC8 version with the corresponding assembly version (from mid-range lesson 6), 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-1Hz

Assembler / Compiler	**Source code (lines)**	**Program memory (words)**	**Data memory (bytes)**
Microchip MPASM	64	49	5
XC8 (Free mode)	32	85	8

Once again, the C source code is less than half as long as the assembler source, while the code generated by XC8 (with optimisation disabled) is significantly 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.

XC8 implementation

Lesson 2 included this piece of code to light the LED on GP1 only when the pushbutton on GP3 is pressed:

```
        sGPIO &= ~(1<<1);            // assume button up -> LED off
        if (GP3 == 0)                // if button pressed (GP3 low)
            sGPIO |= 1<<1;           //   turn on LED on GP1
        GPIO = sGPIO;                // update port (copy shadow to GPIO)
```

If we declare our `sGPIO` union as in example 1, and define symbols to represent the pins:

```
#define sB_LED  sGPIO.GP1            // "button pressed" indicator LED (shadow)
#define sF_LED  sGPIO.GP2            // flashing LED (shadow)
#define BUTTON  GPIObits.GP3         // pushbutton
```

We can rewrite this as:

```
        sB_LED = 0;                  // assume button up -> indicator LED off
        if (BUTTON == 0)             // if button pressed (low)
            sB_LED = 1;              //  turn on indicator LED
        GPIO = sGPIO.port;           // update port (copy shadow to GPIO)
```

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

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

    }   // 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
        sB_LED = 0;              // assume button up -> indicator LED off
        if (BUTTON == 0)         // if button pressed (low)
            sB_LED = 1;          //  turn on indicator LED

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

    }   // 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.

That's the theory, anyway. You might think that, as we did in lesson 2, we could write the code which responds to the pushbutton as:

```
        sB_LED = !BUTTON;        // turn on indicator only if button pressed
```

That's shorter, seems more natural, and is, in theory, equivalent – but in practice it doesn't work properly in this example. This happens because the code generated by XC8[2] to implement this statement does not work

[2] true for version 1.00, running in "Free mode"

correctly if it is interrupted by a routine which also modifies the `sGPIO` union – which of course our ISR, which toggles a bit within `sGPIO` to flash an LED, does. This is despite `sGPIO` being declared as '`volatile`'.

So – in practice, your ISR may interfere in non-obvious ways with your other code, if they are both updating the same variables or structures – especially when you are using a C compiler, where it may not be apparent that the generated code has this susceptibility.

But in general this isn't an issue that you would normally need to worry about. Just remember that it can happen!

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 = 0b111001;              // configure GP1 and GP2 (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 switch (active low)                            *
*                                                                       *
************************************************************************/

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


/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED  sGPIO.GP1           // "button pressed" indicator LED (shadow)
#define sF_LED  sGPIO.GP2           // flashing LED (shadow)
#define BUTTON  GPIObits.GP3        // pushbutton


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = 0b111001;              // configure GP1 and GP2 (only) as outputs

    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 1;         // no prescaler (assigned to WDT)
                                    // -> increment every 1 us

    // enable interrupts
    INTCONbits.T0IE = 1;            // enable Timer0 interrupt
    ei();                           // enable global interrupts


    //*** Main loop
    for (;;)
    {
        // check and respond to button press
        sB_LED = 0;             // assume button up -> indicator LED off
        if (BUTTON == 0)        // if button pressed (low)
            sB_LED = 1;         //  turn on indicator LED

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

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static uint16_t  cnt_t0 = 0;    // counts timer0 overflows

    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 250 clocks = 250 us
    //
    //  Flashes LED at 1 Hz by toggling on every 2000th interrupt
    //      (every 500 ms)
    //
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;              // add value to Timer0
                                    //   for overflow after 250 counts
    INTCONbits.T0IF = 0;            // clear interrupt flag
    // toggle LED every 500 ms
    ++cnt_t0;                       // increment interrupt count (every 250 us)
    if (cnt_t0 == 500000/250)       // on count overflow (every 500 ms),
    {
        cnt_t0 = 0;                 //   reset count
        sF_LED = ~sF_LED;           //   toggle LED (via shadow register)
    }
}
```

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 mid-range 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, 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.

XC8 implementation

In the assembler example in mid-range 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, XC8 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[3]).

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

```
/***** GLOBAL VARIABLES *****/
volatile union {                // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;

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

There is, however, one limitation with the way that bit variables are implemented in XC8 – they cannot be initialised as part of their definition.

That is, we **cannot** write:

```
volatile bit    PB_dbstate = 1;   // debounced pushbutton state (1 = released)
volatile 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 interrupt counter:

```
    static uint8_t  cnt_t0 = 0;     // counts timer0 interrupts
    static uint8_t  cnt_db = 0;     // debounce counter
```

[3] 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.

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).

The debounce routine must be run at some regular interval by the ISR.

In the example in mid-range lesson 6, 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
    ++cnt_t0;                       // increment interrupt count (every 250 us)
    if (cnt_t0 == 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 sB_LED  sGPIO.GP1           // indicator LED (shadow)
#define BUTTON  GPIObits.GP3        // pushbutton
```

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:

```
        cnt_db = 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):

```
        ++cnt_db;                     // increment debounce count
        if (cnt_db == 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
            cnt_db = 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)
        }
```

As ever, it's a question of personal style.

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:

```
            sB_LED = ~sB_LED;              // toggle LED (via shadow register)
```

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


/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED  sGPIO.GP1           // indicator LED (shadow)
#define BUTTON  GPIObits.GP3        // pushbutton


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


/***** GLOBAL VARIABLES *****/
volatile union {                // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;

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


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = ~(1<<1);               // configure GP1 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 1;         // no prescaler (assigned to WDT)
                                    // -> increment every 1 us

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

    // enable interrupts
    INTCONbits.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)
        {
            sB_LED = ~sB_LED;           //   toggle LED (via shadow register)
            PB_change = 0;              //   clear button change flag
        }

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

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static uint8_t  cnt_t0 = 0;     // counts timer0 interrupts
    static uint8_t  cnt_db = 0;     // debounce counter

    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 250 clocks = 250 us
    //
    //  Debounces pushbutton:
    //    samples every 2 ms (every 8th interrupt)
    //    -> PB_dbstate = debounced state
    //       PB_change  = change flag (1 = new debounced state)
    //
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;              // add value to Timer0
                                    //   for overflow after 250 counts
    INTCONbits.T0IF = 0;            // clear interrupt flag

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

    // sample switch every 2 ms
    ++cnt_t0;                       // increment interrupt count (every 250 us)
    if (cnt_t0 == 2000/250)         // until 2 ms has elapsed
    {
        cnt_t0 = 0;                 //   reset interrupt count

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

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 mid-range 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.

XC8 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 uint8_t      db_t_cnt = 0;   // debounce sample timebase counter
    static uint8_t      db_s_cnt = 0;   // debounce sample counter
    static uint16_t     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
        sF_LED = ~sF_LED;           //   toggle LED (via 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 uint8_t      db_t_cnt = 0;   // debounce sample timebase counter
    static uint8_t      db_s_cnt = 0;   // debounce sample counter
    static uint16_t     fl_t_cnt = 0;   // LED flash timebase counter

    //*** Service Timer0 interrupt
    //
    //  TMR0 overflows every 250 clocks = 250 us
    //
    //  Debounces pushbutton:
    //    samples every 2 ms (every 8th interrupt)
    //    -> PB_dbstate = debounced state
    //       PB_change  = change flag (1 = new debounced state)
    //
    //  Flashes LED at 1 Hz by toggling on every 2000th interrupt
    //      (every 500 ms)
    //
    //   (only Timer0 interrupts are enabled)
    //
    TMR0 += 256-250+3;              // add value to Timer0
                                    //   for overflow after 250 counts
    INTCONbits.T0IF = 0;            // clear interrupt flag

    // Debounce pushbutton
    //   use counting algorithm: accept change in state
    //   only if new state is seen a number of times in succession
    //
    // sample switch every 2 ms
    ++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 PB matches current debounce 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
        sF_LED = ~sF_LED;           //   toggle LED (via 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
XC8 (Free mode)	55	154	13

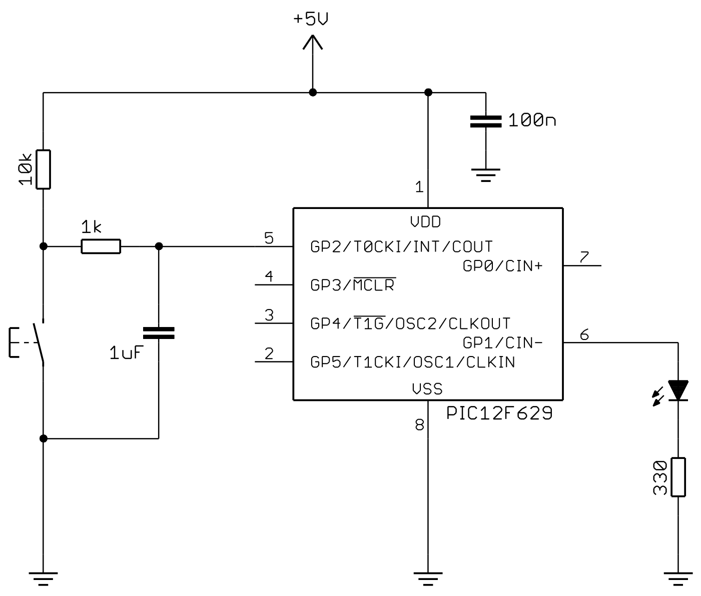
The C source code continues to be around half as long as the assembly source, while the (unoptimised) code generated by XC8 (running in “Free mode”) is more than twice as large as the assembly version.

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 show how to use external interrupts, we can toggle an LED whenever the external interrupt is trigged by a pushbutton press, using the circuit from mid-range lesson 6, shown (with the reset switch and its pull-up resistor omitted for clarity) 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.

To implement this circuit with the Gooligum training board, close jumpers JP3, JP7 and JP12 to enable the 10 kΩ pull-up resistors on MCLR and GP2 and the LED on GP1. You also need to add a 1 µF capacitor (supplied with the board) between GP2 and ground. You can do this via pins 13 ('GP/RA/RB2') and 16 ('GND') on the 16-pin expansion header. There should be no need to use the solderless breadboard – simply plug the capacitor directly into these header pins.

This simple RC filter approach can be used because the 12F629's INT input is a Schmitt trigger type, as explained in baseline assembler lesson 4.

The assembler code in mid-range 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
```

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

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

(also setting 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.

XC8 implementation

Implementing these steps using XC8 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
    OPTION_REGbits.INTEDG = 0;      // trigger on falling edge
```

Then to enable the external interrupt:

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

And finally to service the external interrupt:

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

    // toggle LED
    sB_LED = ~sB_LED;               // (via shadow register)
```

Complete program

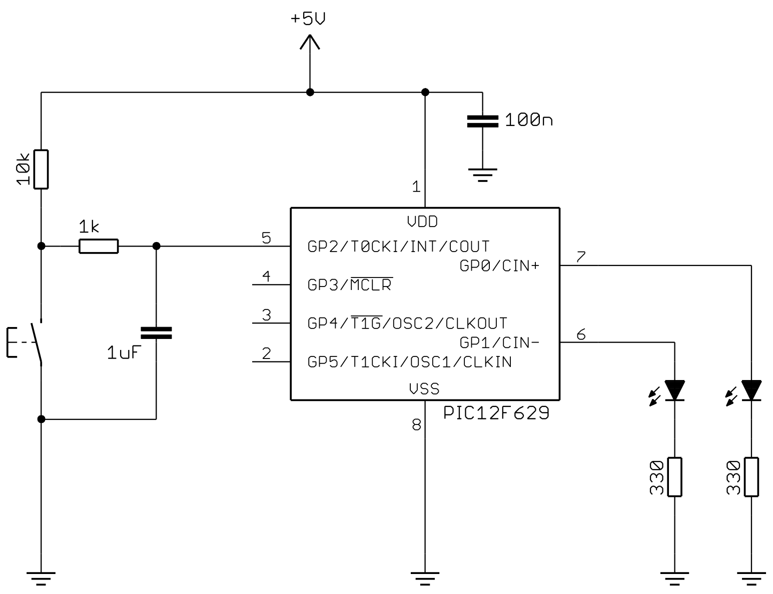
Here is how these code fragments (along with code from the previous examples) fit together:

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

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


/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED  sGPIO.GP1           // indicator LED (shadow)


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = ~(1<<1);               // configure GP1 (only) as an output

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

    // enable interrupts
    INTCONbits.INTE = 1;            // enable external interrupt
    ei();                           // enable global interrupts


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

    }   // 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
    //
    //  Toggles LED on every high -> low transition
    //
    //  (only external interrupts are enabled)
    //
    INTCONbits.INTF = 0;            // clear interrupt flag

    // toggle LED
    sB_LED = ~sB_LED;               // (via shadow register)
}
```

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.

If you have the Gooligum training board, leave it set up as in the last example, but also close jumper JP11 to enable the LED on GP0.

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 mid-range lesson 6.

The program in the example in mid-range 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 is 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
        ;
        ;   Toggles LED on every high -> low transition
        ;
        bcf     INTCON,INTF         ; clear interrupt flag

        ; toggle LED
        movlw   1<<nB_LED           ; toggle indicator LED
        xorwf   sGPIO,f             ;   using shadow register
        goto    isr_end
```

XC8 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 (INTCONbits.INTF)
    {
        // External interrupt handler goes here
    }

    if (INTCONbits.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:

```
    // enable interrupts
    INTCONbits.T0IE = 1;            // enable Timer0 interrupt
    INTCONbits.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 an LED when pushbutton on INT is pressed                    *
*   (high -> low transition triggering external interrupt)              *
*   while another LED flashes at 1 Hz (driven by Timer0 interrupt)      *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP0 = flashing LED                                              *
*       GP1 = indicator LED                                             *
*       INT = pushbutton (active low)                                   *
*                                                                       *
************************************************************************/

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


/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sF_LED  sGPIO.GP0           // flashing LED (shadow)
#define sB_LED  sGPIO.GP1           // "button pressed" indicator LED (shadow)


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = 0b111100;              // configure GP0 and GP1 (only) as outputs
    // configure Timer0
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 1;         // no prescaler (assigned to WDT)
                                    // -> increment every 1 us

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

    // enable interrupts
    INTCONbits.T0IE = 1;            // enable Timer0 interrupt
    INTCONbits.INTE = 1;            // enable external interrupt
    ei();                           // enable global interrupts


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

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static uint16_t     fl_t_cnt = 0;   // LED flash timebase counter

    // Service all triggered interrupt sources

    if (INTCONbits.INTF)
    {
        //*** Service external interrupt
        //
        //  Triggered on high -> low transition on INT pin
        //  caused by externally debounced pushbutton press
        //
        //  Toggles LED on every high -> low transition
        //
        INTCONbits.INTF = 0;            // clear interrupt flag

        // toggle LED
        sB_LED = ~sB_LED;               // (via shadow register)
    }

    if (INTCONbits.T0IF)
    {
        //*** Service Timer0 interrupt
        //
        //  TMR0 overflows every 250 clocks = 250 us
        //
        //  Flashes LED at 1 Hz by toggling on every 2000th interrupt
        //      (every 500 ms)
        //
        TMR0 += 256-250+3;              // add value to Timer0
                                        //   for overflow after 250 counts
        INTCONbits.T0IF = 0;            // clear interrupt flag

        // Flash LED (toggle every 500 ms)
        //
        ++fl_t_cnt;                     // incr interrupt count (every 250 us)
        if (fl_t_cnt == 500000/250)     // until 500 ms has elapsed
        {
            fl_t_cnt = 0;               //   reset interrupt count
            sF_LED = ~sF_LED;           //   toggle LED (via shadow register)
        }
    }
}
```

Summary

These examples have demonstrated that XC8 can be used to implement interrupts, in a very straightforward way. Because the compiler takes care of many of the details, such as saving and restoring processor context, transparently, the C source code can be quite simple and succinct.

On the other hand, we also saw that, because the compiler hides implementation details, it may be harder to uncover and avoid situations where the interrupt code interferes in a non-obvious way with operations performed on variables or structures which are updated in both the ISR and your main code.

In other words, there can be a price to pay for apparent simplicity…

Nevertheless, interrupts are too useful for tasks such as background processes (such as flashing an LED), while responding to and processing events (such as detecting and debouncing key presses), to ignore – they need to remain an important part of our toolkit, whether we're using C or not.

We'll see more examples as topics are introduced in future lessons.

The next interrupt source we'll look at is "interrupt on change", which is commonly used to wake the PIC from sleep mode. It is covered in the next lesson, along with the watchdog timer.