Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 5: Using Timer0

The lessons until now have covered the essentials of baseline PIC microcontroller operation: controlling digital outputs, timed via programmed delays, with program flow responding to digital inputs. That's all you really need to perform a great many tasks; such is the versatility of these devices. But PICs (and most other microcontrollers) offer a number of additional features that make many tasks much easier. Possibly the most useful of all are *timers*; so useful that at least one is included in every current 8-bit PIC.

A timer is simply a counter, which increments automatically. It can be driven by the processor's instruction clock, in which case it is referred to as a *timer*, incrementing at some predefined, steady rate. Or it can be driven by an external signal, where it acts as a *counter*, counting transitions on an input pin. Either way, the timer continues to count, independently, while the PIC performs other tasks.

And that is why timers are so very useful. Most programs need to perform a number of concurrent tasks; even something as simple as monitoring a switch while flashing an LED. The execution path taken within a program will generally depend on real-world inputs. So it is very difficult in practice to use programmed delay loops, as in lesson 2, as an accurate way to measure elapsed time. But a timer will just keep counting, steadily, while your program responds to various inputs, performs calculations, or whatever.

As we'll see when we look at mid-range PICs, timers are commonly used to drive *interrupts* (routines which interrupt the normal program flow) to allow regularly timed "background" tasks to run. The baseline architecture doesn't support interrupts, but, as we'll see, timers are nevertheless very useful.

This lesson covers:

- Introduction to the Timer0 module
- Creating delays with Timer0
- Debouncing via Timer0
- Using Timer0 counter mode with an external clock

Timer0 Module

The baseline PICs provide only a single timer, referred to these days as Timer0. It used to be called the Real Time Clock Counter (RTCC), and you will find it called RTCC in some older literature. When Microchip released more advanced PICs, with more than one timer, they started to refer to the RTCC as Timer0.

Timer0 is very simple. The visible part is a single 8-bit register, TMR0, which holds the current value of the timer. It is readable and writeable. If you write a value to it, the timer is reset to that value and then starts incrementing from there. When it has reached 255, it rolls over to 0, and then continues to increment.

In the baseline architecture, there is no "overflow flag" to indicate that TMR0 has rolled over from 255 to 0; the only way to check the status of the timer is to read TMR0.

As mentioned above, TMR0 can be driven by either the instruction clock (FOSC/4) or an external signal.

The configuration of Timer0 is set by a number of bits in the OPTION register:

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

The clock source is selected by the T0CS bit:

T0CS = 0 selects timer mode, where TMR0 is incremented at a fixed rate by the instruction clock.

T0CS = 1 selects counter mode, where TMR0 is incremented by an external signal, on the T0CKI pin. On the PIC12F508/9, this is physically the same pin as GP2.

> *Note that if T0CS is set to '1', it overrides the TRIS setting for GP2. That is, GP2 cannot be used as an output until T0CS is cleared. All the OPTION bits are set to '1' at power on, so you must remember to clear T0CS before using GP2 as an output. Instances like this, where multiple functions are mapped to a single pin, can be a trap for beginners, so be careful!*
> *These "traps" are often highlighted in the data sheets, so read them carefully!*

T0CKI is a Schmitt Trigger input, meaning that it can be driven by and will respond cleanly to a smoothly varying input voltage (e.g. a sine wave), even with a low level of superimposed noise; it doesn't have to be a sharply defined TTL-level signal, as required by the GP inputs.

In counter mode, the T0SE bit selects whether Timer0 responds to rising or falling signals ("edges") on T0CKI. Clearing T0SE to '0' selects the rising edge; setting T0SE to '1' selects the falling edge.

Prescaler

By default, the timer increments by one for every instruction cycle (in timer mode) or transition on T0CKI (in counter mode). If timer mode is selected, and the processor is clocked at 4 MHz, the timer will increment at the instruction cycle rate of 1 MHz. That is, TMR0 will increment every 1 µs. Thus, with a 4 MHz clock, the maximum period that Timer0 can measure directly, by default, is 255 µs.

To measure longer periods, we need to use the *prescaler*.

The prescaler sits between the clock source and the timer. It is used to reduce the clock rate seen by the timer, by dividing it by a power of two: 2, 4, 8, 16, 32, 64, 128 or 256.

To use the prescaler with Timer0, clear the PSA bit to '0'.

[If PSA = 1, the prescaler is instead assigned to the watchdog timer – a topic covered in lesson 7.]

When assigned to Timer0, the prescale ratio is set by the PS<2:0> bits, as shown in the following table:

PS<2:0> bit value	Timer0 prescale ratio
000	1 : 2
001	1 : 4
010	1 : 8
011	1 : 16
100	1 : 32
101	1 : 64
110	1 : 128
111	1 : 256

If PSA = 0 (assigning the prescaler to Timer0) and PS<2:0> = '111' (selecting a ratio of 1:256), TMR0 will increment every 256 instruction cycles in timer mode. Given a 1 MHz instruction cycle rate, the timer would increment every 256 µs.

Thus, when using the prescaler with a 4 MHz processor clock, the maximum period that Timer0 can measure directly is 255 × 256 µs = 65.28ms.

Note that the prescaler can also be used in counter mode, in which case it divides the external signal on T0CKI by the prescale ratio.

If you don't want to use the prescaler with Timer0, set PSA to '1'.

To make all this theory clearer (hopefully!), here are some practical examples…

Timer Mode

The examples in this section demonstrate the use of Timer0 in timer mode, to:

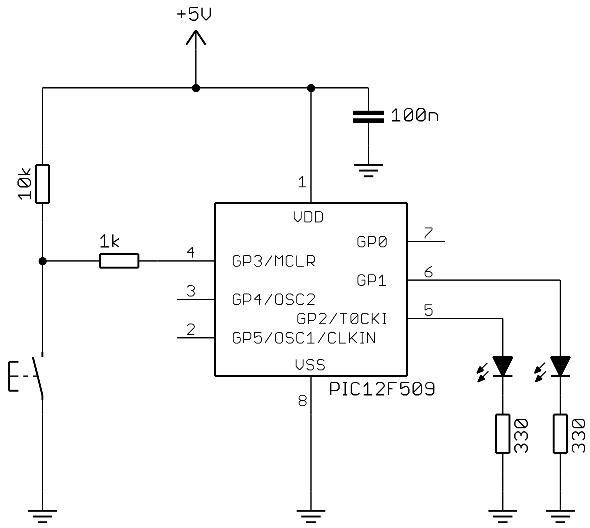
- Measure elapsed time
- Perform a regular task while responding to user input
- Debounce a switch

For each of these, we'll use the circuit shown on the right, which adds an LED to the circuit used in lesson 4.

The second LED has been added to GP2, although any of the unused pins would have been suitable.

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

If you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in lesson 1.



Example 1: Reaction Timer

To illustrate how Timer0 can be used to measure elapsed time, we'll implement a very simple reaction time "game": wait a couple of seconds then light an LED to indicate 'start'. If the button is pressed within a predefined time (say 200 ms) light the other LED to indicate 'success'. If the user is too slow, leave the 'success' LED unlit. Either way, delay another second before turning off the LEDs and restarting.

There are many enhancements we could add, to make this a better game. For example, success/fail could be indicated by a bi-colour red/green LED. The delay prior to the 'start' indication should be random, so that it's difficult to cheat by predicting when it's going to turn on. The difficulty level could be made adjustable, and the measured reaction time in milliseconds could be displayed, using 7-segment displays. You can probably think of more – but the intent of here is to keep it as simple as possible, while providing a real-world example of using Timer0 to measure elapsed time.

We'll use the LED on GP2 as the 'start' signal and the LED on GP1 to indicate "success".

The program flow can be illustrated in pseudo-code as:

```
do forever
        turn off both LEDs
        delay 2 sec
        indicate start
        clear timer
        wait up to 1 sec for button press
        if button pressed and elapsed time < 200ms
                indicate success
        delay 1 sec
end
```

A problem is immediately apparent: even with maximum prescaling, Timer0 can only measure up to 65 ms. To overcome this, we need to extend the range of the timer by adding a counter variable, which is incremented when the timer overflows. That means monitoring the value in TMR0 and incrementing the counter variable when TMR0 reaches a certain value.

This example utilises the (nominally) 4 MHz internal RC clock, giving an instruction cycle time of (approximately) 1 µs. Using the prescaler, with a ratio of 1:32, means that the timer increments every 32 µs. If we clear TMR0 and then wait until TMR0 = 250, 8 ms (250 × 32 µs) will have elapsed. If we then reset TMR0 and increment a counter variable, we've implemented a counter which increments every 8 ms. Since 25 × 8 ms = 200 ms, 200 ms will have elapsed when the counter reaches 25. Hence, any counter value > 25 means the allowed time has been exceeded. And since 125 × 8 ms = 1 s, one second will have elapsed when the counter reaches 125, and we can stop waiting for the button press.

The following code sets Timer0 to timer mode (freeing GP2 to be used as an output), with the prescaler assigned to Timer0, with a 1:32 prescale ratio by:

```
        movlw   b'11010100'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----100          prescale = 32 (PS = 100)
        option                  ;   -> increment every 32 us
```

Assuming a 4 MHz clock, such as the internal RC oscillator, TMR0 will increment every 32 µs.

To generate an 8 ms delay, we can clear TMR0 and then wait until it reaches 250, as follows:

```
        clrf    TMR0            ; clear Timer0
w_tmr0  movf    TMR0,w          ; wait for 8 ms
        xorlw   .250            ;  (250 ticks x 32 us/tick = 8 ms)
        btfss   STATUS,Z
        goto    w_tmr0
```

Note that XOR is used to test for equality (TMR0 = 250), as we did in lesson 4.

In itself, that's an elegant way to create a delay; it's much shorter and simpler than "busy loops", such as the delay routines from lessons 2 and 3.

But the real advantage of using a timer is that it keeps ticking over, at the same rate, while other instructions are executed. That means that additional instructions can be inserted into this "timer wait" loop, without affecting the timing – within reason; if this extra code takes too long to run, the timer may increment more than once before it is checked at the end of the loop, and the loop may not finish when intended.

With 32 instruction cycles per timer increment, it's safe to insert a short piece of code to check whether the pushbutton has been checked, without risk of skipping a timer increment.

For example:

```
        clrf    TMR0            ;   clear Timer0
w_tmr0                          ;   repeat for 8 ms:
        btfss   GPIO,3          ;     if button pressed (GP3 low)
        goto    wait_end        ;       finish delay loop immediately
        movf    TMR0,w          ;
        xorlw   .250            ;    (250 ticks x 32 us/tick = 8 ms)
        btfss   STATUS,Z
        goto    w_tmr0
wait_end
```

This timer loop code can then be embedded into an outer loop which increments a variable used to count the number of 8 ms periods, as follows:

```
        banksel cnt_8ms         ; clear timer (8 ms counter)
        clrf    cnt_8ms         ; repeat for 1 sec:
wait1s  clrf    TMR0            ;   clear Timer0
w_tmr0                          ;   repeat for 8 ms:
        btfss   GPIO,3          ;     if button pressed (GP3 low)
        goto    wait1s_end      ;       finish delay loop immediately
        movf    TMR0,w
        xorlw   .250            ;   (250 ticks x 32 us/tick = 8 ms)
        btfss   STATUS,Z
        goto    w_tmr0
        incf    cnt_8ms,f       ;   increment 8 ms counter
        movlw   .125            ; (125 x 8 ms = 1 sec)
        xorwf   cnt_8ms,w
        btfss   STATUS,Z
        goto    wait1s
wait1s_end
```

The test at the end of the outer loop (`cnt_8ms` = 125) ensures that the loop completes when one second has elapsed, if the button has not yet been pressed.

Finally, we need to check whether the user has pressed the button quickly enough (if at all). That means comparing the elapsed time, as measured by the 8 ms counter, with some threshold value – in this case 25, corresponding to a reaction time of 200 ms. The user has been successful if the 8 ms count is less than 25.

The easiest way to compare the magnitude of two values (is one larger or smaller than the other?) is to subtract them, and see if a *borrow* results.

If A ≥ B, A − B is positive or zero and no borrow is needed.

If A < B, A − B is negative, requiring a borrow.

The baseline PICs provide just a single instruction for subtraction: '`subwf f,d`' – "**sub**tract **W** from **f**ile register", where '`f`' is the register being subtracted from, and, '`d`' is the destination; '`,f`' to write the result back to the register, or '`,w`' to place the result in W.

The result of the subtraction is reflected in the Z (zero) and C (carry) bits in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

The Z bit is set if and only if the result is zero (so subtraction is another way to test for equality).

Although the C bit is called "carry", in a subtraction it acts as a "not borrow". That is, it is set to '1' only if a borrow did *not* occur.

The table at the right shows the possible status flag outcomes from the subtraction A − B:

	Z	C
A > B	0	1
A = B	1	1
A < B	0	0

We can use this to test whether the elapsed time is less than 200 ms (`cnt_8ms` < 25) as follows:

```
        movlw   .25             ; if time < 200 ms (25 x 8 ms)
        subwf   cnt_8ms,w
        btfss   STATUS,C
        bsf     GPIO,1          ;   turn on success LED
```

The subtraction being performed here is `cnt_8ms` − 25, so C = 0 only if `cnt_8ms` < 25 (see the table above). If C = 1, the elapsed time must be greater than the allowed 200 ms, and the instruction to turn on the success LED is skipped.

Complete program

Here's the complete code for the reaction timer, showing how the above code fragments fit together:

```
;************************************************************************
;   Description:    Lesson 5, example 1                                 *
;                   Reaction Timer game.                                *
;                                                                       *
;   Demonstrates use of Timer0 to time real-world events                *
;                                                                       *
;   User must attempt to press button within 200 ms of "start" LED      *
;   lighting.  If and only if successful, "success" LED is lit.         *
;                                                                       *
;       Starts with both LEDs unlit.                                    *
;       2 sec delay before lighting "start"                             *
;       Waits up to 1 sec for button press                              *
;       (only) on button press, lights "success"                        *
;       1 sec delay before repeating from start                         *
;                                                                       *
;************************************************************************
;   Pin assignments:                                                    *
;       GP1 = success LED                                               *
;       GP2 = start LED                                                 *
;       GP3 = pushbutton switch (active low)                            *
;************************************************************************
    list        p=12F509
    #include    <p12F509.inc>

    EXTERN  delay10_R       ; W x 10 ms delay

;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC


;***** VARIABLE DEFINITIONS
        UDATA
cnt_8ms res 1                   ; counter: increments every 8 ms


;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration
        pagesel start
        goto    start           ; jump to main code

;***** Subroutine vectors
delay10                         ; delay W x 10 ms
        pagesel delay10_R
        goto    delay10_R


;***** MAIN PROGRAM *****************************************************
MAIN    CODE
```

```
;***** Initialisation
start
        ; configure ports
        movlw   b'111001'       ; configure GP1 and GP2 (only) as outputs
        tris    GPIO
        ; configure timer
        movlw   b'11010100'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----100          prescale = 32 (PS = 100)
        option                  ;   -> increment every 32 us

;***** Main loop
main_loop
        ; turn off both LEDs
        clrf    GPIO

        ; delay 2 sec
        movlw   .200            ; 200 x 10 ms = 2 sec
        pagesel delay10
        call    delay10
        pagesel $

        ; indicate start
        bsf     GPIO,2          ; turn on start LED

        ; wait up to 1 sec for button press
        banksel cnt_8ms         ; clear timer (8 ms counter)
        clrf    cnt_8ms         ; repeat for 1 sec:
wait1s  clrf    TMR0            ;   clear Timer0
w_tmr0                          ;   repeat for 8 ms:
        btfss   GPIO,3          ;     if button pressed (GP3 low)
        goto    wait1s_end      ;       finish delay loop immediately
        movf    TMR0,w
        xorlw   .250            ;   (250 ticks x 32 us/tick = 8 ms)
        btfss   STATUS,Z
        goto    w_tmr0
        incf    cnt_8ms,f       ;   increment 8 ms counter
        movlw   .125            ; (125 x 8 ms = 1 sec)
        xorwf   cnt_8ms,w
        btfss   STATUS,Z
        goto    wait1s
wait1s_end

        ; indicate success if elapsed time < 200 ms
        movlw   .25             ; if time < 200 ms (25 x 8 ms)
        subwf   cnt_8ms,w
        btfss   STATUS,C
        bsf     GPIO,1          ;   turn on success LED

        ; delay 1 sec
        movlw   .100            ; 100 x 10 ms = 1 sec
        pagesel delay10
        call    delay10
        pagesel $

        ; repeat forever
        goto    main_loop


        END
```

Example 2: Flash LED while responding to input

As discussed above, timers can be used to maintain the accurate timing of regular ("background") events, while performing other actions in response to input signals. To illustrate this, we'll flash the LED on GP2 at 1 Hz (similar to lesson 2), while lighting the LED on GP1 whenever the pushbutton on GP3 is pressed (as was done in lesson 4). This example also shows how Timer0 can be used to provide a fixed delay.

When creating an application which performs a number of tasks, it is best, if practical, to implement and test each of those tasks separately. In other words, build the application a piece at a time, adding each new part to base that is known to be working. So we'll start by simply flashing the LED.

The delay needs to written in such a way that button scanning code can be added within it later. Calling a delay subroutine, as was done in lesson 3, wouldn't be appropriate; if the button press was only checked at the start and/or end of the delay, the button would seem unresponsive (a 0.5 sec delay is very noticeable).

Since the maximum delay that Timer0 can generate directly from a 1 MHz instruction clock is 65 ms, we have to extend the timer by adding a counter variable, as we did in example 1.

To produce a given delay, various combinations of prescaler value, maximum timer count and number of repetitions will be possible. But noting that 125 × 125 × 32 µs = 500 ms, a delay of exactly 500 ms can be generated by:

- Using a 4 MHz processor clock, providing a 1 MHz instruction clock and a 1 µs instruction cycle
- Assigning a 1:32 prescaler to the instruction clock, incrementing Timer0 every 32 µs
- Resetting Timer0 to zero, as soon as it reaches 125 (i.e. every 125 × 32 µs = 4 ms)
- Repeating 125 times, creating a delay of 125 × 4 ms = 500 ms.

The following code implements the above steps:

```
;***** Initialisation
start
        ; configure ports
        clrf    GPIO                ; start with all LEDs off
        clrf    sGPIO               ;   update shadow
        movlw   b'111001'           ; configure GP1 and GP2 (only) as outputs
        tris    GPIO
        ; configure timer
        movlw   b'11010100'         ; configure Timer0:
                ; --0-----              timer mode (T0CS = 0)
                ; ----0---              prescaler assigned to Timer0 (PSA = 0)
                ; -----100              prescale = 32 (PS = 100)
        option                      ;   -> increment every 32 us

;***** Main loop
main_loop
        ; delay 500 ms
        banksel dly_cnt
        movlw   .125                ; repeat 125 times (125 x 4 ms = 500 ms)
        movwf   dly_cnt
dly500  clrf    TMR0                ;   clear timer0
w_tmr0  movf    TMR0,w              ;   wait for 4 ms
        xorlw   .125                ;     (125 ticks x 32 us/tick = 4 ms)
        btfss   STATUS,Z
        goto    w_tmr0
        decfsz  dlycnt,f            ; end 500 ms delay loop
        goto    dly500

        ; toggle flashing LED
        movf    sGPIO,w
```

```
        xorlw   b'000100'       ; toggle LED on GP2
        movwf   sGPIO           ;   using shadow register
        movwf   GPIO

        ; repeat forever
        goto    main_loop
```

Here's the code developed in lesson 4, for turning on a LED when the pushbutton is pressed:

```
        clrf    sGPIO           ; assume button up -> LED off
        btfss   GPIO,3          ; if button pressed (GP3 low)
        bsf     sGPIO,1         ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO
```

It's quite straightforward to place some code similar to this (replacing the `clrf` with a `bcf` instruction, to avoid affecting any other bits in the shadow register) within the timer wait loop; since the timer increments every 32 instructions, there are plenty of cycles available to accommodate these additional instructions, without risk that the "TMR0 = 125" condition will be skipped (see discussion in example 1).

Here's how:

```
w_tmr0                          ;   repeat for 4 ms:
                                ;     check and respond to button press
        bcf     sGPIO,1         ;       assume button up -> indicator LED off
        btfss   GPIO,3          ;       if button pressed (GP3 low)
        bsf     sGPIO,1         ;         turn on indicator LED
        movf    sGPIO,w         ;     update port (copy shadow to GPIO)
        movwf   GPIO
        movf    TMR0,w
        xorlw   .125            ;   (125 ticks x 32 us/tick = 4 ms)
        btfss   STATUS,Z
        goto    w_tmr0
```

Complete program

Here's the complete code for the flash + pushbutton demo.

Note that, because GPIO is being updated from the shadow copy on every "spin" of the timer wait loop, there is no need to update GPIO when the LED on GP2 is toggled; the change will be picked up next time through the timer wait loop.

```
;************************************************************************
;                                                                       *
;   Description:    Lesson 5, example 2                                 *
;                                                                       *
;   Demonstrates use of Timer0 to maintain timing of background actions *
;   while performing other actions in response 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=12F509
    #include    <p12F509.inc>


;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC


;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO

        UDATA
dly_cnt res 1                   ; delay counter


;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k


;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration


;***** MAIN PROGRAM *****************************************************

;***** Initialisation
start
        ; configure ports
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        movlw   b'111001'       ; configure GP1 and GP2 (only) as outputs
        tris    GPIO
        ; configure timer
        movlw   b'11010100'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----100          prescale = 32 (PS = 100)
        option                  ;   -> increment every 32 us

;***** Main loop
main_loop
        ; delay 500 ms while responding to button press
        banksel dly_cnt
        movlw   .125            ; repeat 125 times (125 x 4 ms = 500 ms)
        movwf   dly_cnt
dly500  clrf    TMR0            ;   clear timer0
w_tmr0                          ;   repeat for 4 ms:
                                ;     check and respond to button press
        bcf     sGPIO,1         ;       assume button up -> indicator LED off
        btfss   GPIO,3          ;       if button pressed (GP3 low)
        bsf     sGPIO,1         ;         turn on indicator LED
        movf    sGPIO,w         ;     update port (copy shadow to GPIO)
        movwf   GPIO
        movf    TMR0,w
        xorlw   .125            ;   (125 ticks x 32 us/tick = 4 ms)
        btfss   STATUS,Z
        goto    w_tmr0
```

```
        decfsz  dly_cnt,f       ; end 500 ms delay loop
        goto    dly500

        ; toggle flashing LED
        movf    sGPIO,w
        xorlw   b'000100'       ; toggle LED on GP2
        movwf   sGPIO           ;   using shadow register

        ; repeat forever
        goto    main_loop


        END
```

Example 3: Switch debouncing

Lesson 4 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 switch is deemed to have changed when it has been continuously in the new state for some minimum period, for example 10 ms. This is done by continuing to increment a count while checking the state of the switch. "Continuing to increment a count" while something else (such as checking a switch) occurs is exactly what a timer does. Since a timer increments automatically, using a timer can simplify the logic, as follows:

```
reset timer
while timer < debounce time
      if input ≠ required_state
            reset timer
end
```

On completion, the input will have been in the required state (changed) for the minimum debounce time.

Assuming a 1 MHz instruction clock and a 1:64 prescaler, a 10 ms debounce time will be reached when the timer reaches 10 ms ÷ 64 µs = 156.3; taking the next highest integer gives 157.

The following code demonstrates how Timer0 can be used to debounce a "button down" event:

```
wait_dn clrf    TMR0            ; reset timer
chk_dn  btfsc   GPIO,3          ; check for button press (GP3 low)
        goto    wait_dn         ;   continue to reset timer until button down
        movf    TMR0,w          ; has 10ms debounce time elapsed?
        xorlw   .157            ;   (157 = 10ms/64us)
        btfss   STATUS,Z        ; if not, continue checking button
        goto    chk_dn
```

That's shorter than the equivalent routine presented in lesson 4, and it avoids the need to use two data registers as counters. But – it uses Timer0, and on baseline PICs, there is only one timer. It's a scarce resource! If you're using it to time a regular background process, as we did in example 2, you won't be able to use it for debouncing. You must be careful, as you build a library of routines that use Timer0, if you use more than one routine which uses Timer0 in a program, that the way they use or setup Timer0 doesn't clash.

But if you're not using Timer0 for anything else, using it for switch debouncing is perfectly reasonable.

Complete program

The following program is equivalent to that presented in lesson 4. By using Timer0 for debouncing, it's shorter and uses less data memory:

```
;************************************************************************
;   Description:    Lesson 5, example 3                                 *
;                                                                       *
;   Demonstrates use of Timer0 to implement debounce counting algorithm *
;                                                                       *
;   Toggles LED when pushbutton is pressed then released                *
;                                                                       *
;************************************************************************
;   Pin assignments:                                                    *
;       GP1 = indicator LED                                             *
;       GP3 = pushbutton switch (active low)                            *
;                                                                       *
;************************************************************************

    list        p=12F509
    #include    <p12F509.inc>


;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC


;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO


;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration


;***** MAIN PROGRAM *****************************************************

;***** Initialisation
start
        ; configure ports
        clrf    GPIO            ; start with LED off
        clrf    sGPIO           ;   update shadow
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO
        ; configure timer
        movlw   b'11010101'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----101          prescale = 64 (PS = 101)
        option                  ;   -> increment every 64 us
```

```
;***** Main loop
main_loop
        ; wait for button press, debounce using timer0:
wait_dn clrf    TMR0                ; reset timer
chk_dn  btfsc   GPIO,3              ; check for button press (GP3 low)
        goto    wait_dn             ;   continue to reset timer until button down
        movf    TMR0,w              ; has 10ms debounce time elapsed?
        xorlw   .157                ;   (157=10ms/64us)
        btfss   STATUS,Z            ; if not, continue checking button
        goto    chk_dn

        ; toggle LED on GP1
        movf    sGPIO,w
        xorlw   b'000010'           ; toggle shadow register
        movwf   sGPIO
        movwf   GPIO                ; write to port

        ; wait for button release, debounce using timer0:
wait_up clrf    TMR0                ; reset timer
chk_up  btfss   GPIO,3              ; check for button release (GP3 high)
        goto    wait_up             ;   continue to reset timer until button up
        movf    TMR0,w              ; has 10ms debounce time elapsed?
        xorlw   .157                ;   (157=10ms/64us)
        btfss   STATUS,Z            ; if not, continue checking button
        goto    chk_up

        ; repeat forever
        goto    main_loop


        END
```

Counter Mode

As explained above, Timer0 can also be used to count external events, consisting of a transition (rising or falling) on the T0CKI input.

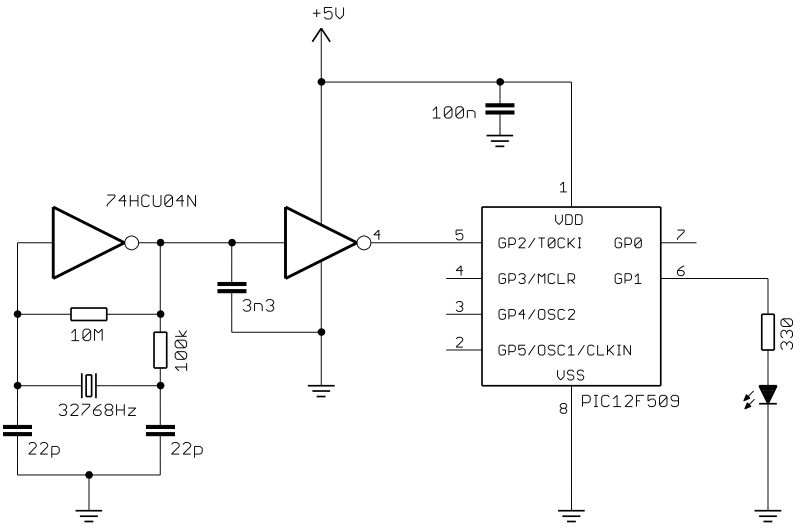
This is useful in a number of ways, such as performing an action after some number of input transitions, or measuring the frequency of an input signal, for example from a sensor triggered by the rotation of an axle. The frequency in Hertz of the signal is simply the number of transitions counted in 1 s.

However, it's not really practical to build a frequency counter, using only the techniques (and microcontrollers) we've covered so far!

To show how to use Timer0 as a counter, we'll go back to LED flashing, but driving the counter with a crystal-based external clock, providing a much more accurate time base.

The circuit used for this is as shown on the right.

A 32.768 kHz "watch crystal" is driven by a CMOS inverter to generate a 32.768 kHz clock signal.

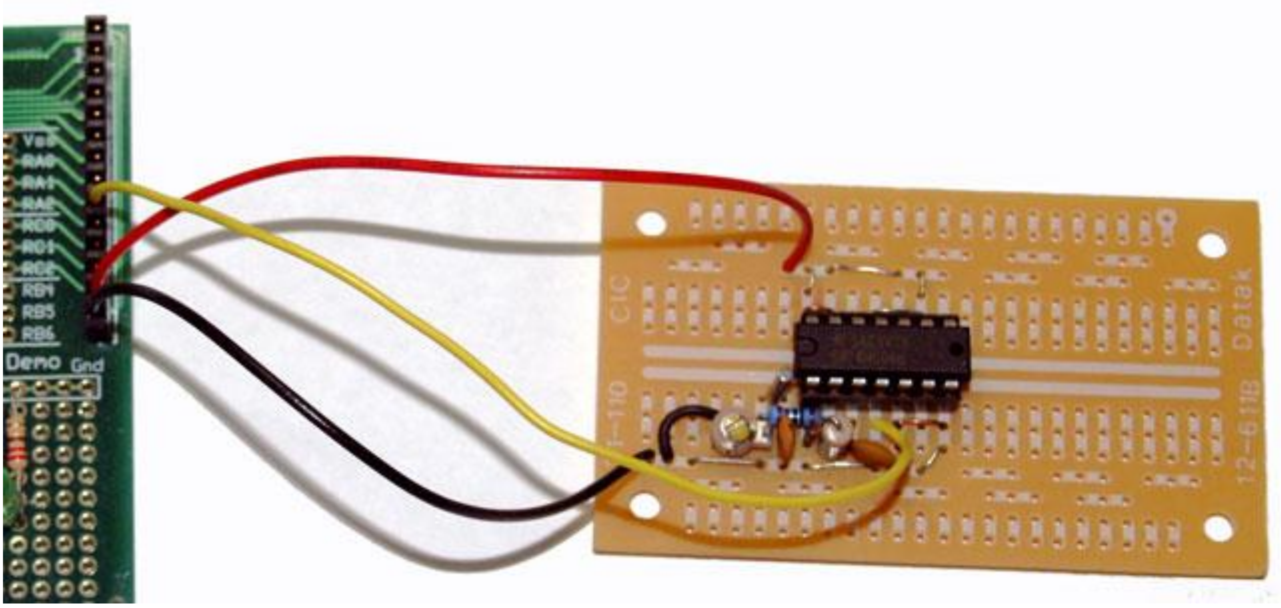


The capacitor and resistor values shown should work with most watch crystals designed for a 12.5 pF load capacitance. The exact value of the capacitor loading the first inverter (3.3 nF here) isn't important – and in theory isn't necessary – but in practice it helps the oscillator start reliably.

The oscillator output is buffered by another inverter, before being fed to the T0CKI (GP2) input on the PIC.

The Gooligum baseline training board already has this oscillator circuit in place (in the upper right of the board) – jumper JP22 connects the 32 kHz clock signal to T0CKI. And, as before, jumper JP12 enables the LED on GP1.

If you have Microchip's Low Pin Count Demo Board, you will need to build the oscillator circuit separately. Since it will be used in a number of lessons, and to limit the effect of stray capacitance, it's a good idea to build it on something more permanent than breadboard, such as a strip prototyping board. You should then connect it to the 14-pin header on the demo board (GP2/T0CKI is brought out as pin 9 on the header, while power and ground are pins 13 and 14), as illustrated in the photograph below:



Note that in this photo, one of the load capacitors is variable – this makes it possible to tune the oscillation frequency, but it's not really necessary.

We'll use this 32.768 kHz signal with Timer0, to generate the timing needed to flash the LED on GP1 at almost exactly 1 Hz (the accuracy being set by the accuracy of the crystal oscillator, which can be expected to be much better than that of the PIC's internal RC oscillator).

Those familiar with binary numbers will have noticed that $32768 = 2^{15}$, making it very straightforward to divide the 32768 Hz input down to 1 Hz.

Since $32768 = 128 \times 256$, if we apply a 1:128 prescale ratio to the 32768 Hz signal on T0CKI, TMR0 will be incremented 256 times per second. The most significant bit of TMR0 (TMR0<7>) will therefore be cycling at a rate of exactly 1 Hz; it will be '0' for 0.5 s, followed by '1' for 0.5 s.

So if we clock TMR0 with the 32768 Hz signal on T0CKI, prescaled by 128, the task is simply to light the LED (GP1 high) when TMR0<7> = 1, and turn off the LED (GP1 low) when TMR0<7> = 0.

To configure Timer0 for counter mode (external clock on T0CKI) with a 1:128 prescale ratio, set the T0CS bit to '1', PSA to '0' and PS<2:0> to '110':

```
        movlw   b'11110110'     ; configure Timer0:
                ; --1-----          counter mode (T0CS = 1)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----110          prescale = 128 (PS = 110)
        option                  ;   -> increment at 256 Hz with 32.768 kHz input
```

Note that the value of T0SE bit is irrelevant; we don't care if the counter increments on the rising or falling edge of the signal on T0CKI – only the frequency is important. Either edge will do.

Next we need to continually set GP1 high whenever TMR0<7> = 1, and low whenever TMR0<7> = 0.

In other words, continually update GP1 with the current value or TMR0<7>.

Unfortunately, there is no simple "copy a single bit" instruction in baseline PIC assembler!

If you're not using a shadow register for GPIO, the following "direct approach" is effective, if a little inelegant:

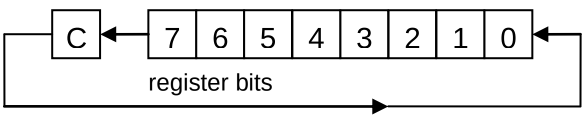
```
start   ; transfer TMR0<7> to GP1
        btfsc   TMR0,7          ; if TMR0<7>=1
        bsf     GPIO,1          ;   set GP1
        btfss   TMR0,7          ; if TMR0<7>=0
        bcf     GPIO,1          ;   clear GP1

        ; repeat forever
        goto    start
```

As described in [lesson 4], if you are using a shadow register (as previously discussed, it's generally a good idea to do so, to avoid potential, and difficult to debug, problems), this can be implemented as:

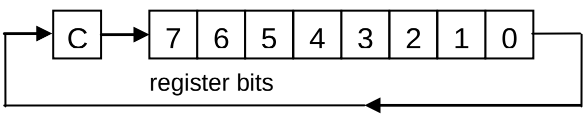
```
loop    ; transfer TMR0<7> to GP1
        clrf    sGPIO           ; assume TMR0<7>=0 -> LED off
        btfsc   TMR0,7          ; if TMR0<7>=1
        bsf     sGPIO,1         ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop
```

But since this is actually an instruction longer, it's only really simpler if you were going to use a shadow register anyway.

Another approach is to use the PIC's rotate instructions. These instructions move every bit in a register to the left or right, as illustrated:



'`rlf f,d`' – "**r**otate **l**eft **f**ile register through carry"



'`rrf f,d`' – "**r**otate **r**ight **f**ile register through carry"

In both cases, the bit being rotated out of bit 7 (for `rlf`) or bit 0 (for `rrf`) is copied into the carry bit in the STATUS register, and the previous value of carry is rotated into bit 0 (for `rlf`) or bit 7 (for `rrf`).

As usual, '`f`' is the register being rotated, and '`d`' is the destination: '`,f`' to write the result back to the register, or '`,w`' to place the result in W.

The ability to place the result in W is useful, since it means that we can "left rotate" TMR0, to copy the current value from TMR0<7> into C, without affecting the value in TMR0.

There are no instructions for rotating W, only the addressable special-function and general purpose registers. That's a pity, since such an instruction would be useful here. Instead, we'll need to rotate the bit copied from TMR0<7> into bit 0 of a temporary register, then rotate again to move the copied bit into bit 1, and then copy the result to GPIO, as follows:

```
        rlf     TMR0,w          ; copy TMR0<7> to C
        clrf    temp
        rlf     temp,f          ; rotate C into temp
        rlf     temp,w          ; rotate once more into W (-> W<1> = TMR0<7>)
        movwf   GPIO            ; update GPIO with result (-> GP1 = TMR0<7>)
```

Note that '`temp`' is cleared before being used. That's not strictly necessary in this example; since only GP1 is being used as an output, it doesn't actually matter what the other bits in GPIO are set to.

Of course, if any other bits in GPIO were being used as outputs, you couldn't use this method anyway, since this code will clear every bit other than GP1! In that case, you're better off using the bit test and set/clear instructions, which are generally the most practical way to "copy a bit". But it's worth remembering that the rotate instructions are also available, and using them may lead to shorter code.

Complete program

Here's the complete "flash a LED at 1 Hz using a crystal oscillator" program, using the "copy a bit via rotation" method:

```
;************************************************************************
;                                                                       *
;   Description:    Lesson 5, example 4b                                *
;                                                                       *
;   Demonstrates use of Timer0 in counter mode and rotate instructions  *
;                                                                       *
;   LED flashes at 1 Hz (50% duty cycle),                               *
;   with timing derived from 32.768 kHz input on T0CKI                  *
;                                                                       *
;   Uses rotate instructions to copy MSB from Timer0 to GP1             *
;                                                                       *
;************************************************************************
;                                                                       *
;   Pin assignments:                                                    *
;       GP1   = flashing LED                                            *
;       T0CKI = 32.768 kHz signal                                       *
;                                                                       *
;************************************************************************

    list        p=12F509
    #include    <p12F509.inc>


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

```
;***** VARIABLE DEFINITIONS
        UDATA_SHR
temp    res 1                   ; temp register used for rotates


;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration


;***** MAIN PROGRAM *****************************************************

;***** Initialisation
start
        ; configure port
        movlw   b'111101'       ; configure GP1 (only) as output
        tris    GPIO
        ; configure timer
        movlw   b'11110110'     ; configure Timer0:
                ; --1-----          counter mode (T0CS = 1)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----110          prescale = 128 (PS = 110)
        option                  ;   -> increment at 256 Hz with 32.768 kHz input

;***** Main loop
main_loop
        ; TMR0<7> cycles at 1Hz, so continually copy to LED (GP1)
        rlf     TMR0,w          ; copy TMR0<7> to C
        clrf    temp
        rlf     temp,f          ; rotate C into temp
        rlf     temp,w          ; rotate once more into W (-> W<1> = TMR0<7>)
        movwf   GPIO            ; update GPIO with result (-> GP1 = TMR0<7>)

        ; repeat forever
        goto    main_loop


        END
```

Conclusion

Hopefully the examples in this lesson have given you an idea of the flexibility and usefulness of the Timer0 peripheral.

With it, we were able to:

- Time an event
- Perform a periodic action while responding to input
- Debounce a switch
- Count external pulses

We'll revisit Timer0 later, and introduce other timers when we move onto the mid-range architecture.

But first, in the next lesson we'll take a quick look at how some of the MPASM assembler directives can be used to make our code easier to read and maintain, which will be important as our programs grow bigger.