Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 6: Introduction to Interrupts

The lessons up until now have re-visited topics covered in the baseline assembler tutorial series, adapting the material to mid-range PICs, and introducing specific features of the mid-range architecture (not found in baseline PICs) where relevant. The most significant of these features, not present in the baseline architecture, is support for *interrupts*. As we will see in this lesson, interrupts make 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.

In summary, this lesson covers:

- Introduction to interrupts on the mid-range PIC architecture
- Interrupt service routines (including saving and restoring processor context)
- Timer-driven interrupts
- Debouncing single switches with timer-driven interrupts
- External interrupts on the INT pin

Interrupts

An *interrupt* is a means of interrupting the main program flow in response to an event, so that the event can be dealt with, 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 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.

Some examples will make this clearer! But first, some more details…

Each interrupt source can be enabled or disabled, independently.

The enable bits for the interrupt sources covered in this lesson are located in the INTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	T0IE	INTE	GPIE	T0IF	INTF	GPIF

To enable an interrupt source, that source's interrupt enable bit must be set:

T0IE enables the Timer0 interrupt, while INTE enables interrupts triggered by the external INT pin.

Interrupts are controlled overall by the global interrupt enable bit, GIE:

If GIE = 0, all interrupts are disabled.

If GIE = 1, interrupts can occur, depending on which interrupt sources are enabled.

For an interrupt to occur, that interrupt's enable bit must be set, in addition to GIE being set.

For example, to enable Timer0 interrupts, you could use:

```
        movlw   1<<GIE|1<<T0IE    ; enable interrupt on Timer0 overflow
        movwf   INTCON
```

Or, if you were setting up a number of interrupt sources and didn't want to allow interrupts to happen straight away, you might write something like:

```
        bsf     INTCON,T0IE       ; enable Timer0 interrupt source

        ...                       ; (initialise some other things)

        bsf     INTCON,GIE        ; enable interrupts
```

Context Saving

When an interrupt occurs, the current instruction completes executing, the address of the next instruction (the *return address*) is pushed onto the stack, the GIE bit is cleared to prevent any more interrupts from occurring while this interrupt is being serviced, and execution jumps to the instruction at address 0004h.

At this point, the only the program counter (PC) has been saved. Every other register holds whatever value it did when the interrupt was triggered.

As mentioned above, the ISR should be transparent to the main program. If the ISR modifies the contents of any register that the main program would "expect" to remain constant, that register should be saved at the start of the ISR, and restored to its original value returning to the main program, so that the main program will never "know" that an interrupt has occurred.

The Microchip-supplied template, '…\MPASM Suite\Template\Object\12F629TMPO.ASM', includes the following code which you can use as the framework of your interrupt service routine:

```
;-------------------------------------------------------------------------------
; INTERRUPT SERVICE ROUTINE
;-------------------------------------------------------------------------------

INT_VECTOR    CODE    0x0004  ; interrupt vector location
        MOVWF   W_TEMP        ; save off current W register contents
        MOVF    STATUS,w      ; move status register into W register
        MOVWF   STATUS_TEMP   ; save off contents of STATUS register

; isr code can go here or be located as a call subroutine elsewhere

        MOVF    STATUS_TEMP,w ; retrieve copy of STATUS register
        MOVWF   STATUS        ; restore pre-isr STATUS register contents
        SWAPF   W_TEMP,f
        SWAPF   W_TEMP,w      ; restore pre-isr W register contents
        RETFIE                ; return from interrupt
```

This code uses two variables, declared in the template code as:

```
INT_VAR     UDATA_SHR   0x20
W_TEMP      RES     1           ; variable used for context saving
STATUS_TEMP RES     1           ; variable used for context saving
```

to save the contents of the W and STATUS registers.

Note that these variables are placed in shared memory[1]. Of course, on the PIC12F629, this is the only type of data memory available; it is all shared. But even on devices with banked as well as shared memory, it is necessary to use shared memory for context saving (at least for W and STATUS), because you cannot know which bank is selected when the interrupt is triggered. If you select a specific bank by changing the bank selection bits (RP0 and RP1) in STATUS, you will lose the original value of these bits unless you save the contents of STATUS first. The only way to do that, without losing the current bank selection, is to copy STATUS to shared memory, before any bank selection is done. And since the only way to save the STATUS register is to copy it to W first, the current contents of W must be saved first.

The instructions to save the contents of W and STATUS are straightforward:

```
movwf   W_TEMP          ; save off current W register contents
movf    STATUS,w        ; move status register into W register
movwf   STATUS_TEMP     ; save off contents of STATUS register
```

After this, any other registers you wish to save (such as PCLATH) can be copied to variables in the same way – and these variables can be in banked memory, because the bank selection bits (in STATUS) have been saved.

For example:

```
movwf   W_TEMP          ; save W and STATUS
movf    STATUS,w        ; to variables in shared memory
movwf   STATUS_TEMP
movf    PCLATH,w        ; save PLCATH
banksel PCLATH_TEMP     ; to variable (can be in banked memory)
movwf   PCLATH_TEMP
```

To restore the context (W, STATUS and any other registers you choose to save, such as PCLATH) at the end of the interrupt routine, you might think that these instructions could simply be reversed:

```
banksel PCLATH_TEMP     ; restore PCLATH
movf    PCLATH_TEMP,w
movwf   PCLATH
movf    STATUS_TEMP,w ; restore STATUS
movwf   STATUS
movf    W_TEMP,w        ; restore W (NO!!! This clobbers Z flag!!!)
```

Unfortunately, **this approach won't work!**

The final `movf` instruction, used above to restore the W register, has a side effect: it affects the Z flag (part of the STATUS register), depending on the value being copied. This means that, whatever value Z had before the interrupt was triggered may be lost. Z will be set or cleared depending on the value in the W, instead of retaining the value it held when the interrupt was triggered. That's almost certain to interfere with the main code – something we must avoid.

[1] The template code explicitly specifies the address (0x20) for this shared memory section. This is unnecessary; the linker can be relied on to place any data section declared by '`UDATA_SHR`' correctly, within shared memory.

To restore W without affecting the Z flag, the template code employs a "trick":

The '`swapf`' instruction – "**swap** nybbles in **f**ile register" – is typically used where data is encoded in the two 4-bit *nybbles* (or "nibbles") comprising an 8-bit byte, as we saw when discussing binary-coded decimal (BCD) in baseline lesson 8. The contents of bits 0-3 of the specified register are swapped with bits 4-7.

If the swap operation is repeated, the nybbles end up back in their original order, leaving the data unchanged.

As with most instructions which operate on a register, the result of the swap operation can be written either back to the register, or to W. This makes it possible to use two successive `swapf` instructions to copy the original contents of a register to W, as in the ISR template code:

```
        swapf   W_TEMP,f        ; restore pre-isr W register contents
        swapf   W_TEMP,w
```

Why use this strange construct, instead of a simple '`movf`'?

The answer is that the `swapf` instruction does not affect any STATUS flags, while `movf` does. That means that it can be used to restore W without affecting STATUS, making the interrupt service routine truly transparent.

The final instruction in the ISR template is '`retfie`' – "**ret**urn **f**rom **i**nterrupt and **e**nable interrupts".

The `retfie` instruction pops the program counter off the stack, returning execution to the main program. It also sets the GIE bit, allowing interrupts to occur again.

Interrupt Flags

Given that a number of different interrupt sources may be enabled, your interrupt service routine must be able to determine which source triggered the interrupt, so that it can respond to that event.

Interrupt flags are used for this – when an interrupt event (such as a timer overflow) occurs, the corresponding interrupt flag is set, to indicate which event has occurred.

The flags for the interrupt sources covered in this lesson are also located in the INTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	T0IE	INTE	GPIE	T0IF	INTF	GPIF

T0IF indicates that Timer0 has overflowed, while INTF indicates that an external interrupt signal has been detected on the INT pin.

If an interrupt flag has been set, and the corresponding interrupt source is enabled, and the global interrupt enable (GIE) bit is also set, an interrupt will occur.

Whenever you service an interrupt, you must always clear its interrupt flag.

> *Whenever any interrupt event is serviced, the interrupt flag corresponding to that event must be cleared, or else the interrupt will be re-triggered immediately after interrupts are re-enabled, when the ISR exits.*

Note that, whenever an event occurs, the interrupt flag for that event will be set, regardless of whether that interrupt source has been enabled.

For example, you can poll the T0IF flag to check to see if Timer0 has overflowed, without having to use interrupts.

Timer0 Interrupts

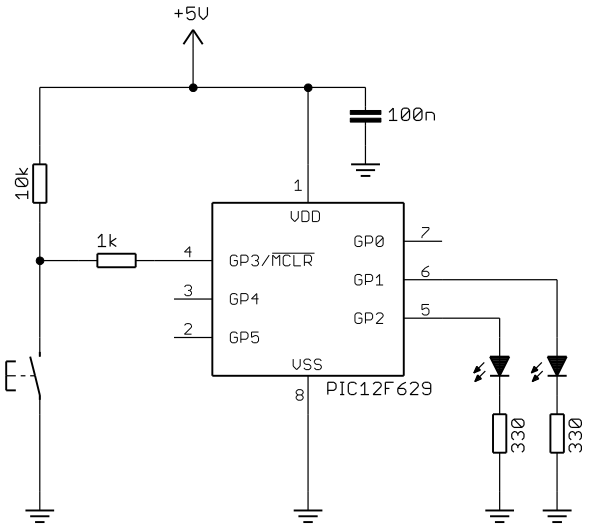
Timer0 can be used to regularly generate interrupts, which can be used to drive "background" tasks, such as:

- Generating a regular output;
  for example flashing an LED
- Monitoring and debouncing inputs

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

We'll use the circuit from lesson 4, shown on the right, to illustrate these techniques.

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 lesson 4 that, given a 1 MHz instruction clock (derived from a 4 MHz processor 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.

Firstly, we'll need some variables to save the processor context during the ISR.

It's also a good idea, when using interrupts to modify a port, to use a shadow register to avoid potential read-modify-write problems (described in baseline lesson 2). It's usually cleaner, and safer (avoiding problems) to have only the interrupt service routine or the main program writing directly to a port (such as GPIO), but not both.

So the variable definitions we need are:

```
CONTEXT     UDATA_SHR       ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO
```

Note that these could have been placed in a single section, but it's good to get into habits that will still be appropriate for larger projects on bigger PICs; the 12F629 is a little unusual in only having a single bank of shared data memory. Giving each logical group of variables its own data section gives the linker more flexibility when allocating memory.

Since the *interrupt vector* is located at address 0004h, while the reset vector (where program execution begins) is at address 0000h, we can't continue to simply place our main program at 0000h; it could only be a maximum of four instructions long!

So it's normal to place the ISR at 0004h, and to place code at 0000h which does nothing more than jump to the start of the main program, somewhere else in memory:

```
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start


;***** INTERRUPT SERVICE ROUTINE
ISR     CODE    0x0004
        ; ISR code goes here
        ; ...
        ; end of ISR


;***** MAIN PROGRAM
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL
```

Note that, because the "`MAIN`" code segment really could be placed anywhere in memory, it is necessary to use a '`pagesel`' directive, in case the main program is located on a different page.

That won't happen on the 12F629, which only has a single page of program memory, but it's a good idea to include the '`pagesel`' anyway, in case you ever move your code to a PIC with more memory.

The main program then starts by calibrating the internal RC oscillator, as shown above, before configuring and initialising the port and Timer0, as we have done before:

```
        ; configure port
        banksel GPIO
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        movlw   ~(1<<nLED)      ; configure LED pin (only) as an output
        banksel TRISIO
        movwf   TRISIO

        ; configure timer
        movlw   b'11000111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        banksel OPTION_REG      ;   -> increment TMR0 every 256 us
        movwf   OPTION_REG
```

There's nothing new or different here – the timer is simply set up as usual.

Now that everything is initialised and ready to go, the Timer0 interrupt can be enabled:

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

Note that there is no need for a '`banksel`' before accessing INTCON, because it is mapped into every bank.

With Timer0 setup, and the Timer0 interrupt enabled, an interrupt will be triggered every 65.5 ms, calling the interrupt service routine at 0004h.

As discussed above, the first thing the ISR must do is to save the processor context:

```
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS
```

We would then normally test specific interrupt flags, to determine the source of this particular interrupt. But since only Timer0 interrupts have been enabled, we know that a Timer0 overflow must have occurred, so we know that this ISR only needs to handle, or service, Timer0 overflow events.

The first thing we must do (or the last, of you prefer – it doesn't matter, as long as you ensure that you do it) is to clear the interrupt flag corresponding to this event.

In this case, because we know this is a Timer0 interrupt, we must clear T0IF:

```
        ; *** 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)
        ;
        bcf     INTCON,T0IF     ; clear interrupt flag
```

The interrupt routine is now free to do whatever it was intended to do; in this case, toggle an LED:

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

Note that only the shadow copy of GPIO is being updated, as discussed above.

Finally, the ISR must restore the processor context, before returning[2]:

```
isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie
```

As mentioned earlier, the `retfie` instruction sets the GIE bit, re-enabling interrupts so that the next event (whenever it occurs) can be serviced.

[2] The '`isr_end`' label isn't actually needed here, as it's not referenced anywhere. However, it helps to mark the end of the ISR, and it's necessary when working with multiple interrupt sources, as we'll see in the final example.

So with the ISR flipping a bit in the shadow copy of GPIO every 65.5 ms, all that remains for the main program to do is to continually copy the shadow register to the GPIO port, to make the changes made by the ISR visible (literally, in this case…):

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

        ; repeat forever
        goto    main_loop
```

Complete program

Here is how these code fragments fit together:

```
;*****************************************************************************
;   Description:    Lesson 6 example 1a                                       *
;                                                                             *
;   Demonstrates use of Timer0 interrupt to perform a background task         *
;                                                                             *
;   Flash an LED at approx 7.6 Hz (50% duty cycle).                           *
;                                                                             *
;*****************************************************************************
;                                                                             *
;   Pin assignments:                                                          *
;       GP2 = flashing LED                                                    *
;                                                                             *
;*****************************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302    ; no "register not in bank 0" warnings
    errorlevel  -312    ; no "page or bank selection not needed" messages


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

; pin assignments
    constant    nLED=2          ; flashing LED on GP2


;***** VARIABLE DEFINITIONS
CONTEXT     UDATA_SHR           ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR           ; general variables
sGPIO       res 1                   ; shadow copy of GPIO


;***** RESET VECTOR ****************************************************
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start
```

```
;***** INTERRUPT SERVICE ROUTINE ***************************************
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS

        ; *** 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)
        ;
        bcf     INTCON,T0IF     ; clear interrupt flag

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


isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie


;***** MAIN PROGRAM ****************************************************
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        movlw   ~(1<<nLED)      ; configure LED pin (only) as an output
        banksel TRISIO
        movwf   TRISIO

        ; configure timer
        movlw   b'11000111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        banksel OPTION_REG      ;   -> increment TMR0 every 256 us
        movwf   OPTION_REG

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

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

        ; repeat forever
        goto    main_loop


        END
```

Example 1b: Slower flashing

The LED in the example above flashed at around 7.6 Hz, which was done by toggling it every 65.5 ms. That's a little too fast.

We saw that, with a 4 MHz processor clock, the longest possible interval between Timer0 interrupts is 65.5 ms. So, to flash an LED any slower than this, 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 (easily done using the `incfsz` instruction).

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.

To flash the LED at 1 Hz, we need to toggle it every 500 ms. That would require N = 7.63.

That's not possible, but we can use N = 8 (prescale ratio of 1:8), which is close – the resulting toggle period is 8 × 256 × 256 µs = 524.3 ms, giving a flash rate of 0.95 Hz.

That's close enough for now!

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

```
GENVAR      UDATA_SHR           ; general variables
sGPIO       res 1                   ; shadow copy of GPIO
cnt_t0      res 1                   ; counts timer0 overflows
                                    ; (incremented by ISR every 2.048 ms)
```

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)
        incfsz  cnt_t0,f        ; increment interrupt count (every 2.048 ms)
        goto    isr_end         ; if count overflow
                                ;   (every 256 interrupts = 524 ms)
        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nLED         ;       using shadow register
        movwf   sGPIO
```

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

```
; configure timer
movlw   b'11000010'     ; configure Timer0:
        ; --0-----          timer mode (T0CS = 0)
        ; ----0---          prescaler assigned to Timer0 (PSA = 0)
        ; -----010          prescale = 8 (PS = 010)
banksel OPTION_REG      ;   -> increment TMR0 every 8 us
movwf   OPTION_REG
```

We should also initialise the timer overflow counter variable:

```
; initialise variables
clrf    cnt_t0          ; zero timer0 overflow count
```

although it's not strictly necessary in this example (its initial value only affects the first flash).

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?[3] Of course this is a contrived example, but there are many cases where an accurate output frequency must be generated; an obvious example is a real-time clock.

It's not possible to achieve this exact timing, using the technique in the example above, where the timer is allowed to run freely, with an interrupt being triggered every 256 ticks. Why? We need to toggle the LED every 500 ms, which, with a 4 MHz processor clock, is 500,000 instruction cycles. And 500,000 is not exactly divisible by 256 – there is no way to count to 500,000, using whole multiples of 256.

To solve this problem, we need to make the timer overflow (triggering an interrupt) every N ticks, where N divides exactly into 500,000. And, of course, since Timer0 is an 8-bit timer, N < 256, so it is not possible for Timer0 to count more than 256 ticks. We also want N to be as high as possible, because if Timer0 overflows less often, fewer interrupts are triggered, and less time is spent servicing interrupts.

In this case, the best result is when N = 250. That is, we want Timer0 to overflow after every 250 ticks.

To make a timer overflow after some number of ticks, you can preload it with an appropriate value. For example, if you had an 8-bit timer, and you wanted it to overflow after 100 ticks, you could load it with the value 156 (equal to 256 – 100), and then start it counting. Since it is starting from 156, after 100 ticks it will have counted to 256, and overflow back to zero. The timer could then be reloaded with 156, and count for another 100 ticks, before repeating the process.

But there are some problems with this approach – some of them specific to Timer0 on mid-range PICs:

- Timer0 is always counting; there is no way to stop it incrementing, load a value, and then restart it[4]
- When a value is written to TMR0, the timer increment is inhibited for the following two instruction cycles.

---

[3] In practice, the internal RC oscillator used in this example is only accurate to around ±2%, varying with V_{DD} and temperature. For higher accuracy, an external crystal should be used.

[4] Timer0 can be halted by selecting counter mode, but then it will be incremented if there is an external signal on the T0CKI pin, so this approach is only possible if GP2/T0CKI isn't being used; it is not a general solution.

The data sheet notes that "the user can work around this by writing an adjusted value to the TMR0 register." In other words, if you wanted to count 100 cycles, you should preload the value 158, not 156, because the timer does not increment for two cycles after the new value is written.

- Preloading a value in this way only gives accurate results when the prescaler is not used.

  The prescaler is a counter, which is not directly accessible. Whenever a value is written to TMR0, the prescaler is cleared. It will then not be incremented for the next two instruction cycles.

  For example, if the prescale ratio is set to 1:8, Timer0 normally increments every eight instruction clocks. So when a value is written to TMR0, ten instruction clocks (two plus the normal eight) will elapse before TMR0 is incremented.

  This means that, if you were using a 1:8 prescaler, and you preloaded a value to 156 to TMR0, the timer will overflow after 802 instruction cycles, not the 800 cycles (8 × 100) that you probably intended. Increasing the preloaded value to compensate for these extra two instruction cycles doesn't help – a value of 157 will cause an overflow after 794 cycles (8 × 99 + 2), not 800.

  To accurately compensate for the timer being inhibited after TMR0 is written, the prescaler should not be used.

- Some time will have elapsed between the timer overflow and the instruction where you load the new value into the timer.

  This is especially true when the timer is being updated within an interrupt service routine; there is some latency between the timer overflow event and the ISR being called (two instruction cycles on a mid-range PIC), and then the ISR must save the processor context, and potentially determine the source of the interrupt, before loading a new value into the timer.

  It's possible to account for this latency, by adjusting the value to be loaded into the timer, but only if there is no other interrupt source which may delay the timer interrupt from being triggered. If another interrupt is being serviced when the timer overflows, some unknown amount of time will elapse before the timer interrupt begins – it would be very difficult to allow for that.

Luckily, it is not difficult to avoid all these problems!

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.

For example, to make Timer0 overflow after 250 cycles, instead of the usual 256 cycles, with no prescaler, you would use:

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

This needs to be done after every Timer0 overflow (i.e. within the interrupt service routine), so that the interrupt is triggered precisely every 250 instruction cycles.

Recall that, with a 4 MHz processor clock, TMR0 will increment every 1 µs.

If we adjust TMR0 in the ISR as shown above, the interrupt will be triggered every 250 µs.

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. And since a single 8-bit variable can only hold a count up to 255, we need more than a single 8-bit variable, so that we can count up to 2000.

This could be done by using two registers to implement a single 16-bit variable (see baseline lesson 11).

However, a more useful approach in this case is to recognise that, if we count 40 interrupts, exactly 10 ms will have elapsed, since 10 ms = 40 × 250 µs. This makes it easy to schedule an operation (such as polling inputs, as we'll see later) every 10 ms – often a convenient time base.

We can then use a second variable to count 10 ms periods. After every 50 × 10 ms, 500 ms has elapsed, and the LED should be toggled.

So to count in units of 10 ms, we need two variables:

```
cnt_t0      res 1               ; counts timer0 interrupts
                                ;   (decremented by ISR every 250 us)
cnt_10ms    res 1               ; counts 10 ms periods
                                ;   (decremented by ISR every 10 ms)
```

We use the first to count for 40 interrupts, to generate a 10 ms time base:

```
        decfsz  cnt_t0,f        ; decrement interrupt count
        goto    isr_end         ; when count = 0 (every 40 interrupts = 10 ms)
        movlw   .40             ;   reload count
        movwf   cnt_t0
```

Note again that it's often easiest to use `decfsz` to count a fixed number of iterations (40, in this case).

Then we can count for 50 of these 10 ms periods, in the same way:

```
        decfsz  cnt_10ms,f      ; decrement 10 ms period count
        goto    isr_end         ; when count = 0 (every 50 times = 500 ms)
        movlw   .50             ;   reload count
        movwf   cnt_10ms
```

After 50 × 10 ms = 500 ms, we can toggle the LED, as we did before:

```
        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nLED         ;       using shadow register
        movwf   sGPIO
```

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

```
        movlw   b'11001000'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----1---          no prescaling (PSA = 1)
                                ;   (prescaler assigned to WDT)
        banksel OPTION_REG      ;   -> increment TMR0 every 1 us
        movwf   OPTION_REG
```

And we should initialise the variables used above:

```
        movlw   .40             ; timer0 overflow count = 40
        movwf   cnt_t0
        movlw   .50             ; 10 ms period count = 50
        movwf   cnt_10ms
```

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, 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 6 example 1c                          *
;                                                                *
;   Demonstrates use of Timer0 interrupt to perform a background task   *
;                                                                *
;   Flash an LED at exactly 1 Hz (50% duty cycle).               *
;                                                                *
;*****************************************************************
;                                                                *
;   Pin assignments:                                             *
;       GP2 = flashing LED                                       *
;                                                                *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302    ; no "register not in bank 0" warnings
    errorlevel  -312    ; no "page or bank selection not needed" messages


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

; pin assignments
    constant    nLED=2          ; flashing LED on GP2


;***** VARIABLE DEFINITIONS
CONTEXT     UDATA_SHR           ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR           ; general variables
sGPIO       res 1                   ; shadow copy of GPIO
cnt_t0      res 1                   ; counts timer0 interrupts
                                    ;   (decremented by ISR every 250 us)
cnt_10ms    res 1                   ; counts 10 ms periods
                                    ;   (decremented by ISR every 10 ms)


;***** RESET VECTOR ****************************************************
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start


;***** INTERRUPT SERVICE ROUTINE ***************************************
```

```
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS

        ; *** Service Timer0 interrupt
        ;
        ;   TMR0 overflows every 250 us
        ;
        ;   Flashes LED at 1 Hz by toggling on every 2000th interrupt
        ;       (every 500 ms)
        ;
        ;   (only Timer0 interrupts are enabled)
        ;
        movlw   .256-.250+.3    ; add value to Timer0
        banksel TMR0            ;   for overflow after 250 counts
        addwf   TMR0,f
        bcf     INTCON,T0IF     ; clear interrupt flag

        ; count for 10 ms (40 interrupts x 250 us)
        decfsz  cnt_t0,f        ; decrement interrupt count
        goto    isr_end         ; when count = 0 (every 40 interrupts = 10 ms)
        movlw   .40             ;   reload count
        movwf   cnt_t0

        ; toggle LED every 500 ms
        decfsz  cnt_10ms,f      ; decrement 10 ms period count
        goto    isr_end         ; when count = 0 (every 50 times = 500 ms)
        movlw   .50             ;   reload count
        movwf   cnt_10ms

        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nLED         ;       using shadow register
        movwf   sGPIO


isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie


;***** MAIN PROGRAM ****************************************************
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        movlw   ~(1<<nLED)      ; configure LED pin (only) as an output
        banksel TRISIO
        movwf   TRISIO
```

```
        ; configure timer
        movlw   b'11001000'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----1---          no prescaling (PSA = 1)
                                ;   (prescaler assigned to WDT)
        banksel OPTION_REG      ;   -> increment TMR0 every 1 us
        movwf   OPTION_REG

        ; initialise variables
        movlw   .40             ; timer0 overflow count = 40
        movwf   cnt_t0
        movlw   .50             ; 10 ms period count = 50
        movwf   cnt_10ms

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


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

        ; repeat forever
        goto    main_loop


        END
```

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 independently, "on its own", regardless of whatever the main program code is doing.[5]

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 4, 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.

In lesson 4, we used this simple piece of code to read the pushbutton and light the LED on GP1 only when it is pressed:

```
        banksel GPIO            ;     check and respond to button press
        bcf     sGPIO,GP1       ;       assume button up -> indicator LED off
        btfss   GPIO,GP3        ;       if button pressed (GP3 low)
        bsf     sGPIO,GP1       ;         turn on indicator LED

        movf    sGPIO,w         ;     update port (copy shadow to GPIO)
        movwf   GPIO
```

[5] Assuming of course that the main program continues to regularly copy the shadow register to GPIO, and does not disable the Timer0 interrupt, nor change the configuration of Timer0.

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

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

        ; repeat forever
        goto    main_loop
```

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

```
main_loop
        ; check and respond to button press
        banksel GPIO
        bcf     sGPIO,nB_LED    ; assume button up -> LED off
        btfss   GPIO,nBUTTON    ; if button pressed (low)
        bsf     sGPIO,nB_LED    ;   turn on indicator LED

        ; continually copy shadow GPIO to port
        movf    sGPIO,w
        movwf   GPIO

        ; repeat forever
        goto    main_loop
```

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.

Note that symbols have been used here, which were defined as:

```
    constant    nB_LED=1            ; "button pressed" indicator LED on GP1
    constant    nF_LED=2            ; flashing LED on GP2
    constant    nBUTTON=3           ; pushbutton on GP3
```

The only other change that has to be made to the code in example 1 is to configure both of the LED pins as outputs:

```
        movlw   ~(1<<nB_LED|1<<nF_LED)    ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO
```

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 6 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)                     *
;                                                                *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302    ; no "register not in bank 0" warnings
    errorlevel  -312    ; no "page or bank selection not needed" messages


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

; pin assignments
    constant    nB_LED=1            ; "button pressed" indicator LED on GP1
    constant    nF_LED=2            ; flashing LED on GP2
    constant    nBUTTON=3           ; pushbutton on GP3


;***** VARIABLE DEFINITIONS
CONTEXT     UDATA_SHR       ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO
cnt_t0      res 1               ; counts timer0 interrupts
                                ;   (decremented by ISR every 250 us)
cnt_10ms    res 1               ; counts 10 ms periods
                                ;   (decremented by ISR every 10 ms)


;***** RESET VECTOR ****************************************************
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start


;***** INTERRUPT SERVICE ROUTINE ***************************************
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS

        ; *** Service Timer0 interrupt
        ;
        ;   TMR0 overflows every 250 us
        ;
        ;   Flashes LED at 1 Hz by toggling on every 2000th interrupt
        ;       (every 500 ms)
        ;
        ;   (only Timer0 interrupts are enabled)
```

```
        ;
        movlw   .256-.250+.3    ; add value to Timer0
        banksel TMR0            ;   for overflow after 250 counts
        addwf   TMR0,f
        bcf     INTCON,T0IF     ; clear interrupt flag

        ; count for 10 ms (40 interrupts x 250 us)
        decfsz  cnt_t0,f        ; decrement interrupt count
        goto    isr_end         ; when count = 0 (every 40 interrupts = 10 ms)
        movlw   .40             ;   reload count
        movwf   cnt_t0

        ; toggle LED every 500 ms
        decfsz  cnt_10ms,f      ; decrement 10 ms period count
        goto    isr_end         ; when count = 0 (every 50 times = 500 ms)
        movlw   .50             ;   reload count
        movwf   cnt_10ms

        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nF_LED       ;       using shadow register
        movwf   sGPIO


isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie


;***** MAIN PROGRAM ****************************************************
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO                    ; start with all LEDs off
        clrf    sGPIO                   ;   update shadow
        movlw   ~(1<<nB_LED|1<<nF_LED)  ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO

        ; configure timer
        movlw   b'11001000'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----1---          no prescaling (PSA = 1)
                                ;   (prescaler assigned to WDT)
        banksel OPTION_REG      ;   -> increment TMR0 every 1 us
        movwf   OPTION_REG

        ; initialise variables
        movlw   .40             ; timer0 overflow count = 40
        movwf   cnt_t0
        movlw   .50             ; 10 ms period count = 50
        movwf   cnt_10ms
```

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


;***** Main loop
main_loop
        ; check and respond to button press
        banksel GPIO
        bcf     sGPIO,nB_LED    ; assume button up -> LED off
        btfss   GPIO,nBUTTON    ; if button pressed (low)
        bsf     sGPIO,nB_LED    ;   turn on indicator LED

        ; continually copy shadow GPIO to port
        movf    sGPIO,w
        movwf   GPIO

        ; repeat forever
        goto    main_loop


        END
```

Example 3: Switch debouncing

Lesson 3 explored the topic of switch bounce, and described a counting algorithm to address it, which was expressed 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.

"Continually … sampling … at a steady rate" sounds like the type of task that could be performed by a regular timer interrupt, and indeed it is common to use interrupts to continually sample and debounce inputs.

Although a number of debouncing algorithms exist, offering varying levels of sophistication, the counting algorithm presented above is effective and is easy to implement in an interrupt service routine.

But when this algorithm was implemented before, in lessons 3 and 4, separate routines were used to wait for and debounce "button up" and "button down" (low → high and high → low transitions). That approach isn't appropriate in an ISR, since it has to run independently of the main program; it can't know what type of transition the main program is waiting for. If we want to detect and debounce both types of transitions, the ISR needs to look for any change in state, and debounce it. And then it needs to have some way of reporting the fact that an input transition (change in switch state) has occurred, in case the main program chooses to act on it.

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.

Normally, this type of signalling from the ISR to the main program is done via a flag which is set by the ISR to indicate that an event 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.

To demonstrate this approach, we'll re-implement example 2 from lesson 3, where the LED on GP1 is toggled each time the pushbutton on GP3 is pressed.

We can re-use and modify the framework from the examples above, where we flashed an LED at 1 Hz.

In those examples, the ISR was triggered every 250 µs, which in turn counted interrupts, to create a 10 ms time base.

However, for sampling a switch input, 250 µs is a little too short (the more often you sample an input, the more time overall is spent in the ISR, leaving less time for the main program), while 10 ms is too long. Many switches stop bouncing within 20 ms, so if you sample every 10 ms, and have a debounce period of 20 ms, you'll be basing the decision that the switch is stable on only two samples – and two samples in a row might be a "fluke"; it's not enough to be sure that the bouncing (or glitches due to EMI) has finished.

Typically a sample rate between 1 ms and 5 ms is recommended; we'll use 2 ms here.

So the timing section of the ISR becomes:

```
; *** Service Timer0 interrupt
;
;   TMR0 overflows every 250 clocks = 250 us
;
movlw   .256-.250+.3    ; add value to Timer0
banksel TMR0            ;   for overflow after 250 counts
addwf   TMR0,f
bcf     INTCON,T0IF     ; clear interrupt flag

; count interrupts to generate 2 ms tick
decfsz  cnt_t0,f        ; decrement interrupt count
goto    isr_end         ; when count = 0
movlw   .2000/.250      ;   reload count for next 2 ms period
movwf   cnt_t0          ;   (2ms / 250 us/interrupt)
```

We also need some variables, discussed above, for the debounce algorithm:

```
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)
cnt_db      res 1           ; debounce counter
```

Note that, because only a single bit is needed to represent the switch state, or to flag that the switch has changed, we can choose to use any of the bits within these variables.

It's most convenient (the coding is simplified) if we use bit 3 of the `PB_dbstate` variable to represent the debounced state of the switch on GP3. This implies that this technique could be extended to debounce other switches at the same time, although in practice, another technique, based on "vertical counters" is more commonly used when debouncing multiple switches. We'll look at it in a later lesson.

Of course these variables should be initialised in the main program:

```
        movlw   1<<nBUTTON      ; initial pushbutton state = released
        movwf   PB_dbstate
        clrf    cnt_db          ; debounce counter = 0
        clrf    PB_change       ; pushbutton change flag = 0
```

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:

```
    constant    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)

Now for the debounce routine, run every 2 ms as part of the interrupt service routine.

First, we need to determine whether the pushbutton has changed (pressed or released) since it was last debounced.

To do so, we need to compare GPIO<3> with `PB_dbstate<3>`, and this means some logic operations:

```
        ; has raw state changed?
        banksel GPIO
        movlw   1<<nBUTTON      ; load raw button state (only) to W
        andwf   GPIO,w
        xorwf   PB_dbstate,w    ; XOR with last debounced state
        btfss   STATUS,Z        ;   (result of XOR is zero if same,
        goto    state_change    ;    so Z flag is clear if state has changed)
```

Note the use of the '`andwf`' instruction – "**and W** with **f**ile register", which ANDs the contents of W with the specified register and writes the results to either the register or W.

It is used here to apply a *mask* to GPIO, so that only GPIO<3> (the only bit we are interested in) is transferred to W. Using AND to mask bits in this way was explained in baseline lesson 8.

As we've seen before, XOR can be used to test for equality. Note however that, if we were using any other bits in `PB_dbstate`, we'd have to mask them out before doing the comparison.

Having determined whether the pushbutton's raw state has changed, we need to deal with both possibilities.

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

```
        ; raw pushbutton state has not changed
        clrf    cnt_db          ; reset debounce count
        goto    debounce_end    ; and exit
```

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:

```
state_change
        ; raw pushbutton state has changed
        incf    cnt_db,f            ; increment count
```

And then we need to check whether the maximum count has been reached, to determine whether the switch really has changed state (and has finished bouncing):

```
        movlw   MAX_DB_CNT          ; has max count been reached yet?
        xorwf   cnt_db,w
        btfss   STATUS,Z            ; if not,
        goto    debounce_end        ;   exit
```

If so, we have a new debounced state, so we can update the variables and flags to reflect this:

```
        ; accept state as changed
        movlw   1<<nBUTTON          ; toggle debounced state
        xorwf   PB_dbstate,f
        clrf    cnt_db              ; reset debounce count
        bsf     PB_change,nBUTTON   ; set pushbutton changed flag
```

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

```
        ; check for debounced button press
        btfss   PB_change,nBUTTON   ; has button state changed?
        goto    pb_press_end
```

If the button has changed state, we 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:

```
        btfsc   PB_dbstate,nBUTTON  ; is button pressed (low)?
        goto    pb_press_end
```

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

```
        ; handle button press
        movlw   1<<nB_LED           ; toggle indicator LED
        xorwf   sGPIO,f             ;   using 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:

```
        bcf     PB_change,nBUTTON   ; clear button change flag
```

And that's all.

It's relatively complex, compared with the equivalent code we saw in lessons 3 and 4, 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. This interrupt-based approach also has the advantage that switch changes are detected quickly, while the main program does not have to respond to them immediately.

Complete program

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

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 6 example 3                           *
;                                                                *
;   Demonstrates use of Timer0 interrupt to implement            *
;   counting debounce algorithm                                  *
```

```
;   Toggles LED when pushbutton is pressed (high -> low transition)     *
;                                                                       *
;************************************************************************
;   Pin assignments:                                                    *
;       GP1 = indicator LED                                             *
;       GP3 = pushbutton (active low)                                   *
;************************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302    ; no "register not in bank 0" warnings
    errorlevel  -312    ; no "page or bank selection not needed" messages


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

; pin assignments
    constant    nB_LED=1            ; "button pressed" indicator LED on GP1
    constant    nBUTTON=3           ; pushbutton on GP3


;***** CONSTANTS
    constant    MAX_DB_CNT=.20/.2   ; max debounce count
                                    ;   = debounce period / sample rate
                                    ;   (20 ms / 2 ms per sample)


;***** VARIABLE DEFINITIONS
CONTEXT     UDATA_SHR       ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO
cnt_t0      res 1               ; counts timer0 interrupts
                                ;   (decremented by ISR every 250 us)
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)
cnt_db      res 1               ; debounce counter


;***** RESET VECTOR ****************************************************
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start


;***** INTERRUPT SERVICE ROUTINE ***************************************
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS
```

```
        ; *** Service Timer0 interrupt
        ;
        ;   TMR0 overflows every 250 us
        ;
        ;   Debounces pushbutton:
        ;     samples every 2 ms (every 8th interrupt)
        ;     -> PB_dbstate<3> = debounced state
        ;        PB_change<3>  = change flag (1 = new debounced state)
        ;
        ;   (only Timer0 interrupts are enabled)
        ;
        movlw   .256-.250+.3    ; add value to Timer0
        banksel TMR0            ;   for overflow after 250 counts
        addwf   TMR0,f
        bcf     INTCON,T0IF     ; clear interrupt flag

        ; count interrupts to generate 2 ms tick
        decfsz  cnt_t0,f        ; decrement interrupt count
        goto    isr_end         ; when count = 0
        movlw   .2000/.250      ;   reload count for next 2 ms period
        movwf   cnt_t0          ;   (2ms / 250us/interrupt)

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

        ; has raw state changed?
        banksel GPIO
        movlw   1<<nBUTTON      ; load raw button state (only) to W
        andwf   GPIO,w
        xorwf   PB_dbstate,w    ; XOR with last debounced state
        btfss   STATUS,Z        ;   (result of XOR is zero if same,
        goto    state_change    ;    so Z flag is clear if state has changed)

        ; raw pushbutton state has not changed
        clrf    cnt_db          ; reset debounce count
        goto    debounce_end    ; and exit

state_change
        ; raw pushbutton state has changed
        incf    cnt_db,f            ; increment count
        movlw   MAX_DB_CNT          ; has max count been reached yet?
        xorwf   cnt_db,w
        btfss   STATUS,Z            ; if not,
        goto    debounce_end        ;   exit

        ; accept new state as changed
        movlw   1<<nBUTTON          ; toggle debounced state
        xorwf   PB_dbstate,f
        clrf    cnt_db              ; reset debounce count
        bsf     PB_change,nBUTTON   ; set pushbutton changed flag
                                    ; (polled and cleared in main loop)
debounce_end


isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie
```

```
;***** MAIN PROGRAM ****************************************************
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        movlw   ~(1<<nB_LED)    ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO

        ; configure timer
        movlw   b'11001000'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----1---          no prescaling (PSA = 1)
                                ;   (prescaler assigned to WDT)
        banksel OPTION_REG      ;   -> increment TMR0 every 1 us
        movwf   OPTION_REG

        ; initialise variables
        movlw   .2000/.250      ; timer0 overflow count = 2ms / 250us/overflow
        movwf   cnt_t0          ;   (-> 2 ms per switch sample)
        movlw   1<<nBUTTON      ; initial pushbutton state = released
        movwf   PB_dbstate
        clrf    cnt_db          ; debounce counter = 0
        clrf    PB_change       ; pushbutton change flag = 0

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


;***** Main loop
main_loop
        ; check for debounced button press
        btfss   PB_change,nBUTTON   ; has button state changed?
        goto    pb_press_end
        btfsc   PB_dbstate,nBUTTON  ; is button pressed (low)?
        goto    pb_press_end

        ; handle button press
        movlw   1<<nB_LED           ; toggle indicator LED
        xorwf   sGPIO,f             ;   using shadow register
        bcf     PB_change,nBUTTON   ; clear button change flag
pb_press_end

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

        ; repeat forever
        goto    main_loop

        END
```

Example 4: Switch debouncing while flashing an LED

Given that 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, demonstrating how a single timer-driven interrupt can be used to schedule multiple concurrent tasks.

Firstly, as before, we need a counter, so that we can count up to 500 ms:

```
cnt_2ms     res 1               ; counts 2 ms periods
                                ;   (decremented by ISR every 2 ms)
```

Note that this counter is intended to count periods of 2 ms each; this is the same as the switch sample period from the previous example. That's not a coincidence! It makes sense to make use of common time bases if possible, to avoid adding unnecessary code. And this is why a sample period of 2 ms was chosen in the last example, instead of 1 ms – to generate a 500 ms delay by counting 1 ms periods, we'd need to count to 500, and that's not possible with a single 8-bit variable. By using a time base of 2 ms, we not only have an appropriate period for sampling the switch, but we only need a single 8-bit counter to generate a 500 ms delay, since 2 ms × 250 = 500 ms, and we can count to 250 with an 8-bit variable.

We should of course initialise this counter, in the main program, before it is used:

```
        movlw   .500/.2         ; 2 ms period count = 500ms / 2ms
        movwf   cnt_2ms         ;   (-> toggle LED every 500 ms)
```

Then, either before or after the debounce routine in the ISR (it doesn't matter, since they both need to run every 2 ms), we need some code to count 2 ms periods, to create a 500 ms delay:

```
        ; toggle LED every 500 ms
        decfsz  cnt_2ms,f       ; decrement 2 ms period count
        goto    toggle_end      ; when count = 0
        movlw   .500/.2         ;   reload count for next 500 ms period
        movwf   cnt_2ms         ;   (500ms / 2ms/tick)
```

And finally, when 500 ms has elapsed, we toggle the LED, using the shadow copy of GPIO, as before:

```
        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nF_LED       ;       using shadow register
        movwf   sGPIO
```

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. 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 ***************************************
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS

        ; *** Service Timer0 interrupt
        ;
        ;   TMR0 overflows every 250 clocks = 250 us
        ;
        ;   Debounces pushbutton:
        ;     samples every 2 ms (every 8th interrupt)
        ;     -> PB_dbstate<3> = debounced state
        ;        PB_change<3>  = change flag (1 = new debounced state)
```

```
        ;
        ;   Flashes LED at 1 Hz by toggling every 500 ms
        ;       (every 250th 2 ms period)
        ;
        ;   (only Timer0 interrupts are enabled)
        ;
        movlw   .256-.250+.3    ; add value to Timer0
        banksel TMR0            ;   for overflow after 250 counts
        addwf   TMR0,f
        bcf     INTCON,T0IF     ; clear interrupt flag

        ; count interrupts to generate 2 ms tick
        decfsz  cnt_t0,f        ; decrement interrupt count
        goto    isr_end         ; when count = 0
        movlw   .2000/.250      ;   reload count for next 2 ms period
        movwf   cnt_t0          ;   (2ms / 250us/interrupt)

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

        ; has raw state changed?
        banksel GPIO
        movlw   1<<nBUTTON      ; load raw button state (only) to W
        andwf   GPIO,w
        xorwf   PB_dbstate,w    ; XOR with last debounced state
        btfss   STATUS,Z        ;   (result of XOR is zero if same,
        goto    state_change    ;    so Z flag is clear if state has changed)

        ; raw pushbutton state has not changed
        clrf    cnt_db          ; reset debounce count
        goto    debounce_end    ; and exit

state_change
        ; raw pushbutton state has changed
        incf    cnt_db,f            ; increment count
        movlw   MAX_DB_CNT          ; has max count been reached yet?
        xorwf   cnt_db,w
        btfss   STATUS,Z            ; if not,
        goto    debounce_end        ;   exit

        ; accept new state as changed
        movlw   1<<nBUTTON          ; toggle debounced state
        xorwf   PB_dbstate,f
        clrf    cnt_db              ; reset debounce count
        bsf     PB_change,nBUTTON   ; set pushbutton changed flag
                                    ; (polled and cleared in main loop)
debounce_end

        ; toggle LED every 500 ms
        decfsz  cnt_2ms,f       ; decrement 2 ms period count
        goto    toggle_end      ; when count = 0
        movlw   .500/.2         ;   reload count for next 500 ms period
        movwf   cnt_2ms         ;   (500ms / 2ms/tick)

        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nF_LED       ;       using shadow register
        movwf   sGPIO
toggle_end
```

```
isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie
```

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.

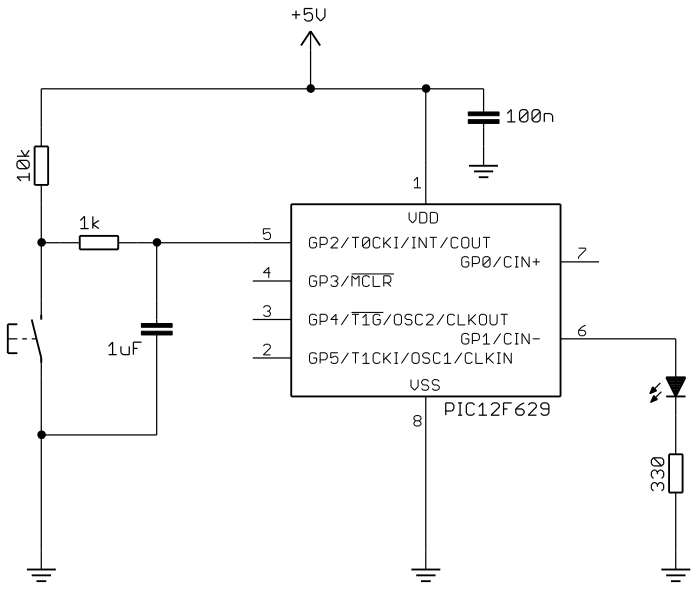
For a very fast response to a digital signal, the external interrupt pin, INT (which shares its pin with GP2) can be used. This pin is *edge-triggered*, meaning that an interrupt will be triggered (if enabled) by a rising or falling transition of the input signal.

Example 5: Using a pushbutton to trigger an external interrupt

To demonstrate how to use external interrupts, we can use a pushbutton to drive the external interrupt pin, and toggle an LED whenever the external interrupt is trigged (i.e. whenever the pushbutton is pressed).

The circuit for this (with the reset switch and its pull-up resistor omitted for clarity) is shown on the right.

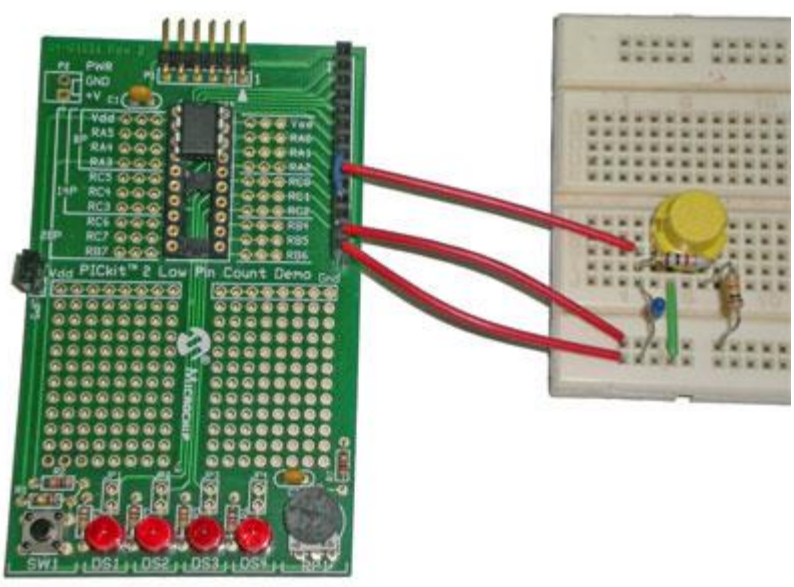
It's quite straightforward, but note the capacitor connected across the switch. This is used, in conjunction with the two resistors, to debounce the pushbutton.



The component values do not have to be exactly as shown. As a guide, the RC time constants should be roughly on the order of the debounce period. The capacitor charges though the 10 kΩ and 1 kΩ resistors, with a time constant of 11 kΩ × 1 µF = 11 ms.
It discharges, when the button is pressed, with a time constant of 1 kΩ × 1 µF = 1 ms. These figures are in line with a debounce period of 10 ms or so, but any values similar to these will be ok.

To implement this circuit with the Gooligum training board, close jumpers JP3, JP7 and JP12 to enable the 10 kΩ pull-up resistors on $\overline{\text{MCLR}}$ 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 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.



If you are using Microchip's Low Pin Count Demo Board, you can build this circuit on a solderless breadboard with a pushbutton switch, resistors and capacitor (which you will have to supply yourself), connected to the 14-pin header on the demo board, as shown on the left.

In this picture, a link has been added, from pin 8 to pin 11 on the header, so that the LED labelled 'DS2' on the demo board lights up when GP1 goes high.

When hardware debouncing was discussed in baseline lesson 4, it was pointed out that this type of simple RC filter is only effective when driving a Schmitt trigger input. Luckily, the 12F629's INT input is a Schmitt trigger type, so this simple form of hardware debouncing is quite adequate.

Of course, the switch debouncing could be done in software, but it is difficult to do for an edge-triggered interrupt, while retaining a fast response (e.g. a short glitch will trigger the interrupt, but should really be ignored – this simple circuit will effectively filter out such glitches).

As mentioned above, the external interrupt can be triggered on either the rising or falling edge of the signal on the INT pin.

The type of edge is selected by the INTEDG bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{GPPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

INTEDG = 0 selects interrupt on falling edge.

INTEDG = 1 selects interrupt on rising edge.

In this example, since we want the LED to toggle as soon as the pushbutton is pressed (which in this circuit creates a high → low transition on INT), we need to select the falling edge:

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

We then need to enable the external interrupt, by setting the INTE bit (as well as GIE, as always):

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

Finally, within the ISR, we need to service the external interrupt.

Since the INT pin is the only interrupt source enabled, it is safe to assume that every interrupt is externally triggered, so all we need to do is clear the INTF interrupt flag (recall that the interrupt flag for any interrupt source has to be cleared, when that interrupt has been serviced), and toggle the LED:

```
        bcf     INTCON,INTF         ; clear interrupt flag

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

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

Complete program

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

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 6 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)                            *
;                                                                *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302    ; no "register not in bank 0" warnings
    errorlevel  -312    ; no "page or bank selection not needed" messages


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

; pin assignments
    constant    nB_LED=1        ; "button pressed" indicator LED on GP1


;***** VARIABLE DEFINITIONS
CONTEXT     UDATA_SHR           ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR           ; general variables
sGPIO       res 1                   ; shadow copy of GPIO


;***** RESET VECTOR ****************************************************
```

```
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start


;***** INTERRUPT SERVICE ROUTINE ***************************************
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W                ; save W
        movf    STATUS,w            ; save STATUS
        movwf   cs_STATUS

        ; *** 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)
        ;
        bcf     INTCON,INTF         ; clear interrupt flag

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


isr_end ; *** Restore context then return
        movf    cs_STATUS,w         ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f              ; restore W
        swapf   cs_W,w
        retfie


;***** MAIN PROGRAM ****************************************************
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF              ; retrieve factory calibration value
        banksel OSCCAL              ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL              ;   then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO                ; start with all LEDs off
        clrf    sGPIO               ;   update shadow
        movlw   ~(1<<nB_LED)        ; configure LED pin (only) as an output
        banksel TRISIO
        movwf   TRISIO

        ; configure external interrupt
        banksel OPTION_REG
        bcf     OPTION_REG,INTEDG   ; trigger on falling edge (INTEDG = 0)

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

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

        ; repeat forever
        goto    main_loop


        END
```

Example 6: Multiple interrupt sources

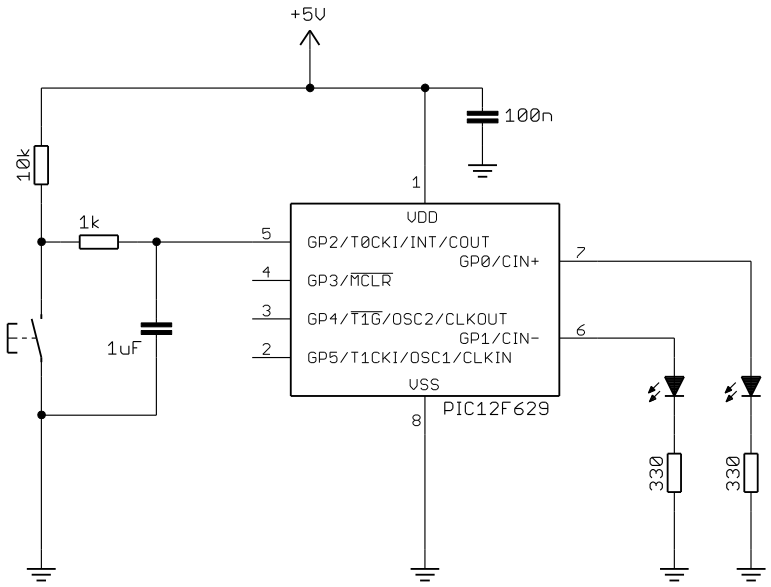
So far we've only used a single interrupt source, but it is common for more than one source to be active; for example, one or more timers scheduling background tasks, while servicing events such as external interrupts.

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 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.



In the main program, having configured Timer0 and selected the appropriate edge (falling) for the external interrupt, we need to enable both interrupt sources (as well as global interrupts):

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

The interrupt service routine must include code to service both types of interrupt, but first we need to determine which source has triggered this interrupt – and that can be done by testing the various interrupt flags, as follows:

```
        ; *** 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
```

The order is important, because it is possible that more than one interrupt source has triggered – that is, more than one of these flags may be set. That's possible because more than one interrupt-triggering event, such as a timer overflow or an external signal, may have occurred while interrupts were disabled (for example, while another interrupt was being serviced).

So, if some interrupt sources (such as external events) are more important than others (such as timer overflows), you should structure your ISR so that the highest-priority interrupt sources are serviced first.

Note that the last instruction, '`goto isr_end`', should never be executed. It is there to handle the case where an interrupt is triggered by a source that you haven't written a handler for. If your hardware has a means of logging or informing the user of an error condition, you could use that capability here. Or it might be safest to reset your hardware, because clearly something has gone wrong! In this example, we just ignore the problem by immediately exiting the ISR. If you're sure that nothing can ever go wrong, you could leave out this "catch all" `goto`.

The individual interrupt handlers are the same as before, except that they must finish with an instruction that skips to the end of the ISR, so that the other handlers are not executed.

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
```

Of course, the handler immediately preceding the end of the ISR doesn't need this '`goto isr_end`' instruction, since it is at the end of the ISR anyway, but it's a good idea to include it regardless, because it makes it easier to add more interrupt handlers later, without having to remember to add this '`goto`'.

Complete program

Here is the complete "toggle LED via external interrupt while flashing LED via timer interrupt" program, so that you can see how it all fits together:

```
;*****************************************************************
;   Description:    Lesson 6 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 = "button pressed" indicator LED                    *
;       INT = pushbutton (active low)                           *
;                                                               *
;*****************************************************************
```

```
    list        p=12F629
    #include    <p12F629.inc>

    errorlevel  -302    ; no "register not in bank 0" warnings
    errorlevel  -312    ; no "page or bank selection not needed" messages


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

; pin assignments
    constant    nF_LED=0            ; flashing LED on GP0
    constant    nB_LED=1            ; "button pressed" indicator LED on GP1


;***** VARIABLE DEFINITIONS
CONTEXT     UDATA_SHR       ; variables used for context saving
cs_W        res 1
cs_STATUS   res 1

GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO
cnt_t0      res 1               ; counts timer0 interrupts
                                ;   (decremented by ISR every 250 us)
cnt_5ms   res 1             ; counts 5 ms periods
                                ;   (decremented by ISR every 5 ms)


;***** RESET VECTOR ****************************************************
RESET   CODE    0x0000          ; processor reset vector
        pagesel start
        goto    start


;***** INTERRUPT SERVICE ROUTINE ***************************************
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W            ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS

        ; *** 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


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


t0_int  ; *** Service Timer0 interrupt
        ;
        ;   TMR0 overflows every 250 clocks = 250 us
        ;
        ;   Flashes LED at 1 Hz by toggling every 500 ms
        ;       (every 250th 2 ms period)
        ;
        movlw   .256-.250+.3    ; add value to Timer0
        banksel TMR0            ;   for overflow after 250 counts
        addwf   TMR0,f
        bcf     INTCON,T0IF     ; clear interrupt flag

        ; count interrupts to generate 5 ms tick
        decfsz  cnt_t0,f        ; decrement interrupt count
        goto    isr_end         ; when count = 0
        movlw   .5000/.250      ;   reload count for next 5 ms period
        movwf   cnt_t0          ;   (5ms / 250us/interrupt)

        ; toggle flashing LED every 500 ms
        decfsz  cnt_5ms,f       ; decrement 5 ms tick count
        goto    flash_end       ; when count = 0
        movlw   .500/.5         ;   reload count for next 500 ms period
        movwf   cnt_5ms         ;   (500ms / 2ms/tick)

        movf    sGPIO,w         ;   toggle LED
        xorlw   1<<nF_LED       ;       using shadow register
        movwf   sGPIO
flash_end
        goto    isr_end


isr_end ; *** Restore context then return
        movf    cs_STATUS,w     ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f          ; restore W
        swapf   cs_W,w
        retfie


;***** MAIN PROGRAM ****************************************************
MAIN    CODE
start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   (stored at 0x3FF as a retlw k)
        movwf   OSCCAL          ;   then update OSCCAL

;***** Initialisation

        ; configure port
        banksel GPIO
        clrf    GPIO                    ; start with all LEDs off
        clrf    sGPIO                   ;   update shadow
        movlw   ~(1<<nB_LED|1<<nF_LED)  ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO
```

```
        ; configure timer
        movlw   b'11001000'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----1---          no prescaling (PSA = 1)
                                ;   (prescaler assigned to WDT)
        banksel OPTION_REG      ;   -> increment TMR0 every 1 us
        movwf   OPTION_REG

        ; initialise variables
        movlw   .5000/.250      ; timer0 overflow count = 5ms / 250us/overflow
        movwf   cnt_t0          ;   (-> 5 ms per tick)
        movlw   .500/.5         ; 5 ms tick count = 500ms / 5ms
        movwf   cnt_5ms         ;   (-> toggle LED every 500 ms)

        ; configure external interrupt
        banksel OPTION_REG
        bcf     OPTION_REG,INTEDG   ; trigger on falling edge (INTEDG = 0)

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


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

        ; repeat forever
        goto    main_loop


        END
```

Conclusion

Although this lesson has barely scratched the surface of what can be done with interrupts on mid-range PICs, we've seen, especially in examples 4 and 6, how interrupts make it possible to maintain background tasks (such as flashing an LED), while responding to and processing events (such as detecting and debouncing key presses), in a way that would be much more difficult to achieve if interrupts were not available.

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.